Laboratoire d'Informatique
Fondamentale de Lille

N° Ordre : 1418

Année : 1995

THESE

présentée à

L'Université des Sciences et Technologies de LILLE

pour obtenir le titre de

Docteur de l'Université

spécialité

INFORMATIQUE

par

Howaida SHERIF AHMED
(Ingénieur B.Sc.- M.Sc.)

Multi-Résolution de Programmes PROLOG

Soutenue le vendredi 20 Janvier devant la commision d'examen :

Membres du Jury :

Président	J.M.GEIB	Professeur - LIFL
Rapporteurs	C.PERCEBOIS	Professeur - IRIT - Toulouse
	B.PLATEAU	Professeur - ENSIMAG - Grenoble
Directeur de thèse	B.TOURSEL	Professeur - EUDIL
Codirecteurs de thèse	G.GONCALVES	Professeur - Université d'Artois
	P.LECOUFFE	IUT-A - Villeneuve d'Ascq
Examinateurs	J.CHASSIN de KERGOMMEAUX	CNRS - IMAG/LGI - Grenoble
	Ph.DEVIENNE	CNRS - LIFL

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Remerciements

Cette étude a été effectuée au Laboratoire d'Informatique Fondamentale de Lille (LIFL - UA 369 du CNRS), dirigé par Monsieur le Professeur J.M.Geib qui a bien voulu me faire l'honneur de présider le jury de cette thèse.

Je remercie Monsieur le Professeur C.Percebois, qui a participé au jury en tant que rapporteur, pour ses remarques constructifs qui ont permis d'améliorer la rédaction du document.

Je remercie Madame B.Plateau d'avoir accepté de rapporter mon travail.

Je remercie Monsieur J.Chassin de Kergommeaux et Monsieur P.Devienne d'avoir accepté de participer au jury.

Je remercie Monsieur le Professeur B.Toursel qui m'a bien accueillie dans son équipe et qui a suivi la progression de ce travail.

Je remercie Monsieur le Professeur G.Goncalves qui a suivi l'évolution de ce travail, pour sa disposition à tout moment et ses discussions utiles.

J'adresse mes vifs remerciements à Monsieur P.Lecouffe, qui est l'initiateur de cette étude, pour son aide précieux, ses conseils et son soutien tout au long du travail.

Mes remerciements sont aussi adressés à Madame M.P.Lecouffe pour sa sympathie dans les moments les plus difficiles.

Je remercie Madame I.Deligniers qui a bien assuré la lecture de ce document.

Les simulations ont été effectuées à l'IUT-A (Informatique), je tiens à remercier les personnels pour leur patience et leur coopération.

J'adresse également mes remerciements aux collègues de l'équipe de recherche PALOMA pour leur soutien.

Je remercie la Mission Française de Recherche et de Coopération au Caire pour m'avoir proposé une bourse d'étude tout au long de mon travail depuis 1991.

Je tiens enfin à remercier Monsieur H.Glanc qui a assuré la reprographie de ce mémoire dans des délais très rapides.

Avec reconnaissance,

A ma famille

Preface

The past few years have seen an explosion of interest in the field of logic programming. An indication of interest is the numerous number of workshops and conferences held and the journals that are now devoted exclusively to logic programming.

Much of the current research involves techniques for implementing and improving logic programming languages. One of the attractions of logic programming is the clean separation between semantics and control. It is easy to separate *what* a program should compute from *how* an implementation can efficiently compute it. Accordingly, though early implementations were quite inefficient compared to conventional programming languages, there had been a promise for more efficient implementations since it is possible to experiment different implementation techniques without violating the semantics.

The major advantage of the separation of semantics from control, however, is the potential of parallelism. Logic programming languages, like languages based on applicative models, are often inefficient compared to traditional languages when implemented on the von Neumann architectures. More hope for efficient implementation lies in parallel architectures. In fact, some argue the other side of the coin: languages based on *non* von Neumann models provide the key to the acceptance of large scale parallel machines, due to inherent difficulties of exploiting parallelism in von Neumann languages.

Prolog has always been the mother tongue of logic programming languages. Since its existence, research efforts have never ceased to understand, study or improve Prolog. Different techniques were proposed to enhance the execution model of Prolog programs, whether sequential or parallel.

The main feature that characterises the classic execution model of Prolog is the *backtracking*. The use of traditional backtracking to explore a search space top down starts with the initial

state as the current state. Then, for each forward derivation step, one of the operators applicable to the current state is used to derive a new state. This forward execution is repeated until either a solution state is reached, and success is reported, or the set of unused operators applicable to the current state is empty. At this point, the search *backtracks*. The current state is dropped, its predecessor is redefined as the current state and forward execution restarts. If backtracking beyond the initial state is required, failure to find any more solutions is reported.

Thus, after reaching a failing state, the system simply returns to the previous states. Sometimes, however, doing this does not prevent the repetition of the same failure. Focusing on a failing state, a thrashing traffic can result where the system performs an exhaustive search over a sub space which is irrelevant to the failure. Several efforts were proposed to achieve intelligent backtracking that succeeded in reducing the number of retried states attempted to find all the solutions of a given problem.

In this thesis, an attempt is made to eliminate the classic deep backtracking while executing a finite non deterministic Prolog program. No returning to predecessor states takes place; each state is tried *once and only once*. The resolution resembles greatly a breadth-first search without the enormous evolution of memory space due to a synchronous OR layer that assembles all the possible solutions from an OR node, before attempting the following state. The execution starts from the initial state, where all alternatives are attempted, then all the resulting solutions are assembled into packets, successes and failures together, and these packets propagate in a forward sense to successive states and the same operation is repeated until the final state is attempted. The final outcome is a packet of solutions. This packet conforms normally to the outcome of standard Prolog, i.e. same number of solutions and same order as standard Prolog.

This model necessitates a detailed study. The idea is to change the manner of finding the solutions. We are faced with a packet of solutions propagating between the different states. Accordingly, variables are *multi-instantiated,* in the sense that a variable may be bound to more than one value at a time. Semantically, this means that the system attempts to solve the goal by one scan to its search space and stores all the possible alternative solutions during the multi-resolution.

Consequently, the standard unification algorithm implemented in the standard resolution model is not sufficient to support all the unification operations between two terms. The *multi-unification* algorithm specifies the manner of unifying two terms, whether non, mono

or multi-instantiated. It is based upon the classical unification algorithm but with added modifications to support multi-unification.

The failures occuring during the multi resolution were treated, at the first place, statically after the end of the resolution at the moment of the display of all possible solutions. When running large benchmarks, it proved to be inefficient due to its large consumption of memory space and computation time. We then attempted to treat, as much as possible, the failures during the unification operations dynamically to discard all useless instantiations during the following attempts to satisfy the coming states.

A simulator is built based on a meta interpreter, written in Prolog, with the objective to observe the behaviour of the proposed model. It executes the Prolog programs in the multi-resolution mode, performs the multi-unification, identifies correctly the multi-instantiations of the variables and produces the correct results. The simulator also supports the execution of programs including arithmetic and relational operations. Comparing the results obtained to that produced by standard Prolog, the output of the simulator conforms to Prolog, producing the same number of solutions, in the same order as Prolog.

The results are promising, in the sense that employing the multi-resolution model may result in speedups when compared to the standard resolution model, even in the sequential mode. This is due to the elimination of redundant computations where all the solutions are produced after one full scan of the search states.

Another potential of the model emerges from the fact that we succeeded to represent the ensemble of solutions in a vector-like form. We discuss different underlying intrinsic parallelism features that may be exploited in the multi-resolution model.

Organization of the Thesis

Chapter One :

In this chapter, a survey study is presented on logic programming, namely Prolog. This includes the classic execution model of Prolog and the main features that is of our interest. A special attention is given to the backtracking feature, where we discuss thoroughly its definitions, types and all the already attempted propositions and optimisations to ameliorate the execution of a standard Prolog program. To better understand the problem, we present a simulation of the AND/OR process model of Prolog that observes the backtracking behaviour. We terminate with the objectives of the work presented in this thesis.

Chapter Two :

Chapter two discusses the basic idea of the execution model that we are presenting. We quickly pass through the different phases, demonstrating the role of each phase in the proposed multi-resolution. Different underlying problems that might occur due to the replacement of the deep backtracking by the multi-instantiated objects are made clear.

Chapter Three :

Here, we define the new model for the execution of Prolog programs, the multi-resolution model. The main characteristics of the model are presented including the representations of the variables, the multi-unification algorithm, and the internal data structures that are employed to store the information of the search in the different states. We prove the soundness of our model by overcoming all the previous ambiguities discussed in the previous chapter.

Chapter Four :

In this chapter, we discuss the failures occuring during the multi-resolution of Prolog programs. Different techniques are proposed to treat different types of failures. A comparison between these techniques is given together with a full criticism on their performance.

Chapter Five :

This chapter is dedicated to a discussion on the evaluable arithmetic operations and predefined predicates and how to execute such instructions in the multi-resolution model. A representation of an arithmetic multi-instantiation is discussed followed by the multi-unification algorithm for arithmetic and relation operations. We discuss how I/O predicates may be implemented, taking the *write* as an example. It also serves for the display of the solutions to the users. We present two algorithms to display the solutions with a full comparison between them.

Chapter Six :

A detailed description of the simulator is presented. It is written in LPA MacProlog. It is based on a meta-interpreter for the standard resolution model as well as the multi-resolution model. Tested benchmarks are presented that demonstrate the performance parameters comparing multi-resolution to classical resolution and to related work discussing the same idea. We also discuss the complexity of the multi-unification algorithm with respect to the standard unification algorithm.

Chapter Seven :

Terminating the discussion on the model, we recapitulate the different properties of the multi-resolution model on situating each aspect with respect to the already existing research domains. We discuss the perspectives of this work, pointing out the underlying the potentials of parallelism in the multi-resolution model.

Contents

II Basic Idea: A Look Through

III The Multi-execution model

IV Failures in Multi-resolution

V Arithmetic and predefined predicates

VI Model Performance

VII Summary and Perspectives

Introduction

Ce travail est consacré à l'amélioration de temps d'exécution de programmes Prolog. Nous présentons le modèle de multi-résolution de Prolog dans lequel le retour arrière profond est éliminé. Ce modèle est basé sur une résolution OU synchrone et une gestion des multi-instanciations. Nous discutons toutes les phases différentes du modèle proposé. Nous comparons des résultats de simulation basée sur un meta-interpreteur avec la résolution standard. Nous terminons avec une ouverture sur les travaux futurs.

1. Introduction

L'accroissement des performances nécessaires au développement des applications en intelligence artificielle utilisant la programmation logique, notamment Prolog, nécessite une accélération des temps d'execution, pouvant etre obtenue en séquentiel et en parallèle.

Ce travail est une conséquence de plusieurs travaux menés précédemment au LIFL : d'abord sur un modèle d'exécution à grain fin basé sur l'arbre ET/OU [22,23] puis sur des améliorations du modèle multi-séquentiel [3] au niveau de la gestion de taches et des instanciations multiples. L'exécution parallèle de Prolog posant des problèmes au niveau des instanciations multiples, nous avons cherché à mettre en oeuvre un modèle articulé autour de ces instanciations multiples. Cela nous a conduit à un modèle gérant des environnements multiples, synchronisé au niveau OU, sans redondances d'exécution et sans le retour arrière classique. On appelle ce modèle **la multi-résolution de Prolog.**

Ce modèle n'est pas a priori un modèle d'exécution parallèle, mais doit être vu comme un modèle général d'exécution de programmes Prolog, non déterministes. Ce modèle peut être utilisé pour accélérer l'exécution séquentielle en bénéficiant de la non redondance de certains traitements. Nous montrons quelques résultats de simulations où des accélérations intéressantes ont été obtenues. De plus, les instanciations multiples des variables peuvent être tout naturellement représentées par des vecteurs, permettant de ce fait un traitement sur des machines travaillant en mode SPMD.

2 Optimisations du retour arrière

Pour accélérer l'exécution d'un programme Prolog, il y a trois approches différentes : la première est d'améliorer l'implantation de la WAM [45,46,47] pour une exécution plus efficace, la deuxième est d'exploiter le parallélisme du modèle d'exécution standard de Prolog et exécuter le programme sur une machine parallèle [6,7,9,10,18,48], et la troisième est de définir d'autres modèles d'exécution pour les programmes [12,15,26,30,34,38].

Nous avons choisit la troisième approche en tenant de conserver les potentialités offertes par les deux autres aspects. L'accélération séquentielle nous a tenté à l'étudier pour

plusieurs raisons : on utilise les machines séquentielles qui existent déjà et on ne change rien dans la syntaxe de programmes déjà écrits.

L'élément caractéristique du modèle d'exécution classique de Prolog est le retour arrière. Dans les programmes non déterministes, à cause des échecs et des retours arrière, on ne peut pas empêcher la répétition de certains calculs plusieurs fois. Pour mieux observer ce phénomène, des simulations ont été effectuées et nous avons remarqué que la ré-exécution de sous buts à cause du retour arrière pouvait être fréquente.

Pour avoir une modèle d'exécution plus efficace, on peut essayer de diminuer la redondance due au retour arrière.

Dans le cadre d'une étude bibliographique, nous présentons quelques travaux réalisés précédemment pour optimiser le comportement du retour arrière dans le modèle de la résolution standard de Prolog. Deux approches différentes peuvent être choisies: l'optimisation du nombre de retours arrière effectués ou l'élimination partielle du retour arrière. Nous discutons chaque approche plus en détail.

1- L'optimisation du comportement du retour arrière

a- Le retour arrière intelligent

Cette approche est basé sur la mémorisation de la cause des échecs précédents [4,5,12,13,19,30,31,35,36]. Considérons le programme suivant :

p(a). *p(b).*
q(1). *q(2).*
r(b,1). *r(b,2).*

et la question *:- p(X), q(Y), r(X,Y).*

Dans une résolution standard de Prolog, la variable *X* est lié à *a* et *Y* à *1*. Le sous-but *r(a,1)* échoue. Un retour arrière est effectué et *Y* est lié à *2* et essayer de résoudre *r(a,2)*, conduit à un nouvel échec.

Un interpréteur intelligent analyse la cause d'échec. Il mémorise que le sous-but *r(a,_)* échoue toujours à cause de la présence de l'atome *a* en premier argument. Dans ce cas,

quand le sous-but *r(X,Y)* donne un échec, le système effectue un retour arrière jusqu'au point de choix *p(X)* (et pas *q(Y)*) sans perdre de temps à essayer des alternatives différentes pour *Y*.

b- Le retour arrière semi-intélligent

Considérons la question:

:-p(A), q(B), r(A).

avec les mêmes faits que pour le retour arrière intelligent. Quand *r(A)* échoue, on peut sauter *q(B)* dans le retour arrière parce que ce sous but ne peut pas donner une alternative de *A* pour que *r(A)* réussisse. Cette approche est valable pour les sous-buts independent (qui ne partagent pas les variables) [34].

2- L'élimination partielle du retour arrière

Dans les deux approches précédentes, on a essayé d'optimiser le nombre de foi le retour arrière est effectué. D'autres travaux ont tenté d'éliminer partiellement le retour arrière.

DAP Prolog, proposé par Kacsuk et al. [26], est une extension du Prolog standard tournée vers une exécution SIMD. Dans cette approche, seulement des parties de programmes peuvent s'exécuter sans retour arrière. Ces parties ne comportent du non déterminisme que sur des faits. Pour des règles différentes dans un point de choix, le retour arrière classique est effectué.

Dans MultiLog[38,39,40], on élimine partiellement le retour arrière en ajoutant un opérateur (**disj**) avant quelques sous buts non déterministes. Ces sous buts sont résolus en essayant toutes leurs alternatives puis on passe une disjonction de solutions au sous but suivant. Dans ce modèle il y a un encombrement de taille mémoire du à la multiplication des environnements : en effet, chaque fois qu'une nouvelle alternative est explorée, on copie tous les environnements.

Une étude bibliographique plus complète sur ces travaux est présentée au chapitre 1.

3 Notre proposition

Tout les approches pour éliminer le retour arrière déjà proposées concernent des parties de programmes (base de faits en DAP Prolog). Le programmeur doit sélectionner des parties de programmes pour ajouter des annotations (*set_mode* pour DAP Prolog ou *disj* pour MultiLog). Nous proposons une approche plus générale, transparente à l'utilisateur, où on élimine complètement le retour arrière profond quelle que soit la nature du programme (base de faits, règles, récursivité) et ceci sans modifier la syntaxe originale des programmes.

Notre idée de base est de ne pas répéter un travail déjà réalisé. Nous éliminons totalement le retour arrière profond: on résout chaque sous but une seule fois en essayant tous les alternatives (têtes de clauses) mais on conserve le retour arrière superficiel. Ensuite, on rassemble toutes les instanciations des variables avant de tenter le sous but suivant. On ne retourne jamais en arrière au sous but précédent.

Dans le chapitre 2, nous présentons plus en détail les idées de base du modèle.

Considérons le début du programme de génération de nombres premiers suivant :

```
gener(1). gener(2). gener(3). gener(4).
impair(X) :- X mod 2 is 1.
premier(X):- ...
```

et la question :- *gener(X), impair(X), premier(X).*

En Prolog classique, à cause des échecs et sorties de solutions, le sous-but *impair* est exécuté 4 fois, le sous-but *premier* 2 fois et 2 solutions sont sorties pendant la résolution. Dans notre modèle, il n'y a pas de retour arrière profond, les sous-buts *impair* et *premier* ne sont exécutés qu'une seule fois, ce qui implique que les 4 cas possibles pour le sous-but *gener* soient exécutés avant de passer au sous-but *impair*. Plus précisément les 4 faits *gener* doivent être exécutés, les 4 instanciations différentes de *X* doivent être mémorisées. Quand toutes les alternatives d'un sous-but ont été exécutées (ce n'est pas toujours des faits comme ici) une phase de synchronisation OU est effectuée. Pour chaque variable de sous-but, les instanciations sont regroupées sous forme

d'instanciations multiples. Pour l'exemple ci-dessus, les 4 instanciations de *X* sont regroupées, de façon ordonnée, dans une structure notée :

$$\{ 1, 2, 3, 4\}$$

La variable *X* est alors instanciée à la structure précédente, elle est dite multi-instanciée et sera reconnue comme telle par la suite par rapport aux variables mono-instanciées classiques qui coexistent. Le premier sous-but étant complètement exécuté, le deuxième sous-but *impair* peut être à son tour exécuté. La variable *X* étant multi-instanciée; pour 2 instanciations il y aura succès et pour les deux autres il y aura échec. Bien entendu comme la variable *X* est multi-instanciée l'unification classique ne peut pas être utilisée et il faut utiliser la multi-unification décrite ci-dessous. Pour la suite de la multi-résolution, il est nécessaire de garder trace de ces succès et de ces échecs. Plusieurs solutions sont alors possibles, comme:

- modifier la variable X, mais cela pose des problèmes si le sous but a plusieurs alternatives.
- ne pas modifier la variable et garder des informations sur l'échec; c'est la solution qui a été retenue.

La prise en compte des échecs, mais aussi des problèmes techniques de mise en oeuvre du modèle, nécessitent d'ajouter à chaque instance de variable une information supplémentaire dite *date d'instanciation*. Pour une exécution séquentielle chaque unification (ou multi-unification) effectuée est numérotée (datée) par un entier positif et toutes les variables instanciées lors de cette unification ont cette date qui leur est associée. De plus, on numérote les points de choix dans un ordre croissant selon leur apparition dans la multi-résolution. Ainsi, la variable *X* de l'exemple précédent sera effectivement multi-instanciée à la structure :

$$\{ \mathbf{1}, [(\underline{1}, 1), (\underline{2}, 2), (\underline{3}, 3), (\underline{4}, 4)]\}$$

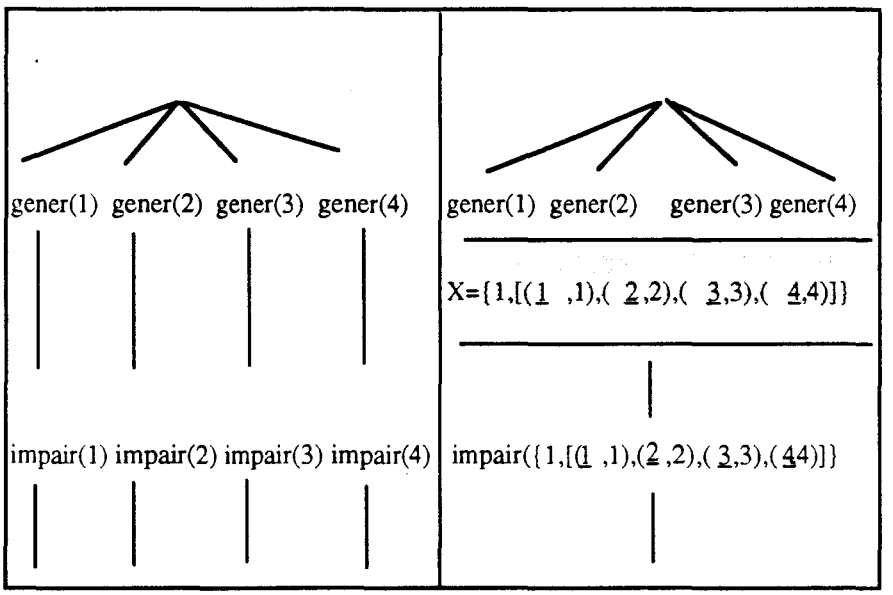
le chiffre en gras est le numéro du point de choix, et les petits chiffres soulignés correspondent aux dates d'instanciations des différentes instances. La création de cette multi-instanciation se passe dans la phase OU synchrone de ce point de choix.

Le numéro du point de choix et les dates ont deux rôles essentiels: d'une part déterminer si deux variables ont été instanciées en même temps ou pas, d'autre part mémoriser les échecs.

Les détails du modèle sont présentés en chapitre 3.

4 Le modèle de multi-résolution

Le modèle de multi-résolution diffère sensiblement de la résolution classique. La multi-résolution ressemble a priori beaucoup à un parcours d'arbre en largeur d'abord, mais ce n'est pas le cas à cause de la synchronisation OU. Voici une partie des arbres de résolution de l'exemple précédent dans le cas classique et dans le cas de multi-résolution :



Arbre de résolution classique *Arbre de multi-résolution*

Dans l'arbre de multi-résolution, la synchronisation OU est représentée par la zone grisée. On remarque que le sous but *impair* n'est exécuté qu'une seule fois, mais avec une multi-instanciation.

La résolution classique de Prolog fait un parcours d'arbre de gauche à droite en profondeur d'abord. Dés qu'une solution est trouvée, elle peut être sortie. La multi-

resolution fait un parcours d'arbre de gauche a droite, quasi largeur d'abord et les phases OU synchrone créent les multi-instanciations, alors toutes les solutions apparaissent en même temps à la fin de la multi-résolution.

Donc, le modèle de multi-résolution est divisé en deux phases : la phase de multi-exécution et la phase de multi-sorties.

Ci-dessous nous discutons chaque phase en plus de détails.

1- La phase de multi-exécution

C'est la première phase du modèle de multi-résolution. Elle correspond à la résolution multiple d'un but donné. Les entrées sont un programme Prolog et une question. Chaque sous-but est résolu une seule fois en essayant toutes les alternatives de manière séquentielle (dans l'ordre d'écriture dans le programme). A la suite de chaque sous-but, est exécuté une phase OU synchrone où les multi-instanciations sont crées. Ceci est local à chaque point de choix.

Du fait de la présence des variables multi-instanciées, l'algorithme standard d'unification ne peut pas être utilisé. C'est pourquoi on a définit l'algorithme de multi-unification.

L'algorithme de multi-unification

La multi-unification est l'unification standard de Prolog à laquelle a été ajoutée la prise en compte des multi-instanciations. Les détails de cet algorithme est présenté en chapitre 3, et on le compare avec l'algorithme d'unification standard.

La figure ci-dessous précise le fonctionnement de la multi-unification pour tous les cas où des multi-instanciations apparaissent. X et Y sont des variables, x et y sont des constantes, x_i et y_i sont des variables ou des constantes, $\{C_i,...\}$ représente une multi-instanciation, d_i est la date de branche du sous-terme x_i. ':=' est une opération d'affectation, et '=' symbolise une opération de multi-unification. Les premières colonnes sont les termes à unifier, la troisième indique les actions à effectuer et la dernière indique la complexité de ces actions.

terme X	terme Y	action(s)	complexité
X	{Cy,...}	X:={Cy,...}	*O(1)*
{Cx,...}	Y	Y:={...}	*O(1)*
x	{Cy,[(d_1,y_1),...,(d_n,y_n)]}	$x=y_i$ pour i=1-n	*O(n)*
{Cx,[(d_1,x_1),...,(d_n,x_n)]}	y	$y=x_i$ pour i=1-n	*O(n)*
{C,[(d_1,x_1),...,(d_n,x_n)]}	{C,[(d1,y_1),...,(dn,y_n)]}	$x_i=y_i$ pour i=1-n	*O(n)*
{Cx,[(d_1,x_1),...,(d_n,x_n)]}	{Cy,[(d_1,y_1),...,(d_m,y_m)]}	$x_i=y_j$ pour i=1-m et j=1-n	*O(m*n)*
f1({C,[(d_1,x_1),...,(d_n,x_n)]}	f2(t)	échec	*O(1)*

Deux algorithmes de multi-unification sont présentés : le premier pour les opérations non arithmétiques qui sont représenté par partage de données, et le deuxième pour les opérations arithmétiques qui sont présenté par copie. Ce dernier est discuté au chapitre 5.

Durant la multi-unification, lors des accès aux sous-termes, des dates de branche sont rencontrées. Ces dates sont assemblées pour former ce qu'on appelle des chemins d'accès aux sous-termes. Ce sont ces chemins qui servent à mémoriser les échecs.

Dans Prolog classique, l'unification d'un sous-but conduit soit à un échec soit à un succès. Une des particularités de la multi-unification est de conduire à trois cas possibles dans le cas où il y a des variables multi-instanciées :

- il y a succès pour toutes les combinaisons des variables multi-instanciées, on a alors un succès total.
- il y a succès pour certaines combinaisons des variables multi-instanciées et il y a échec pour certaines combinaisons des variables multi-instanciées, on a alors un échec/succès partiel.
- il y a échec pour toutes les combinaisons des variables multi-instanciées, on a lors un échec total.

En cas d'échec total, on dé-instancie toutes les variables qui ont été instanciées, et un retour arrière superficiel est effectué pour essayer la tête de clause suivante comme en Prolog classique.

La multi-unification doit aussi mémoriser des informations concernant les échecs partiels et traiter les échecs éventuels provenant de multi-unifications précédentes. Ces échecs

partiels sont gardés dans une structure de données spéciale qu'on appelle la base des échecs.

Dans chapitre 4, nous présentons en détail le traitement des échecs dans le modèle de multi-résolution.

2 Le phase de multi-sorties

C'est le deuxième phase du modèle de multi-résolution. Ses entrées sont les structures de données crées dans la phase de multi-éxecution, c'est-à-dire les multi-instanciations des variables de la question, la base des échecs et l'arbre de multi-résolution. C'est ici, qu'on affiche les solution à l'utilisateur.

Au début du travail, nous avons écrit un algorithme qui reconstitue l'arbre de résolution standard en profondeur d'abord et de la gauche à droite. Avant de considérer une instanciation d'une variable, on vérifie en consultant la base des échecs que cette instanciation n'a pas donné un succès/échec partiel sur la branche courante. Cet algorithme permet de sortir les mêmes solutions que Prolog (même nombre de solutions et même ordre).

Après quelques simulations, il est apparu que cet algorithme consomme souvent un temps plus élevé que le temps de multi-exécution, ceci à cause de la recréation de l'arbre de la résolution standard qui est fréquemment complexe.

Nous avons donc écrit un autre algorithme de sortie de solutions plus efficace pour profiter pleinement des performances du modèle dans la phase de multi-exécution. Nous avons abandonné la reconstitution de l'arbre de résolution standard : les solutions sont produites à l'aide des multi-instanciations et de la base des échecs. Cet algorithme sort les bonnes solutions, mais ni dans le même ordre ni le même nombre que Prolog standard : Prolog standard peut produire (de façon qu'on peut juger redondante) des solutions identiques qui n'apparaissent qu'en un seul exemplaire dans cet algorithme.

Les deux algorithmes sont présentés en détail au chapitre 5, avec quelques prédicats prédéfinits. Les différences en terme de performances du modèle en employant chaque algorithme de multi-sorties sont présentés dans le chapitre 6 avec d'autres résultats de simulations.

5 Performances du modèle

Le modèle de multi-résolution a été validé à l'aide d'un simulateur basé sur un méta-interpréteur écrit en Prolog. Les entrées sont un programme Prolog standard, sans aucune modification de la syntaxe et une question à résoudre, et les sorties sont les solutions possibles.

Nous avons étudié deux types de performances dans ce modèle: l'accélération et la consommation de mémoire. Pour le premier, nous avons mesuré l'accélération théorique et mesures de temps.

Nous avons mesuré le temps passé dans chaque phase de la multi-résolution : la multi-éxecution et la multi-sorties. Pour la dernière, les chiffres indiquent la différence de performance entre les deux algorithmes de sortie de solutions.

Nous avons simulé aussi le modèle standard de l'exécution de Prolog où tout est écrit de façon aussi semblable que possible. Nous avons calculé l'accélération d'exécution du modèle multi-résolution par rapport au modèle standard. Dans certains cas, on a eu une amélioration très intéressante.

La performance du modèle est présenté au chapitre 6.

Une autre mesure de performance est l'accélération théorique. Cette mesure est inspiré du travail du Kacsuk et al. [26]. Ils mesurent l'accélération en terme de taille de l'arbre de résolution de leur modèle par rapport au modèle standard. Nous avons fait le même avec le modèle de multi-résolution en mesurant la taille de l'arbre de multi-résolution et la comparant avec celle de la résolution standard. Les chiffres donnés montrent la différence dans taille de l'espace de recherche de deux modèles.

L'accélération de l'exécution d'un programme donné se fait souvent au détriment de l'encombrement mémoire. Cet encombrement mémoire est le dernier paramètre que nous avons mesuré. Il est évident que notre modèle d'exécution occupe plus de mémoire que le modèle standard pour exécuter le même programme. Par contre, à cause de la synchronisation OU, la multi-résolution n'a pas la complexité d'un parcours d'arbre en largeur d'abord. L'encombrement mémoire théorique du modèle standard de Prolog (profondeur d'abord) est de l'ordre de $O(d)$ où d est la profondeur moyenne de l'arbre de

résolution. Pour le modèle largeur d'abord cet encombrement est de l'ordre $O(b^d)$ où b est le nombre moyen de branche de point de choix. Dans le modèle multi-résolution, la profondeur et la largeur moyennes de l'arbre multi-résolution sont les même mais par contre l'encombrement de mémoire théorique est de l'ordre $O(b*d)$. Ça montre que le modèle de multi-résolution se situe entre le modèle de profondeur d'abord et le modèle de largeur d'abord.

6 Conclusion

Nous avons présenté le modèle de multi-résolution qui est une amélioration de l'exécution de certains programmes écrits en Prolog. La seule contrainte est que ça soit un programme fini (déterministe ou non déterministe). Nous avons éliminé le retour arrière profond et conservé le retour arrière superficiel. Chaque sous but est résolu une seule fois, en essayant toutes les alternatives. Les solutions sont assemblés pour créer des multi-instanciations. Des algorithmes pour les multi-unifications pour des données représentées par copie ou par partage sont présentés. Les échecs sont gérés et nous avons comparé l'ensemble des solutions qui sortent avec celles fournis par Prolog standard.

Nous avons ecrit une simulation, basée sur un meta-interpreteur de la multi-resolution et l'avons comparé avec le modèle standard de Prolog vis à vis le temps d'exécution et de l'encombrement mémoire. Des accélérations encourageantes sont obtenues.

Le modèle présenté possède des potentialité de parallélisme au niveau des données. Ca merite une étude plus profonde pour exploiter ce type de parallélisme. Des propositions pour des travaux futurs à effectuer concernent l'implantation du modèle en séquentiel et en parallèle. Une adaptation de la WAM au modèle de multi-résolution serait intéressante à étudier pour pouvoir faire tourner réellement le modèle sur une machine séquentielle, voir parallèle.

Chapter One

Motivations and Objectives

Abstract

A survey study is presented on logic programming, namely Prolog. Certain basic concepts in the language that we will be frequently recalling are emphasised. Existing optimisations to enhance the performance of the execution of Prolog programs are discussed. Finally, the motivations and objectives of this work are highlighted.

I.1 Introduction

The phrase *logic programming* refers to the use of formulae of first order predicate logic as statements of a programming language. The key idea underlying logic programming is *programming by description*. In traditional software engineering, one builds a program by specifying the operations to be performed in solving a problem, that is by saying *how* the program could be solved. In logic programming, one constructs a program by describing its application area, that is, by saying *what* is true. At the heart of the program is an application independent inference procedure, which accepts queries from users, the facts in its knowledge base, and deduces conclusions, and sometimes recording these conclusions in its knowledge base. Thus, the attractive feature of logic programming is the *clean separation* of semantics and control. It is easy to separate specification of what a program should compute from how an implementation can efficiently compute it.

A logic program may be defined as a set of axioms, or rules, defining relationships between objects. A program defines a set of consequences. A computation of a logic program is a deduction of the consequences of the program. The art of logic programming is in constructing concise and elegant programs that have the desired meaning. When interrogated by the user, the inference procedure replies after drawing conclusions from the facts in the knowledge base.

The first logic programming system, Prolog, was developed by Colmerauer and his colleagues at Marseilles[6] growing out of a project to implement an automatic theorem prover. Since then, the semantics of logic as a programming language have been formalized and there have been a number of implementations of Prolog, a high level language that extends the formalism of logic programming in ways that makes it more useful and efficient for problem solving.

A pure Prolog program is basically a logic program, in which an order is defined for both clauses in the program and goals in the body of the clause. Two major decisions have to be taken into account to convert the abstract interpreter for logic programs into a form suitable for a concrete programming language. First, the arbitrary choice of which goal in the resolvant to be reduced, i.e. the scheduling policy, must be specified, and second the nondeterministic choice of the clause from the program to effect the reduction must be implemented.

Several logic programming languages exist. Loosely, there are two categories; Prolog and its extensions (Prolog II, IC Prolog, MU Prolog) which are based on sequential execution [16,32]. Other languages such as Parlog, Concurrent Prolog, GHC are based on parallel execution [5,18,43]. The distinction between Prolog and its extensions is in the choice of the goal to reduce. Prolog's execution mechanism is obtained from the abstract interpreter by choosing the leftmost goal instead of an arbitrary one. This strategy will be fully explored in the following sections. Also nondeterministic choices of a clause are replaced by defining search for a matching clause and backtracking.

The remaining sections of this chapter explore different specific aspects of Prolog. The objective is not to recite previous definitions, but to emphasize certain basic concepts in the language before starting to discuss the work presented in this thesis. First, we quickly recall the definitions of the different elements that construct a Prolog program. We then highlight mostly on the execution model of Prolog, together with the backtracking phenomenon which is of major interest to our work. How Prolog behaves when a failure is reported, and how unification takes place are two important concepts that we are keen to discuss. Finally, we terminate this chapter by a discussion on the motivations and objectives of the work that we present in this thesis.

I.2 Basic Elements in a Prolog Program

The basic constructs of a Prolog program are terms and statements inherited from logic. There are three basic statements; facts, rules and queries. There is one single data structure; the logical term. We present each of these constructs, together with demonstrating examples.

I.2.1 Facts

Facts are a means of stating a relationship between objects. It is also called a *predicate*. An example is

father(john,mary).

The above statement is a fact, called *father*, stating that john is the father of mary. The objects that are enclosed within the round brackets are called the *arguments*.

Facts that have the same name are normally defined consecutively in a logic program. Semantically, this means that the relationship *father* is applicable to a certain set of argument pairs.

A finite set of facts constitutes a Prolog program, which is in its simplest form. The program in this case could also be regarded as a database.

I.2.2 Queries

The second form of the statements in a logic program is a *query*. It is the sole means for retrieving information from a Prolog program. It is basically a question, the user interrogating the program about the validity of a certain relationship between certain objects. Thus,

:- father(john,mary).

is a query whose answer is yes. We sometimes call a query a goal.

A simple query consists of a simple goal. A more general query consists of a number of subgoals such as:

:-p(X), q(Y), r(X,Y,Z).

How does a program respond to a query? If the query is a simple interrogation of facts, this is straight forward. If a fact identical to the query is found, then the answer is *yes*, otherwise the answer is *no*. But facts are simple statements in a Prolog program. More complex statements , called *rules*, may exist. Following, we present the definition of a rule, then in section I.3, we describe how rules are solved.

I.2.3 Rules

Rules, sometimes called implications, are statements of the form

A:- $B_1,B_2,\ldots,B_n$

We define A as the head of the rule, and $B_1,\ldots,B_n$ are the body of the rule A.

The above rule is read declaratively *A is implied by the conjunction of the B_i's* and interpreted procedurally by *to solve A, solve the conjunctive query (subgoals) $B_1,...B_n$.*

An example is a rule expressing the *son* relationship,

son(X,Y):-father(Y,X),male(X).

Procedurally, reading the above rule we have to prove that *X* is the son of *Y*, prove that *Y* is the father of *X* as well as *X* is male.

Rules serve in two different ways. They are the means of expressing new or complex queries in terms of simple queries as well as implying a set of facts to prove a rule. Here, a new query *son* relationship has been built from simple queries of *father* and *male* relationships.

I.2.4 The Logical Term

The term is a single data structure in logic programs. It could be a constant, a variable or a compound term. Constants and variables are terms, e.g. *a, john, 1,* and *X, Result,...*etc. A compound term is a *functor* and a number of arguments with a certain *arity*. The functor consists of its name, which is an atom and the arity of the functor is the number of arguments, e.g. $f(a_1,...,a_n)$ where *f* is the functor, a_i is the i^{th} argument, and the arity of this compound term is *n*.

I.2.4.1 The Logical Variable

What if the required query is to ask the program *who is the father of mary?* There are several possible ways to answer this query. One possible way is to ask all the facts sequentially until one responds with a correct answer. Apparently, this is a costy and tedious way. A better way is to make the query resemble actually the desired question. In this case, the above query is represented as

:- father(X,mary).

where *X* is an unknown term, namely a *logic variable*, that may be assigned to *john* as a result of the query.

Hence, a query containing a variable asks whether there is a value for the variable that makes the query a possible logical consequence of the program.

At the beginning of the resolution, variables are unbound (free or uninstantiated), i.e their values are unknown. During execution, variables take values, thus become bound, or instantiated to another term. This other term becomes the value of the variable. Once the variable is instantiated, it keeps its value. If a variable occurs in many places, every occurrence takes the same value at the same time.

I.3 Control in Prolog

The control in Prolog programs is like in conventional procedural languages as long as the computation progresses forward. Goal invocation corresponds to procedure invocation, and the ordering of goals in the body corresponds to the sequence of statements. We define the term *resolvent* which is the current goal at any stage of the computation. Prolog employs a procedural reading where each goal atom is viewed as a procedure The clause,

A:- $B_1,\ldots,B_n$

can be viewed as a definition of a procedure A similar to

procedure A
  call B_1,
  call B_2,
  ...
  call B_n
end.

To satisfy A, Prolog attempts to satisfy each goal, from B_1 to B_n, in turn by searching a matching goal in the database. All goals have to be satisfied in order for the satisfaction of the main goal A. Assuming there are different alternatives for each subgoal, Prolog attempts to satisfy the first subgoal of B_1. If this reports a success, Prolog will mark its place with a place marker and attempts to satisfy the second subgoal. If the second subgoal is satisfied, then Prolog will also mark that goal's place in the database and starts to attempt B_3.,...and so on. The resolution continues in the forward sense, from left to right, until B_n is satisfied. At this point, we say that the goal A is satisfied and success is reported.

I.4 Matching &Unification

A prerequisite to solving a resolvant is to attempt to unify the goal with a clause (fact or rule) head. Unification is a pattern matching operation. Initially, all variables in both terms should be free or uninstantiated. Two terms are unifiable if they are syntactically identical, or if variables in either terms can be replaced by terms in order to make them identical. The terms are identical if they have the same function symbol, the same arity, and the corresponding argument terms are unifiable.

When unifying two terms, a variable can be replaced by another term, including another variable, as long as the replacement is consistent throughout the two terms. For example, *q(f(a))* and *q(X)* can be unified, since when *X* is replaced by *f(a)* in *q(X)* both terms are equivalent; *q(f(a))*.

The two input terms to be unified must not have any variables in common. The scope of a variable is the clause that contains it, and hence the effect of a binding is confined to the resolvent, and the variables in the input clauses are not modified.

Unification is the operation that binds variables during a proof. The set of bindings created during unification is known as the substitution. After unifying two terms, Prolog applies the substitution generated during unification to the remaining occurrences of the variables in the two input clauses in order to form the resolvent.

The rules for deciding whether a goal matches the head of a clause are as follows:

- An uninstantiated variable will unify with any object, whether another variable, an atom or a structure. As a result, this object will be what the variable stands for. In the case of two variables, as soon as one is instantiated, the other will immediately be instantiated to that same value.
- An atom (integer or symbol) will only match with itself.
- A structure will match with another structure having the same functor name and the same number of arguments, taking into account that all the corresponding arguments must match.

We will rediscuss the standard unification algorithm in more details in chapter 3.

I.5 Resolution Tree

For a Prolog program, a search tree (sometimes called goal tree) is a tree where each node represents a goal statement. Immediate descendants of a node N are goal statements derivable from N in one inference step. The root of the tree is the goal, G, typed by the user and the interior nodes are the subgoal statements, and the leaves of the tree are either null statements or failure nodes.

There is an edge leading from a node N for each clause in the program P whose head unifies with the selected goal. Each branch in the tree from the root is a computation of G by P. Leaves of the tree are the either success nodes or failure nodes, depending whether the selected goal has been reached or not. Success nodes correspond to the solutions of the root of the tree. The number of success nodes is the same for any search tree of a given goal with respect to a program. Search trees contain multiple success nodes if the query has multiple solutions.

Some conventions are considered when searching trees in Prolog. The leftmost goal of a node is the goal always selected. The edges are labeled with substitutions that are applied to the variables in the leftmost goal. These substitutions are computed as part of the unification algorithm.

We define a number of definitions that we will be frequently recalling afterwards:

I.5.1 A Choice Point

A choice point is a node representing a subgoal, that includes more than one alternative (clause head) in the program. Often, choice points offer more than one solution to the same goal.

Figures (I.1a) and (I.1b) point out the difference between a single clause head and a choice point.

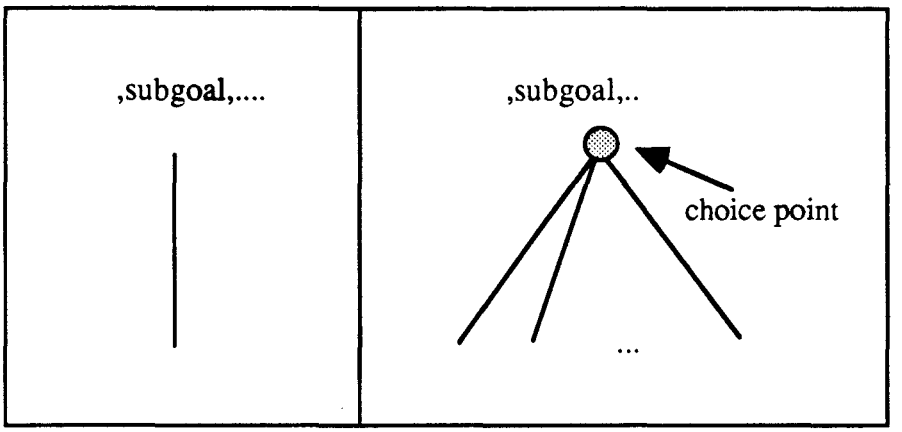


Figure (I.1a) :
A single clause head

Figure (I.1b) :
A choice point

I.5.2 Solution Paths

A solution path is a continuous path in the resolution tree starting from the root, representing the goal, and terminating by a leaf node, representing a solution. For a multiple-solutions goal, we have multiple solution paths in the same resolution tree.

Example:

Consider the program,

```
p(a).        p(b).        p(c).
q(a,1).      q(a,2).      q(c,4).      q(c,6).
r(Y):- Y < 6.
s(a).        s(b).
```

solve the query,

```
:- p(X), q(X,Y), r(Y), s(Z).
```

The resolution tree representing the different solutions of X,Y and Z is given by:

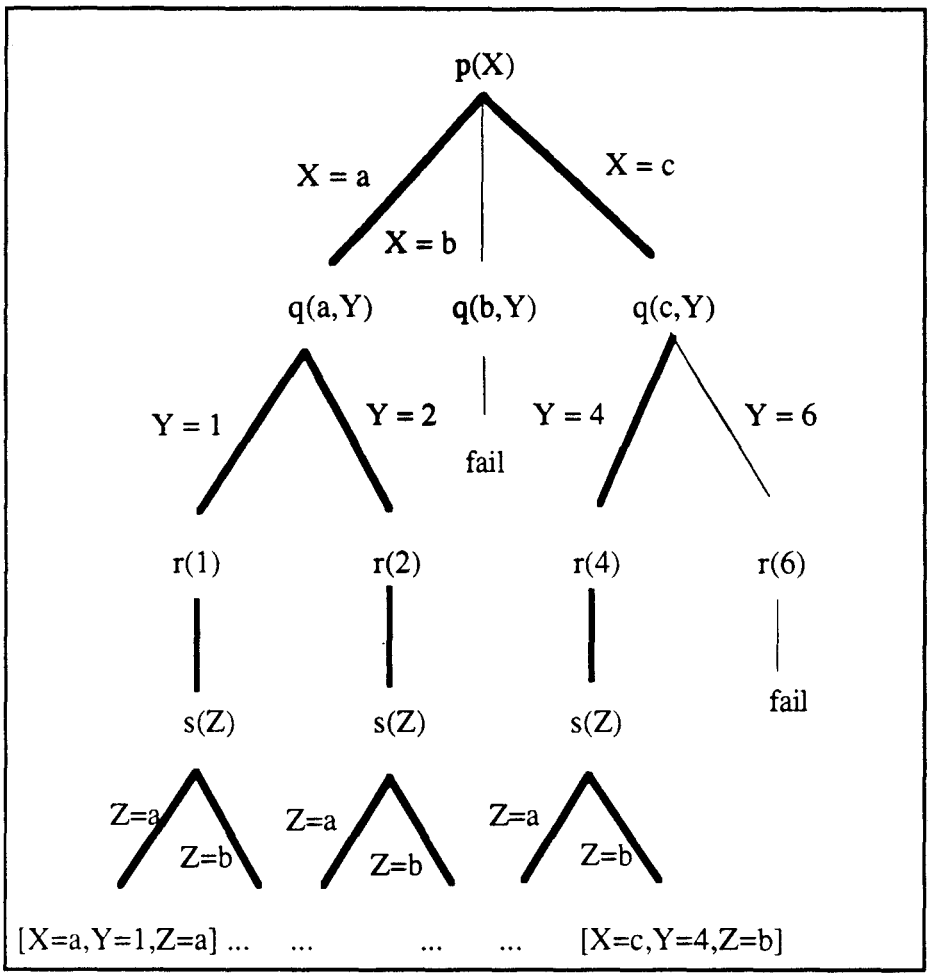


Figure (1.2) : Solutions paths in a standard resolution tree

In the above graph, the bold lines represent the paths that resulted in a success. A continuous path is an ensemble of paths, starting from the root node until a leaf is reached which represents one solution path of the goal. We count 6 leaf nodes, indicating 6 distinct solutions to the given query.

I.6 Failures in Prolog

Prolog responds to the query asked by the programmer by trying to satisfy the conjunction of goals, whether they appear in a rule body or in the question itself, using the given set of rules in the program. It attempts to satisfy these goals from left to right, i.e. no trials to satify a certain goal will take place unless its neighbour to the left has been satisfied.

When a failure occurs, the flow of control in Prolog returns back along the way until the last choice point. At this point, it attempts to resatisfy the goal by finding another alternative clause. First it undoes (uninstantiate) all the instantiations of the variables that took place during the previous goal satisfaction. Then, it searches in the database where the last placemarker was put. If it finds another matching possibility, it marks the place and the resolution continues normally. Now all the goals to the right of this choice point will be tried from scratch, i.e. attempting all possible alternatives for each, beginning from the very first. It is worth noting that Prolog will satisfy them and *not* resatisfy them. If no other matching possibility is possible, then the goal fails and Prolog returns further backwards to the previous choice point and the same operation is repeated.

When a goal fails, it reports its failure to its neighbour to the left. If it has no neighbours, then it reports its failure to the goal that caused it to be invoked.

I.7 Determinism of a Goal

It is clear that Prolog attempts to solve the goal under investigation by computing one solution at a time. Due to the simple declaration of rules, Prolog has been popular since its existence in implementing artificial intelligence techniques. These, by default, include various solutions to a problem. Hence, we are faced by a class of problems, or goals, with more than one possible solution. Such problems are nondeterministic. We could distinguish a deterministic goal from a non deterministic goal as follows:

I.7.1 Deterministic goal

A deterministic goal is a goal which has exactly one output solution for each distinction combination of its inputs (only one solution exists).

I.7.2. Nondeterministic goal

If there is more than one solution, then the goal is nondeterministic. For nondeterministic goals, we have two types, *don't know* nondeterminism interpretation of the goal , and *don't care* nondeterminism interpretation of the goal.

I.7.2.1 Don't know nondeterminism

The don't know nondeterminism interpretation implies that the programmer need not know which of the choices specified in the program is the correct one. It is the responsibility of the execution of the program to choose correctly when several transitions are enabled.

Don't know nondeterminism simply could be sensed in a manner where failing computations 'don't count' and only successful computations may produce observable results.

I.7.2.2 Don't care nondeterminism

On the other hand, the don't care interpretation of nondeterminism requires that results of failing computations be observable. Hence, a don't care nondeterminism may produce partial output even if it is not known whether the computation will eventually succeed or fail.

Although nondeterminism of abstract computational models is commonly interpreted as don't know nondeterminism, such models are also open to the don't care interpretation. In the logic programming domain, Prolog takes the don't know interpretation whereas concurrent logic languages often take the don't care interpretations.

Formally, the two interpretations of nondeterminism induce different notions of equivalence on the set of programs. For example, in logic programs, two computations on the same initial goal are equivalent if they have the same answer substitution and the same mode of termination. Under two don't know interpretations, two programs are equivalent if they have successful computations. Under don't care interpretation, two programs are equivalent if they have equivalent computations, whether successful or not.

However, it is not possible in general, to know in advance whether a computation will succeed or fail.

I.8 Backtracking: Definition

As previously mentioned, Prolog starts searching the database of rules from the very top. A matching fact may exist and the goal is satisfied immediately. A marker is placed in the base pointing to the current matched goal. If there are variables then they will be instantiated.

The matching fact may be a rule, thus reducing the task to a conjunction of subgoals. Each of these subgoals should be satisfied to satisfy the original goal.

If a goal cannot be satisfied due to nonexisting clause heads or due to a unification error, we say that the goal *failed*. At this point, Prolog initiates *backtracking*. Backtracking consists of *undoing* what has actually been done, and attempting to resatisfy the goals by finding an alternative way to satisfy them. Accordingly, all previously instantiated variables will be uninstantiated, then the search is resumed in the database beginning from where the goal's place marker was put. This new 'backtracked' goal might succeed or fail in the same manner.

I.8.1 Types of backtracking

According to the execution model of standard Prolog, there are two types of backtracking; *shallow* backtracking and *deep* backtracking. To clearly introduce a distinction of the difference between the two types, we will always refer to the following example,

:- m, p.

with the following base,

m1. *m2.* *m3.*

p.

where *m* has three alternatives and *p* has only one alternative.

I.8.1.1 Shallow backtracking

As previously mentioned, in nondeterministic problems, several clause heads might appear for the same problem, i.e. more than one solution might exist. When a goal is attempted, the Prolog program starts to search the database from the very top starting with the leftmost goal. Since *m* has three different alternatives; m_1,m_2 and m_3, resolution attempts to satisfy the objective goal *m* by matching it first with *m1*. If unification succeeds, then the marker is put at this alternative and resolution continues classically to satify the next goal to the right, that is *p*.

Alternatively, a unification with *m1* could also result in a failure during unification. At this point, Prolog backtracks to the goal, the *last* choice point, to attempt another alternative. This

backtracking is *shallow* as it takes place between different alternatives of a clause. No variables have been instantiated to undo them. No previous matching has occurred and accordingly, Prolog continues to satisfy the goal under investigation by continuing the search starting from the next rule after the place marker has been put.

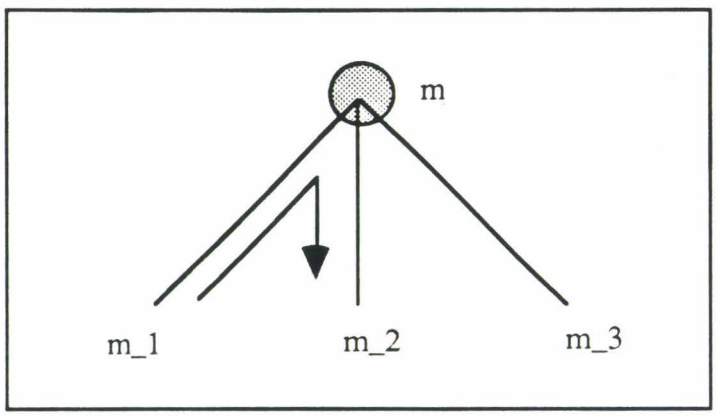


Figure (I.3) : Shallow backtracking

If m_2 satisfies the goal, we say that the variables are bound to the instantiations due to the success of this unification process. If no more goals exist, then we say we have a solution and success is reported, otherwise, resolution then continues to attempt the satisfaction of the next goal to the right of m, p.

I.8.1.2 Deep Backtracking

What if another solution was demanded after the success of p, where p, as previously mentioned, is a unique alternative clause. In this case, Prolog returns back to the previous goal, m. This is *deep* backtracking. The resolution backtracks to the previous goal to the left, in this case m, undoing all the instantiations that resulted from the last unification and starts to attempt to resatisfy m by trying the next alternative after the marker, m_3. According to the result of this attempt, resolution continues.

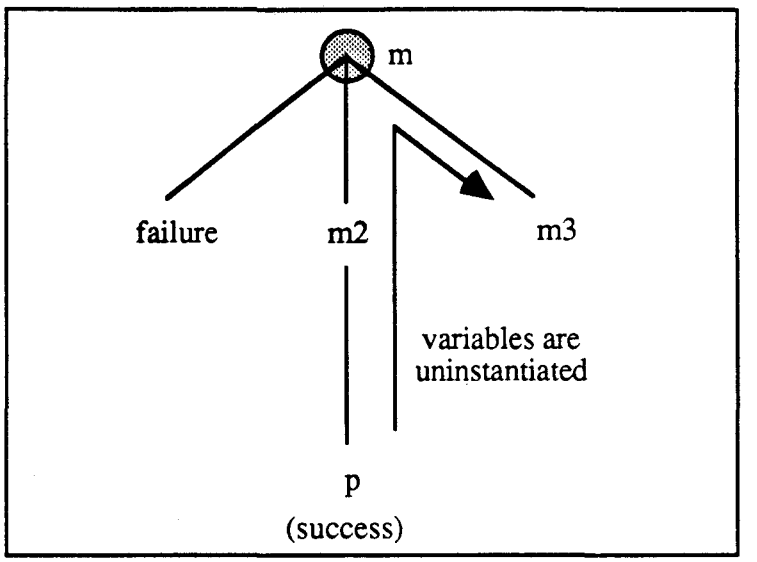


Figure (I.4) : Deep backtracking

If *m3* failed, we have no more alternatives for a shallow backtracking of *m*. Hence, Prolog reports a failure to find more solutions.

I.8.2 Overhead of backtracking

In the above example, if the attempt to satisfy *m3* succeeded, then in this case Prolog resumes resolution to instantiate the variables, attempting the body clauses if *m3* was a rule. Normally after satisfying the body clauses, Prolog attempts *p* again. This is the interesting part; *p* has a unique alternative only, i.e. the same rule has been reexecuted but with different instantiations of the variables. Considering a general program and all the possible backtracks made during the resolution, we can see that *p* will be solved numerous times.

The number of times a subgoal is reattempted is what we consider as the overhead of backtracking. It is the work done until all solutions are produced.

For a complex program, it is obvious that the overhead of backtracking over certain subgoals is costy due to a tedious repetition of the same operations. To understand better the backtracking phenomenon, we wrote a meta-interpreter of an AND/OR process model of Prolog.

I.8.3 Standard AND/OR

In this section, we present the meta-interpreter by which we tried to observe the behaviour of the backtracking phenomenon during the execution of a standard Prolog program. We first present the AND/OR model followed by the architecture of the simulator, the benchmarks tested, the results obtained and how we interpreted them. From this discussion, we focus on the different aspects that motivated us to proceed with the work presented hereafter.

I.8.3.1 AND/OR process model

In the AND/OR process model, there are 2 types of processes; the AND processes and the OR processes. An AND process is created to solve a goal statement, a conjunction of one or more subgoals. An OR process is created by an AND process to solve exactly one of these subgoals. If there is a nonunit clause in a procedure of a subgoal, the OR process will start an AND process for the body of the clause.

A computation in a Prolog program can be described by an AND/OR tree of processes, with the initial goal statement defining an AND process at the root of the tree. Messages are used to start and cancel descendants (in case of failures) and return results to higher levels of the tree.

I.8.3.2 Principle

The simulator is written in LPA Prolog. It is a meta-interpreter that observes the real execution of Prolog programs following the AND/OR model. This observation is accomplished by the aid of a trace of the execution of the programs. This trace is obtained by inserting the *spy* predicate between the different literals of the clause to be executed.

This simulator is in fact built to achieve two objectives; the first is to construct the complete AND/OR tree of the goal, and the second is to store all information related to each node (subgoal) for further observation of different parameters, including the number of unification operations that took place in each node together with the corresponding instantiations of the variables, during the computation of the different solutions for a given query.

The output of this simulation is the number of times each node in the AND/OR tree was invoked, together with the different bindings of the variables during each invokation. We

analysed this information (the number of invokations of each node) in terms of shallow backtracks and deep backtracks.

Moreover, a graph of the AND/OR tree that corresponds to th execution of a given query is produced by this simulation. In this graphical output, AND nodes are represented by squares and OR nodes by circles. The size of each node is proportional to the number of unifications attempted and the number of solutions that ascended. This output gives a quick estimtation of the exhausted nodes when attempting to find the different solutions of the given query.

I.8.3.3 Running several benchmarks

In this section, we will discuss several tested benchmarks.

1- The *permute* problem:

The program listing is given in appendix A.

The queries are:

:- permute([a,b,c],R).
:- permute(a,b,c,d],R).

We actually chose this problem because it is an example of recursive programs where no failures take place.

When running the corresponding program with only **3** elements in its list as an input parameter, the graphic output of the program is as shown in fig.(I.5).

In this case, there are **6** solutions. Observing the maximum number of unification operations occuring, we find that the *insert* subgoal was unified **6** times though there were no failures that occured during the unification operations. Every time a new solution is demanded, the same subgoals are reattempted. In other words, **6** deep backtrackings took place to produce all the solutions.

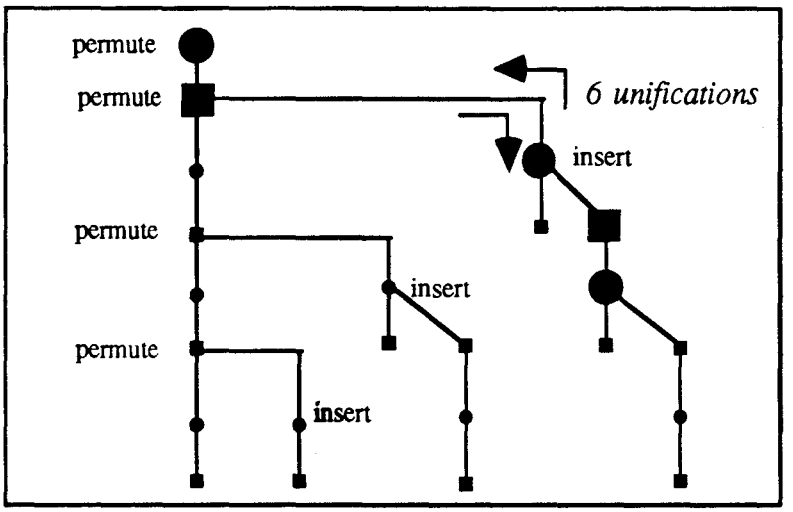


Figure (I.5) : The AND/OR tree for permute([a,b,c],R)

Increasing the complexity of the problem to **4** elements, the number of output solutions reaches **24** different alternatives. Though no failures existed, the *insert* subgoal was unified **24** different times for **24** instantiations of the variables in the clause header. Again, we understand that **24** deep backtracking to the *insert* subgoal took place.

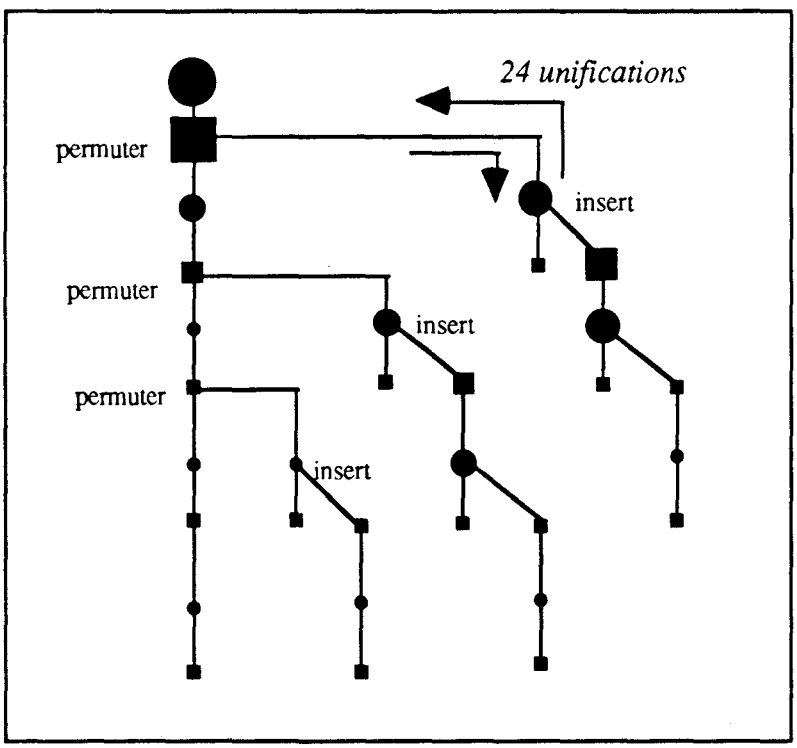


Figure (I.6) : The AND/OR tree for permute([a,b,c,d],R)

The same thing for a *permute* with **5** elements, **120** solutions are produced. Again, the *insert* subgoal was unified **120** times to produce these solutions.

Our first comment is that even in the absence of failures, since the backtracking feature is the only control in the execution model of Prolog there is an overhead due to the repetitive times a subgoal is reattempted in order to find all the solutions.

2- The *bits-palondromic* problem:

A more motivating example is the *bits-palondromic* problem that includes different types of failures. It is inspired from [28]. It is a program that generates 6 elements, each having one of two values, then reversed to produce a symmetric list. The program listing is given in appendix A.

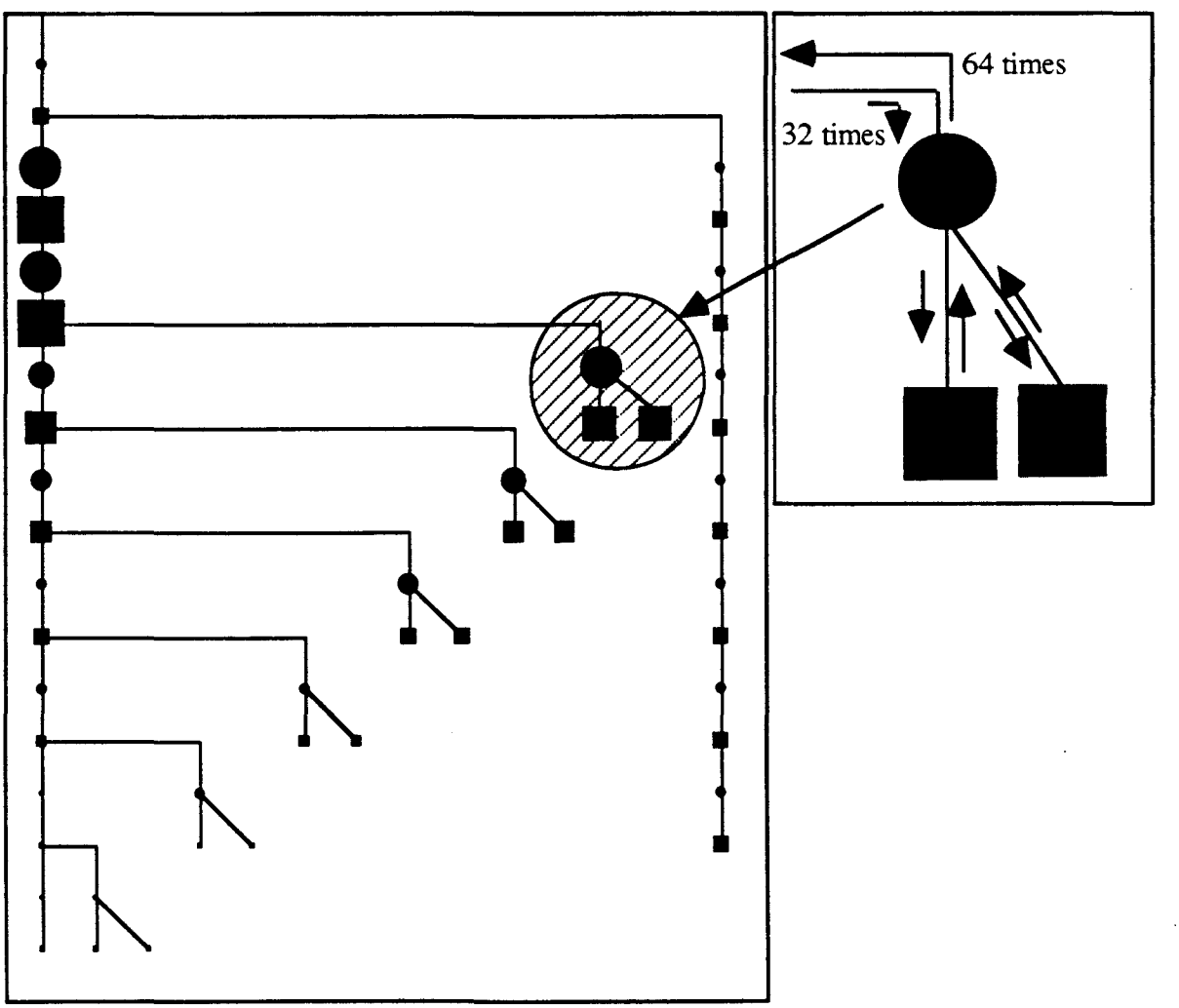


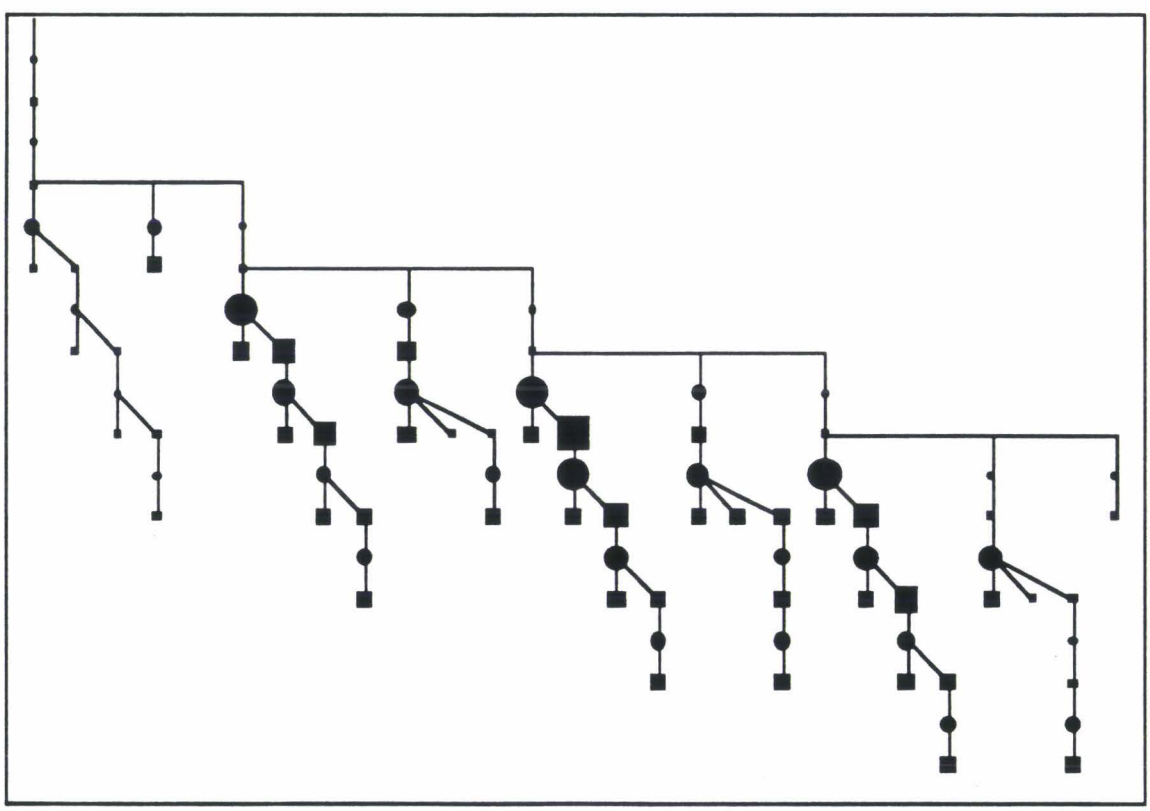
Figure (I.7) : The AND/OR tree for bits-palondromic of order 6

In the above figure, if we zoom on the shaded choice point, we find that **32** descents took place. This means that the system deeply backtracked **32** times. Since this choice point has two alternatives, then we understand why **64** solutions ascended from this node. In other words, to produce all the solutions (**16**), **32** deep backtracks and **2** shallow backtracks took place to this subgoal. We cannot avoid the shallow backtracking if we are interested in all the possible solutions. On the other hand, it is a waste of time to reattempt the same subgoal **32** different times until all the solutions are produced.

We will return to this example once more in chapter 6.

3- The *n-queens* problem:

The last benchmark executed was the *n-queen* model with different indices. For $n=4$, we have **2** distinct solutions, with a subgoal like *index* was unified **24** times. For $n=6$, the program suggests only **4** possible moves, and at the same time, a subgoal like *save* is executed **216** times. For $n=8$, we arrive to a figure of **4544** unification operations with the subgoal *index*, while the suggested solutions are **120**!



Figure(1.8): The AND/OR tree for the 4 queens problem

I.8.3.4 Recapitulation

Given all the above figures, the complexity of the standard computational model of Prolog is quite remarkable. For different instantiations to different variables, the resulting backtracking feature, needed to explore all the different possibilities leads to a high traffic over the same subgoals again and again. This, obviously, results in enormous execution times due to redundant attempts to repeat the same operations.

I.8.4 Optimization of Backtracking

Several efforts have tried to enhance the resolution of Prolog programs by trying to optimise the number of backtracks that take place. We discuss the most popular approaches.

I.8.4.1 More Intelligent backtracking Systems

In such systems, a control mechanism is applied that optimises the performance of the naive backtracking while executing a program. In other words, more 'intelligent' backtracking is defined. Such systems are related to the occurence of failures. When a failure occurs, the system analyses the cause of this failure, and memorises the instantiations that led to this failure. This is to serve two objectives: the first is to backtrack to the effective choice point, i.e. the choice point that will change the instantiation that led to the previous failure, and secondly, to assure that this failure will not be repeated in the future. Such systems that include such a mechanism are called intelligent backtracking systems.

Such systems are based on the variable dependence between the different subgoals. Two wide classes are defined: the intelligent backtracking and the semi-intelligent backtracking. We detail each of these classes.

I.8.4.1.1 Intelligent Backtracking

Intelligent backtracking was introduced independently by Cox [19] and Perira et Porto [35,36] to reduce the number of backtracks. This approach is actually related to the occurence of failures in the standard execution model. When a failure takes place, the system analyses the cause of that failure.Consider the following set of unit clauses,

p(a). *p(b).*
q(1). *q(2).*
r(b,1). *r(b,2).*

with the goal statement,

:- p(X), q(Y), r(X,Y).

A depth first interpreter first solves *p(X)*, binding *X* to a, then solves *q(Y)*, binding *Y* to *1*, and then tries to solve *r(a,1)*. When the latter fails, the interpreter backtracks. The most recent choice point is in the selection of *q(Y)*; when this is redone, uninstantiating *Y*, another solution is found binding *Y* to *2*, and the next goal will be *r(a,2)*, which will also fail.

Both of these calls to r fail because the solution of *p(X)* binds *X* to a value that cannot be used to sove *r(X,Y)*. When the ineterpreter backs up only as far as *q(Y)*, it cannot fix this erroneous choice, and by re-solving *q(Y)* and binding *Y* to a different value, it is wasting time.

An interpreter designed and implemented by Pereira et al. performs this type of analysis[ ]. In the example given above, it finds that any goal of the form *r(a,_)* fails because of the presence of the term a in the first argument position. Since *X* was bound to a in the call to *p(X)*, the interpreter backs up past the call to *q(Y)*, all the way to a choice point in the solution of *p(X)*. When *p(X)* is solved again, binding *X* to b this time, the entire goal list could be solved, without the wasteful attempt to resolve *r(a,2)*.

A first attempt to introduce intelligent backtracking in a Prolog complier was proposed by Lin et al. [30,31]. Codognet et al. extended the WAM architecture to include the intelligent backtracking feature. An extended unification-related instructions was introduced mainly to rememorise the source of the bindings. They proposed the DIB machine [12] then the WAMIB [13] that resulted in a more efficient implementation where speedups upto a factor of 10 for nondeterministic programs are achieved.

I.8.4.1.2 Semi Intelligent backtracking

This is the case when there is no variable dependence between two subgoals. Here, the analysis of a failure and the decision making to decide the backtracked choice point are simpler. An example is

:- p(A), q(B), r(A).

When *r(A)* fails, *q(B)* could be skipped on backtracking since it does not produce any values that affect the solution of *r(A)*. This is a case where it is not necessary to analyse the exact cause of the failure; it is only necessary to notice that a new solution of *q(B)* cannot help solve *r(A)* as *q(B)* and *r(A)* have no variables in common. This has been called semi intelligent backtracking[34] since it is not quite as effective as the intelligent backtracking. For example, a semi intelligent backtracking could not make the correct backtracking choice for the first example in the section.

The above discussed types of backtracking will be re-discussed when coming to the presentation of related work in chapter 7.

I.8.4.3 Partial elimination of backtracking

The backtracking feature, though a powerful control in the computational model of Prolog, yet it results in redundant computing while solving a goal, hence consuming more execution time as well as memory space to store all required information on already attempted choice points in order to be able to backtrack in case of failures or further trials.

The above presented optimisations represent partial enhancements to reduce the number of times that the system backtracks. But alternatively, regarding previous research, we observe several efforts to present execution models for extended Prolog that partially eliminate backtracking. All these efforts were oriented to parallelism in their execution. We cite the most close ideas here.

An example is DAP Prolog [26] which is an extension of Prolog. It focuses on large relational database systems and tends to eliminate backtracking. It is actually a set-oriented view of Prolog, where two new data structures, together with their relevant support code are added to the system. It is described by: DAP Prolog = Prolog + Sets + Arrays. The DAP Prolog programmer thinks in terms of sets rather than individual binding values. He is responsible to define the code segments that will be treated in a set mode. These segments are large database predicates. For rules, normal backtracking takes place and is not eliminated. Hence list-oriented progams could not benefit from this approach.

MultiLog [38,39,40] is another approach where a prefix unary operator **disj** is added to the syntax of Prolog. The programmer may annotate any subgoal by this operator, whose role is

to attempt to gather some of the solutions that solve the given subgoal and produce a set of environments. Such environments partially replace backtracking as the operational embodiment of disjunction.

Firebird [44], which is a commited choice logic programming language bears also some similarity to the same idea. In a non deterministic derivation step, if there is any unbound domain variable X in the system, with the domain $\{a1,...,an\}$, Firebird will execute each branch with a constraint: $X = ai,\ 1 \leq i \leq n$. . Here, all constraints are tried in each branch, but in parallel.

I.9 Motivations and objectives

The above study motivated us to think on a more global platform. To understand the behaviour of the standard execution model of Prolog program, we wrote the meta-interpreter of the AND/OR execution model that proved that to produce all the solutions of a given problem, redundant work took place. We believe that this redundant work resulted in longer execution times.

Our main objective is actually to reduce the execution time of Prolog programs. This problem has been tackled by different research domains. Several approaches exist that may result in a faster execution: the first is to optimise the current implementation of Prolog (the WAM architecture [45,46,47]), the second is to introduce parallelism [5,6,718,43], and thirdly is to define other execution models for Prolog sources. We chose the third approach to achieve our objective.

From here, we started to think of a way to optimise backtracking. We are not interested in adding any modifications to the existing standard Prolog syntax, which confirms to the Edinburgh syntax. Modifying a language means simply rewriting already existing programs, and this, we believe, is not an optimum way to tackle the problem.

Also, our interest is not to present a model that treats a certain programming style as that proposed by DAP Prolog, but rather a very general approach that may be applied to any style of Prolog programs (databases, recursive clauses, sequential list searching,etc.).We simply state that our target program is any finite nondeterministic standard Prolog program.

The idea that we had in mind is not to repeat any previous attempt in solving subgoals. This means that we want to construct the resolvant of the given query by passing *once and only once* over each subgoal. We thought of eliminating the classical deep backtracking feature and preserving the shallow backtracking. The resulting solutions from each choice point are assembled. We do not want to reattempt the choice point that solves this subgoal again. All the possible solutions propagate in the forward sense of the resolution (from left to right) and the resolution never returns backwards.

This model is transparent to the programmer, in other words, no added load lies on the user, where no modifications in the syntax is required, nor a certain programming style is essential. We called this model, the multi-resolution model for Prolog programs.

The main feature that characterises the proposed multi-resolution model is the presence of multiple bindings of variables. What we present in this thesis is a detailed discussion on the treatment of such multi-instantiated variables that replace the deep backtracking feature. These variables are involved in the different phases of the multi-resolution. In the work presented in this thesis we study:

- the multi-resolution model, concerning the different phases of execution,
- the methodology of creation of multi-instantiated variables (when and how),
- the impact of the presence of such variables on the unification operations for numeric and non numeric operations, on the occuring failures, and on the display of solutions, and finally
- the result of employing multi-instantiated objects on the performance of the execution (time-wise, memory-cost-wise) with respect to standard Prolog.

By proposing this model, we aim to enhance the performance compared to previous related work. We are interested in the amelioration of the execution in the *sequential* mode as well as in the *parallel* mode.

All details concerning each phase in this multi-resolution will be clearly discussed in the remaining chapters of this thesis.

Chapter Two

Basic Idea: A Look Through

Abstract

We discuss the basic idea of the multi-resolution execution model. The structure of the model is presented, demonstrating the role of each phase. Different underlying problems of replacing deep backtracking by multi-instantiated variables are discussed.

II.1 Introduction

The multi-resolution execution model of Prolog programs is a new execution model with which we aim to enhance the execution of Prolog programs in both modes; sequential and parallel. It is a powerful model for nondeterministic finite problems as well as clause problems related to large databases. The main underlying theme is to eliminate completely the deep backtracking feature defined in the standard Prolog execution model. This means that for a query such as:

?- subgoal$_1$,subgoal$_2$,...,subgoal$_i$,...

each *subgoal$_i$* is traversed only once. *subgoal$_i$* is attempted with each of its clause heads sequentially, i.e. preserving shallow backtracking. After exploring all its alternatives, i.e. no more clause heads exist, the different solutions are gathered. Now, when the multi-resolution proceeds with the following subgoal, *subgoal$_{i+1}$*, control never returns backwards to *subgoal$_i$*. In other words, *subgoal$_i$* will never be reattempted.

After the attempt of all the subgoals, the model produces the solutions. These are the correct combinations derived from the different solutions gathered throughout the multi-resolution of the query.

We will demonstrate clearly the basic idea of the multi-resolution model with the following example.

Example:

Given the following program,

number(1). number(2). number(3). number(4). number(5).

odd(X):- X mod 2 is 1.

prime(X):- ....

consider the following query,

?- number(X), odd(X), prime(X).

We will explain the difference between the standard resolution and our proposed multi-resolution. In the standard execution model of Prolog, the execution starts by selecting the first subgoal to be resolved, which is *number(X)* . Unifying it with the first clause head,

number(1), will result in a success and X is instantiated to *1*. The standard resolution directs control to the next adjacent right subgoal, *odd(X)*, where X is already instantiated to *1*. The unification operation with the clause head succeeds and the execution of the body will also succeed. Afterwards, the third subgoal, *prime(X)*, is attempted, which will also succeed and a first solution is produced, $X = 1$.

Now to produce further solutions, deep backtracking takes place and the control returns to the *odd* clause, which due to the fact that it has only one alternative, i.e. not a choice point, will pass the control to the *number* choice point as it is the last choice point encountered. The variable X is deinstantiated and the second clause head is considered. The standard resolution continues in the same manner as discussed in chapter one until all solutions are produced.

The interesting point is that due to the fails and deep backtracking operations that take place during the standard resolution, the *odd* goal is executed *5* times though it consists of a unique alternative only. The *prime* goal is executed *3* times resulting in *3* solutions.

The standard resolution tree could be represented as shown in fig. (II.1).

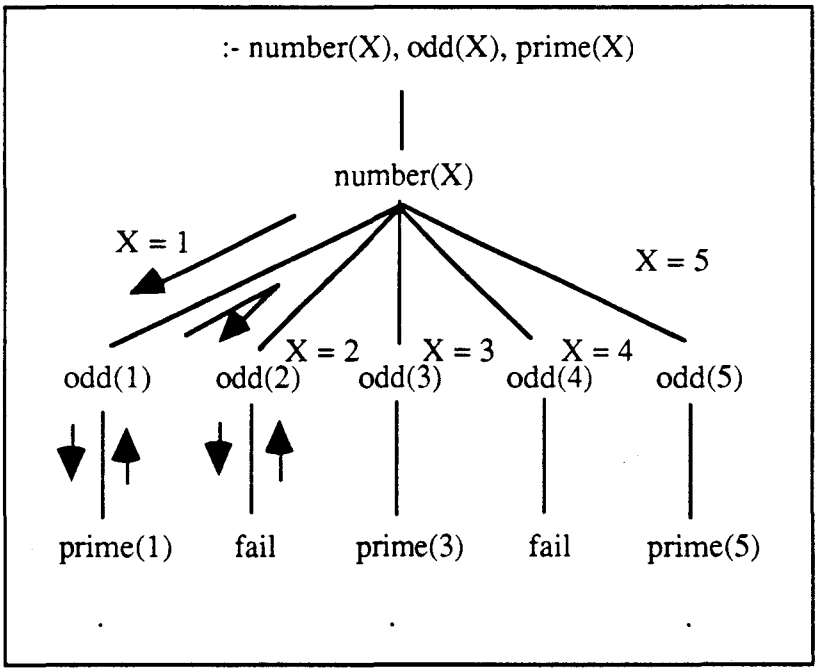


Figure (II.1): Standard resolution tree

It is worth noting that once a solution path reaches a leaf node, the solution is displayed to the user then deep backtracking takes place to compute a new solution.

What we propose is a multi-resolution approach which is more efficient in executing such a program as we aim to eliminate redundant work. Here, we select each subgoal once, attempting all its alternatives sequentially before proceeding with the next subgoal. This implies that the *number(X)* subgoal should be executed 5 times, as we preserve the shallow backtracking feature, before passing to the *odd* subgoal. Hence, the subgoals *odd(X)* and *prime (X)* are attempted once and only once. It should be noted that before attempting the *odd* subgoal, it should be memorised that X was instantiated to 5 different alternatives. We say that the variable X is *multi- instantiated.* To differentiate it from the list structure defined in Prolog, we have chosen the '{}' to represent a multi-instantiated variable. Hence, X is represented as follows:

$$X = \{ 1 , 2 , 3 , 4 , 5 \}$$

Note that now onwards, X will be bound to these five alternatives and the subgoal *number* will never be re-attempted.

We state a number of relevant definitions that we will be frequently recalling throughout our discussion.

Following, we present a naive definition of a multi-instantiation.

Definition 2.1: A Multi-instantiation

A multi-instantiation is given by:

$$\{ a_1, a_2, ..., a_n \}$$

where
- the structure {...} denotes a multi-instantiation, and
- a_i is the i^{th} instantiation which could be any *multi-term.*

Definition 2.2: A Multi-term

A multi-term is an atom, a variable, a compound term or a multi-instantiation.

Returning to our example, we then come to the *odd* subgoal. The variable X in the subgoal head is already multi-instantiated to the 5 values. A single unification operation is attempted to all the 5 values of X. The odd test will take place on each of the 5 instantiations sequentially resulting in 2 failures for the values $X = 2$ and $X = 4$, and 3 successes for $X =$

1, *X = 3* and *X = 5*. The multi-resolution continues with the *prime* subgoal in the same fashion.

Fig. (II.2) shows the resolution tree of our multi-resolution model.

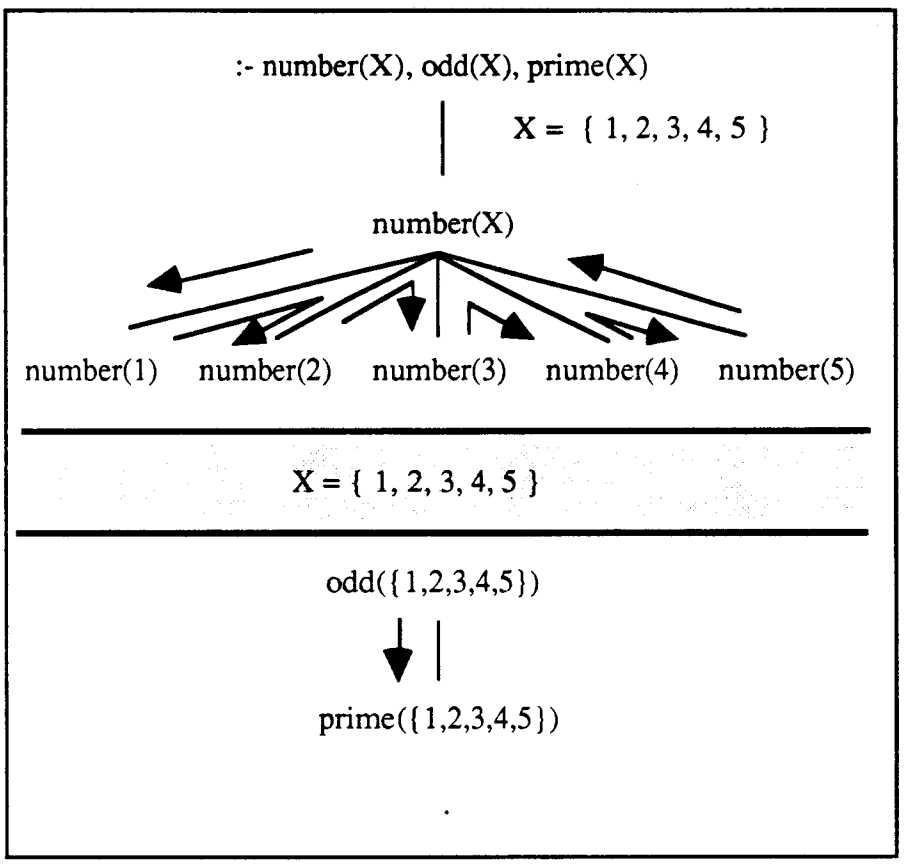


Figure (II.2): The multi-resolution tree

In the above figure, the multi-resolution tree may be viewed as if it consists of different steps. Each step corresponds to a subgoal in the given query with all its alternatives attempted and terminated by a phase where the solutions are gathered. We called this phase the *local-synchronous-OR-phase*. It is represented in the above figure by the dotted area bounded by the bold horizontal lines. It is in this phase where the variables are multi-instantiated. It has a vital role in the multi-resolution which will be discussed in the following sections of this chapter.

The steps' order corresponds to the order by which the subgoals are written. The first step corresponds to *number(X)*, the second to *odd(X)* and the third to *prime(X)*. By terminating the attempts of the last subgoal, the multi-resolution is over.

II.2 Model Structure

The multi-resolution model is represented in fig. (II.3). It takes a standard written Prolog program as its input without any special modifications in the syntax of the program (Edinburgh syntax). It produces all the solutions of the given query as its output.

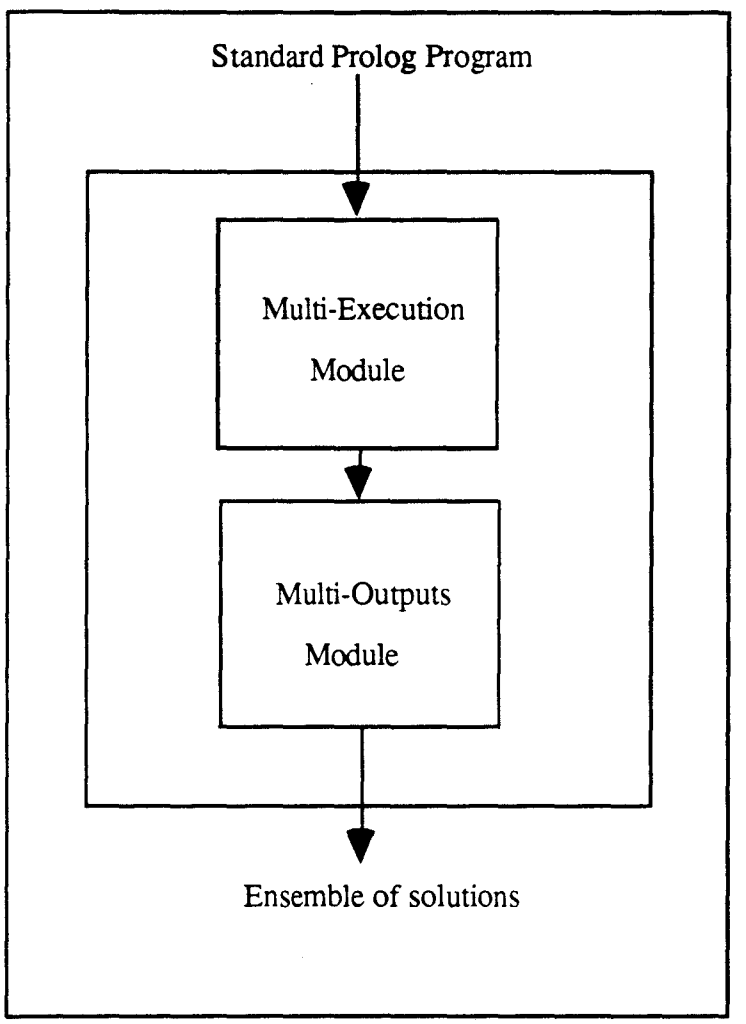


Figure (II.3): The multi-resolution model structure

Inherently, it is divided into two phases; the *multi-execution* phase and the *multi-outputs* phase. The former is responsible for the actual resolution of the goal in a multi-resolution fashion, whereas the latter is responsible for the production of the ensemble of the correct solutions. The output of the first phase is the input of the second phase. We will detail the roles of both phases in the following sections.

II.3 The Multi-Execution phase

The input to this phase is the query to be solved. It may consist of one, or more, subgoals. In either cases, this phase handles one subgoal at a time. For each subgoal, the different clause heads are examined sequentially, in the same order as they are written in the program. This is why we mentioned previously that we preserve the shallow backtracking feature. For each clause header, a special unification operation which we will call *multi-unification* takes place between the subgoal and the clause head. If the multi-unification succeeds, then the body clauses are attempted before proceeding with the next clause head.

After exploring all the alternatives, the multi-resolution proceeds to assemble all the resulting solutions. This takes place in a certain phase during the multi-resolution, that we called the *synchronous OR* phase. There is a local synchronous OR phase for each choice point. Instead of proceeding the multi-resolution with different instantiations of the variables, in this phase all solutions are assembled in a certain structure (multi-instantiation) to be assigned to the original variables in the subgoal head. We called such variables *multi-instantiated* variables.

Multi-instantiated variables could be *multi-unified* to other multi-terms. This is the role of the *multi-unification* algorithm. It manipulates all types of operations dealing with multi-instantiated variables. Besides, this algorithm processes the failures occuring during the multi-resolution. We will return to this point later in this chapter.

When all the subgoals are fully attempted, the multi-execution phase terminates its role by passing the saved data structures to the multi-outputs phase. These data structures contain all the information of the multi-resolution encountered when attempting to solve the original query to be processed by the output phase, including the instantiations of the different variables, the occured failures and the multi-resolution tree, to produce the correct ensemble of solutions.

Now, we will discuss thoroughly each phase in the multi-execution phase.

II.3.1 Search Strategy

The search strategy adopted in the multi-resolution model resembles greatly the breadth first strategy. To solve a goal, its subgoals are selected sequentially, starting from the leftmost

subgoal. All the alternatives of each subgoal are explored, in a sequential manner, before proceeding to the next subgoal.

Multi-unifying a subgoal with each clause head will result in a number of instantiations to the variables appearing in the head of the subgoal. If originally these clause heads are facts, such as *number(X)* in the example discussed in section II.1, then the different solutions are assembled to be assigned to the variable *X*, and now onwards we say that the variable *X* is multi-instantiated. Fig.(II.2), discussed in the previous section, demonstrates clearly this case.

On the other hand, if any of the clause heads was a rule, we investigate the rule to the very end. In other words, the body clauses of this rule are attempted before attempting the next clause head of the original subgoal. These body clauses are investigated in the same manner; one at a time, from left to right, searching for all possible solutions for each subgoal. When all the body clauses are attempted, control returns to the next alternative of the original subgoal and the multi-resolution continues in the same manner. A scheme representing the proposed strategy is given in fig.(II.4).

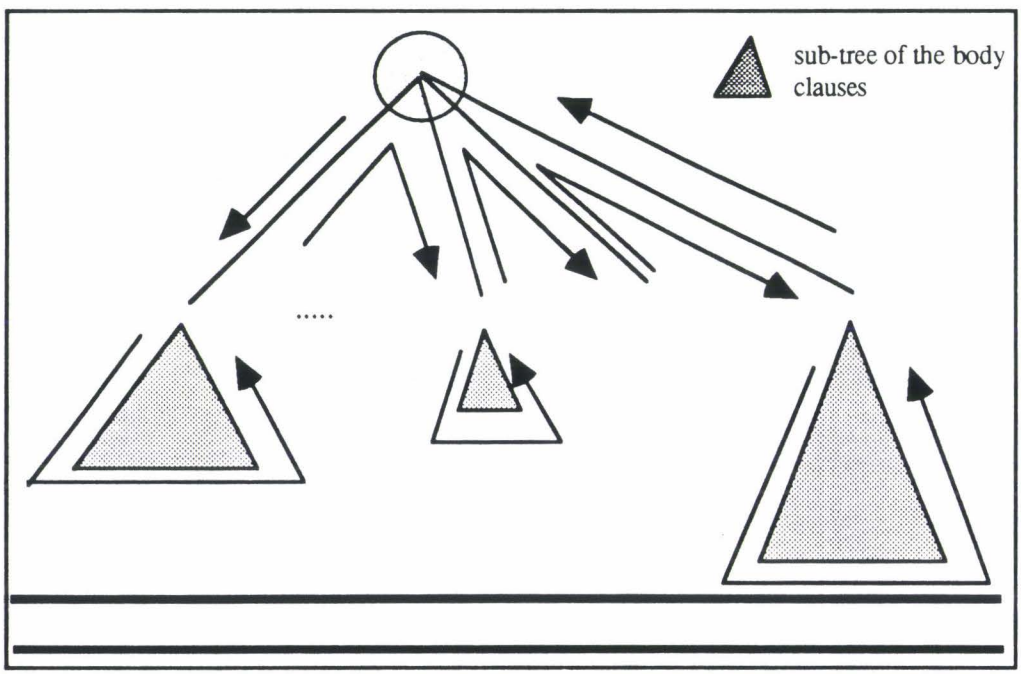


Figure (II.4): The combined breadth-first, depth-first search strategy for a choice point in the multi-resolution model

If the body clauses are not executed before the following alternatives, this will turn into a breadth-first strategy, with its disadvantageous high memory consumption.

In the following example, we illustrate the difference between the depth-first search (adopted in the standard resolution model of Prolog), the breadth-first search, and the search strategy adopted in the multi-resolution model.

Example:

Given the program,

p(a). *p(X):-s(X).* *p(z).*
s(b). *s(c).*
q(a,aa). *q(c,cc).*

consider the query,

:- p(X), q(X,Y).

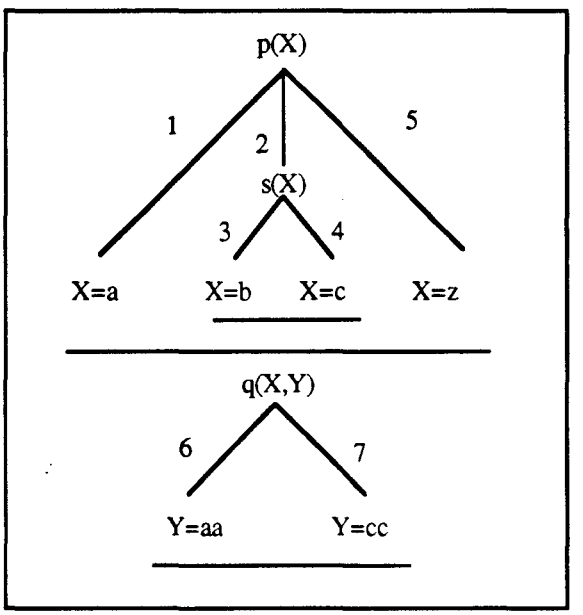


Figure (II.5a):
Multi-resolution search

The first subgoal to be attempted is *p(X)*. In the multi-resolution model, multi-unification takes place between *p(X)* and the first clause head, *p(a)*, resulting in the instantiation of *X=a* (branch number 1). Shallow backtracking takes place to attempt the second clause head, *p(X)*. This clause head contains a body clause, *s(X)*. Here, the multi-resolution proceeds to solve the body clauses before returning to the following clause head.

Multi-resolving *s(X)* will result in the instantiations *X=b* and *X=c*. The local-synchronous-OR phase of this choice point, *s*, assembles the different instantiations and multi-instantiates *X* to *{b,c}*.

Returning to the third clause head, *p(z)*, *X* will be instantiated to *z*. The local-synchronous-OR phase of the choice point *p* will multi-instantiate *X* to *{a, {b, c}, z}*.

In the above figure, (2.5a), we did not emphasize what actually happens in the local-synchronous-OR phases because this is not our objective when discussing this example. Here, we are interested in how the branches are searched until all solutions are produced. In this example, the number of traversed branches in the multi-resolution model is **7**.

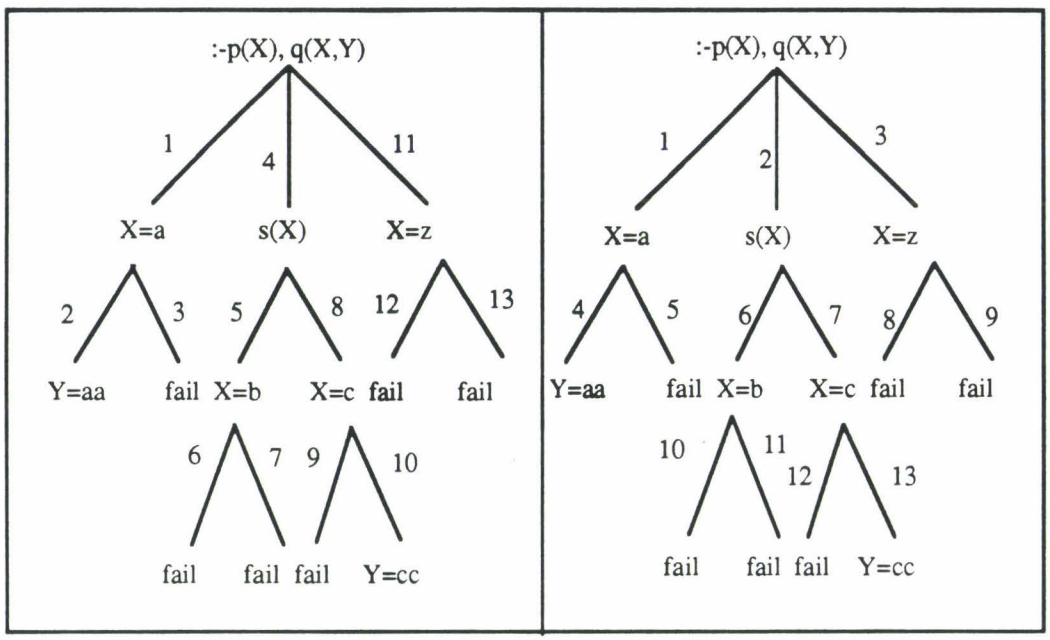


Figure (II.5b):
Depth-first search

Figure (II.5c):
Breadth-first search

In the depth-first strategy, fig. (II.5b), when the unification operation between *p(X)* and *p(a)* succeeds, the resolution ignores (temporarily) the rest of the clause heads, and proceeds to satisfy the following subgoal, *q(X,Y)*. The first alternative, *q(a,aa)*, succeeds and a first solution is obtained *X=a, Y=aa*.

To find the rest of the solutions, the system backtracks (shallow) to the last choice point, *q(X,Y)*, and attempts the following clause head, *q(c,cc)*, that will result in a failure. At this moment, deep backtracking takes place to the predecessor choice point, *p*, and tends to attempt the second clause head, *p(X)*, which contains a body clause, *s(X)*.

The resolution attempts *s(X)*, with its first clause head, *s(b)*, resulting in the instantiation of *X* to *b*, then proceeds to the following subgoal, to reattempt again *q(X,Y)* with its two alternatives. Both will result in a failure, resulting in a deep backtracking to the second alternative of *s*, *s(c)*. Repeating the same operation, the first alternative of *q* will fail, while the second succeeds, resulting in a second solution *X=c, Y=cc*.

Since no more alternatives exist for *s*, deep backtracking takes place and the third clause head of *p* is attempted, instantiating *X* to *z*. Satifying *q(z,Y)*, will result in a failure when attempting both alternatives.

In the depth-first search strategy, the number of traversed branches until all solutions are produced is **13**. Comparing this figure to that in the multi-resolution model, a reduction of almost **47%** is achieved when adopting the multi-resolution model. This is an interesting figure that we will discuss in detail when we present our results (chapter 6).

In the breadth-first search strategy, fig. (II.5c), we have the same number of traversed branches as in the depth-first strategy, **13**, but the search is done in another manner. Here, the search takes place on a breadth level. The different branches of the same level are attempted sequentially, from left to right. In this case, *p(X)* will be unified to the first clause head, *p(a)*, resulting in the instantiation of *X* to *a*, then to the second, resulting in a new subgoal *s(X)*, then the third, instantiating *X* to *z*.

After the complete investigation of the first level, the search starts to resolve the branches of the second level. *q(a,Y)* is attempted, so as *s(X)* and *q(z,Y)*. The search continues in the same manner on all levels until the same solutions are produced.

II.3.2 Memory representation of a multi-instantiation

It is worth noting that comparing the memory cost between the multi-resolution search and the breadth-first search, we find that the memory consumption of the latter is far higher than that of the former. This is because in the multi-resolution strategy all instantiations concerning each variable are assembled before proceeding with the following subgoal, whereas in the breadth-first strategy, the different instantiations of the variable are stored in the memory. At the same time, data sharing is adopted in the multi-resolution model, i.e. we do not create a copy of the variable each time a new alternative is invoked, but rather the same variable is shared among all the alternatives. This is also responsible for the reduction of the memory consumption in the multi-resolution search. In other words, the environment in the multi-resolution model is a single environment that includes multi-instantiations.

II.3.3 Coherency of an instantiation

In the multi-resolution model, single assignment is employed; i.e. once a variable is multi-instantiated, its value cannot be modified.

We return to the definition of a multi-instantiation (definition 2.1). Logically, it is a structure that includes different instantiations assigned to the same variable resulting from different alternatives of a choice point. Each of these instantiations may be any multi-term defined in the multi-resolution model (definition 2.2).

But what does a multi-instantiation actually represent? It represents an ensemble of instantiations that Prolog produces in a sequential manner (one instantiation at a time). A multi-instantiation *{ a, b, c}* represents actually 3 different terms represented by Prolog; *a*, *b*, and *c*.

An example of a compound multi-term is *p(f({q,r}))*, where the argument of the functor *f* is multi-instantiated to 2 values. It represents *p(f(q))* and *p(f(r))*.

Another compound multi-term is *[{a,b}/{c,d}]*. This is a list whose elements are multi-instantiations. Actually, this multi-term represents 4 different lists represented in Prolog, which are the result of the different combinations between both multi-instantiations. The four lists are: *[a/c]*, *[a/d]*, *[b/c]* and *[b/d]*.

Though simple, the representation of a multi-instantiation, as given in definition 2.1, may lead to ambiguous interpretations. Consider the following example:

Example:

Consider the query,

:- (X=a, Y = c ; X = b, Y = d), A=[X/Y].

Multi-resolving the first subgoal will result in *X = {a,b}* and *Y = {c,d}*. The second subgoal results in the instantiation of *A = [{a,b} / {c,d}]*. In multi-resolution, this would represent 4 lists ;*[a/c]*, *[a/d]*, *[b/c]*, and *[b/d]*. Double-checking with the standard resolution of Prolog, we find that only 2 lists exist; *[a/c]* and *[b/d]*.

This illustrates that the multi-resolution model may represent incoherent multi-terms with respect to those produced by the standard execution of the program.

Problem 2.1

A multi-instantiation, as defined in definition 2.1, may create incoherent multi-terms. The given representation does not include all the necessary information to avoid the creation of

incoherent multi-terms. The following chapter discusses thoroughly the representation that we propose and how the coherency of multi-terms is ensured in the multi-resolution model.

II.3.4 Coherency of an operation including 2 multi-instantiations

The same concept of coherency of a multi-term applies to the coherency of any performed operation that involves two multi-instantiations. It is imperative that a verification of the coherency of the first multi-term with respect to the second should take place. This is to ensure that the performed operations in the multi-resolution model are the same as those that take place in the standard resolution model.

Two multi-instantiations may result from the same subgoal, or from different subgoals. To make clear this concept, we present the following examples, that point out certain ambiguities with respect to standard Prolog.

Example 1:

Given the program,

p(a). *p(b).*

q(c). *q(d).*

Solve the query,

:- p(X), q(Y) .

After the multi-resolution of the above query,

$$X = \{a,b\} \quad \text{and} \quad Y = \{c,d\}.$$

When we want to know how many solutions are actually reached, we try all the different combinations between the instantiations of X to those of Y. We find that we have 4 solutions:

$$X = a\ ,\ Y = c$$
$$X = a\ ,\ Y = d$$
$$X = b\ ,\ Y = c$$
$$X = b\ ,\ Y = d$$

Comparing the results to that produced by the standard resolution of the program, we obtain the same results. This example is the case of creating multi-instantiations from different subgoals (different multi-unification operations).

Example 2:

For the program,

p(a,c). *p(b,d).*

q(X,Y,Z):-...

solve the query,

:- p(X,Y), q(X,Y,Z)

then the different instantiations of the variables are $X = \{a,b\}$ and $Y = \{c,d\}$. To attempt q(X,Y,Z), the variables X and Y are multi-instantiated. The different combinations of X and Y are:

$$X = a\ ,\ Y = c$$
$$X = a\ ,\ Y = d$$
$$X = b\ ,\ Y = c$$
$$X = b\ ,\ Y = d$$

whereas standard Prolog considers only:

$$X = a\ ,\ Y = c$$
$$X = b\ ,\ Y = d$$

That is, not all combinations between the multi instantiations are permitted systematically. This is the case of two multi-instantiations resulting from the same subgoal (same multi-unification operation).

Problem 2.2

An important aspect in the multi-resolution model is to check that the multi-resolution model performs the same operations, as in the standard resolution. The given representation of the

multi-instantiations (definition 2.1) is not sufficient to detect the order by which the variables were multi-instantiated to ensure the coherency of the performed operations.

II.3.5 Multi-Unification Operations

It is clear that the standard unification algorithm does not support all multi-unification operations dealing with multi-instantiated variables. Here, we define an appropriate algorithm for multi-unifying two multi-terms in the multi-resolution. This algorithm is called the multi-unification algorithm. The multi-unification algorithm includes the standard unification operations together with all operations of multi-instantiated variables. It is worth noting that basically it respects all rules of the standard unification algorithm defined by Robinson [37].

The different added cases are summarised in the table shown in table (II.1). *X* and*Y* are variables, *x* and *y* are atoms, x_i and y_i are variables or atoms, *{...}* represents a multi-instantiation. ':=' is an assignment operation, and '=' represents a multi-unification operation. The first two columns are the two multi-terms to be multi-unified, the third indicates the actions that will take place.

term X	term Y	action(s)
X	{...}	X:={...}
{...}	Y	Y:={...}
x	$\{y_1,...,y_n\}$	$x=y_i$ for i=1 to n
$\{x_1,...x_n\}$	y	$y=x_i$ for i=1to n
$\{x_1,...x_n\}$	$\{y_1,...,y_n\}$	$x=y_i$ for i=1 to n
$\{x_1,...x_n\}$	$\{y_1,...,y_m\}$	$y_j=x_i$ for i=1to n and j=1 tom

Table (II.1): Multi-Unification cases including multi-instantiatied variables

The detailed presentation of this algorithm is defined in the following chapter.

II.3.5.1 Multi unifying a multi instantiation to another multi-term

1- Total-success:

The first is the case of *total-success* where all the instantiations of a multi-instantiation result in a success when being multi-unified with the corresponding parameter in the clause head. In this case, the multi-resolution continues normally.

In standard Prolog, unifying a subgoal containing instantiated variables in its head results normally in one of two cases; either a success or a failure. In the multi-resolution model, there exist three cases when trying to unify a subgoal containing multi-instantiations with a clause head.

Example:

Given the following program,

p(a). p(b).
q(M,N).

with the query,

?- p(X), q(X,X).

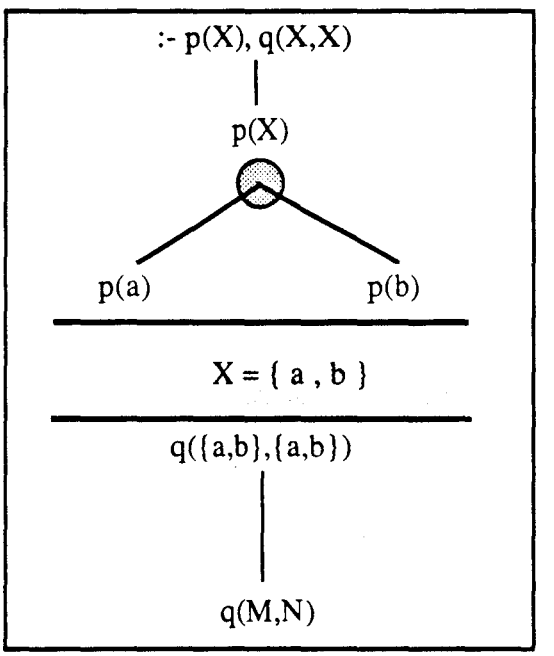


Figure (II.6): Total success

After the first subgoal, X will be multi-instantiated to $\{a , b\}$. The second subgoal has X as its arguments, which is already multi-instantiated. Multi-unifying this clause to the corresponding clause head, the result of this operation is a total success with all the different sub-terms resulting in $M = \{a,b\}$ and $N = \{a,b\}$.

2- Total-failure:

The second case is the *total-failure* and that is when each instantiation of the multi-instantiation results in a failure during the same multi-unification operation.

Example:

In the previous example, modifying the above clause head of *q* to

q(x,y).

the above query will lead to a total failure where none of the multi-instantiations of *X* resulted in a success. This is because multi-unifying either values of *X* to *x* or to *y* will result in a complete failure.

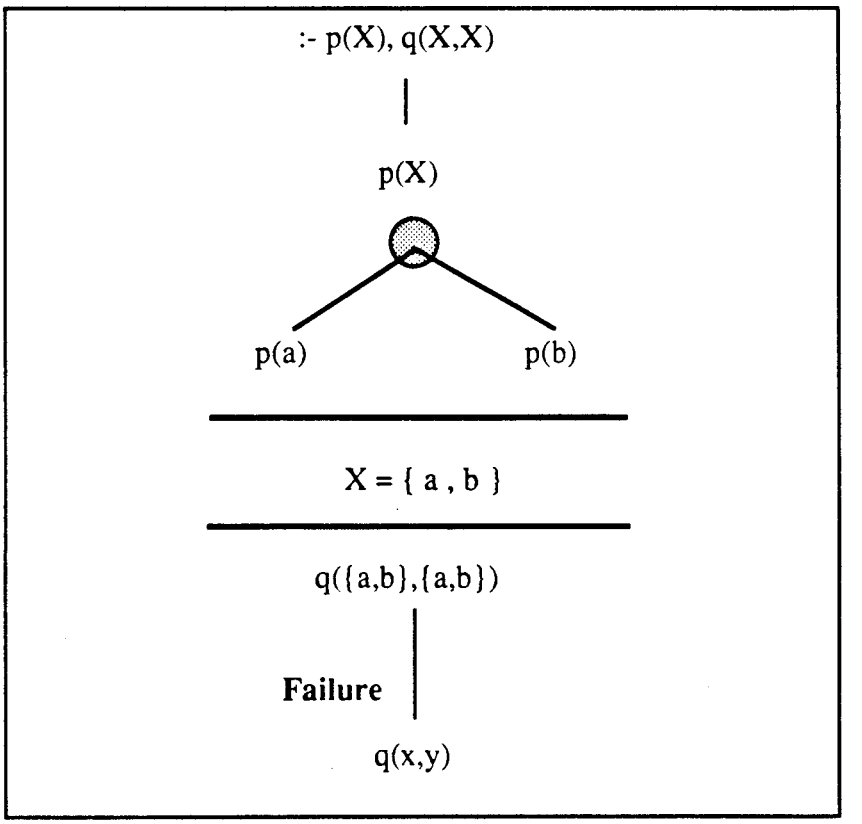


Figure (II.7): Total failure

We note that once a total failure is encountered, all the variables that were instantiated during the failing multi-unification operation will be deinstantiated as in standard Prolog.

3- Partial success/failure:

The interesting case is the third case when a *partial success/failure* is achieved, where some instantiations of a multi-instantiation result in a success of the subgoal while others, in the same multi-unification operation, result in a failure. To illustrate clearly this case, consider the following example.

Example:

Consider the following program,

p(a). *p(b).*

q(I,m):- s(I). *q(b,f).*

s(a). *s(z).*

r(f).

with the query,

:- p(X), q(X,Y), r(Y).

First, *p(X)* is attempted resulting in the multi-instantiation of *X* to *{ a , b }*. The subgoal *q* has two alternatives. The first alternative will result in the multi-instantiation of *I* to *{a,b}*. The body clause, s({a,b}), is to be solved before trying the next clause head. When attempting to solve *s(X)*, the first alternative results in a partial success/failure due to the fact that multi-unifying the second instantiation of *X* to *a* will result in a failure. On the other hand, multi-unifying either values of *X* with *z* will result in a complete, or total, failure.

Fig.(II.8) illustrates the multi-resolution of the above query showing the resulting states of the multi-unification operations.

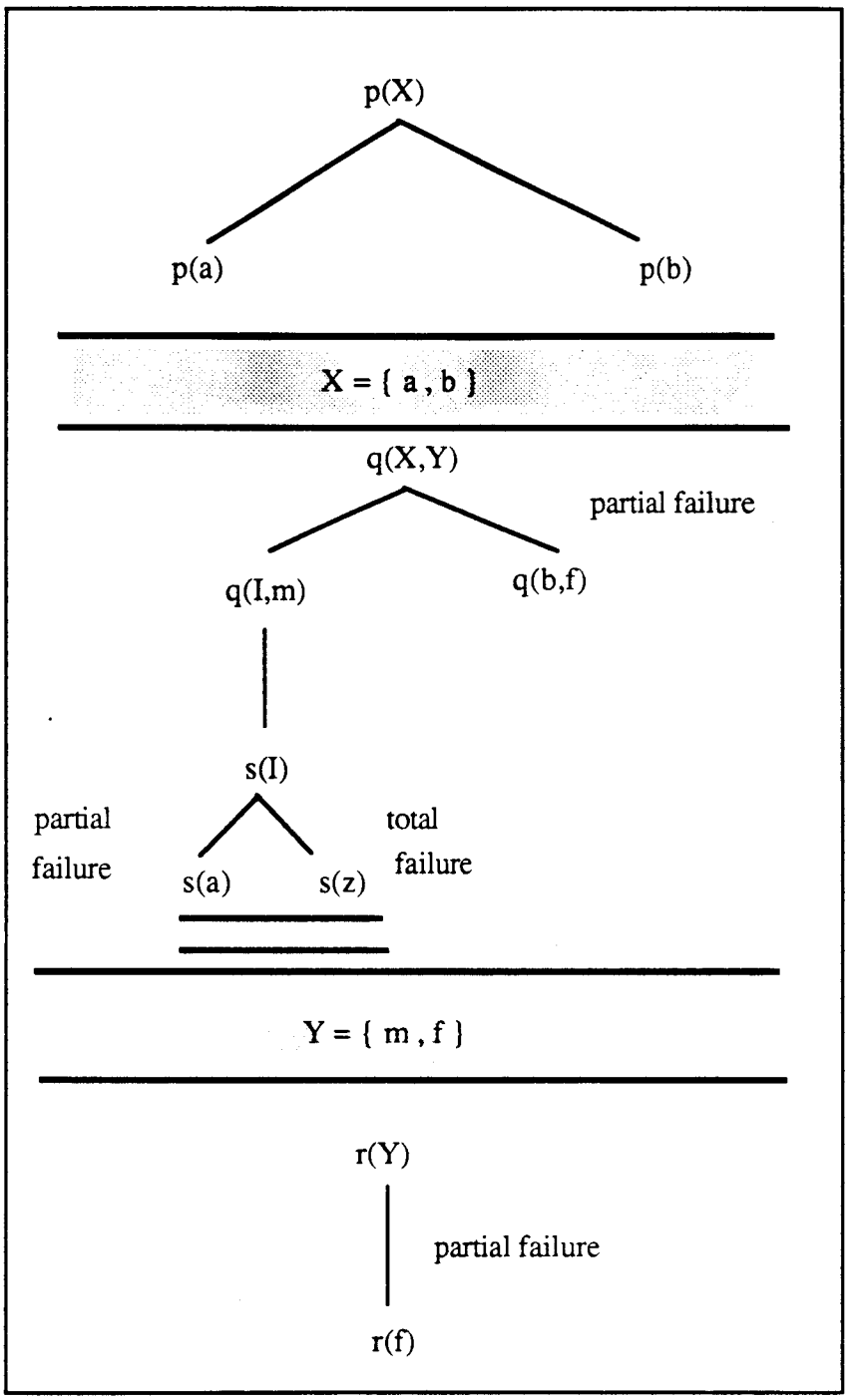


Figure (II.8) Different types of failures; partial and total

If we examine closely the different instantiations of X and Y, we find that, potentially, there are 4 solutions:

$$X = a \, , Y = m$$
$$X = a \, , Y = f$$

$$X = b\ ,\ Y = m$$
$$X = b\ ,\ Y = f$$

Now considering the solutions that are produced by standard Prolog, we find the only solution is $X = b\ ,\ Y = f$.

Problem 2.3

The case of partial success/failure necessitates a certain technique to identify occuring failures and consider them lately when dealing with multi-instantiations that include already failing instantiations.

II.3.5.2 Treatment of partial success/failures

Generally speaking, there are several proposals to treat partial success/failures commonly occuring during the multi-unification operations. We present different propositions to treat such type of failures.

1- Variable modification:

An elementary approach was to modify the multi-instantiations of the variables. As previously mentioned, variables sharing is adopted in the multi-resolution model, and hence this solution presents problems. Consider the following example:

Example:

Given the program,

p(a). *p(b).* *p(c).*

q(a). *q(c).*

with the query,

:- p(X),q(X).

Multi-resolving *p(X)*, the variable *X* will be multi-instantiated to *{ a,b,c }*. We now attempt to solve *q(X)*. There are two clause heads, so the first is considered, *q(a)*. Multi-unifying *X*, which is already multi-instantiated, to *a*, will result in a partial success/failure as the values *b*

and *c* will result in a failure. Now if we modify *X* to include only the succeeding values, then in this case it will be equal to *{ a }*.

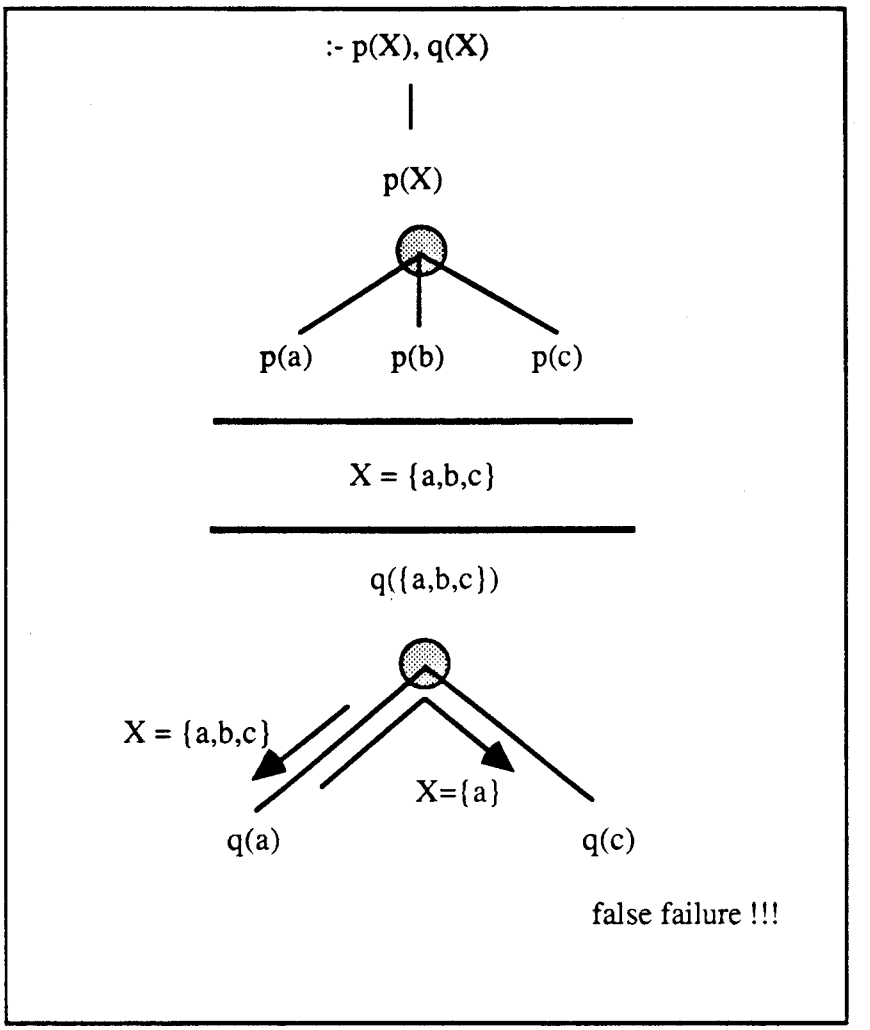


Figure (II.9): Failing variable's modification

The next step is to attempt the second clause head, in this case, *q(c)*. Given that *X* is now instantiated to *{ a }*, a total failure is achieved, while this is not true when comparing with the standard resolution. We should have multi-unified the three values of *X* to *c*, resulting in another partial success/failure, but since the contents of *X* were modified, then the multi-resolution did not proceed correctly (did not confirm to Prolog behaviour).

Another point of view was to keep the contents of multi-instantiation as it is, and simply adding a status flag to each instantiation. When a failure occurs, the corresponding instantiation that caused this failure is *disabled*. Afterwards, in the multi-resolution, this instantiation will never be considered.

The drawback of this approach is due to the problems resulting from sharing the multi-instantiations between the alternatives of a choice point. Modifying a multi-instantiation in one alternative will result in incomplete multi-resolution of the following alternatives.

Another important problem results from the case of having different multi-instantiations that belong to the same choice point. In this case, when a modification is made on a multi-instantiation, all the corresponding multi-instantiations should be modified accordingly. But there are cases when this is not quite evident:

Example:

Considering the following multi-instantiations; $X = \{a,b,c\}$ and $Y = [\ m\ /\ \{\ a,b,c\}]$, where X and Y belong to the same choice point, if X was modified to $\{c\}$, due to a partial success/failure, we cannot guarantee that Y will be accordingly modified to $[\ m\ /\ \{c\}]$.

In DAP Prolog [26], a similar approach is implemented by employing a mask to each set, where relational database programs were treated (and not rules). Adopting this approach in the multi-resolution model, a status mask was created for each choice point. A status bit 1 indicates that the instantiation is valid, and a status bit 0 means that the corresponding instantiation has resulted in a partial success/failure. The resolution takes place as follows:

The initial status masks of the multi-instantiations are true before attempting any alternatives. For each new alternative, a copy of the initial status of the masks is created. During the multi-resolution of each alternatives, partial success/failures may occur. In this case, this bit is reset to indicate a failure. A reset bit may not be attempted.

At the local synchronous OR level of this choice point, an OR operation takes place between the different copies of the status masks to create the resultant status mask of the initial multi-instantiations.

Due to the sharing representation and the association of several multi-instantiations to the same choice points, false results were produced. An example of such a case is as follows:

Example:

Considering the following multi-instantiations; $X = \{a,b\}$ and $Y = \{a,b\}$, where X and Y belong to different choice points, and the query $:-(X=Y ; X=\backslash=Y)$. The multi-resolution tree including the status masks are as shown in the following fig. (II.10).

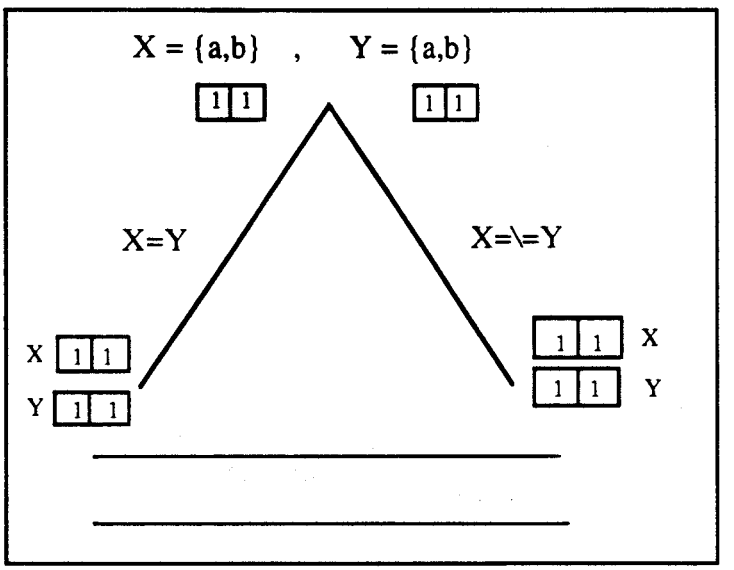


Figure (II.10): Status masks for multi-instantiations

In the first alternative, $X=a$ succeeds with $Y=a$ (i.e. the first bits are set in both masks), whereas the same instantiation ($X=a$) fails with $Y=b$. This is the first conflict that might occur. Another problem is that $X=b$ succeeds with $Y=b$ (the second bits are set in both masks) but this is not evident when we examine the status masks as there are no failures that are recorded to indicate that partial success/failures took place between particular instantiations.

We conclude that it is impossible to adopt the solution of modifying a multi-instantiation to treat an occuring partial success/failure.

2- Variable Copying:

A second approach is to work with a copy of the modified variable. If we apply this idea to the previous example, then we create a new copy of the variable each time we attempt to multi-unify the subgoal with a clause head, and create a copy of the resulting succeeding values. Fig.(II.11) represents such a case.

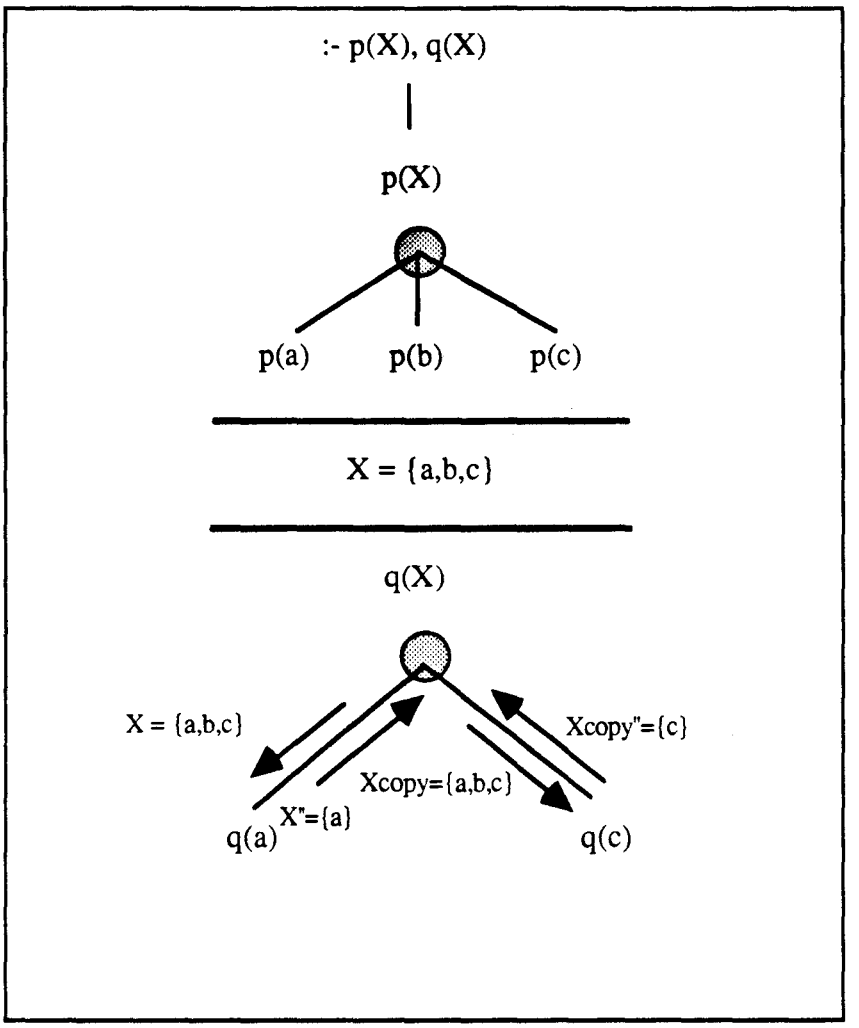


Figure (II.11): Failing variable copying

The same aspects discussed in the previous approach represent the sources of inconvenience of this solution. Shared multi-instantiations and having more than one multi-instantiation that originate from the same choice point are the main sources of confusion.

Imagining a system of multi-instantiated variables together with their duplicates, surely an elevated memory consumption will be the end result and this is what we are aiming to avoid at the first place. Hence we conclude the inconvenience of this proposition.

3- Saving failures:

The third proposal is to keep all the original multi-instantiations as they are, and when a partial success/failure is reached, the system memorises the failing instantiations. In our example, we should record that the second and the third instantiations resulted in a partial

success/failure in the first alternative of *q*, while the first and second instantiations resulted in another partial success/failure of the second alternative. In this case the multi-resolution continues with the initial ensemble of multi-instantiations, and utilises the failures information when necessary.

Actually, this is the solution that we adopted in our model. Chapter 5 is dedicated to a detailed discussion of the failures phenomenon in general and a profound presentation of the partial success/failures in particular where we explain how a special treatment was applied for optimal performance.

II.3.5.3 Types of partial success/failures

When multi-unifying 2 multi-terms, a partial success/failure might take place. Following are the different types of partial success/failures.

1- During the same unification operation:

Some partial success/failures are vital to be treated in the multi-unification or else useless multi-execution continues. Consider the following example.

```
p(1).     p(2).     p(3).
q(1,2).   q(2,3).
:- p(X), q(X,X).
```

In this program, *X* will be multi-instantiated to *{ 1, 2, 3 }*. When attempting to solve *q(X,X)*, the first clause head is selected. Multi-unifying *X* to *1* will result in a partial success/failure with the instantiations *2* and *3*. Since a total failure was not reported, multi-unification continues to multi-unify the second pair of arguments *X* and *2*. Again, partial success/failures are detected. The same operation is repeated with the second clause head.

Standard Prolog fails when attempting to solve the above query. Thus, it is expected that the multi-unification algorithm detects a total failure in this case.

Problem 2.4

Detection of total failures that occur as a result of an ensemble of partial success/failures in the same multi-unification operation.

2- During different unification operations:

Problem 2.5

In the general case, when attempting to solve a subgoal, some partial success/failures may occur. It would be optimum if the following subgoals detect such failures to optimise the performance by neglecting the instantiations that led to previous failures and will serve no more to find a solution.

Example:

For the query,

:- (A = [a,a] ; A = [b,b] ; A = [c,c]) , A =\= [b,b], A = [B,C].

A will be multi-instantiated to *{ [a,a] , [b,b] , [c,c] }*. The second subgoal results in a partial success/failure. If this partial success/failure is taken into account during the third subgoal, the variables B and C will be equal to:

B = { a , c } and not *{ a , b, c }*
C = { a , c } and not *{ a , b , c }*

reducing the memory consumption.

II.3.5.4 Recapitulation

From the above discussion, it is evident that the definition of a multi-instantiation (definition 2.1) is insufficient to solve the different problems that we explored throughout the discussion of the multi-unification algorithm. We summarise these problems:

- problem (2.1): ensuring the coherence of an instantiation,
- problem (2.2): ensuring the coherence of an operation involving two multi-instantiations,
- problem (2.3): memorising instantiations that led to partial success/failures,
- problem (2.4): recognizing total-failures resulting from partial success/failures in the same multi-unification operation, and
-problem (2.5): taking into account already occurred partial success/failures in the following subgoals.

It is clear that the individual instantiations bound to each variable play an important role in the multi-resolution. Hence, a more clear, nonambiguous representation of multi-instantiations is required. This is what we present in the following chapter.

II.3.6 The Multi-Resolution Tree

In the multi-execution phase, we traversed completely the multi-resolution tree resolving the given query. The multi-resolution tree is a tree representing the different succeeding traversed choice points, and the different succeeding branches representing the succeeding alternatives for each subgoal.

Conceptually, the multi-resolution tree differs from the standard OR resolution tree in the same way as the multi-resolution model differs from the standard resolution. Each choice point is represented only once, as deep backtracking is eliminated. No solution paths are visible on the multi-resolution tree. Figures (II.1) and (II.2) of this chapter showed clearly a standard OR resolution tree and its equivalent multi-resolution tree. Other figures throughout this chapter explained in detail the concept of the multi-resolution tree.

II.4 The Multi-Outputs phase

In the above sections, we explored thoroughly the different steps of the multi-execution phase, which was the first phase in the multi-resolution model of Prolog programs. We eliminated deep backtracking and we introduced multi-instantiated variables. Now, we come to the final phase which is the display of the ensemble of solutions. Given such a system, how could we display for the user all the possible true solutions for the nondeterministic problem under investigation?

To answer this question, we first state what we have and what we want to produce. At this level, the complete multi-execution of the given query took place, resulting in a number of data structures. Using these data structures, we want to display all the solutions of the given query to the user.

II.4.1 Existing Data Structures

The data structures produced by the multi-execution phase are the input to the multi-output phase. At this level, we have the following data structures stored in the memory of the system:

- the multi-resolution tree
- instantiated objects (mono instantiated or multi-instantiated), and
- encountered failures.

Now, the question is using the existing data structures how could we display the correct solutions?

II.4.2 Display of solutions

The phase of display of solutions is an important phase in the multi-resolution. It is here where the partial success/failures are really processed and the succeeding values are displayed to the user.

A first approach was to produce same order and the same number, of solutions as standard Prolog. To satisfy this constraint, this necessitated the reconstruction of the standard resolution tree with the aid of the stored data structures. We observed that this approach consumes a considerable time. Since a remarkable speedup was achieved from the multi-execution phase, which we will discuss in detail in chapters 6, we found that the time taken to process the existing data structures to produce the solutions consumed a considerable time. We intended to reduce the time taken to display the solutions as much as possible.

Accordingly, our second approach depends on the multi-instantatiations of the variables mentioned in the query subgoals without the reconstruction of the standard resolution tree. Such an approach does not guarantee the order of the displayed solutions to be similar to that of standard Prolog. Nevertheless, it produces the same results.

We present each of these approaches.

II.4.2.1 Reconstruction of the Standard Resolution Tree

In this approach, we actually construct the different solution paths of the query in a standard resolution tree. A valid solution path is defined by a list of succeeding alternatives of different choice points, starting from the root, until a leaf is reached, making sure that no partial success/failures occured along this solution path.

The input to this algorithm is the list of variables in the query, together with the failures' information and the multi-resolution tree. By default, it starts to construct the different

solution paths starting from the root. At each level, it retrieves from the tree the succeeding successors. Selecting one successor at a time, it builds a solution path. The algorithm favours the left-most successor. After the completion of this solution path, the same operation is repeated on all brothers.

Each time before the system appends a new branch to the solution path under construction, it should verify that no partial success/failures occured in this branch. If a failure is recorded, then this branch, and accordingly this solution path, are omitted, otherwise it continues, depth-wise, the construction of the solution path.

A complete path is detected when no more successors exist for the current branch, and that no failures were encountered along the constructed path. If this is the case, then the system displays the solutions of the variables corresponding to this solution path. Afterwards it resumes the above routine.

This algorithm follows the standard execution model of Prolog; it favours the first succeeding son, i.e. left-most. Deep backtracking takes place, but with no unification operations. Only the correct solution paths are constructed. This is why, it was expected to produce the same number of solutions as that produced by Prolog and in the same order as well, as it adopts the same search criteria.

This approach is a valid approach, it produces the same solutions as standard Prolog, in order and number. The drawback of such an approach is the long tedious algorithm that occupies a considerable execution time. We compared the processing time versus the time required for the display of the solutions of such a method when running large benchmarks, and it was quite surprising. It almost took about 60-70% of the execution time to display the solutions. This motivated us to try to display the solutions alternatively without the dependance on the resolution tree. The corresponding algorithm is presented in the following section.

II.4.2.2 Without the reconstruction of the standard resolution tree

This approach relies on the different multi-instantiations. Given the list of variables in the query, different combinations of the instantiations take place to produce all the solutions. Hence, coherency tests are necessary in such a case. Also, checking with the partial success/failures should take place to produce the correct solutions only.

This algorithm consumes considerably less time than the previous one as it is less complex because we do not reconstitute the standard resolution tree. It is worth noting that it is possible that it does not guarantee the order by which the solutions are produced.

II.5 Conclusion

In this chapter, we presented the basic idea of the multi-resolution model. After a thorough look through the different phases of the model, substituting deep backtracking by multi-instantiated variables is as simple as it seems. Certain ambiguous cases occur due to the naive representation of the multi-instantiations. The following chapter resolves such ambiguities with a more sound representation.

Chapter Three

The Multi-Execution Model

Abstract

In this chapter, we present the different characteristics of the model including the representation of multi-instantiated variables assuring the elimination of all the discussed problems in the previous chapter. A detailed study of the multi-unification algorithm is presented that includes coherency tests and treatment of failures.

III.1. Introduction

The previous chapter was dedicated to a full discussion of all the underlying difficulties in the multi-resolution model due to the replacement of the deep backtracking feature by the multi-instantiated variables. We concluded that to overcome the mentioned problems, a major modification in the simple structure representing a multi-instantiation was necessary.

Recalling definition 2.1, a multi-instantiation is represented by $\{a_1,..,a_n\}$, where a_i is the i^{th} instantiation and could be any multi-term (definition 2.2). When running several benchmarks, we found many special cases where the given representation of a multi-instantiation does not ensure the coherency of an instantiation (problem 2.1) or the coherency of an operation including 2 multi-instantiations (problem 2.2). An example is the case when two multi-instantiations, originating from the same choice point, are involved in an operation.

A first attempt was to give each branch (alternative) in the multi-resolution tree a distinct date. This date is the multi-unification number. Instantiating any variable in a branch, the instantiated value will be associated to the date of the branch where the multi-unfication took place. Thus, we represented a multi-instantiation by:

$$\{ (\underline{date}_1,a_1) ,..,(\underline{date}_n,a_n) \}$$

where $\underline{date}_i$ is the i^{th} date associated to the i^{th} instantiation.

In the case of two multi-instantiations that are originally from the same choice point, we remarked, when running several benchmarks, that they might not have the same number of instantiations due to occuring failures. Given the above representation, it was difficult to assure the coherency features due to long complicated routines.

On the other hand, in the treatment of partial success/failures, we concluded that memorising the occurred partial success/failures was the solution that we decided to adopt in our model (problem 2.3). In an early stage of this work, we represented partial success/failures in terms of the dates of the failing instantiations which were not processed except in the multi-outputs phase.

Again, when running large benchmarks, we found that there are several important counter aspects:
- Treating partial success/failures at the very end in the multi-outputs phase was not the optimum decision due to the large number of partial success/failures that have occurred (we are considering complex benchmarks). This resulted in a considerable time to produce the solutions which spoilt the speedup that we gained in the multi-execution phase.

- At the same time, postponing the processing of the partial success/failures to the very end resulted in some cases, in useless work that should not have been done, if these failures were treated during the multi-execution. An example is the case of partial success/failures that lead to a total-failure (problem 2.4).

- Another aspect was the optimisation of the memory size. If previously occurred partial success/failures are considered at the moment of a new multi-unification operation, then the new created multi-instantiations will not include the failing instantiations that will reduce the size of the multi-instantiation and hence the memory size used (problem 2.5).

To sum up, we modified the representation of the multi-instantiation to include the choice point number, which will have a major role in the multi-resolution, which we will discuss later. We also introduced different mechanisms to treat the failures during the multi-execution phase that will assure the correctness of solutions and avoid any false solutions (problem 2.4). We consider the aspect of problem 2.5 an optimisation aspect, that we will not discuss in this chapter.

Recalling what was previously mentioned, we need a simple representation that contains all the necessary information to solve the problems that were previously discussed:

- Problem 2.1: detection of incoherent multi-terms,
- Problem 2.2: detection of incoherency in operations that involve two instantiations,
- Problems 2.3: structured enough to memorise partial success/failures,
- Problem 2.4: treating partial-failures that might lead to a total-failure (same multi-unification operation).

III.2 Multi-execution model

We present here an informal algorithm of the multi-execution of a query that takes place in the multi-execution phase. We clearly detail the methodology of the creation of the multi-instantiations.

III.2.1 The multi-execution algorithm

The model performs the following operations:

For each subgoal:

A- Check if a choice point exists or not. A choice point exists, if, in the program, there exists more than one clause head to solve that query. If yes, then:

1-A new unique sequential choice-point-number is given to this choice point. It is the number of apparition of this choice point in the multi-resolution. A choice point having a choice-point-number= **2** is a choice point that appeared in the multi-resolution before another choice point whose choice-point-number = **3**, for example.

2- Clause heads are selected from the left to the right, i.e. in the order by which they were written in the program (similar to Prolog).

3- For each clause head (alternative or branch):

1- A new branch-date is generated. This is a distinct date, generated independently from the numbering of the choice points, and that tags all instantiations that will occur in this alternative.

2- A multi-unification operation is attempted between the subgoal and this clause head.

3- If the multi-unification with the clause head succeeds, then each instantiation that took place will be tagged by the branch-date. Hence, the created structure will be in the form (branch date,value), which we will call an *instantiation-pair*.

4- If this clause head is a rule, then the body clauses are invoked, in the same manner.

5- If the multi-unification fails totally, then this alternative is completely omitted and any variables that were instantiated in this branch will be de-instantiated. The multi-resolution proceeds to the next clause head.

6- If partial success/failures occur, then these failures are recorded in a *failures database* in terms of the *date-paths* of both failing terms, together with the *failing-branch-location*. Following is a clarification of these terms.

7- The same operation takes place each time the subgoal is being unified to a new clause head.

4- After no more clause heads exist, the *local-synchronous-OR-phase* for this choice point performs the following operations:

1- It gathers the different solutions, or instantiation-pairs, of the different variables in the clause heads, from the different branches of this choice point.

2- It creates the multi-instantiated variables, in terms of the choice-point-number and the instantiation-pairs.

3- Multi-execution proceeds to the following subgoal.

B- If a choice point for this subgoal does not exist, then a multi-unification is attempted between the subgoal and the unique clause head. The result of this multi-unification is a total-success, a total-failure or a partial success/failure. In the case of a total-failure, then a failure is reported. In the case of partial /success failure, the failing branch-location is predecessor branch of this subgoal. If no predecessor exists, then it is recorded to the root level so as to sense this partial failure on a global level to avoid any further operations including the two instantiations that caused this partial-failure.

III.3 Representation of multi-instantiation

III.3.1 A multi-instantiation

Now onwards the naive representation of a multi-instantiation given by definition 2.1 (page II.4) is no more valid. We redefine the representation of a multi-instantiation.

Definition 3.1: A multi-instantiation

We present a formal form of a multi-instantiation as follows:

$$\{\textbf{choice-point-number},\ [\ \text{Instantiation-pairs}\ ]\ \}$$

where,

- the structure, {...}, represents a multi-instantiation,
- **choice-point-number** is a unique *choice point number*,
- *Instantiation-pairs* = $(\underline{date}_1,value_1)$, $(\underline{date}_2,value_2)$,..., $(\underline{date}_n,value_n)$
- $(\underline{date}_i,value_i)$ is the $i^{\underline{th}}$ instantiation-pair of a variable V,
- $\underline{date}_i$ represents the *date* of the $i^{\underline{th}}$ instantiation-pair of V, and
- $value_i$ is corresponding $i^{\underline{th}}$ *sub-term* associated to the $i^{\underline{th}}$ date of V.

Definition 3.2: A sub-term

A sub-term is a value to which a variable was instantiated in a certain branch. It could be any multi-term in the multi-resolution.

III.3.2 Memory representation of a multi-instantiation:

A general scheme representing a multi-instantiation is given in fig. (III.1):

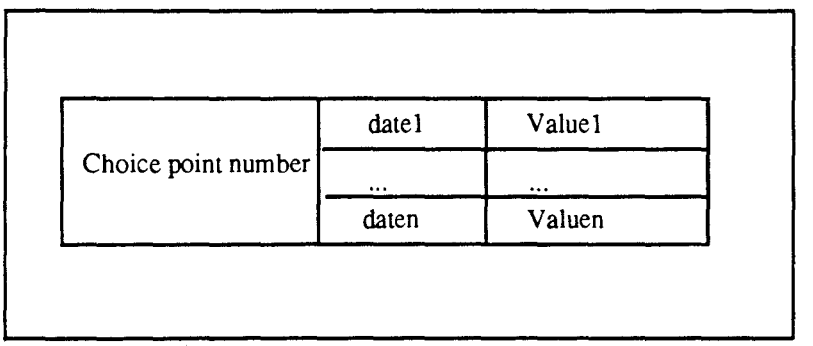


Figure (III.1) : A multi-instantiation

We store in the memory the branch-date for each sub-term, together with the choice point where this multi-instantiation was created.

We emphasize that multi-instantiations are shared in the multi-resolution. Creation of new sub-terms that are equal to (or include) a multi-instantiation is done by a simple memory

reference. No copying takes place, except for certain special cases that we will discuss in the following chapter. Multi-instantiation examples are presented hereafter.

Example 1:

This is the case of a simple multi-instantiation. All sub-terms are atoms. Each sub-term is tagged by the corresponding branch-date.

1- X = { **1** , [(1,a), (2,b), (3,c)] }

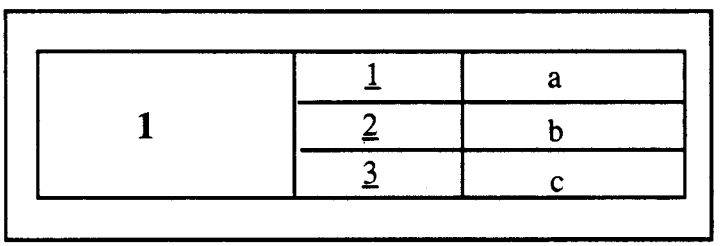


Figure (III.2): A variable multi-instantiated to 3 atoms

Example 2:

X = { **2** , [(4,a) , (5,{ **1** , [(1,a) , (2,b) , (3,c)] }) , (6,m)] }

The given multi-term is a more complex multi-instantiation. The second sub-term is a multi-instantiation. When X is being represented in the memory, a memory reference is made to the second multi-instantiation. No copying takes place.

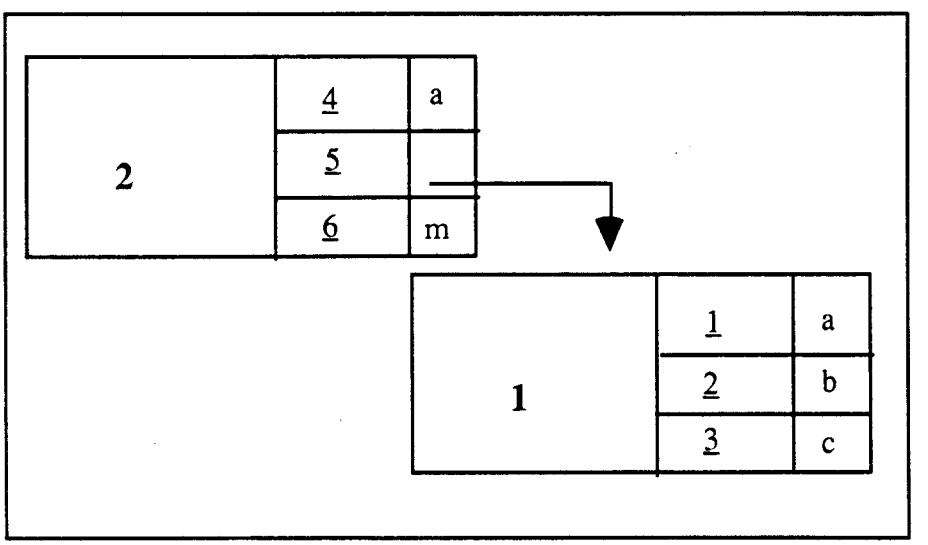


Figure (III.3): A multi-instantiated complex object

Example 3:

X = { 3 , [(5,[a,b]) , (6,[])] }

This case is the case of a multi-instantiation of lists.

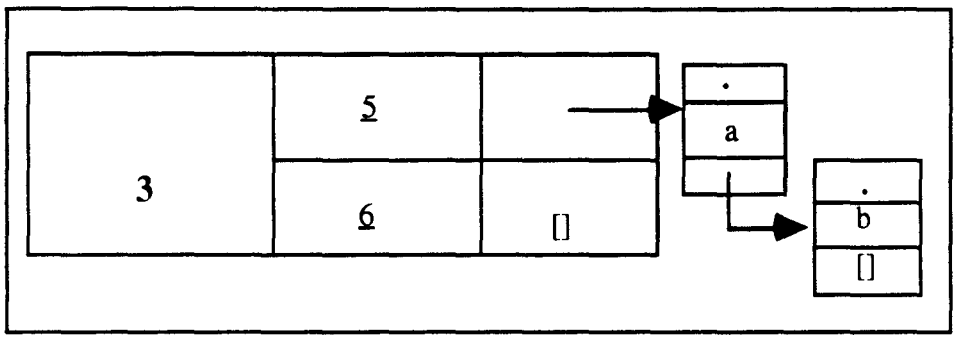


Figure (III.4): A variable multi-instantiated to different lists

Example 4:

X = [{ 1 , [(1,a) , (2,b)] } | { 2 , [(3,c) , (4,d)] }]

This is the case of a list whose elements are multi-instantiated. Each element location includes a pointer that points to the multi-instantiation in the memory.

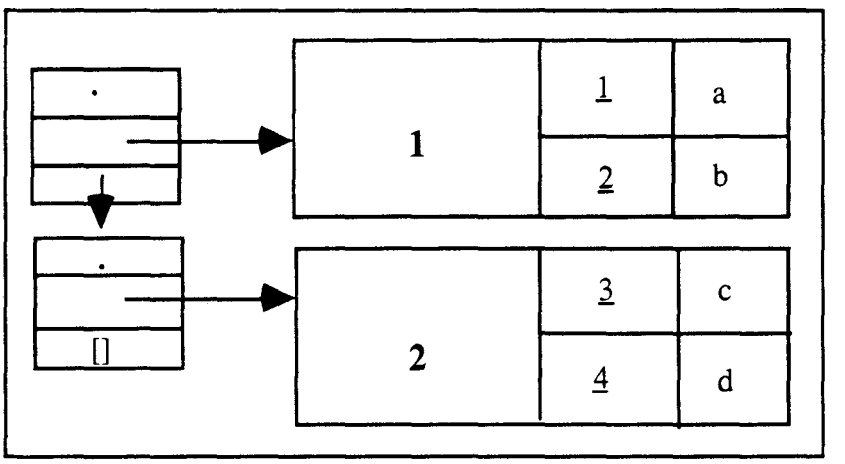


Figure (III.5): A mono instantiated object including multi-instantiated variables

III.3.3 A Choice Point

Definition 3.3: A choice-point-number:

It is a unique number assigned to each new choice point. The choice points are numbered in the order by which they appear during the multi-execution of the query.

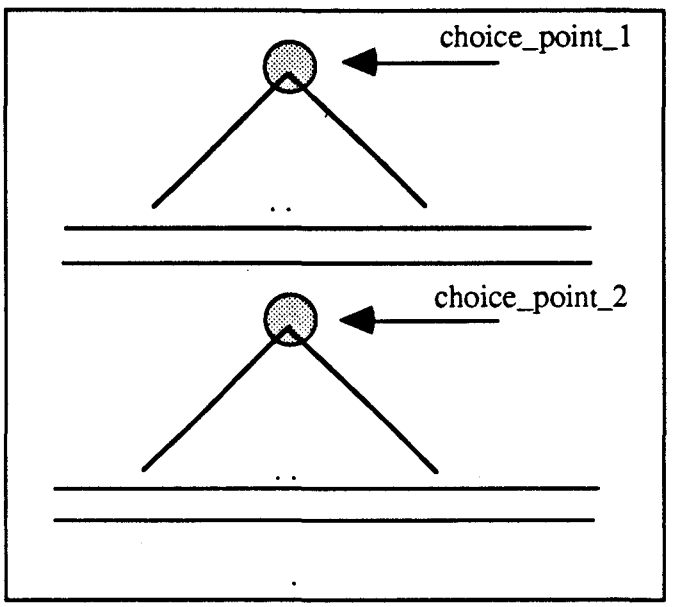


Figure (III.6): Numbering of choice points in the multi-resolution tree

If given two multi-instantiations that include choice_point_1 and choice_point_2, respectively, we may deduce the following:

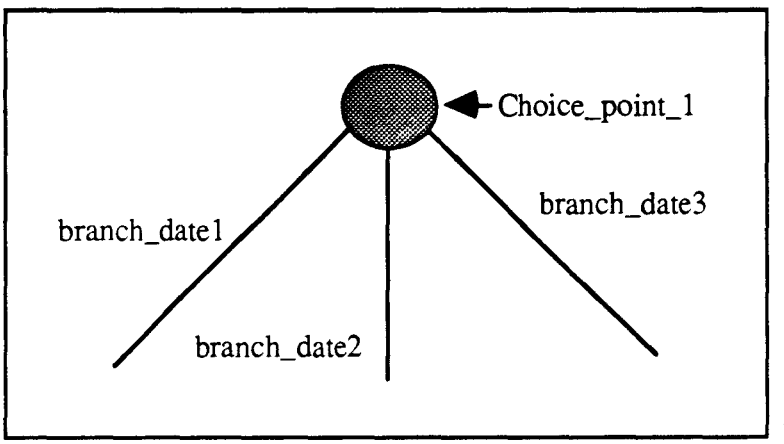
-if choice_point_1 is equal to choice_point_2 then both multi-instantiations originated from the same choice point.
-if choice_point_1 is not equal to choice_point_2 then the two multi-instantiations originated from different choice points.

We will see afterwards that associating this information about the choice point to the representation of the multi-instantiation will facilitate enormously the coherency tests (problems 2.1 and 2.2). A full discussion is given in sections III.4.2.4 and III.4.2.5 of this chapter.

III.3.4 A branch in a choice point:

Definition 3.4: A branch-date:

In a choice point, each alternative (clause head) is given a unique branch number, which we called *branch-date*. It is actually the number of the multi-unification operation that multi-unifies a subgoal to this clause head.The numbering system of the branches of a choice point is independant of the numbering system of the choice points. Branches are numbered sequentially. The ordering of the branches is not of any significance in the multi-execution.



Figure(III.7): Numbering of different branches of a choice point

After the exploration of all alternatives, the local synchronous OR phase of this choice point, the choice point number together with all its branch dates are stored.

Definition 3.5: A branch-location:

The branch-location of an alternative in a choice point is defined in terms of its date associated to the choice_point_number of the choice point to which it belongs. It is given by the term (**choice-point-number**,branch-date).

III.3.5 A Date-path

Definition 3.6: A date-path:

A date-path of a sub-term is a list of branch-locations. It represents the path to access a multi-term in a multi-instantiation. It is constructed sequentially. For the multi-instantiation,

$$\{\ \mathbf{1},\ [\ (\underline{1},a),\ (\underline{2},\{\ \mathbf{2},\ [\ (\underline{3},x),(\underline{4},y),(\underline{5},z)\ ]\},\ (\underline{6},c)\ ]\}$$

the date-path of the first sub-term, *a*, is [(**1**,1)]. Given the new representation of multi-instantiation, this information is very clear as follows:

$$\{\ \mathbf{1},\ [\ (\underline{1},a),\ (\underline{2},\{\ \mathbf{2},\ [\ (\underline{3},x),(\underline{4},y),(\underline{5},z)\ ]\},\ (\underline{6},c)\ ]\}$$
$$\quad |\_\_\_\_\_|$$

The date-path is constructed in terms of the choice-point-number **1** and the branch-date associated to the sub-term *a*, in this case 1.

The date-path of the multi-instantiation (the second sub-term) is [(**1**,2)].

$$\{\ \mathbf{1},\ [\ (\underline{1},a),\ (\underline{2},\{\ \mathbf{2},\ [\ (\underline{3},x),(\underline{4},y),(\underline{5},z)\ ]\},\ (\underline{6},c)\ ]\}$$
$$\quad |\_\_\_\_\_|$$

To access the sub-term *x* in the multi-instantiation, we encounter the first branch-location (**1**,2) that is the date-path of the second multi-instantiation, then we traverse (**2**,3) to reach *x*. Hence, its date-path is given by [(**1**,2), (**2**,3)].

$$\{\ \mathbf{1},\ [\ (\underline{1},a),\ (\underline{2},\{\ \mathbf{2},\ [\ (\underline{3},x),(\underline{4},y),(\underline{5},z)\ ]\},\ (\underline{6},c)\ ]\}$$
$$\quad |\_\_\_\_\_|\quad |\_\_\_|$$

Similarly, the date-paths of the other sub-terms of the given multi-instantiation could be retrieved in the same way. Table (3.1) represents the different date-paths.

Sub-term	Date-path
a	[(**1**,1)]
{**2**,[(3,x),(4,y),(5,z)]}	[(**1**,2)]
x	[(**1**,2),(**2**,3)]
y	[(**1**,2),(**2**,4)]
z	[(**1**,2),(**2**,5)]
c	[(**1**,6)]

Table (III.1): Date paths of different sub-terms of a multi-instantiation

III.3.5.1 Construction a date-path

The date-path is constructed in terms of branch-locations. As previously mentioned, a branch-location is given by (**current choice point number**, current branch-date). The date-path is constructed sequentially. We illustrate this with the following example:

Example:

For the multi-instantiated variable *X*, given by

$$X = \{\ 3\ ,\ [\ (7,a)\ ,\ (8,\ \{\ \mathbf{1},\ [\ (1,x),(2,y),(3,z)\ ]\ \}\ )\ ]\ \}$$

it includes two sub-terms; *a*, tagged by the branch-date 7, and a multi-instantiation, { **1**, [(1,x),(2,y),(3,z)] }, tagged by the branch-date 8. The date-path of the former is [(**3**,7)], while that of the latter is [(**3**,8)].

Now, if we are to access the sub-term *x* in the multi-instantiation, we have already created a part of its date-path that begins with [(**3**,8)]. To really reach *x*, we have to traverse another branch-location, which is (**1**,1). Before appending this branch-location to the already existing date-path, we have to perform a number of coherency tests (problems 2.1 and 2.2, page II.14) that we will discuss later in this chapter. If these tests are satisfied, then the branch-location (**1**,1) is appended to the path [(**3**,8)] resulting in the new date-path [(3,8),(1,1)]. This new date-path represents a coherent sub-term. The details of the coherency tests are presented in the sections III.4.2.4 and III.4.2.5.

Each time a new branch-location is appended to the already constructed date-path, the system checks whether the encountered sub-term, corresponding to the existing date-path, might be multi-unified to the other multi-term or not. If this is the case, then a multi-unification takes place between the encountered sub-term and the other multi-term, otherwise, it continues the construction of the date-path until another sub-term is encountered.

Example 1:

multi-unify (X,Y), where

X = {**1**, [(1,a), (2,b)]} and Y = a

Since *X* is multi-instantiated, then each sub-term will be multi-unified independently. We will detail the different cases of the multi-unification in the following section, meanwhile we are interested in the construction of the date-paths. To access the sub-term *a*, the corresponding date-path is [(**1**,1)]. The system tends to multi-unify the encountered sub-term, that is *a*, to the atom *a* which will result in a success.

Example 2:

multi-unify(p({**{1, [** (1,a), (2,b)**]}**),q(X)).

The multi-unification fails (as in standard Prolog) before even starting to construct the date-paths of the different sub-terms of the first multi-term. This is because the functor names are different and hence non unifiable.

The above example points out the case where a multi-unification operation might take place before encountering the actual sub-term constituting a multi-instantiation. We will clarify this point again throughout the discussion of the different cases of the terms to be multi-unified.

Example 3:

multi-unify({**1, [** (1,a), (2,b)**]}**,Y)

Since *Y* is a variable, a direct assignment takes place with neither the construction, nor validation of the date-paths of the different sub-terms assigned to *X*.

III.3.5.2 Features of a date-path

There are some rules concerning the construction of the date-paths:

1- A date-path of any sub-term should never include two different branch-locations of the same choice point. Assuming a date-path is given by:

date-path = [..., (**choice-point-i**,$date_i$) , (**choice-point-j**,$date_j$),...]

then **choice-point-i** is never equal to **choice-point-j** unless if $date_i$ = $date_j$.

This is to satisfy the rule restricting the coherency of a sub-term that states that it is prohibited to access two alternatives of the same choice point. This immediately solves the problem 2.1. That is, just knowing the choice points from which two branch-locations originated, we can easily perform the coherency tests.

2- A date-path may include redundant branch-locations. Consider the following example:

Example:

For the multi-instantiated variable *X* given by:

$$X = \{\ 1\ ,\ [(1,a)\ ,\ (2,\ \{\ 2\ ,\ [\ (3,x)\ ,\ (4,\ \{\ 1\ ,\ [\ (2,b)\ ]\})\ ]\})\ ]\ \}$$

the date-path of the sub-term *b* is [(**1**,2) , (**2**,4) , (**1**,2)].

This feature is accepted as long as the coherency rules are not violated. An optimisation to be proposed is to eliminate any redundancy existing in a date-path. This is to ameliorate the performance.

3- A date-path includes unsorted branch-locations.

Example:

$$\{\ 1\ ,\ [(1,a)\ ,\ (2,\ \{\ 3\ ,\ [\ (5,x)\ ,\ (6,\ \{\ 1\ ,\ [\ (2,\{\ 2\ ,\ [\ (3,m,(4,n))\ ]\})\ ]\})\ ]\})\ ]\ \}$$

The date-path to access the sub-term m is [(**1**,2), (**3**,6), (**1**,2), (**2**,3)]. This is an unsorted date-path, but it is a coherent correct one.

An optimisation is to sort the date-path. This feature influences the treatment of failures. We will discuss the sorting problem of a date-path in chapter 4, when examining the failures treatment.

4- date-path for a ground term = [].

III.4 Multi-unification algorithm

Before presenting a detailed discussion of the multi-unification algorithm, we recall several aspects concerning the standard unification algorithm.

III.4.1 The Standard Unification Algorithm

The following program, *unify(X,Y)*, represents the standard unification algorithm expressed in a Prolog program without the *occur-check* included. The case when the two terms *X* and *Y* are not equal are separated for analogy reasons when examining the multi-unification algorithm.

```
unify(X,Y):-
        atomic(X), atomic(Y), !, X == Y.

unify(X,Y):-
        var(X), nonvar(Y),! ,X = Y.

unify(X,Y):-
        nonvar(X), var(Y), !, Y= X.

unify(X,Y):-
        var(X), var(Y),! ,X = Y.

unify(X,Y):-
        nonvar(X), nonvar(Y), functor(X,F,N), functor(Y,F,N), !,
        unify_arguments(N,X,Y).

unify_arguments(0,X,Y).
unify_arguments(N,X,Y):-
        arg(N,X,Xn), arg(N,Y,Yn), unify(Xn,Yn),
        N1 is N-1, unify_arguments(N1,X,Y).
```

Program(III.1): The Standard Unification Algorithm without the occur check

In the unification algorithm presented above, the relation *unify(X,Y)* succeeds if *X* unifies with *Y*. The clauses of *unify* outline the possible cases. The first case is that of two atoms. Two variables unify in standard Prolog. On the other hand, if *X* (or *Y*) is a variable, then *X* (or *Y*) unifies with *Y* (or *X*).

Finally, if *X* and *Y* are compound functions, with the same functor and the same arity, then their corresponding arguments are unified respectively.

Alternatively, if there arrives a case outside these stated above, a failure is detected and the algorithm terminates. If not, then the terms unify.

We might represent the different inputs and outputs of this algorithm as shown in figure (III.8).

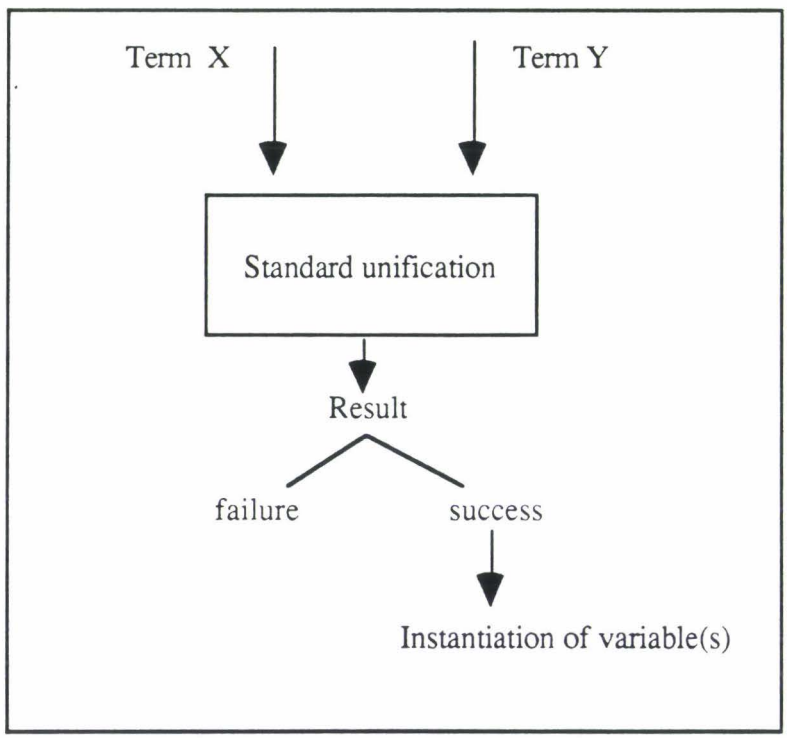


Figure (III.8): Inputs and outputs of the standard unification algorithm

The inputs to the algorithm are *X* and *Y*; the two terms to be unified. The output is either a total-success or a total-failure. In the case of a total-success, variables, if any existed, will be instantiated. On the other hand, in case of a total-failure, all variables that were instantiated in this unification operation will be deinstantiated.

III.4.2 The Multi-Unification Algorithm

The main feature of the multi-resolution model is the presence of multi-instantiations. It is evident that the standard unification algorithm does not support all unification operations dealing with such multi-instantiated variables. Accordingly, we defined the multi-unification algorithm to support all unification operations in the multi-resolution model. The multi-unification algorithm is an extension of the standard unification algorithm, including the standard unification operations together with all operations of multi-unifying multi-instantiated variables.

The multi-unification algorithm performs the classical unification operations, together with other features:

- it supports all unification operations between any two multi-terms,
- it constructs the date-paths of the accessed sub-term to ensure their coherency according to the rules, previously stated (problem 2.1, page II.12),
- it ensures the coherency of the multi-unification operation between the two sub-terms to be multi-unified (problem 2.2, page II.14),
- it signals any partial success/failures occuring during multi-unification (problem 2.3, page II.20), and
- it recognises certain failures and treats them (problem 2.4, page II.25).

The inputs to the multi-unification algorithm are the two multi-terms, X and Y to be multi-unified. Moreover, the *current choice-point-number* and the *current-branch-date*, constituting the *current-branch-location* are also inputs to the algorithm.

A scheme representing the inputs and output of the algorithm is given in fig. (III.9).

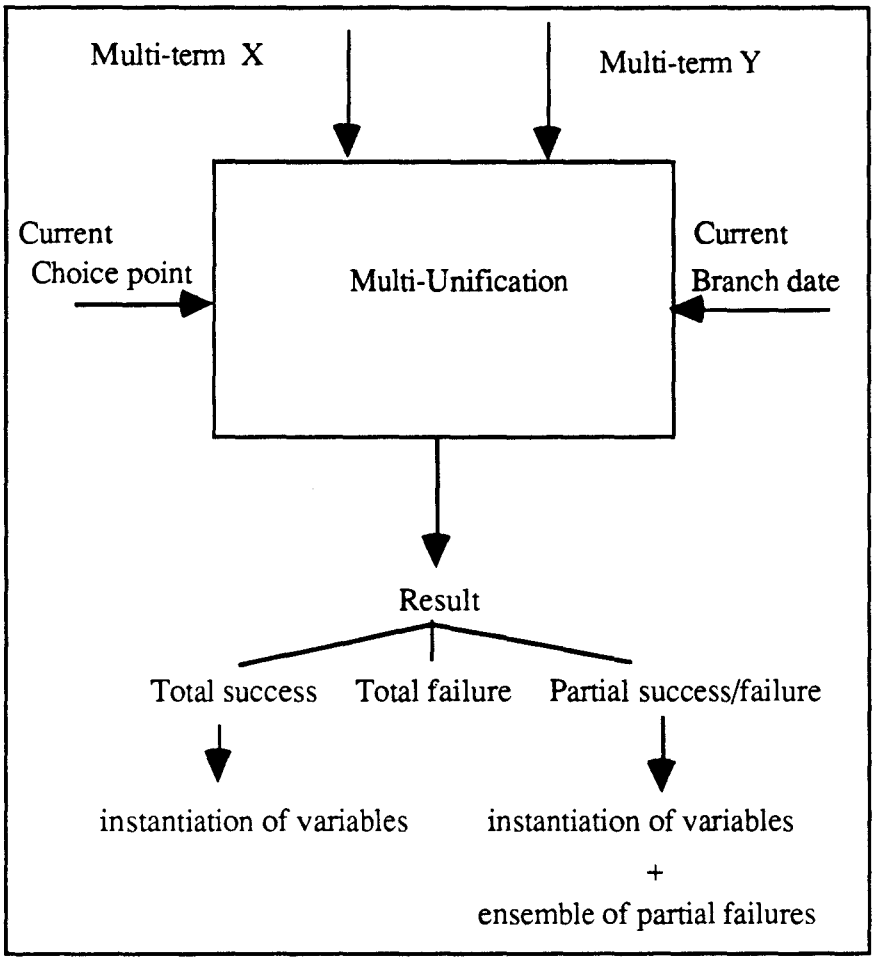


Figure (III.9): Inputs and outputs of multi-unificationalgorithm

The output of the multi-unification algorithm is one of three cases; a total-failure, a total-success or a partial success/failure. In the case of total-failure, a failure to multi-unify the two multi-terms is reported. If total-success is achieved, then the variables are instantiated. In the case of partial success/failures, an ensemble of the partial-failures that took place will be reported together with the different instantiations of the variables resulting from this multi-unification.

Before discussing the different unification cases that take place in the multi-unification algorithm, we present the failures' database (the solution to the problem 2.3):

III.4.2.1 Failures database

Problem 2.3 (page II.20) was concerned with the methodology of memorising partial success/failures that might occur in a multi-unification operation. Given the concrete representation of a multi-instantiation, we decided to save the partial success/failures in a

database, which we called the *failures' database*. Each time a partial failure is encountered, a new record is 'asserted' that contains all the information of this partial success/failure.

As discussed in section II.3.5.2, we discussed the different possibilities to treat the partial failures. We concluded that we will memorise the failures aside. A failure occurs between two sub-terms in a certain branch-location. We represent a partial failure in terms of the date paths of both sub-terms that led to this failure, together with the current branch-location as the failing branch-location, which is given by: (**current-choice-point-number**, current-branch-date).

Fig. (III.10) shows the record structure of the failures' database.

date-path-1	date-path-2	failing branch-location

Figure (III.10): Failures' database record structure

III.4.2.2 Multi-Unification Cases

If the tests, mentioned above in section III.4.2, are satisfied, then we say that we have two coherent terms and that the multi-unification operation between these two terms is a coherent operation. In this case, the system proceeds to the actual unification between these two terms. We enumerate the different cases handled by the multi-unification algorithm. The following program segment states the different cases of both terms:

```
multi_unify(X,Y):-
  atomic(X),atomic(Y),!,
  ( X == Y -> true; signal_failure).

multi_unify(X,Y):-
  var(X), var(Y), !, X=Y.

multi_unify(X,Y):-
  var(X), nonvar(Y), !, X = Y.

multi-unify(X,Y):-
  var(Y), nonvar(X), !, Y=X.

multi_unify(X,Y):-
  nonvar(Y), Y = {Choice_point_y, Instantiation_pairs_y},!,
  varsin(X,Vx),
  findall(Vx,
            multi_unify_mono_multi(X,Choice_point_y,Instantiation_pairs_y),
            List_of_solutions),
  assign(Vx,List_of_solutions).

multi_unify(X,Y):-
  nonvar(X), X = {Choice_point_x,Instantiation_pairs_x},!,
  varsin(Y,Vy),
  findall(Vy,
            multi_unify_mono_multi(Y,Choice_point_x,Instantiation_pairs_x),
            List_of_solutions),
  assign(Vy,List_of_solutions).

multi_unify(X,Y):-
  nonvar(X), functor(X,Fx,Ax),
  nonvar(Y), functor(Y,Fy,Ay),!,
  ( (Fx==Fy),(Ax==Ay)->multi_unify_arguments(Ax,X,Y);
                        signal_failure, fail ).

multi_unify_mono_multi(X,Choice_point_y,[(Date1,Value1)|Rest]):-
   check_coherency(..,...,...)
   check_failures(..,...,...),
  multi_unify(X,Value1).

multi_unify_mono_multi(X,Choice_point_y,[(Date1,Value1)|Rest]):-
  multi_unify_mono_multi(X,Choice_point_y,Rest).

multi_unify_arguments(0,_,_).
multi_unify_arguments(A,X,Y):-
  arg(A,X,Xn), arg(A,Y,Yn), multi_unify(Xn,Yn),
  A1 is A - 1, multi_unify_arguments(A1,X,Y).
```

Program (III.2): The Multi-unification Algorithm

We enumerate the different cases of X and Y, keeping in mind that either X or Y, each maintains a unique date-path.

1- X = atom , Y = atom:

The treatment in this case is similar to that in the standard unification algorithm. An equality test takes place between X and Y. If the result of this test is true, then the multi-resolution continues normally. Alternatively, if the result is a failure, here a failure is signaled between both terms. This failure is stored in the failures database.

Example:

Assume that is required to multi-unify the two atoms *a* and *b*, with their corresponding date-paths; *d1* and *d2* at the branch-dated 7 in the choice-point-number3. The result of this test is a failure. This failure is recorded in the failures database as the following entry:

d1	d2	(3, 7)

Figure (III.11): Failures' database

This data is stored temporarily until further processing, and the multi-unification reports a failure.

Note:

In case of a single clause head, and a partial success/failure occurred, then the failing-branch-location is the branch-location of the predecessor branch of this clause head.

2- X = Variable , Y = Variable:

As in the standard unification algorithm, the two terms unify, with the result $X = Y$.

3- X = Variable , Y = Nonvariable:

By nonvariable, we mean any multi-term. Here, a direct assignment operation takes place. The nonvariable Y is assigned to X. We say that $X = Y$.

In case of *Y* is a multi-instantiation, we point out that we do not explore the multi-instantiation, i.e. we do not construct the date-paths for the respective sub-terms, and hence the coherency tests are not performed. This is the main reason why we have incoherent sub-terms in a multi-instantiation.

But at the same time, the fact that we assign the multi-instantiation to a variable in one step is the power of the algorithm. This is because a remarkable execution time is reduced in this step. The case of multi-unifying a variable to a multi-instantiation is commonly occuring in Prolog programs. Thus, we expect that an enhancement in the execution takes place when the multi-resolution model is adopted compared to the standard model. Chapter 6 is dedicated to a full discussion of the speedups attained when running several benchmarks.

Example:

For the operation multi-unify(X,{1,[(1,a) , (2,b) , (3,c)]}), the variable *X* will be equal to these three alternatives in one assignment operation, (without accessing the different sub-terms of the multi-instantiation) instead of 3 assignment operations when considering the standard resolution. The result is

$$X = \{1,[\ (1,a)\ ,\ (2,b)\ ,\ (3,c)]\}$$

4- X = Nonvariable , Y = Variable:

This is the reverse case. The result is that *Y* = *X*, irrespective of the contents of *X*.

5- X is a non variable , Y is multi-instantiated:

$Y = \{Choice\_point_Y\ ,\ [\ (date_1,value_1),...(date_n,value_n)]\}$:

Multi-unifying a multi-term, *X*, to a multi-instantiation, *Y* (containing *n* sub-terms), results in a single multi-unification operation where inherently a multi-unification operation takes place between this multi-term and each sub-term of the multi-instantiation. These multi-unification operations are invoked sequentially, in the order of the multi-instantiations.The algorithm in this case is given by:

For i = 1 to n

multi-unify (X,Y_i).

Naturally, any of these invoked multi-unifications might resemble any of the above mentioned cases.

Example:

multi-unify(x,{1,[(1,a) , (2,x)]}) will result in 2 inherent multi-unification operations:

multi-unify(x,{1,[(1,a) , (2,x)]})

|_____| -------> multi-unify(x,a), then

|_____| -------> multi-unify(x,x).

6- X = multi-instantiated , Y is a non variable:

The reverse operation of the above case takes place.

7- Optimisations:

The previous two cases (5 and 6) are the cases of multi-unifying a multi-instantiation to a multi-term. Since a multi-term may be a multi-instantiation (definition 2.2, page II.4), frequently the multi-unification performs the multi-unification operation between two multi-instantiations. Widely speaking, there are two cases, the case when both multi-instantiations belong to the same choice point (same choice-point-number), and the case when they belong to different choice points (different choice-point-numbers). The algorithm as shown in program (III.2), handles these cases, but for clarity reasons we detail these two cases as optimisations:

7.a - X is multi-instantiated , Y is multi-instantiated:

$X = \{Choice\_point_X , [(date_1,value_1),...(date_n,value_n)]\}$:
$Y = \{Choice\_point_Y , [(date_1,value_1),...(date_m,value_m)]\}$:

This is the case of the multi-unifying 2 multi-instantiations originating from two different choice points; $Choice\_point_X$ and $Choice\_point_Y$. It is the most complex case. It is as if we are trying to multi-unify two sub-trees, with different alternatives.

Here, the multi-unification algorithm invokes multi-unification operations between all the combinations of the sub-terms in both multi-instantiations, i.e. a multi-unification between the first sub-term of *X* with *Y*, then between the second sub-term of *X* with *Y*, and so on. Each of such multi-unification operations resembles any of the cases mentioned above.

The difference between explicitly mentioning this case in the multi-unification algorithm or not is the order by which the multi-unification takes place. Leaving the above cases to handle the multi-unification, no control could be super imposed on the order by which the loops representing the multi-instantiations are multi-unified. Alternatively, here, we can state explicitly which loop starts before which according to a certain condition (superiority, inferiority).

Ordering of loops in the multi-unification operation is a technical aspect, that concerns the optimisation of the treatment of failures. Same results are produced in both cases. The algorithm as shown in program (III.2) will not allow any control to perform the loops in a certain order. Stating the cases of multi-instantiations explicitly in the multi-unification algorithm will allow to these operations to be performed in a certain order (ascending or descending). We will return to this point again when discussing failures in chapter 4.

Example:

Given two multi-instantiated variables *X* and *Y* such that:

X = { **1,** [(1,a), (2,B)]} and
Y = { **3,** [(5,a), (6,bb)]}

it is required to multi-unify(X,Y).

Since both variables are multi-instantiated, the algorithm examines their choice-point-numbers. Since they are different, then this means that they belong to two different choice points. This is because the numbering of the choice point is distinct; each choice point is given a unique number to distinguish it from other choice points.

A number of multi-unification operations are invoked between each sub-term of *X* with all sub-terms of *Y* as follows:

multi-unify({ **1,** [(1,a), (2,B)]},{ **3,** [(5,a), (6,bb)]})

|_____| ---> multi-unify(a,{ **3,** [(5,a), (6,bb)]}) then
|_____| ---> multi-unify(B,{ **3,** [(5,a), (6,bb)]}).

We remind the reader that the date-path of the first term, *a*, in the first invoked multi-unification is [(**1**,1)] while that of *B*, in the second multi-unification operation is [(**1**,2)].

Each of the invoked multi-unifications resembles the previous case. The algorithm handles each of them separately, and in order. The first invoked operation is the first to be treated, resulting in the following invoked multi-unifications:

multi-unify(a,{ **3,** [(5,a), (6,bb)]})

|_____| ---> multi-unify(a,a) then

|_____| ---> multi-unify(a,bb).

The first multi-unification results in a success, while the second results in a partial failure. This failure is recorded in the failures database in terms of the date-paths of both terms:

[(1, 1)]	[(3, 6)]	current branch-location

Figure (III.12): Failures' database

The algorithm then handles the second sub-term of *X*, *B*. The same operation is repeated where a multi-unification operation is invoked between *B* and all the sub-terms of *Y*.

multi-unify(B,{ **3,** [(5,a), (6,bb)]})

This is the case of multi-unifying a variable to a nonvariable. A direct assignment takes place and we say that B = { **3,** [(5,a), (6,bb)]}).

By this the multi-unification between the multi-instantiated multi-terms X and Y is accomplished.

7.b- X is multi-instantiated, Y is multi-instantiated:

X = {Choice_point , [($date_1$,$value_1$),..]}:
Y = {Choice_point, [($date_1$,$value_1$),..]}:

This is the case of multi-unifying two multi-instantiated variables, belonging to the same choice point. Here, the corresponding pairs of sub-terms having the same-branch-dates are multi-unified. In such a case, a multi-unification operation is invoked between every correponding pair of terms in *X* and *Y*. No combinatorial tests are allowed.

Every two corresponding terms are called by the multi-unification algorithm with their corresponding date-paths. Each multi-unification resembles any multi-unification case stated above, with the same procedure of manipulation of dates.

Example:

multi-unify({1,[(1,a) , (2,b)]},{1,[(1,m), (2,b)]}),

The above multi-unification operation between the two multi-instantiated variables will result in two inherent multi-unifications.

multi-unify({1,[(1,a) , (2,b)]},{1,[(1,m), (2,b)]}),
|_____| ---> multi-unify(a,m) then
|_____| ---> multi-unify(b,b).

The first multi-unification operation between the sub-term *a* of the first multi-term and the sub-term *m* of the second multi-term will result in a partial success/failure.

Note:

An important characteristic which we emphasize is that in the case of two multi-instantiations belonging to the same choice point, this does not necessarily imply that both multi-instantiations have the same length (same number of sub-terms constituting each multi-instantiation).

Example:

Given,

p1(X,A):- X=[A/a].
p2(X,B):- X=[b/B].

and the query,

?- (X = [a/a] ; X = [b/b]), p1(X , Y), p2(X , Z), Y = Z.

The variables *X*, *Y* and *Z* will be multi-instantiated as follows:

X = { **1** , [(1,[a|a]) , (2,[b|b])] },
Y = { **1** , [(1,a)] }, and
Z = { **1** , [(2,b)] }

This example is the case of the multi-instantiation that includes a single instantiation.

Now we come to the test *Y = Z*. Both variables are multi-instantiated, with the same choice point number, **1**, but with different branch-dates. It is prohibited to perform the test between *a* and *b* due to the fact that these two values are generated from two different alternatives, 1 and 2, of the same choice point, **1**. This test violates the semantics of standard Prolog resolution where one and only one alternative of each choice point is considered during the whole resolution. Hence, the result of this multi-unification operation is a complete failure.

Another remark that we would like to add is that this case of multi-instantiations belonging to the same choice point is also handled by the coherency test procedure that ensures that the date-paths of both multi-terms results in a permitted coherent multi-unification operation.

9- X and Y are functors:

The last case is the multi-unification between two compound terms, namely functors. The test is similar to that in standard unification, where a test occurs to assure that the two functors' names are identical, and that the number of arguments in both terms is the same. If both conditions are satisfied, then multi-unification operations are invoked between every two corresponding arguments. These operations could be any of the above mentioned cases.

Note that the multi-unification operations between the different pairs of arguments are independent. No coherency is checked between the different arguments of a clause head, even if these arguments are (or include) multi-instantiations.

Example 1:

multi-unify(p(X), p({**1**,[(1,a) , (2,b)]}).

Since both functors are the same and include only one argument, then the result is the invokation of the operation:

multi-unify(X,{**1**,[(1,a) , (2,b)]})

which, in turn, will multi-instantiate X to {**1**,[(1,a) , (2,b)]}.

Example 2:

multi-unify(f1({**1**,[(1,a) , (2,b)]}),f2({**1**,[(1,a) , (2,B)]})).

The above operation will fail before invoking any multi-unifications due to the difference in the functors' names. The failure took place before exploring the different sub-terms of the multi-instantiated arguments.

III.4.2.3 A general example

multi_unify (m,{**3**,[(6,{**1**, [(2,A) , (3,{ **2**, [(4,a) , (5,b)]})]}) , (7,m)]}).

This is a multi-unification operation between *m* and a multi-instantiation. The latter includes two multi-terms; a multi-instantiation and a single instantiation (mono-instantiation), tagged by the branch-dates 6 and 7 respectively. Hence the date-path of the multi-instantiation is [(**3**,6)], while that of the mono instantiation is [(**3**,7)].

multi_unify (m,{**3**,[(6,{**1**, [(2,A) , (3,{ **2**, [(4,a) , (5,b)]})]}) , (7,m)]}).

The result is the invokation of 2 multi-unification operations as follows:

multi-unify(m,{**1**, [(2,A) , (3,{ **2**, [(4,a) , (5,b)]})]}), and
multi-unify(m,m).

The second multi-unification is simple, and will lead to a success.

The first invoked multi-unification operation is between *m* and another multi-instantiation, which in turn will invoke two other multi-unifications between *m* and each sub-term of this multi-instantiation.

multi-unify(m,{**1,** [(2,A) , (3,{ **2,** [(4,a) , (5,b)]})]}), and

resulting in the following multi-unification operations:

multi-unify(m,A), and
multi-unify(m,{ **2,** [(4,a) , (5,b)]})]}), and

where the variable *A* has a date-path = [(**3**,6),(**1**,2)] and the multi-instantiation has a date-path = [(**3**,6),(**1**,3)].

The first multi-unification operation is the case when one term is a variable and the other is a non variable. Here, a direct assignment takes place and *A* will be equal to *m*.

On the other hand, the multi-unification operation is between the atom *m* and the multi-instantiation { **2,** [(4,a) , (5,b)]})]}. This will lead to two other multi-unification operations between *m* and each sub-term of this multi-instantiation as follows:

multi-unify(m,{ **2,** [(4,a) , (5,b)]})

resulting in:

multi-unify(m,a),
multi-unify(m,b).

The date-path of the sub-term *a* is [(**3**,6), (**1**,3), (**2**,4)] while that of *b* is [(**3**,6), (**1**,3), (**2**,5)].

The first multi-unification operation will fail. The failure is stored temporarily in the failures database as follows:

[]	[(3, 6),(1, 3),(2, 4)]	...

Figure (III.13): Failures' database

The first date-path is [] as the first multi-term is a ground term.

Now, attempting to multi-unify the atom *m* to the mono instantiation *b*, another failure is detected, that is stored in the failures database.

[]	[(3, 6),(1, 3),(2, 4)]	...
[]	[(3, 6),(1, 3),(2, 5)]	...

Figure (III.14): Failures' database

The last multi-instantiation led to a total-failure as none of the invoked multi-unification operations led to a success (the two branches of the choice point number **2** have failed). This implies a failure of the branch that invoked this choice point. In this case, the choice-point-number **2** is fully omitted and we record a partial failure of the instantiation whose date-path is [(**3**,6),(**1**,3)]. Hence the above records in the failures database are modified to:

[]	[(3, 6),(1, 3)]	...

Figure (III.15): Failures' database

III.4.2.4 Coherency of a sub-term

As previously mentioned, while constructing a date-path, each time a new sub-term is encountered, before appending its new branch-location to the already constructed parts of the date-path, several tests are performed to assure the coherency of the constructed date-path until this moment.

A test is made to check if the choice-point-number of the branch-location to be appended already exists in the constructed part of the date-path. If this is true, then a test is made to make sure that both branch-locations: the one already existing and the one to be appended, belong to the same alternative, i.e. have the same branch-dates. This is to ensure, by

analogy to the standard resolution, that we are traversing the same solution path and not considering 2 alternatives of the same choice point.

The following Prolog program performs the above test:

```
check_coherency_1(Choice_point_number,Branch_date,Path):-
   on((Choice_point_number,Date),Path),!,
   Date = Branch_date.
```

Program (III.3): Object coherency test

It is clear that with the new representation of multi-instantiations (definition 3.1), it is quite evident how simple a coherency test may be performed.

In case of incoherency, a failure is reported and the multi-unification operation that was handling this incoherent sub-term is abandonned.

Example 1:

Given a variable *X* such as

$$X=\{1,[\ (1,a)\ ,\ (\ 2,\ \{2,[\ (\ 3,x)\ ,\ (4,\{1,[(1,c)\ ,\ (2,d)]\})]\})]\}$$

to be multi-unified to another multi-term, say *Y*, each sub-term in *X* will be examined before performing this operation. By examining a sub-term, we mean that we create its date-path and test its coherency.

The following table enumerates the corresponding date-paths for the different sub-terms of *X*:

Object	Date_path
a	[(1,1)]
{2,[(3,x),(4,{1,[(1,c),(2,d)]})]}	[(1,2)]
x	[(1,2),(2,3)]
{1,[(1,c) , (2,d)]}	[(1,2),(2,4)]
c	[(1,2),(2,4),(1,1)]
d	[(1,2),(2,4),(1,2)]

Table (III.2): Objects and their corresponding date-paths

In the above table, when we observe the different date-paths of the different instantiations, we find that they all respect the rules defining a date-path, except for that corresponding to the instantiation *c*. Its date-path includes 2 branch-locations belonging to the same choice point; i.e. the same choice-point-number mentioned twice with two different branch-dates. This violates the rule that only one branch-date of each choice point should be considered at a time.

Given the representation of the multi-instantiations, we proved in section III.3.5.1 that it is sufficient to indicate the incoherency of any sub-term. Note that the test of coherency takes place before actually performing any operation. In case of incoherency, this operand is completely omitted and the multi-resolution continues with the following sub-term.

III.4.2.5 Coherency of a multi-unification operation

A multi-unification operation requires two multi-terms as its inputs. To ensure the coherency of the performed operation, we have to ensure the coherency of the two multi-terms. To check the coherency between these two multi-instantiations, we have to check the coherency of the date-path under construction with respect to the second-date-path.

Appending a new branch-location to the date-path under construction, the system checks whether a branch-location belonging to the same choice point exists in the already constructed path and the date-path of the other term or not. If this is true, then we should validate that the branch-dates of the three branch-locations: those in both paths and that to be appended, have the same branch-date. If this condition is not satisfied, then an incoherency is signaled and a failure is reported.

If the appended branch-location does not exist on the already constructed path, but exists in the date-path of the second term, then, again, the algorithm should make sure that the branch-dates are equal otherwise it is an invalid operation for the same reason mentioned above; i.e. those two terms cannot be unified as they belong to different solution paths.

The above tests validating the coherency of sub-terms and operations are performed by the following Prolog program:

```
check_coherency_2(Choice_point_number,Branch_date,First_path,Second_path):-
    check_coherency_1(Choice_point_number,Branch_date,First_path),
    check_coherency_1(Choice_point_number,Branch_date,Second_path).
```

Program(III.4): Operations coherency tests

where check_coherency_1 is the program (III.2).

We present a detailed example, where we show how date-paths are constructed and coherency checks take place.

Example 1:

For the query,

(A = [a | a] ; A = [b | b]) , A = [B | C] , A = D , (Z = [B | C] ; Z = D) , Z = A.

the different variables will be multi-instantiated to:

$$A = \{\ 1\ ,\ [\ (\underline{1},\ [a|a]\ )\ ,\ (\underline{2},\ [b|b]\ )\ ]\ \}$$

$$B = \{\ 1\ ,\ [\ (\underline{1},\ a\ )\ ,\ (\underline{2},\ b\ )\ ]\ \}$$

$$C = \{\ 1\ ,\ [\ (\underline{1},\ [a]\ )\ ,\ (\underline{2},\ [b]\ )\ ]\ \}$$

$$D = \{\ 1\ ,\ [\ (\underline{1},\ [a|a]\ )\ ,\ (\underline{2},\ [b|b]\ )\ ]\ \}$$

$$Z = \{\ 2,\ [(\underline{5},\ [\ \{\ 1,\ [\ (\underline{1},a)\ ,\ (\underline{2},b)\ ]\}\ |\ \{\ 1\ ,\ [\ (\underline{1},[a])\ ,\ (\underline{2},[b])\ ]\}\ ])\ ,$$
$$(\underline{6},\ \{\ 1,\ [\ (\underline{1},[a|a])\ ,\ (\underline{2},[b|b])\ ]\})]\}$$

When coming to test the equality between *Z* and *A*, the sub-terms are accessed by their date-paths. We compare the date-paths of each pair of operands. If no coherency test are performed then the 16 operations given in the following table. It is here where we can detect any incoherency between the operands. Once an incoherency is detected in the date-path of either terms, this operation will be totally omitted, and control proceeds with the following combination. We enumerate the coherent cases:

T1	Date path1	T2	Date path2	Results
a	[(1,1)]	a	[(2,5),(1,1)]	coherent
[a]	[(1,1)]	[a]	[(2,5),(1,1)]	coherent
a	[(1,1)]	a	[(2,6),(1,1)]	coherent
[a]	[(1,1)]	[a]	[(2,6),(1,1)]	coherent
b	[(1,2)]	b	[(2,5),(1,2)]	coherent
[b]	[(1,2)]	[b]	[(2,5),(1,2)]	coherent
b	[(1,2)]	b	[(2,6),(1,2)]	coherent
[b]	[(1,2)]	[b]	[(2,6),(1,2)]	coherent

Table (III. 3): Testing date-paths for coherency of operations

The algorithm checks the date-paths of both multi-terms involved in the same multi-unification operation. It should validate that the T1 is a coherent sub-term(the previous test), and that T2 is a coherent sub-term, and that T1 and T2 are coherent to be involved in the same operation. This last test is done by validating that all branch-locations encountered for both terms are coherent, i.e. if the same choice point is encountered in both date-paths, then their branch-dates should be the same, otherwise an incoherency is signaled, and both pairs are neglected without performing this false operation. The algorithm continues to test the remaining pairs of terms in the same fashion.

To ensure the correctness of the performed test operations, we present the standard resolution tree of the above query, fig. (III.16).

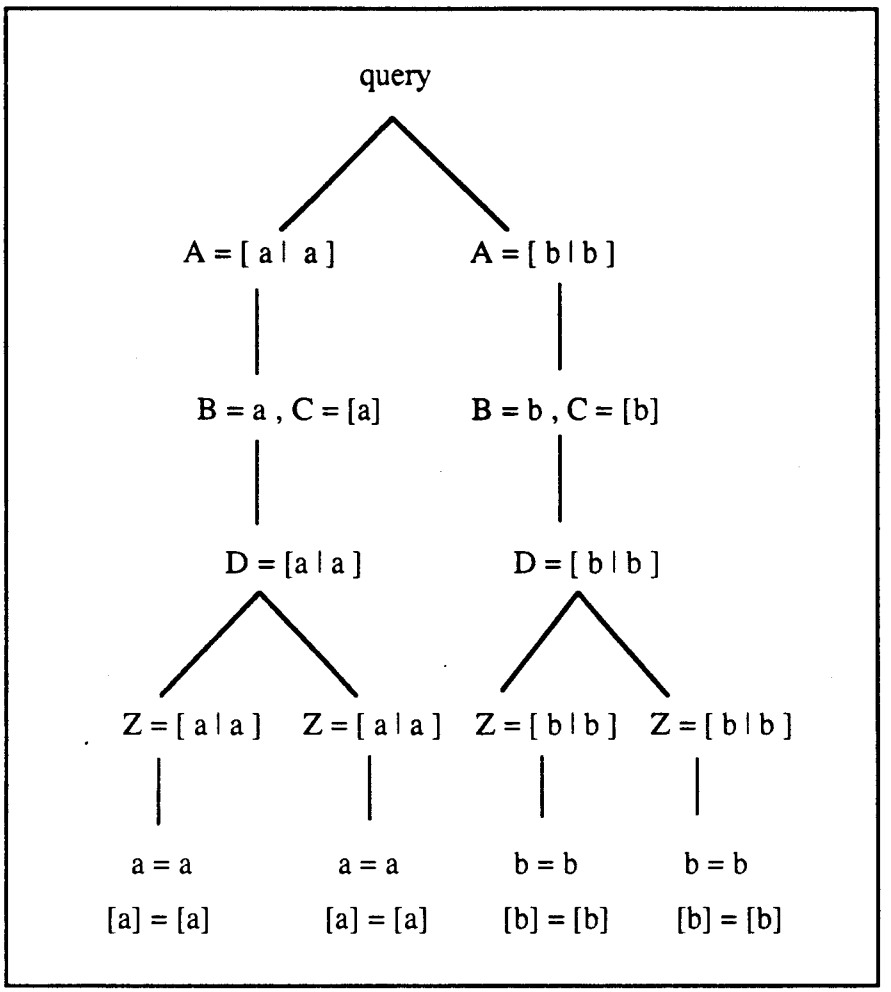


Figure (III.16): Standard resolution tree

We find that the number of performed unification operations are only 8 operations, which are the same as those performed in the multi-resolution model. By this, we conclude the soundness of the representation of the multi-instantiated variables from which incoherency of sub-terms and operations could be easily detected (problem 2.2 (page II.14) is resolved).

III.4.2.6 Partial success/failures in the same multi-unification operation

While constructing the date-path of a sub-term, the algorithm ensures that no partial success/failures have occured between this date-path to be constructed and the date-path of the second term in this same multi-unification operation (problem 2.4, page III.25). If a failure is encountered in the failures database, then this unification operation is omitted as it is a waste of time to multi-unify two terms that already proved to be failing terms.

This encountered failure might have occurred during the same multi-unification operation, but with previous arguments, or during another multi-unification operation that took place before the current operation. We have proposed a treatment of the first type of encountered failures.

The following Prolog program treats this case:

```
check_failure(Branch_location,First_path,Second_path):-
(failures([Branch_location|First_path],Second_path,Current_branch_location);
 failures(Second_path,[Branch_location|First_path],Current_branch_location) )
                ->fail;      true).
```

Program (III.5): Treatment of failures while multi-unifying two sub-terms

This program checks the occurrence of a failure that includes the given branch-location in the failures database. If this is the case, then a failure is reported and the multi-unification operation is dropped, whereas if the database does not include this entry, then it performs the multi-unification operation between the corresponding two sub-terms.

Example:

Given the program,

p(a). *p(b).*

q(a,b). *q(b,c).*

solve the following query,

:- p(X), q(X,X).

After attempting the first subgoal, the variable *X* will be multi-instantiated to:

X = { **1** , [(1,a), (2,b)] }

To solve *q(X,X)*, we attempt the clause head *q(a,b)*. Multi-unifying the first multi-term of the subgoal to the first multi-term of the clause head, the first sub-term in *X* will result in a success, while the rest will result in partial success/failures. The current choice point number

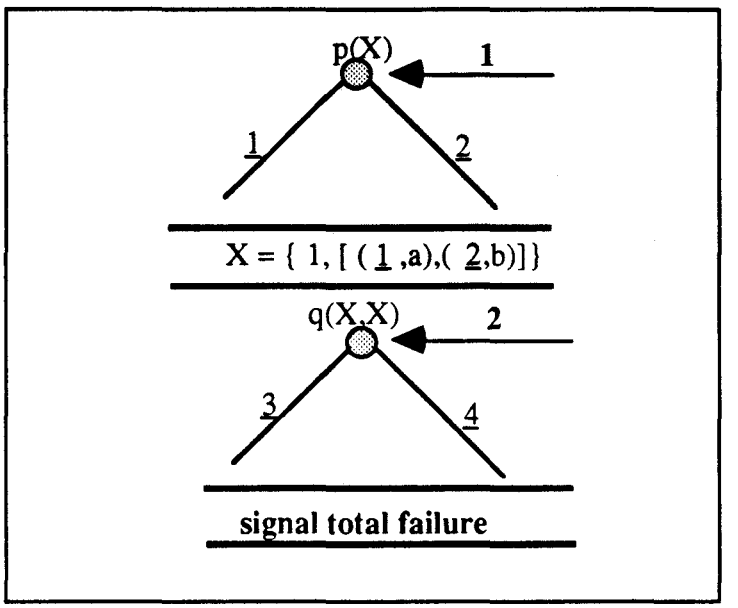


Figure (III.17): Multi-resolution tree

is **2**, and the current branch-location is 3. This partial success/failure are stored in the failures database as follows:

[(1, 2)]	[]	(2, 3)

Figure (III.18): Failures' database

Now, coming to multi-unify the second multi-term in the subgoal, which is *X*, and the second multi-term in the clause head, always in the same multi-unification operation, the algorithm attempts each sub-term instantiated to *X* to the atom *b*. Starting with*a*, by checking the failures database, the system does not recognize any previous failures, but multi-unifying a to b results in a failure that will be appended to the above mentioned records.

[(1, 2)]	[]	(2, 3)
[(1, 1)]	[]	(2, 3)

Figure (3.19) : Failures' database

Now, the algorithm considers the following sub-term, which is *b*. Its date-path is [(1,2)]. By scanning the failures database, the system recognises that the date-paths of the sub-terms to be multi-unified to *b* have already caused a failure. Accordingly, this alternative is abandonned and the multi-unification resumes the operation with the following sub-terms.

In the same manner, the system detects that all the other sub-terms of *X* will fail due to the same reason, hence reporting a total-failure of the multi-unification of the subgoal to this clause head. The same phenomenon is repeated with all the other clause heads resulting in a total-failure of the subgoal *q*.

By this the problem (2.4) is resolved.

III.4.2.7 Recapitulation of failures in the multi-unification

Let us now recapitulate the different locations where a total-failure might result in the multi-unification of any two multi-terms:

1- atom / atom,
2- functor / functor,
3- incoherent sub-term (problem 2.1),
4- incoherent multi-unification between two sub-terms of two multi-instantiations problem (2.2),
5- total-failure of all sub-terms of a multi-instantiation when being multi-unified to another multi-term, and
6- result of double checking with failures database due to a previous partial success/failure during the same multi-unification operation (problem 2.4).

These are actually the different cases where the multi-unification algorithm results in a total-failure. In the first two cases, partial success/failures are stored in the failures database in terms of the date-paths of both sub-terms that caused the failure, together with the failing branch-location where this failure took place. In the rest of the cases, a general failure is reported to the system and the algorithm abandons the current multi-unification operation that caused this failure.

III.5 The Multi-Resolution Tree

Before terminating the discussion about the multi-resolution phase, we present the multi-resolution tree. It was graphically presented in all the discussed examples, but we did not approach the algorithm that creates it.

In the multi-execution phase, we traversed completely the multi-resolution tree resolving the given query. The multi-resolution tree is a tree representing the different succeeding traversed choice points, and the different succeeding branches representing the succeeding alternatives for each subgoal.

As previously stated, the multi-resolution tree differs from the standard OR resolution tree in the same way as the multi-resolution model differs from the standard resolution. Each choice point is represented only once, as deep backtracking is eliminated. No solution paths are visible on the multi-resolution tree. In the previous chapter, we presented several figures demonstrating a comparison between the traversed multi-resolution tree with respect to its equivalent standard resolution tree. Other figures throughout this chapter explained in detail the concept of the multi-resolution tree.

What we want to emphasize is that the multi-resolution tree is created dynamically during the multi-execution of the query. By the termination of the multi-execution phase, we may consider the multi-resolution tree as a static tree.

In our model, each time a choice point is completely investigated, a new level in the multi-resolution tree is appended to the already existing part of the tree. This takes place in the local-synchronous-OR-level of this choice point.

The multi-resolution tree is represented in terms of a list including father-son pairs.

Definition 3.7: A father-son pair

Each father-son pair is defined as:

([list of predecessor branche-dates] , [list of successor branche-dates])

where,

list of predecessor branche-dates are the branch-dates of all the succeeding branches belonging the same choice point. If only one branch suceeded, then it includes only one date. The same concept applies to the **list of successor branche-dates.**

Example:

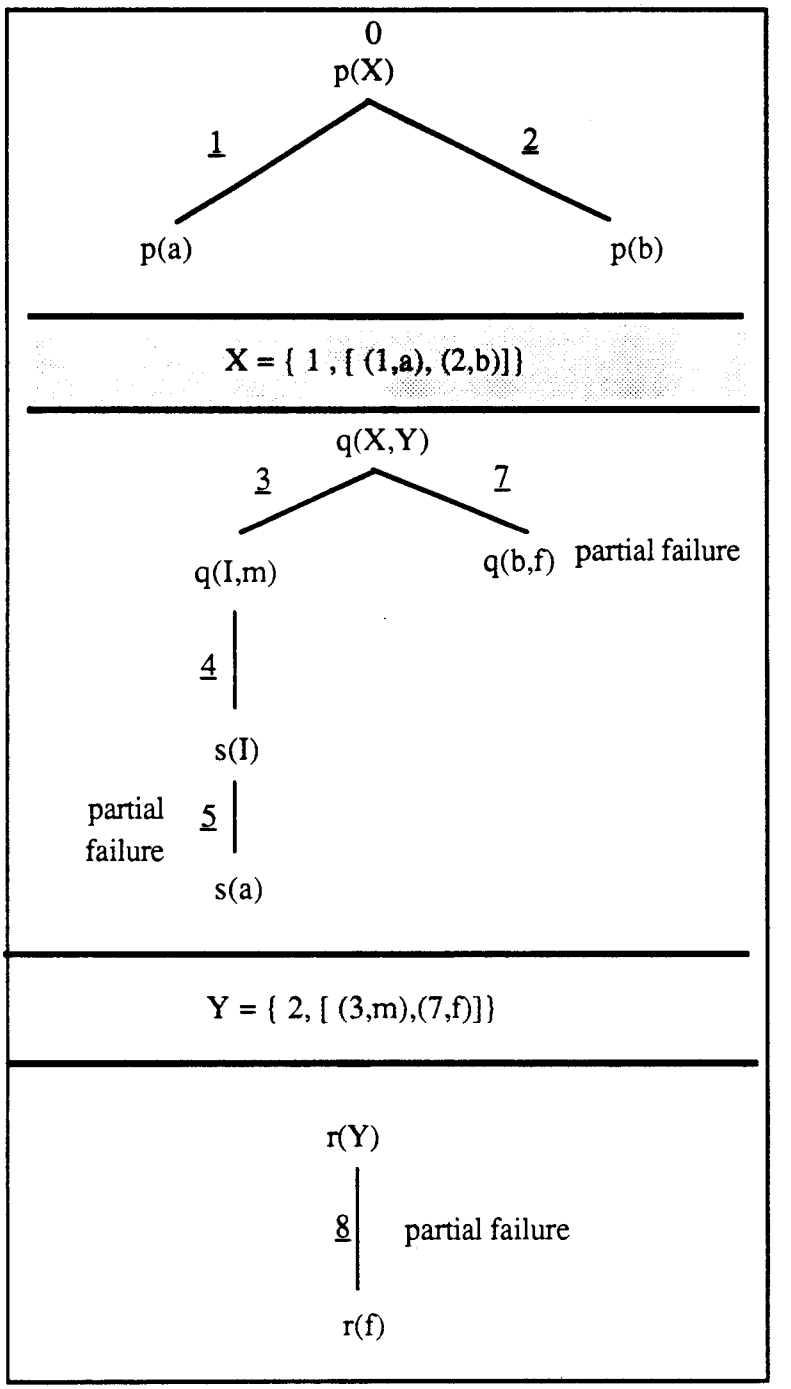
We recall the example mentioned in section II.3.4.1 when we discussed partial success/failures.

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

q(I,m):- s(I).
q(b,f).

s(a).    s(z).

r(f).
```

with the query,

```
:- p(X), q(X,Y), r(Y).
```

The multi-resolution tree is represented as follows:



Figure(III.20): The multi-resolution tree

We explain what actually happens to create this data structure:

- The multi-execution starts from the root level. The first subgoal is *p(X)*. A corresponding choice point exists, hence it is given a unique number, **1**. Both alternatives succeed, with dates 1 and 2 respectively.

- At the synchronous OR level of that choice point, the branch-dates of the succeeding alternatives are assembled in a list.

- The system recalls the predecessor of this choice point, which is the root level, denoted by 0.

- The first entry in the multi-resolution tree representing the first examined level is the father-son pair:

$$\text{Tree} = [([0] , [1,2])].$$

- The next subgoal to be investigated is *q(X,Y)*. A corresponding choice point exists, which is identified by the choice point number **2**. Multi-unification with the first alternative (branch-date 3) results in a complete success: *I* is multi-instantiated to {1,[(1,a),(2,b)]} and a first solution of *Y* is (3,m). The body clause of this alternative is invoked that tests *s({1,[(1,a),(2,b)]})*.

The first alternative for the subgoal s (branch-date 4) will result in a partial success/failure, and the second brnach (dated 5) will lead to a total failure. Since there are no more alternatives, then comes the role of the local synchronous OR phase for this choice point, *s*, where all the branch dates of the current choice point are assembled to add a new entry in the tree data structure. The only succeeding branch is 4, that was invoked by its predecessor 3, hence the new tree information is:

$$\text{Tree} = [([0], [1,2]), ([3], [4])]$$

The body clause of the first alternative has been totally explored. Now, the multi-resolution returns to the second alternative of *q*. The current branch-date is 6 (because 5 was the date of the branch that led to the total failure). The multi-unification results in a partial success/failure that will be stored in the failures' database.

All the alternatives of the subgoal *q* have been attempted, and hence the local synchronous OR phase will assemble the different solutions to multi-instantiate *Y* to {2,[(3,m),(6,f)]}. All branch-dates of this choice point are assembled [3,6], the predecessor of which is the

previous choice point, having the branch-dates [1,2]. Accordingly, a new father-son pair is added to the tree.

Tree = [([0], [1,2]), ([3], [4]), ([1,2], [3,6])]

The last subgoal to multi-resolve is *r(Y)*. This is a single clause having the branch-date 7, that will lead to a partial success/failure. Its predecessor is the choice point, having the ensemble of branch-dates [3,6]. At this stage, the multi-resolution tree is given by:

Tree = [([0], [1,2]), ([3] , [4]), ([1,2], [3,6]), ([3,6],[7])]

Checking with fig.(III.20), this representation fits totally to the given graph.

In the above representation, we observe that the branch 3 is a predecessor to the branch 4 and the branch 7, though they are not included in the same term. This is to indicate that if a branch has 2 successors, then one is its body clause, while the second is the following subgoal. We differentiated between the two cases by their corresponding branch-dates. The date 4 is less than the date 7 indicating that 4 was encountered before 7. We deduce that 4 is the body clause while 7 is the next subgoal, as we decided from the very beginning that we multi-execute the body clauses of a clause head before proceeding either with the next clause head or the next subgoal.

This data structure will be used together with failures database to reconstruct the standard resolution tree. The details of this algorithm is given in the chapter 5 which is dedicated to a discussion on predefined predicates.

III.6 Conclusion

This chapter was dedicated to a deep study of the multi-execution which is the first phase of the multi-resolution model. We explained how and where multi-instantiations were created. We discussed the multi-unification algorithm and compared it to the standard unification algorithm. We will be comparing their complexities in chapter 6. We also showed how this model was able to resolve all the problems mentioned in the previous chapter.

Chapter Four

Treatment of Failures

Abstract

This chapter considers the multi-resolution from the failures' point of view. We explain the difference between the failures encountered in the standard resolution and those in the multi-resolution. We present the different treatments proposed for the different types of failures. We demonstrate how the failures' database is a dynamic database throughout the multi-execution phase ready to be processed statically in the multi-outputs phase to display the ensemble of solutions.

IV.1 Introduction

A logic program is a program, including facts and/or relations and a query, which is to be satisfied. Answering a query with respect to a program is actually determining whether the query is a logical consequence of the program. The query is attempted to be unified to the different clause heads and an answer is expected. If the answer is *yes (true)*, then the query is proved. If the answer is no, then we say that a *failure* has occurred. It means that the posed query is not a logical consequence of the program. This answer does not reflect on the truth of the query; it merely says that the system failed to prove the query from the given program.

There are several reasons for the occurrence of a failure. To present a comparison of the different classes of failures that occur in the standard resolution as well as in the multi-resolution models, we will recall several definitions and phenomena already discussed in previous chapters to demonstrate how we treated each class clearly.

IV.2 Classes of Failures

We can widely classify the failures occuring during the execution of a Prolog program into two classes; the explicit failures and the implicit failures. We detail each class as follows:

IV.2.1 Explicit Failures

An explicit failure is a failure due to an explicit *fail* stated in the body of a clause (or goal). The impact of an explicit fail is the same in either models; the standard resolution model or the multi-resolution model. When such type of a failure is encountered, an immediate failure is signaled to the system. All variables instantiated in this alternative (or goal) will be deinstantiated and we say that the clause (or goal) has failed.

Example:

For the program,

p. *q.*
.. ..
p. *q.*

r.

solve the query :

:- p,q,fail,r,..

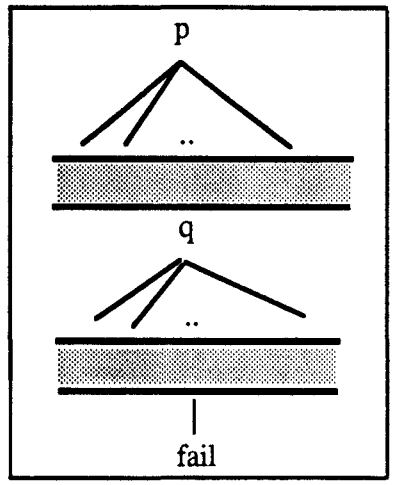


Figure (IV.1):
Explicit failure in multi-resolution

Assuming that both *p* and *q* result in success, the query fails after solving *q*, and *r* will never be attempted.

IV.2.2 Implicit Failures

In the standard resolution model, implicit failures may occur while unifying two terms. Examining closely the different unification operations, we find that an implicit failure occurs if:

- 2 different atoms are to be unified, or
- 2 different functors (different functor names and/or different arities) are to be unified.

When such a failure is encountered, the alternative where this failure took place is abandonned. Variables already instantiated during this unification operation are deinstantiated and backtracking takes place to attempt to solve the subgoal with another clause header.

By analogy, in the multi-resolution model, an implicit failure may occur while multi-unifying two multi-terms. The multi-unification cases that might lead to an implicit failure are:

- 2 different atoms are to be multi-unified
- 2 different functors (different functor names and/or different arities)
- incoherent sub-term (problems 2.1)

- incoherent multi-unification operation (problem 2.2)
- previous failures that occured in the same multi-unification operation (problem 2.4)
- previous failures in previous multi-unification operations (problem 2.5)
- failure of all sub-terms of a multi-instantiation to be multi-unified to another multi-term.

In the following sections, we detail the different types of implicit failures in the multi-resolution model, together with the different proposed algorithms for the treatment of each type.

IV.3 Implicit failures in the multi-resolution

In the multi-resolution model, we have replaced deep backtracking by multi-instantiated variables. When attempting to solve a goal containing multi-instantiated variables, we might encounter one of two types of implicit failures; either partial success/failures or total failures.

IV.3.1 Partial success/failures

This is the case when attempting to multi-unify a multi-instantiation to any multi- term. Here, each instantiation is to be multi-unified to that multi-term, as explained in section II.3.4.1. A partial success/failure may be encountered due to any of the reasons mentioned above. It is when some sub-terms result in a success, while others, in the same multi-unification operation, result in a failure.

Problems 2.3 & 2.5 are concerned with the treatment of partial success/failures.

IV.3.2 Total Failures

In a multi-unification operation between a multi-instantiation and a multi-term, if all the sub-terms resulted in a failure, then a total-failure is reported. Basically, there are two causes for a total-failure, the direct total failures and the indirect total failures.

IV.3.2.1 Direct total failures

It is the case when attempting to multi-unify a multi-instantiation to another multi-term and that not one instantiation leads to a success (all instantiations fail). We call this a *direct total failure*. *Total* because all instantiations failed, and *direct* because the cause of this failure is

clear as the multi-unification algorithm records an immediate failure while multi-unifying these two multi-terms.

Example:

multi-unify({1, [(1,a), (2,b)]},c)

In the above example,all the sub-terms will fail when being multi-unified to *c*. This is the case of a direct-total failure.

IV.3.2.2 Indirect Total Failures

A total failure may result indirectly from an ensemble of partial failures. This is why we call this type of failure indirect-total-failure. This was previously represented in the problem 2.4. A typical example of such a failure is the following program.

Example:

p(1).
p(2).
p(3).

q(1,2).
q(2,3).
q(3,4).

solve the query,

:-p(X),q(X,X).

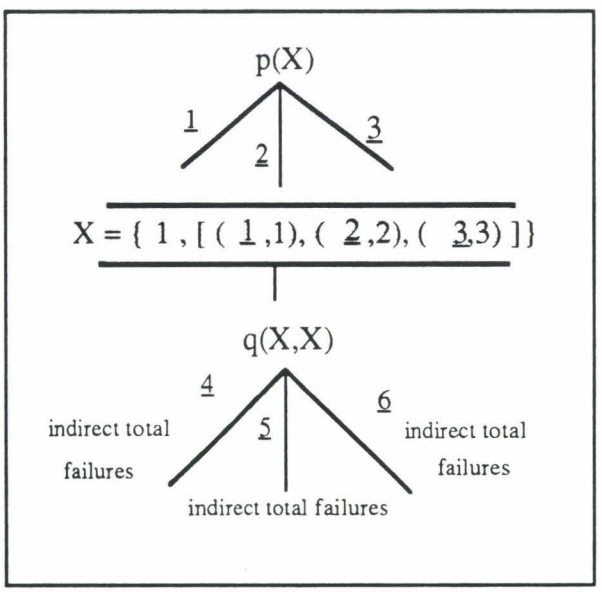


Figure (IV.2): Example on indirect total failures

This example was previously discussed in section III.4.2.6.

IV.4 Treatment of Failures

We presented the different types of failures that might be encountered in the multi-execution phase. In section II.3.4.2, we decided that memorising the occured failures will be the

solution that we are going to adopt in our model. Failures are stored in a data structure that we called the failures' database. We detail the record structure of this database together with the different operations that manipulate its entries in both phases; the multi-execution phase and the multi-outputs phase.

IV.4.1 Failures' database

Partial success/failures are stored in a failures' database. For any program, there is only one global failures' database. During the multi-execution phase, it is a dynamic data structure. When a partial success/failure occurs, it is stored in this database. The contents may be updated due to any redundancy that might exist. Once, the multi-execution phase terminates its role, the multi-outputs phase processes this data structure statically. It may be considered as a lookup table that is to be checked before the solutions are displayed to the user.

IV.4.2 Record Structure

Partial success/failures are stored in terms of the two multi-terms that caused this failure. We store the date-path for each multi-term as well as the branch-location when this partial success/failure took place.

A record representing this information is appended to the failures' database. It is represented as follows:

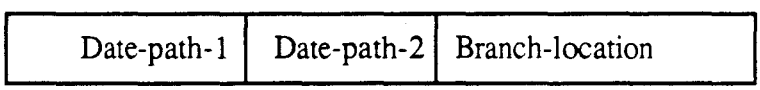


Figure (IV.3): Record structure of the failures' database

where the *branch-location* is given by the current choice-point-number and the current branch-date of the alternative where the failure took place.

Afterwards, the multi-execution continues, always with the same multi-instantiations of the variables.

IV.4.3 Representation of a partial success/failure

The class of partial success/failures that occured in multi-unification operations is a complicated type of failure to treat. In all the material that we presented until now, all partial success/failures occured during the multi-unification are due to two multi-terms that failed to

be multi-unified or as a result of an error in a multi-arithmetic operation. In multi-arithmetic operations, we either have single-operand instuctions or double-operands instructions.

Hence, we normally have two multi-terms that result in this partial success/failure. Examples of operations that might result in a failure are: *multi-unify(X,Y)*, *>(X,Y)*, *=(X,Z)*, etc.(The arithmetic operations are discussed in the following chapter). As we mentioned, this failure is stored in terms of the date-path for each multi-term in the failing operation, together with the failing branch-location.

To recognize a failing sub-term, we check the contents of the database. If its corresponding date-path is present, then it means that this sub-term has already failed and that any further processing is fruitless.

Example:

Given the multi-instantiated variables,

X = { **1,** [(1,a) , (2,b)]} , Y = { **2,** [(3,a) , (4,c)]}

it is required to solve the query,

:- ..., X=Y, write([X,Y]).

After the equality test, the following failures will be stored in the failures' database.

[(1, 1)]	[(2, 4)]	...
[(1, 2)]	[(2, 3)]	...
[(1, 2)]	[(2, 4)]	...

Figure (IV.4): Failures' database

When coming to the next subgoal, the different values of *X* and *Y* are computed. Since both multi-instantiations belong to different choice points, then normally, if no failures have occured, all combinations of the different sub-terms are permitted.

Before writing each combination, a check is made with the failures' database to validate that both sub-terms have never previously resulted in a partial success/failure. If the date-paths of

both terms are not an entry in the database, then this combination is a permitted (safe) combination. On the other hand, if a records exists, having the date-paths of both sub-terms, then the operation is abandonned as this combination is a failing combination.

This is a simple case. A more compound problem arises from a user predefined predicate such as ===(X,Y,Z) (which is read as X=Y=Z) or >>>(X,Y,Z) (X is greater than Y and Y is greater than Z). If a partial success/failure occurs in such a case, then we have to store the date-paths of the three sub-terms of X, Y and Z that caused this failure.

The failures' database record structure consists of only two date-paths, and hence a problem arises. Several possible solutions are:

- modifying the database record structure to store the three date-paths, or
- merging the date-paths of the three sub-terms into one date-path, or
- performing this operation on two internal steps, the first between the first two operands (X=Y or X>Y) and then between the second two operands (Y=Z or Y>Z).

We discuss the pros and cons of each of the proposed solutions.

1- Modifying the record structure:

This solution is an unpractical solution, because what if another user defined a predicate that handles more than three operands. This means that the model is not a fixed model, and that it solves a certain class of problems and this is contradicting to our objectives. We are keen to solve any problem expressed in Prolog and thus designing the failures' database with a certain number of date-path fields is not a practical approach. Hence, we reject the first solution.

2- Merging the date-paths of the failing multi-terms:

We attempted the second solution at the early phases of this research, but it proved its weakness. Merging the date-paths of all the sub-terms, two or more, will result in the loss of some information about the failures that might be required afterwards. We clarify this point with the following example.

Example:

Assuming that we have the three multi-instantiations:

X = { **1**, [(1,a),(2,b)]}
Y = { **2**, [(3,a),(4,b)]}
Z = { **3**, [(5,a),(6,b)]}

and it is required to perform the rest of the query,

:- ..., ===(*X*,*Y*,*Z*), *X*=*Z*.

To perform the first equality operation, several partial success/failures will take place that will be stored in the failures' database in terms of a merged date-path of the date-paths of the corresponding sub-terms of X,Y and Z. We can imagine the database in this case to be:

[(1, 1), (2, 3), (3, 6)]	[]	...
[(1, 1), (2, 4), (3, 5)]	[]	...
[(1, 1), (2, 4), (3, 6)]	[]	...
[(1, 2), (2, 3), (3, 5)]	[]	...
[(1, 2), (2, 3), (3, 6)]	[]	...
[(1, 2), (2, 4), (3, 5)]	[]	...

Figure (IV.5): Merging failing date-paths in one date-path

When we come to multi-execute *X*=*Z*, again, we try all the possible combinations of the sub-terms since both multi-instantiations belong to different choice points, making sure that each pair of sub-terms did not cause previous partial success/failures. The different possibilites of the sub-terms of X and Z are

[(**1**,1)] , [(**3**,5)]
[(**1**,1)] , [(**3**,6)]
[(**1**,2)] , [(**3**,5)]
[(**1**,2)] , [(**3**,6)]

For the second case, checking the database, we cannot retrieve a partial success/failure between [(**1**,1)] and [(**3**,6)], though it is implicitly present in the merged date-path. For more complicated programs, this problem is more evident.

We concluded that by merging the date-paths of the failing multi-terms, we lost all information about any partial success/failure between any two specific multi-terms. In other words, this solution proved to be inefficient.

3- Dividing the required operation into n-cascaded steps:

The third solution is a general and very convenient solution. Irrespective of the degree of the problem, we guarantee the multi-resolution of the problem, by splitting the original operation into ***n-cascaded*** stages. Each stage deals with two multi-terms at a time and hence any partial success/failures that might occur are stored in terms of two date-paths only.

This solution, though simple, but is very critical. Several deduction rules are required to continue the multi-resolution of the following subgoals of the query, which will use the failures' database, to identify the succeeding sub-terms from the failing sub-terms.

When a new subgoal, that includes several multi-instantiations, is to be multi-resolved, we consider the different combinations of sub-terms. For each pair of sub-terms, we have 2 distinct date-paths, *date-path-i* and *date-path-j*. We examine the failures' database for each date-path independently. We start with *date-path-i*:

- if no entries exist that includes this *date-path-i*, then we deduce that the corresponding sub-term did not cause any previous partial success/failures.

- if *date-path-i* is found (date-path-1 or date-path-2), we observe the other date-path in the database:

  - if it is equal to *date-path-j*, then the desired operation will not be performed since we found an entry in the failures' database between these two sub-terms.

  -if it is not equal to *date-path-j*, then we store this date-path, which we will call a *suspected date-path*, in a global list until the verification procedure is over.

The same procedure is repeated for *date-path-j*, storing any date-path in the same list of the suspected date-paths as *date-path-i*.

After the termination of the checking of the database, we consider the temporary data structure containing the suspected date-paths. If we encounter all alternatives dates of the same choice point, irrespective which one in the multi-resolution tree, then the desired operation is a failing operation since an intermediate choice point has totally failed. The information about the relevant branch-dates to a choice point are stored during the multi-resolution. If this condition is not satisfied, then the required operation between the two sub-terms will be performed.

The above deduction rule could be clarified by the following graph:

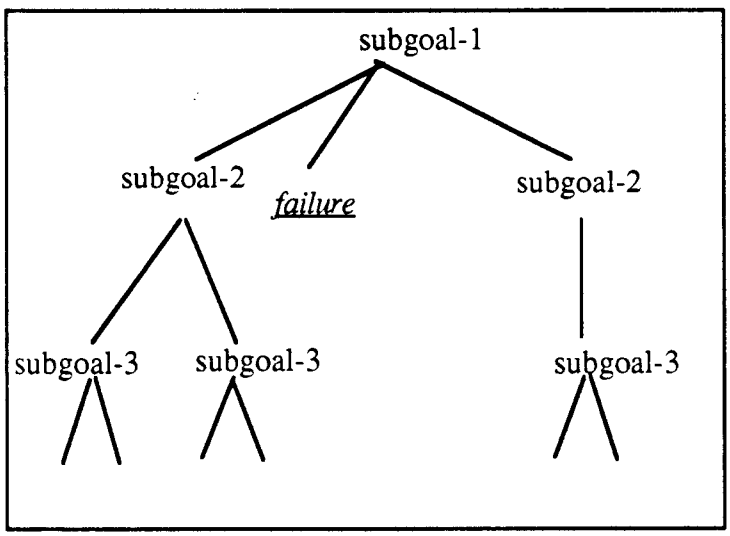


Figure (IV.6): Standard resolution tree

Considering the standard resolution, assume that we wish to execute subgoal-3. The second branch of subgoal-1 led to a failure of all alternatives of subgoal-2, making it impossible to proceed with subgoal-3 for this alternative.

To demonstrate how treatment of failures may take place in this case, we reconsider the previous example.

X = { **1**, [(1,a),(2,b)]}
Y = { **2**, [(3,a),(4,b)]}
Z = { **3**, [(5,a),(6,b)]}

The first equality test will be performed in two steps.

===(X,Y,Z):- =(X,Y), =(Y,Z).

The failures' database in this case will be represented by:

[(1, 1)]	[(2, 4)]	...
[(1, 2)]	[(2, 3)]	...
[(2, 3)]	[(3, 6)]	...
[(2, 4)]	[(3, 5)]	...

Figure (IV.7): Failures database when performing the test on 2 cascaded steps

Coming to the following subgoal $X=Z$, we want to know which are the permitted combinations of the sub-terms between X and Z. We enumerate the same date-paths mentioned above:

[(**1**,1)] , [(**3**,5)]
[(**1**,1)] , [(**3**,6)]
[(**1**,2)] , [(**3**,5)]
[(**1**,2)] , [(**3**,6)]

Considering the first operation, we examine the failures' database for the first date-path, [(**1**,1)]. The suspected date-path is [(**2**,4)] which will be stored in a list. Considering the date-path of the second sub-term, [(**3**,5)], no correponding records exist in the database. Since this date-path is not equal to the suspected date-path, then we could perform the desired operation ($X=Y$) which will result in a success.

The same operation is repeated for the date-paths of the second pair of sub-terms. The suspected date-paths list includes: [[(**2**,4)], [(**2**,3)]]. These two suspected date-paths are all the alternatives of the choice point **2**. Accordingly, the required operation will not be performed due to a total failure of an intermediate choice point.

The same phenomenon is encountered for the third pair of date-paths, that will fail for the same reason, due to a total failure of the choice point **2**.

The equality test will be performed on the last pair of sub-terms as only one date-path was suspected [(2,3)], which is not equal to neither suspected date-paths of the sub-terms.

IV.5 Transactions of the failures' database

As we previously explained, the failures' database is a data structure that serves to store all partial success/failures that occur during the multi-resolution of a query. We classify any transaction into one of two classes; either an update of the database, or a utilisation of the database. We discuss each of these classes in the following sections.

IV.5.1 Update of the failures' database

By updating the database, we mean addition, deletion or modification of records. We describe when and how each of these transactions may take place.

IV.5.1.1 Total failures in the multi-execution phase

Before describing the algorithm to treat a total failure, we first should know how to detect a total failure. After the exploration of an alternative, i.e. attempting to multi-unify a clause head and a subgoal, and before the shallow backtracking proceeds to multi-unify the subgoal to the following clause head, the system checks if this last alternative resulted in a total failure. In case of success, the multi-resolution proceeds normally, as previously explained, to attempt the following clause head, whereas in the case of a total failure all the records in the failures' database representing all the failures that took place during this alternative,will be removed from the database to minimise its size as much as possible. These records are distinguished by their failing branch-locations. They are the records that are tagged with the current failing branch-location = (current choice-point-number, current branch-date).

Moreover, all the variables instantiated (or multi-instantiated) in this alternative will be deinstantiated, and no information about this branch is appended to the multi-resolution tree. It is as if this branch was never traversed, and the multi-resolution proceeds to multi-unify this subgoal to the following clause head.

We emphasise that the above treatment is valid after each attempt to multi unify the subgoal to a clause head (whether it belongs to a choice point or not).

IV.5.1.2 Treatment of failures in the local synchronous OR phases

The failures' database includes the records of all the partial success/failures that have occured in the multi-execution. Given a number of deduction rules, it tends to optimise the information stored by discarding either instantiations or alternatives to optimise the performance and to reduce the time taken to multi-resolve the rest of the query. By the termination of the multi-execution phase, the information stored in the failures' database is the optimum information about all types of failures that were traversed while multi-executing the query.

The synchronous OR phases are intermediate phases between different subgoals, where certain processing takes place. In the previous chapters, we have seen that the local synchronous OR phase is responsible for the creation of the multi-instantiated variables. Another important role of these phases is the treatment of partial success/failures encountered while resolving a subgoal. This treatment is not evident until all the clause heads to solve a subgoal are attempted, i.e. the corresponding choice point is fully explored, afterwhich we detect if only one alternative succeeded or more than one alternative succeeded. We explain how the database is updated in both cases.

It is worth noting that a choice point is also explored when attempting to multi-unify a multi-term to a multi-instantiated variable. What we present as treatment in the synchronous OR phase is exactly similar to what happens after the termination of a multi-unification operation between a multi-instantiation and another multi-term.

1- Single Clause:

This is the case when only one clause (or one sub-term of a multi-instantiation) succeeds from a choice point. After the termination of the exploration of all alternatives, in the local synchronous OR phase of this choice point, the system will discover that only one clause has solved the subgoal.

An algorithm searches in the failures' database if any sub-term of a multi-instantiation caused a partial success/failure in this alternative. If any entries are found, these failures are *rippled up* to its predecessor, where the records representing the partial success/failures in this alternative are deleted and new records are rewritten with the predecessor branch-location as the failing branch-location.

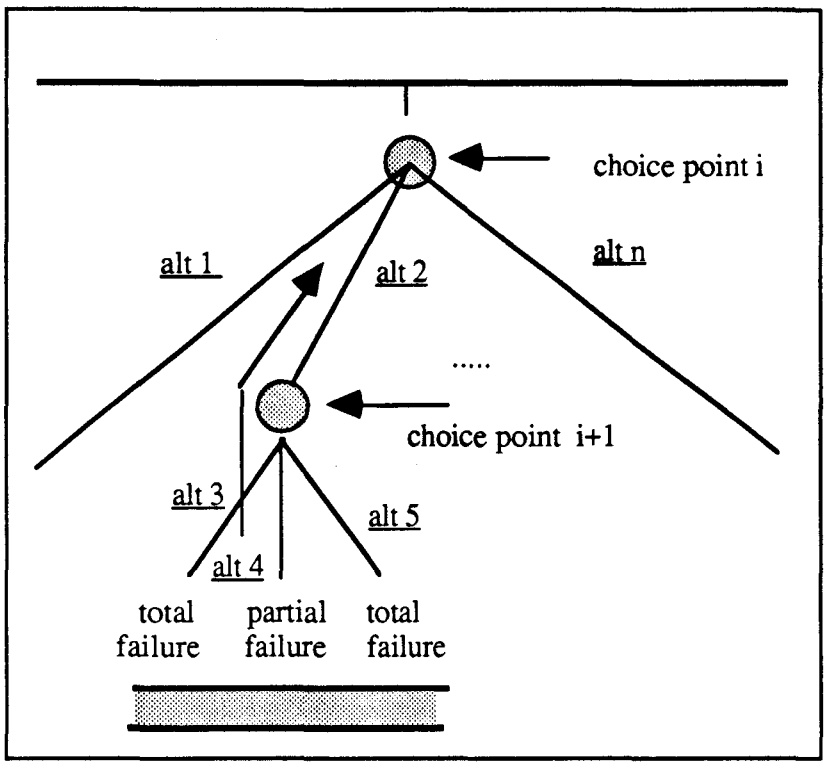


Figure (IV.8): Rippling up of partial success/failures of a single alternative

Example:

For the program,

p(a). p(b). p(c).
q(1). q(X):- s(X).
s(m). s(b).

solve the query,

:- p(X), q(X).

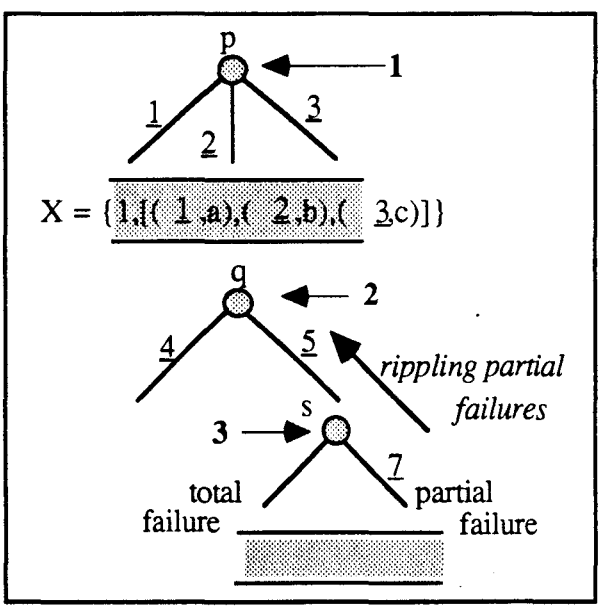


Figure (IV.9): Rippling up partial success/failures from a single alternative of a choice point

In this example, *X* is multi-instantiated to { **1,** [(1,a), (2,b), (3,c)]}. The first alternative of *q* results in a total success. The second alternative includes a choice point, *s*, where only one alternative succeeds partially (branch number 7), resulting in the following entries in the failures' database:

[(1, 1)]	[]	(3, 7)
[(1, 3)]	[]	(3, 7)

Figure (IV.10): Partial success/failures before rippling up

In the local synchronous OR phase of choice point **3**, these partial success/failures are rippled up to the predecessor branch (dated 5) since only one alternative succeeded (branch-date 7). These records are modified to:

[(1, 1)]	[]	(2, 5)
[(1, 3)]	[]	(2, 5)

Figure (IV.11): Partial success/failures after rippling up

The main benefit in doing this is to sense the partial success/failures on a more global level to avoid using these sub-terms in future computations.

2- A Choice Point:

This is the case when attempting to multi-unify a subgoal, including at least a multi-instantiated variable, to a number of clause heads representing a choice point. Here, we consider one sub-term that led to a partial success/failure during the multi-unification to all the clause heads.

When a sub-term of a multi-instantiated variable leads to a consistent partial success/failure when attempting all the clause heads of a subgoal, it should be signalled that this sub-term always leads to a failure. At the local synchronous OR phase of this choice point, a check is made to test if a certain date-path caused repeatedly a partial success/failure on all the

consecutive branch-locations of this choice point. If so, then these failures, will be rippled up to its predecessor level, by deleting all the records representing the partial failures at the different branches, and rewriting a new record signaling this failure at its predecessor branch-location.

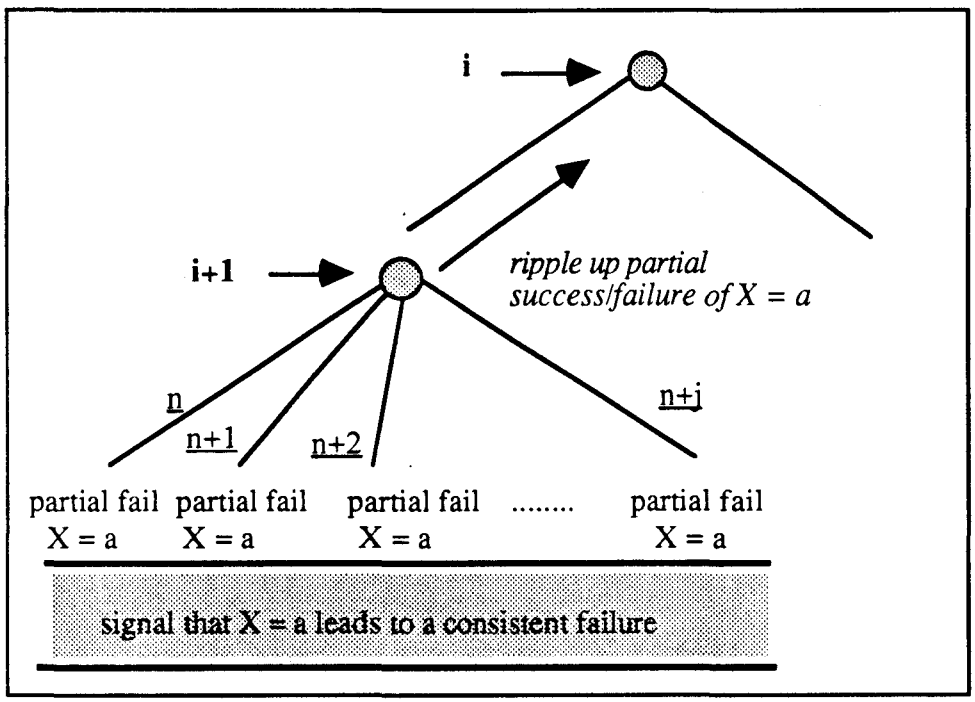


Figure (IV.12) Consistent partial success/failure of a sub-term of a multi-instantiation

We come to this predecessor, which might be either an alternative or a choice point. In the case of an alternative, the result is simple, as the failing branch-location will be the branch-location of this alternative. On the other hand, if its predecessor is another choice point (synchronous OR phase of the previous choice point), and not a branch, then this means that the total failure trapped should be recognised on a global level, i.e. at the root to avoid any further processing of this instantiation during the multi-execution of the query.

This operation is a very optimising operation in the contents of the database. First, a number of records are replaced by only one record, leading to a space-wise optimisation. Secondly, redundant information is eleminated so that the multi-resolution senses any failures that will occur at an early stage.

Example:

For the program,

p(a). p(b). p(c).
q(I). q(X):- s(X).
s(b). s(c).

solve the query,

:- p(X), q(X).

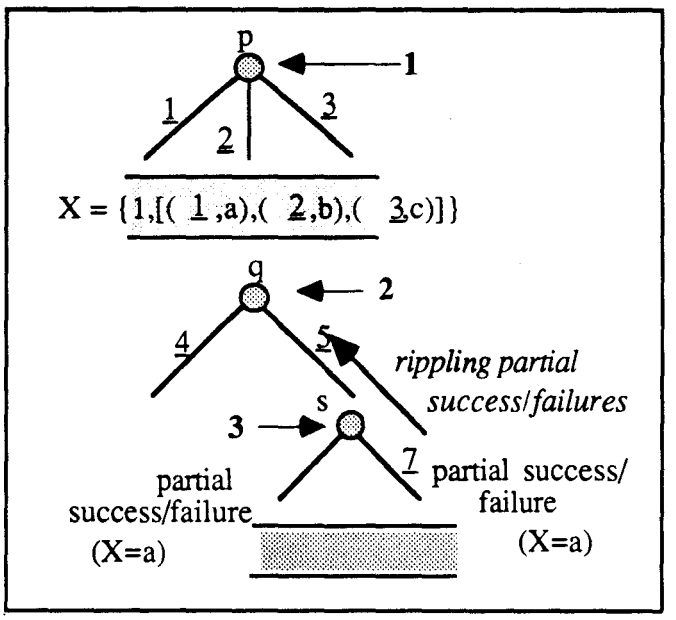


Figure (IV.13): Rippling up partial success/failures of a certain sub-term of a multi-instantiation

In this example, when the subgoal *s(X)* is being multi-resolved, its local synchronous OR phase will realise that *X=a* always leads to a partial success/failure by a simple examination of the failures' database, figure (IV.14) .

[(1, 1)]	[]	(3, 6)
[(1, 3)]	[]	(3, 6)
[(1, 1)]	[]	(3, 7)
[(1, 2)]	[]	(3, 7)

Figure (IV.14): Partial success/failures before rippling up

We notice that the date-path [(**1**,1)] has resulted in a partial success/failure in all the alternatives of the choice point **3** (branch dates 6 and 7). We can report this partial success/failure at the predecessor level. This takes place by deleting all the entries representing these failures and writing a new record representing this failure, but having the predecessor branch-location as the failing branch-location. Hence the entries in the database will be:

[(1, 3)]	[]	(3, 6)
[(1, 2)]	[]	(3, 7)
[(1, 1)]	[]	(2, 5)

Figure (IV.15): Partial success/failures after rippling up

This will be of great use in the multi-outputs phase, while reconstituting standard resolution tree. Rippling up a failure information to predecessor levels will avoid creating the long solution paths that already led to failures. This is n optimisation made to ameliorate the performance of this algorithm.

IV.5.2 Utilising the failures' database

In the above sections, we presented the different treatments of failures in the different phases of the multi-execution phase. We demonstrated the different operations that could be performed to update, or moreover optimise, the information content of the failures' database. To make this discussion complete, we now present the locations where the same database is accessed (statically) to utilise the information stored for optimisation of the performance of the multi-resolution as a whole. These are:

- checking the failures' database when accessing two sub-terms before performing a certain operation, and
- displaying the ensemble of solutions in the multi-outputs phase.

IV.5.2.1 Treatment of failures in the multi-unification algorithm

As detailed in chapter 3, the multi-unification algorithm is responsible to perform all the multi-unification operations between any two multi-terms in the multi-execution phase. We enumerated the different cases of the two multi-terms to be multi-unified. Here, we are concerned with the cases involving a multi-instantiated variable. These cases are the case of multi-unifying a mono instantiated variable to a multi-instantiated variable, the reverse case, or multi-unifying two multi-instantiated variables. In each of these cases, each sub-term of the multi-instantiated variable is to be multi-unified to the second multi-term, according to the rules previously mentioned when discussing the algorithm. Similar to coherency test, there

are the failures tests that verifies whether the two multi-terms to be multi-unified have eventually caused partial success/failures.

The main role of the failures' database is to store all the already occured partial success/failures so as to avoid, as much as possible, performing useless computations. In chapter 2, we demonstrated the necessity of double checking with the failures' database for 2 cases; failures that occured during the same multi-unification operation (problem 2.4) and failures that occured in previous multi-unification operations (problem 2.5). We describe how we treated each of these problems.

IV.5.2.1.1 Partial success/failures in the same branch-loaction

This case may take place in the same multi-unification operation, or in different multi-unification operations.

1- Same multi-unification operation:

This is the case when a sub-term results in a partial success/failure during a multi-unification operation. Later, in the same operation if any sub-term having the same date-path that failed is involved in an invoked multi-unification operation, the corresponding sub-term will be abandoned as it already proved to be a failing sub-term.

Considering a more general example than that we mentioned in section IV.3.2.2 when discussing indirect total failures.

Example:

For the shown program,

p(a,a). *p(b,b).*
q([a/b]). *q([b/a]).*

solve the query,

:- p(X,Y), (Z=m; Z=[X/Y]), q(Z).

Multi-resolving the first subgoal, we have the following multi-instantiations:

X = { **1** , [(1,a) , (2,b)] } and Y = { **1** , [(1,a) , (2,b)] }

The second subgoal will result in the multi-instantiation of *Z* to:

$$Z = \{\ 2\ ,\ [\ (\underline{3},m)\ ,\ (\underline{4},\ [\ \{1,[(\underline{1},a)\ ,\ (\underline{2},b)]\}\ |\ \{1,[(\underline{1},a)\ ,\ (\underline{2},b)]\}\ ])\ ]\ \}$$

To multi-unify *q(Z)* to the first clause head, we attempt to multi-unify each sub-term with *[a/b]*. The first sub-term *m* will result in a partial success/failure as we are trying to unify an atom to a list. The second sub-term (which is a list of multi-terms) will invoke a number of internal multi-unifications. The multi-unification of the heads of the list will result in the following partial success/failures:

[(2, 3)]	[]	(3, 5)
[(2, 4),(1, 2)]	[]	(3, 5)

Figure (IV.16): Partial success/failures entries

When coming to the tails' multi-unification, the algorithm tends to multi-unify the sub-term *a* to the corresponding argument in the clause head. Checking with the failures' database, there is no entry that indicates that the date-path of this sub-term ([(2,4),(1,1)]) has caused a partial success/failure in the same multi-unification operation (3,5). Performing the operation, a partial success/failure is detected, which will be appended to the database.

[(2, 3)]	[]	(3, 5)
[(2, 4),(1, 2)]	[]	(3, 5)
[(2, 4),(1, 1)]	[]	(3, 5)

Figure(IV.17): Failures' database

Coming to the next sub-term, *b*, again, a check if any failures with the same branch-location resulted from the same date-paths. It is found, accordingly, the current multi-unification operation is dropped as it already caused a failure and hence any further computation with this sub-term is just a waste of processing time.

The algorithm resumes the following multi-unifications in the same manner.

2- Different multi-unification operations:

Consider the following multi-instantiations:

X = { 1, [(1,{ 2, [(2,a),(3,b)]}),(4, { 3, [(5,c),(6,d)]})]}
Y = { 1, [(1,x),(4,y)]}

and the query,
:- ..., X=\=a, X=\=b, write(Y).

For the first equality test, we have the following partial success/failure:

[(1, 1),(2, 2)]	[]	(0, 0)

Figure (IV.18): Partial success/failures entries

The current branch-location is given by (**0**,0) because the failure took place in a single clause (not a choice point), hence any occuring failures are sensed on a global level.

After the second equality test, a new record is added:

[(1, 1),(2, 2)]	[]	(0, 0)
[(1, 1),(2, 3)]	[]	(0, 0)

Figure (IV.19): Partial success/failures entries

We observe that all the alternatives of the choice point **2** have failed in the same branch-location (**0**,0) resulting in an indirect total failure of the choice point **2**. If no special treatment is made, then when writing the different values of *Y* (the following subgoal), we will not take know that the first sub-term has failed.

In this case, these records that represent an indirect total failure are deleted and a new entry indicating that the sub-term whose branch-location is [(**1**,1)] has failed at (**0**,0). Accordingly,

when displaying the different values of Y, checking with the failures' database, the information that the first sub-term is a failing sub-term is clear and retrievable.

IV.5.2.2 In the multi-outputs phase

To make this discussion complete, another important role of the failures' database is during the display of solutions. Before considering a sub-term of a multi-instantiation, a double-check is made with the failures' database to make sure that it did not result in a partial success/failure. The algorithms for displaying the solutions will be discussed in the next chapter.

IV.6 Conclusion

We discussed in this chapter how failures are treated in the multi-resolution model. We justified our choice to the way by which we treated these failures. We pointed out how the failures database is updated dynamically during the multi-execution phase to optimise its information content. We also showed how it is treated statically during the multi-outputs phase to display the solutions. The correponding algorithms will be discussed in more detail in the following chapter.

Chapter Five

Arithmetic & Predefined Predicates

Abstract

We explain how arithmetic operations are performed in the multi-resolution model. The multi-unification algorithm for multi-arithmetic operations is presented. We also discuss the possibility of treating several predefined predicates in our model, terminating our discussion with a detailed explanation of the different algorithms implemented to display the solutions.

V.1 Introduction

Within the formalism of logic programming, there are two methods for doing arithmetic operations. One tedious method is possible by giving the relationship explicitly as a set of assertions of the relationship, for example, *sqr(X,Y)* could be defined in the form of a table for all possible different cases. In this case, arithmetic operations will be table searches. Obviously, the entire infinite relation cannot be stored and the defined subset will consume a large amount of memory space. Alternatively, the relation can be computed, which is the case in Prolog. In Prolog, arithmetic is performed by metalogical evaluable predicates analogous to the built-in primitive functions of applicative languages. The evaluable predicates are metalogical because arithmetic is done by escaping from the system of resolution and using another formal system, in this case, the underlying machine hardware. Prolog uses the machine for arithmetic because it is faster. Numbers are a subset of atoms, and the evaluable predicates implementing arithmetic operations are restricted to operating on terms representing numbers.

V.2 A standard arithmetic operation

Arithmetic is performed by a number of predefined arithmetic operators (+, -, *, /, mod, ++, --, **, //, <, >,≥, ≤, =:=, =\=) or the evaluable predicate *is*.

The arithmetic operators are defined to the system as infix operators. A call of the form

Operation(number,Value) or
Operation(number1,number2,Value)

will bind the variable *Value* to a numeric value, where *number* is a numeric atom or a variable bound to a number.

The evaluable predicate *is* is a binary predicate that can be used as an infix operator. The second argument must be a legal arithmetic expression, constructed from the usual operators and integer terms, and the first argument can either be an integer or a variable. When *is* is called, the expression is evaluated, and the first argument is unified with the value of the expression. If it is a variable, then it is bound to this value.

A simple example is the goal *X is 3+5*, which has the solution X=8. The goal *8=3+5* succeeds. The goal *3+5 is 3+5* fails, because the left hand argument, *3+5*, does not unifiy with 8, the result of the evaluation of the expression.

V.3 A multi-arithmetic operation

It corresponds to an arithmetic operation in the multi-resolution model. The main difference from the standard arithmetic operation is that the required arithmetic operations may operate on multi-instantiations.

Since the standard arithmetic operators are predefined in Prolog, calling them in the multi-resolution model will fail if operands are multi-instantiated. Accordingly, we trap such operations and perform them differently in the multi-resolution mode.

When regarding the structure of an arithmetic instruction, we find it in the form:

Operation(A,B,R)

where *Operation* is any standard arithmetic operation,

A, B are operands of the operation, and

R is the result (always a variable).

In the multi-resolution model, *A* and *B* may be any two multi-terms (non variables), but *R* should be strictly a variable, though it might be multi-instantiated to other variables. Performing an arithmetic operation on multi-instantiated operands and storing the result in another multi-instantiation in the same step is a complex operation .

Example:

+({**1**,[(1,10) , (2,{**3**,[(5,1) , (7,2)]}), (4,{**4**, [(11,100) , (12,200) , (13,300)]})]},
{**3**,[(5,5) , (7,10)]},
{**8**,[(15,A) , (16,{**7**,[20,B) , (21,C)]})]})

Performing the addition operation between the first two multi-instantiations is already a complicated operation. Adding to that, we have to map the result of this addition operation to the third multi-instantiation, the resultant is a very complicated operation.

Accordingly, we defined another mechanism to perform any multi-arithmetic operation, which is totally transparent to the user. It is a special algorithm that is responsible to perform the multi-arithmetic operation into two steps as follows:

Step 1:
First, it multi-unifies the operands and multi-performs the desired multi-arithmetic operation on them. The output of this phase is a ***multi-result*** stored in an intermediate variable. A special algorithm is responsible to execute this phase which we will describe in the following section.

Step 2:
Second, it multi-unifies the multi-result (the intermediate variable)with the original variable, *R*, where the result is to be stored in. It is a standard multi-unification operation, and may be any of the cases mentioned in the previous chapter. It is possible that this variable *R* might be already multi-instantiated to another ensemble of variables.

Hence, the above operation is analysed to

```
Operation(A,B,R):- perform_operation(Operation,A,B,R1), multi-unify(R1,R).
```

A special multi-unification algorithm for arithmetic operations was defined, that is reponsible to multi-unify the operands, and performs the multi-arithmetic operation (step 1). All operations concerning the dates are manipulated. Previously discussed coherency features impose (for a multi-term and a multi-arithmetic operation), and failures are also treated.

We describe the different functions performed by the multi-unification algorithm for arithmetic operations.

V.4 Multi-unification algorithm for multi-arithmetic operations

The algorithm described in the previous chapter deals with multi-terms resulting from non arithmetic operations. Here, we define the algorithm that handles arithmetic multi-terms. The following program is an example on how to handle double operand instructions in the multi-resolution model.

```
double_op(O,A,B,R1):-
  atomic(A), atomic(B), !, O(A,B,R1).

double_op(O,A,B,R1):-
  nonvar(A), A = {Choice_pt,XX},!,
  varsin(R1,V),
  findall(V, (multi_mono_double_op(O,XX,B,R1),L),
  assign_values(V,L).

double_op(O,X,Y,R1):-
  nonvar(A), A = {Choice_pt,YY},!,
  varsin(R1,V),
  findall(V, (mono_multi_double_op(O,A,YY,R1),L),
  assign_values(V,L).


multi_mono_double_op(O,[X1|Xn],B,R1):-
  check_coherency (...),
  double_op(O,X1,B,R1).

multi_mono_double_op(O,[X1|Xn],B,R1):-
  multi_mono_double_op(O,Xn,B,R1).
```

Program (V.1): Multi-unification algorithm for a double operand arithmetic operation

V.4.1 Unification cases

In all the following cases, *R1* is a variable.

1- A is an atom, B is an atom:

This is the standard case. The result is performing the required operation (addition, subtraction, etc.) storing the result in the intermediate variable R'. If a failure occurs, then this failure is stored in the same manner in the failures' database in terms of the date-paths of both multi-operands and the current branch-location as the failing branch-location.

Examples:

1 - +(2,6,R) will result in the instantiation of an intermediate variable to 8. This variable, in return will be bound to R.

2- /(5,0,R).

This operation is false operation that results in a failure. This failure is stored in the failures' database.

2- A is a multi-term, B is a multi-instantiation:

This resembles the mono/multi case in the standard multi-unification algorithm. The desired operation is performed between the operand *A* and each operand in *B*. The result is a multi-instantiated variable R' where the results of all the performed operations are stored.

Example:

For the operation ,

+(1,{1, [(1,10), {2, [(2,20), (3,30)]}, (4,40)]}, R1)

the result is R1 = {1, [(1,11), {2, [(2,21), (3,31)]}, (4,41)]}.

3- A is a multi-instantiation, B is a mono-instantiation:

It is the reverse operation.

4- Optimisations:

The 2 above cases (2 and 3) handle a multi-operation between a multi-instantiation and any multi-term. This latter may be another multi-instantiation. We might mention this case (2 multi-instantiations) explicitly in the algorithm. The only difference is the control to override the original loops' order so that the multi-arithmetic operation may be perfomed according to a certain condition (ascending/descending order of choice-point-numbers). The ordering of the loops when multi-unifying two multi-instantiations is an optimisation issue with respect to failures.Imposing a certan order to execute the loops is to impose a certain ordering on the date-paths when failures occur. This is a technical aspect that facilitates the search in the failures' database.

As we previously explained, there are two cases of optimisations; performing a multi-arithmetic operation on two multi-instantiations belonging to the same choice point, or two multi-instantiations that belong to two different choice points.

We discuss each of these cases.

4a - A = { C,...}, B ={C,...};

A = { **Choice_point** , [(d1,v1) ..., (dn,vn)]}
B = { **Choice_point** , [(d1,v1) ..., (dm,vm)]}

Here, we have two multi-instantiated operands, originating from the same choice point. The result is that we perform the arithmetic operations on the operands that have the same branch-dates.

Example:

+({**1**,[(1,10) , (2,20)]},{**1**,[(1,1) , (2,2)]},R).

This operation results in two simple addition operations between the corresponding elements of the two multi-terms. The result is the multi-instantiation {**1**,[(1,11) , (2,21)]} which will then multi-unifed with the resultant variable.

Note:

In case of 2 multi-instantiated variables, originating from the same choice point, but with different lengths, only the terms having the same branch-dates are multi-unified, otherwise a failure is recorded.

Example:

For the instruction,

+({**1**,[(1,1) , (2,2) , (3,3) , (4,4)]},{**1**,[(1,10) , (5,20) , (4,40)]},R)

only two addition operations will take place between the terms dated 1 and 4. The resulting multi-instantiation is {**1**,[(1,11) , (4,44)]}.

4b- A = {Ca,...}, B = {Cb,...}:

A = { **Choice_point_a** , [(d1,v1) ..., (dn,vn)]}
B = { **Choice_point_b**, [(d1,v1) ..., (dm,vm)]}

As for the multi-operands instantiated at different choice points, consider two multi-instantiations of lengths *m* and *n* respectively. In this case, all combinations are possible between the different operands. The desired operation is performed between each operand of *A* with all the operands in *B*. Thus, *m*n* operations will be invoked. Each of the resulting operations converge to the second case mentioned in this algorithm (mono/multi).

Example:

For the instruction,

+({**1**,[(1,1),(2,2),(3,3)]},{**2**,[(7,10),(8,20)]},R)

all combinations between the two terms are permitted, i.e. 6 addition operations take place. The resulting multi-instantiation is

{**1**,[(1,{**2**,[(7,11) , (8,21)]}) , (2,{**2**,[(7,12) , (8,22)]}) , (3,{**2**,[(7,13) , (8,23)]})]}

which in turn will be multi-instantiated to the variable *R*.

Note:

We note that a certain order is considered when performing the operations. We consider the choice points in an ascending order.

If the order by which the above operations were reversed, then the resulting multi instantiation is

{**2**,[(7,{**1**,[(1,11) , (2,12) , (3,13)]}),(8,{**1**,[(1,21) , (2,22) , (3,23)]})]}

Both structures represent the same operations. The only difference is the order of loops execution by which the multi-instantiation were created. This ordering problem is a feature that concerns mainly the failures' treatment. Chapter 5 discusses this aspect in more details.

V.4.2 Memory representation of arithmetic multi-instantiations

The arithmetic multi-instantiation is represented formally similar to the non arithmetic multi-instantiation. A scheme of the representation of an arithmetic multi-instantiation in the memory is given in fig. (5. 1).

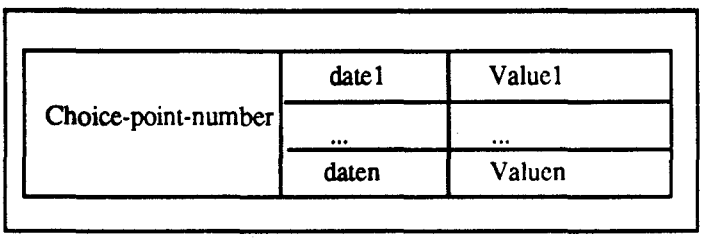


Figure (V.1): An arithmetic multi-instantiation

From the above discussion, it is clear that the arithmetic multi-instantiation is not represented by data-sharing. No memory references can take place in a multi-arithmetic operation as the desired operation is perfomed while multi-unifying the two multi-terms, and not left to be applied at the end of the multi-resolution. Hence, in the multi-resolution model, when dealing with multi-instantiations, *data-copying* is employed on the multi-instantiation each time a new alternative is attempted.

This is a costy representation due to the copying of already large data structure.

Example:

Consider the following program:

number(10). *number(20).* *number(30).*

For a non arithmetic query:- *number(X), number(Y), Z = [X/Y].* the variables *X* and *Y* are given by:

X = { **1,** [(1,10) , (2,20) , (3,30)]}
Y = { **2,** [(4,10) , (5,20) , (6,30)]}
Z = [{ **1,** [(1,10) , (2,20) , (3,30)]} | { **2,** [(4,10) , (5,20) , (6,30)]}].

These variables are represented in the memory as shown in fig. (V.2).

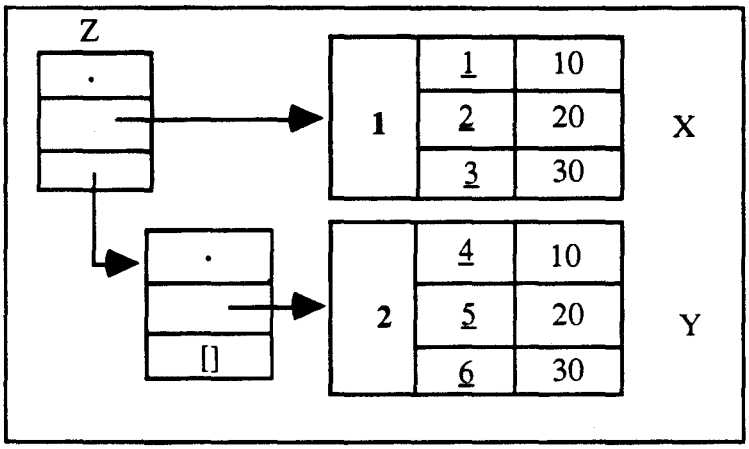


Figure (V.2): Data sharing is implemented to create non-arithmetic multi-terms

X and *Y* are created and stored in the memory. Any new structure (resulting from a non arithmetic subgoal) that includes either *X* or *Y* will not copy the already existing structures. A memory reference is enough to create this new structure.

If we change the last subgoal in the above query to

```
:- ..., +(X,Y,Z).
```

in this case, the memory representation of the multi-instantiated variable is as follows:

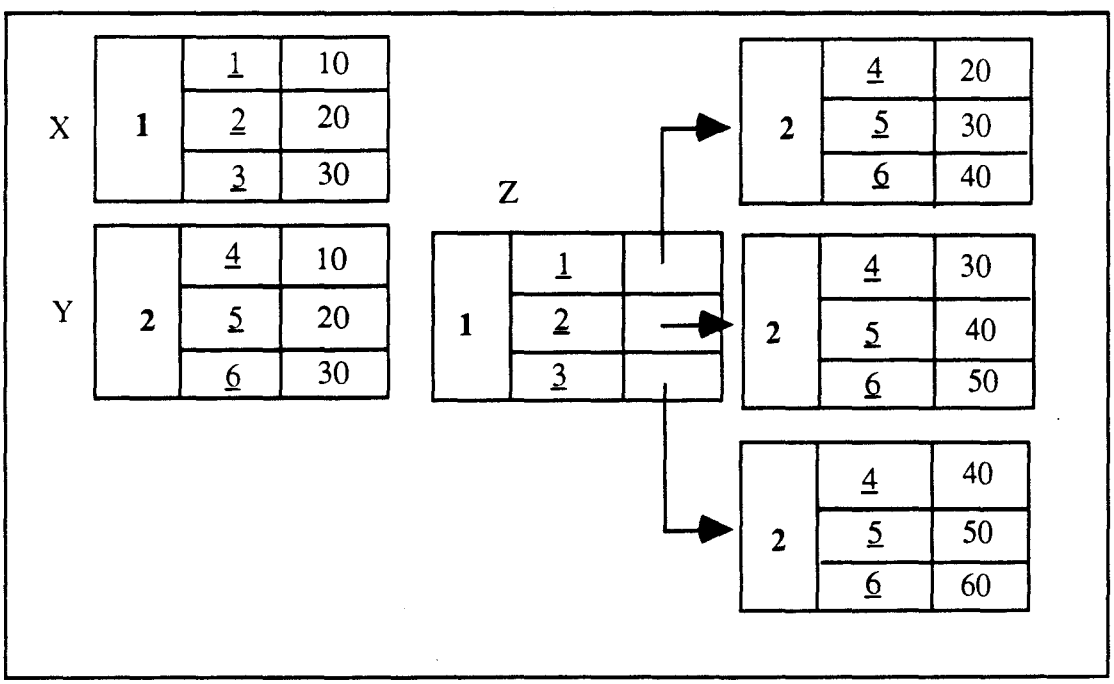


Figure (V.3) : Data copying is implemented to create arithmetic multi-terms

Here, copying of the original multi-terms takes place to create a new multi-instantiation. In arithmetic multi-terms, there are no memory references. Each time a multi-instantiation is considered, the desired multi-arithmetic operation is perfomed on the different sub-terms of this multi-instantiation resulting in another multi-instantiation.

It is worth noting that a multi-instantiation represented by data-sharing may be represented by data-copying, but the reverse case is not true. In our example, $Z = [X/Y]$ may be represented as shown in fig. (V.4):

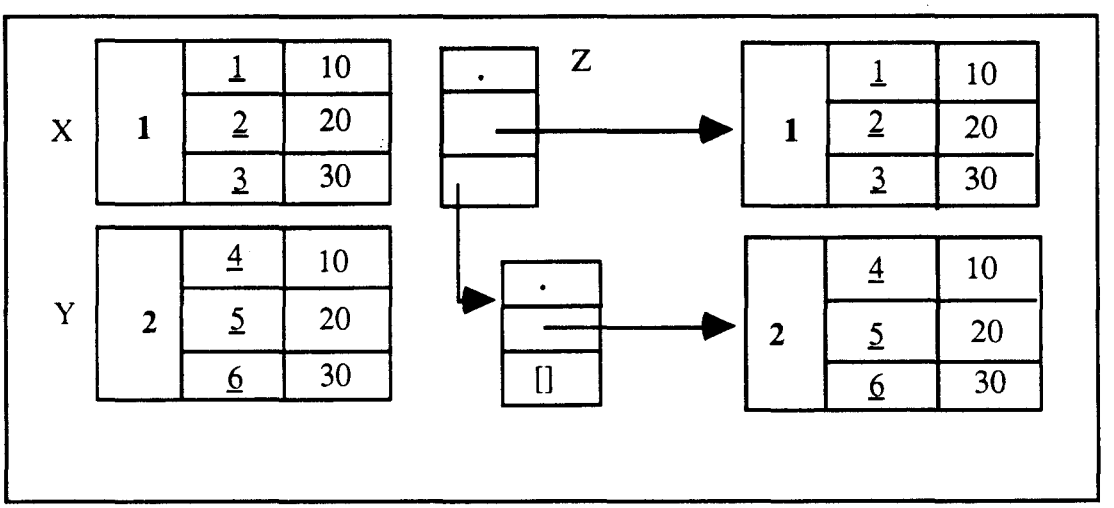


Figure (V.4) : A non arithmetic multi-instantiation represented by data-copying

For the same variable Z, the two representations, given in figures (V.2) and (V.4) are equivalent. For non arithmetic multi-instantiations, we adopt data-sharing to avoid waste of memory space, whereas in the case of arithmetic multi-instantiations we cannot help not representing these multi-instantiations by data-copying because the required multi-arithmetic operation has to be performed.

V.4.3 Creation of date-paths of an arithmetic sub-term

To access a sub-term of a multi-instantiation, its corresponding date-path is created. The manner of creation of date-paths of arithmetic sub-terms is similar to that of non arithmetic sub-terms, as discussed in section III.3.5. The same features, previously discussed, are applicable. Similarly, coherency checks impose for each sub-term and for each multi-arithmetic operation.

V.4.4 Coherency of an arithmetic sub-term

It respects the same concepts of the coherency of a normal multi-term, that we discussed in section II.3.3 (problem 2.1). Each time a new sub-term is encountered, its branch-location is appended to the already constructed date-path. Before systematically appending this branch-location, the coherency of this sub-term with respect to the operation is checked. This check is performed by the same program mentioned in section III.4.2.4.

V.4.5 Coherency of the performed multi-arithmetic operation

Before explaining how incoherency of a multi-arithmetic operation is detected, we present the follwoing example which demonstrates a common case of incoherency on the multi-arithmetic operation level (even when multi-instantiations are represented by data-copying).

Example:

Consider the following base of facts,

number_i(1). *number_i(2).*

number_j(10). *number_j(20).*

with the following query,

:- number_i(I), number_j(J), R1 is I+J, R2 is J+I, R is R1-R2.

I will be multi-instantiated to { **1** , [(1,1) , (2,2)] },
while *J* will be multi-instantiated {**2** , [(3,10) , (4,20)] }.

After the first addition operation, *R1* will be multi-instantiated to:

R1 = {**1,** [(1,{**2** , [(3,11) , (4,21)]}) , (2,{**2,** [(3,12) , (4,22)]})]}

Similarly, *R2* will be multi-instantiated to:

R2 = {**2,** [(3,{**1** , [(1,11) , (2,12)]}) , (4,{**1,** [(1,21) , (2,22)]})]}

(The difference in the contents of *R1* and *R2* resulted from the difference in the order of the loops by which *R1* and *R2* were created respectively. Both structures are equivalent. It is when partial success/failures will occur that the corresponding date-paths of *R1* will be sorted, while those of *R2* will not be sorted. Sorted date-paths will facilitate the search in the failures' database).

Coming to the last sub goal, *R* is the result of the subtraction operation between *R1* and *R2*. Since *R1* and *R2* belong to different choice points (1 and 2 respectively), then all combinations of the multi-instantiations of *R1* and *R2* are tried. *R1* contains four values, and *R2* contains also four values. If no coherency checks are performed then we meet the same problem of coherency of operations (problem 2.2) resulting in the performance of 16 subtraction operations.

Double checking with standard Prolog, we regard the classical resolution tree, fig. (V.5). There are only four solutions (zeroes) are produced! That points out that there are a number of unnecessary, evenmore *false*, operations that could be performed during the multi-resolution of the given query if no coherency tests were performed.

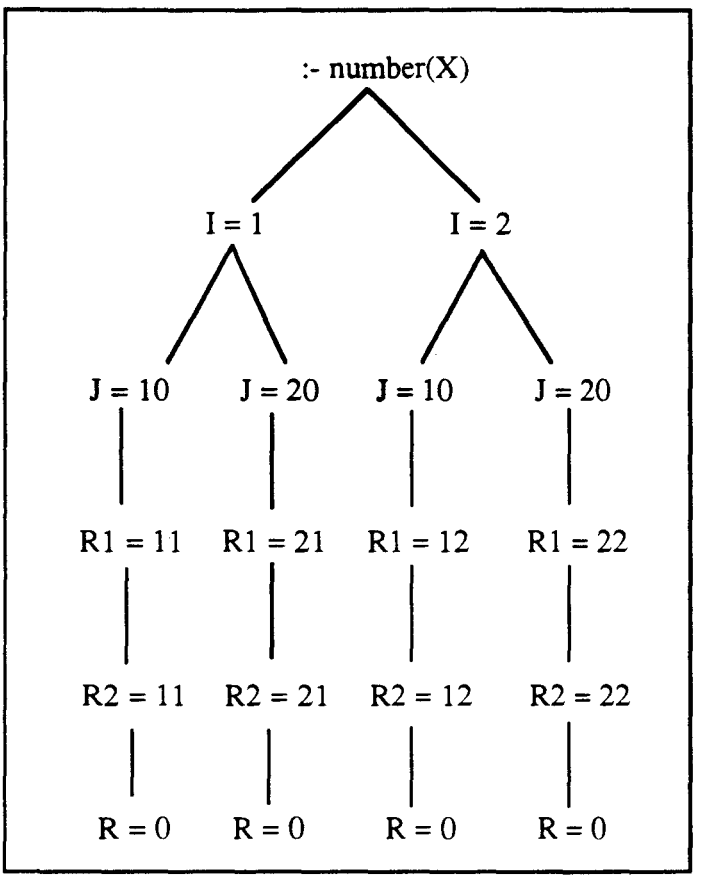


Fig. (V.5): Standard resolution tree

The illegality of the performed subtraction operation is proved by violating the strict rule that no two multi-terms resulting from two distinct alternatives of the same choice point could be utilized in the same operation. Accordingly, it is required to sense the validity of the operation to be performed.

The program that performs this operation is the same as that responsible to detect incoherencies in non arithmetic operations. It was discussed in detail in the previous chapter (programs III.3 and III.4).

V.4.6 Treatment of failures

The encountered failures are the same as that discussed in the normal multi-unification; total failures and partial success/failures. Treatement and types, discussed throughout section II.3.5 are applied in the multi-arithmetic operations.

The same problems concerning failures (problems 2.3, 2.4 and 2.5) exist in multi-arithmetic operations. Partial failures are stored in the failures' database in terms of the date-paths of the two operands and the failing branch-location. Partial failures leading to total failures (problem 2.4) are treated in the multi-unification algorithm for arithmetic operations in the same manner.

V.5 Predefined predicates

In Prolog, there are several predefined predicates to perform certain operations ,such as the input/output operations, or to impose control on the resolution as the *fail, !,* or *not* predicates. We examined the behaviour of some of these predicates in the multi-resolution model and we found out that a special treatment was required to handle each of them.

Following is a discussion of selected predicates and how we propose to treat them in the multi-resolution model.

V.5.1 Input/Output Predicates

A very important class of predicates is the input/output predicates. In Prolog, the basic predicate for input is *read(X)*. This goal reads a term from the current input stream. The term that has been read unifies with X, and read succeeds or fails depending on the result of the unification. The basic predicate for output is *write(X)*. This goal writes the term X on the current output stream.

V.5.1.1 read(X)

The normal use of read is with a variable argument X, which acquires the value of the first term in the current input stream. When backtracking to a previous choice point takes place, each time a solution is demanded. *read(X)* succeeds with a (possibly) different value for X.

In the multi-resolution model, we pass each subgoal only once and no deep backtracking takes place. Hence, only one value will be entered and thus processed (instead of several values). We examined the different cases where a *read* may exist. Its presence is relatively rare in the known benchmarks. Nevertheless, we discuss several propositions for its treatment.

Considering a simple goal that includes only one *read*. A proposition is that the user enters all the required values of *X* in one step, which will be defined as a multi-instantiation to the system. As we previously said, any multi-instantiation is characterised by its choice point number and its branch-dates. What we are creating here is an *artificial* choice point. Its choice point number is the choice-point-number of the predecessor choice point. Afterwards, the multi-resolution proceeds normally as previously discussed.

The following example demonstrates the case of a single *read* in a query.

Example:

For the program,

p(1). p(2). p(3).

consider the query,

:- p(X), read(Y), Z = [X/Y].

In the standard resolution, each time *p(X)* succeeds, *X* will be instantiated and the system waits for the user to enter *Y*. Since three backtracking operations take place, we will asume that the three values the user has entered are *a,b,c* corresponding to each value of *X* respectively. The solutions displayed are

X = 1 , Y = a , Z = [1/a]
X = 2 , Y = b , Z = [2/b]
X = 3 , Y = c , Z = [3/c]

In the multi-resolution model, *p(X)* is attempted that results in the multi-instantiation of *X* to 3 different sub-terms. *X* will be multi-instantiated to { **1,** [(1,1), (2,2), (3,3)]}, and the subgoal *p(X)* will never be reattempted.

To obtain the same results as in the standard resolution, we expect that the user enters all the different values of *Y* in one step. Since there are three values to be entered (*a,b,c*), then it is as if we are creating a choice point for *Y*, i.e. *Y* will be multi-instantiated. To obtain the same solutions as Prolog, the choice-point-number of *Y* has to be that of *X*, so that the solutions of *X* and *Y* are the sub-terms that have the same branch-date. In this case, *Y* will be given by:

$$Y = \{ \mathbf{1}, [(\underline{1},a), (\underline{2},b), (\underline{3},c)]\}$$

whose choice-point-number is **1**, and the branch-dates are from $\underline{1}$ to $\underline{3}$. The different solutions of *Z* are produced by the multi-outputs module, which will be the same as that produced by standard Prolog.

Actually, this is a simple proposition to solve the problem arising from the *read* predicate. Considering a more complicated case where the predecessor choice-point is not a simple choice point, as that in the previous example.

Example:

Consider the program,

```
p(1).    p(X):-s(X).
s(2).    s(3).
```

solve the query,

```
:-p(X), read(Y), Z = [X/Y].
```

Multi-resolving the first subgoal, *X* will be multi-instantiated to

$$X = \{ \mathbf{1}, [(\underline{1},1), (\underline{2}, \{\mathbf{2}, [(\underline{3},2),(\underline{4},3)]\})]\}$$

Now, if the user enters all the solutions of *Y*, which we will assume that they will also be *a,b* and *c*, the created choice point of *Y* has to be created analogous to *X*, i.e.

$$Y = \{ \mathbf{1}, [(\underline{1},a), (\underline{2}, \{\mathbf{2}, [(\underline{3},b),(\underline{4},c)]\})]\}$$

In more complicated predecessor choice points, i.e. the resulting multi-instantiations include nested multi-instantiations too, we have to ensure the coherency of *Y* with respect to the already created multi-instantiations. This could be the case if several reads may exist. Consider the following example:

Example:

```
p(1).    p(2).    p(3).
```

:- p(X), p(Y), read(A), p(Z), read(B).

The standard resolution tree is given by

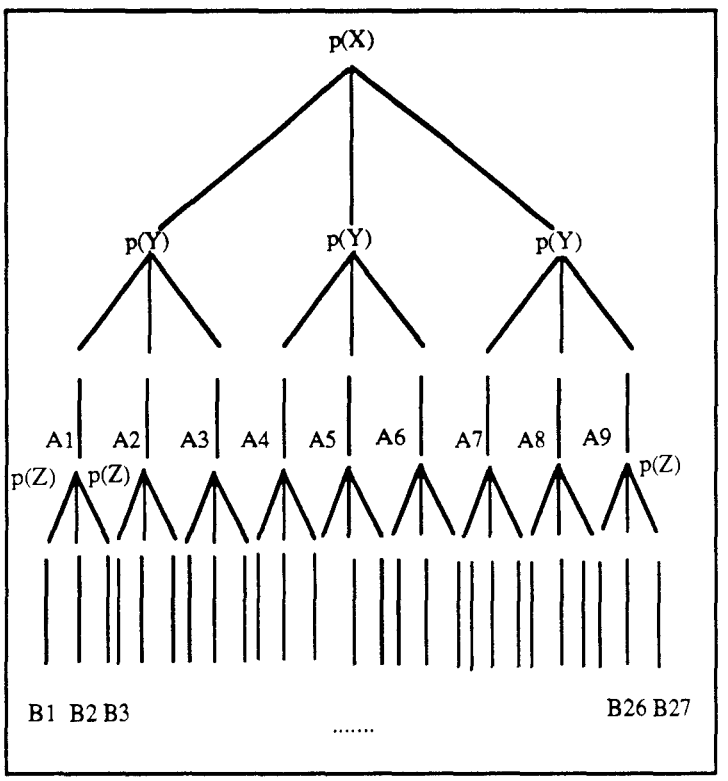


Figure (V.6): Standard resolution tree

In the multi-resolution model, the variables *X* and *Y* are given by:

$$X = \{\ \mathbf{1},\ [\ (\underline{1},1),\ (\underline{2},\ 2),\ (\underline{3},3)\ ]\}$$
$$Y = \{\ \mathbf{2},\ [\ (\underline{4},1),\ (\underline{5},\ 2),\ (\underline{6},3)\ ]\}$$

To enter the different alternatives of *A*, they should be assembled in the multi-instantiation,

$$A=\{\quad \mathbf{1},[(\underline{1},\{\mathbf{2},[(\underline{4},A1),(\underline{5},A2),(\underline{6},A3)]\}),$$
$$(\underline{2},\{\mathbf{2},[(\underline{4},A4),(\underline{5},A5),(\underline{6},A6)]\}),$$
$$(\underline{3},\{\mathbf{2},[(\underline{4},A7),(\underline{5},A8),(\underline{6},A9)]\})\quad]\}$$

Multi-resolving *p(Z)* will result in the multi-instantiation

$$Z = \{\ \mathbf{3},\ [\ (\underline{7},1),\ (\underline{8},\ 2),\ (\underline{9},3)\ ]\}$$

For the second read, *B* will be mapped to the sub-tree of all the predecessor choice points as follows:

```
B={1,[(1,{2,[(4,{3,[(7,B1),(8,B2),(9,B3)]}),
                (5,{3,[(7,B4),(8,B5),(9,B6)]}),
                (6,{3,[(7,B7),(8,B8),(9,B9)]})]}),
          (2,{2,[(4,{3,[(7,B10),(8,B11),(9,B12)]}),
                (5,{3,[(7,B13),(8,B14),(9,B15)]}),
                (6,{3,[(7,B16),(8,B17),(9,B18)]})]}),
          (3,{2,[(4,{3,[(7,B19),(8,B20),(9,B21)]}),
                (5,{3,[(7,B22),(8,B23),(9,B24)]}),
                (6,{3,[(7,B25),(8,B26),(9,B27)]})]})      ]}
```

which is relatively a complex structure.

From all the above examples, we conclude that treating the *read* predicate in the multi-resolution model, though complex, yet it is possible to obtain the same semantics of the standard Prolog execution. It requires that the user enters all the required data at once, taking into consideration the structure of the predecessor sub-tree.

V.5.1.2 write(X)

If *write(X)* was a subgoal in the query to be solved, the standard resolution will display the current value of *X* on the output stream. In the multi-resolution model, *write(X)* will display all the possible values of *X*. Given that the variables may be multi-instantiated, the multi-resolution utilises these multi-instantiations together with the failures' database and the multi-resolution tree to write the correct values of *X*. The same procedure takes place inherently in the multi-outputs module, where all variables' instantiations are treated to produce an ensemble of solutions.

We detail the algorithms for the display of solutions as follows:

V.5.1.3 Display of solutions

To display the required solutions of a variable, we have two alternatives, either respect the same order and number of solutions that are displayed in standard Prolog, or neglect this constraint. We explain in details each algorithm.

V.5.1.3.1 Same order and same number of solutions as Prolog

In this approach, we actually construct the different solution paths of the query in a standard resolution tree. The standard resolution tree is reconstructed by deep backtracking. Only the paths that produce solutions are constructed. No unification takes place. Each time a new solution path is constructed, a double check is made with the failures' database to make sure that this solution path did not cause any partial success/failures in the multi-resolution.

A solution path is defined by a list of succeeding branch-dates starting from the root, 0, until a leaf (solution) is reached, making sure that no partial success/failures occured along this solution path. It is given by,

Solution path = [branch_date$_n$, branch_date$_{n-1}$,..., branch_date$_i$,....,0]

The input to this algorithm is the list of variables (after being multi-executed) in the query. By default, it starts to construct the different solution paths starting from the root, which we give a date 0. At each level, it retrieves from the tree information the succeeding successors, in terms of branch-dates. Selecting one successor at a time, it builds a solution path. The algorithm favours the left-most successor. After the completion of this solution path, the same operation is repeated on all brothers.

Each time before the system appends a new branch-date to the solution path under construction, it cross checks with the failures' database to ensure that no partial success/failures occured in this branch. If a failure is recorded, then this branch, and accordingly this solution path, are omitted, otherwise it continues ,deep-wise, the construction of the solution path.

A complete path is detected when no more successors exist for the current branch, and that no failures were encountered along the constructed path. If this is the case, then the system

displays the solutions of the variables corresponding to this solution path. Afterwards it resumes the above routine.

This algorithm follows the standard execution model of Prolog it favours the first succeeding son, i.e. left-most. This is why, it was expected to produce the same number of solutions as that produced by Prolog and in the same order as well, as it adopts the same search criteria.

This approach is a valid approach, it produces the same solutions as standard Prolog, in order and number. The drawback of such an approach is the long tedious algorithm that occupies a considerable execution time. We compared the processing time versus the time required for the display of the solutions of such a method when running large benchmarks, and it was quite surprising. It almost took about 60-70% of the execution time to display the solutions. This motivated us to try to display the solutions alternatively without the dependance on the resolution tree.

More details on the model performance are given in chapter 6.

V.5.1.3.2 Not necessarily same order nor same number of solutions as Prolog

In this case, the different instantiations of a variable are considered. Possible permitted combinations between the instantiations produce the different solutions. For each solution, the system checks if the date-paths of any sub-term in the multi-instantiation has previously resulted in a partial success/failure. The manner in treating these failures is more clearly discussed in the following chapter which is dedicated to the treatment of different types of failures.

Anyway, once a failure is encountered, then this solution is abandonned, otherwise it is displayed to the user.

This algorithm consumes less time than the previous one. The order of the solutions is not the same as that of standard resolution of Prolog. And at the same time, there are cases when this algorithm does not produce the same solutions as Prolog. These are certain redundant cases when not all the solutions are displayed to the user. This redundancy is not eleminated systematically. An example is if X={ **1,** [(1,a),(2,a)]} then two solutions are produced; *X*= *a* and *X*= *a* (similar to Prolog).

Alternatively, if we had the query *X=a , (true;true)*, Prolog displays two solutions, while this algorithm produces only one solution; *X= a*.

V.5.2 fail

As mentioned in section IV.2.1, an explicit failure is a failure due to an explicit *fail* stated in the body of a clause (or goal). When such type of a failure is encountered, an immediate failure is signalled to the system. All variables instantiated in this alternative will be deinstantiated and we say that the clause (or goal) has failed.

The impact of an explicit fail is the same in either models; the standard resolution model or the multi-resolution model.

V.5.3 The !

It is a predefined predicate that affects the procedural behaviour of the program. Its main function is to reduce the search space of Prolog computations by dynamically pruning the search tree. The cut can be used to prevent Prolog from following fruitless computation paths that the programmer knows could not produce solutions. It is used to commit a choice. In terms of tree representation, the cut eleminates all the part that includes OR nodes that are present in the sub-tree whose root is the predecessor of the cut, including the predecessor subgoal, if it belongs to an OR node itself. We call this part in the resolution tree the scope of a cut.

Example:

For the program,

```
p(a) .
p(b).
p(c).
q(x).
q(y).
q(X):- p(X),!.
q(z).
```

solve the query,

```
:- q(X) .
```

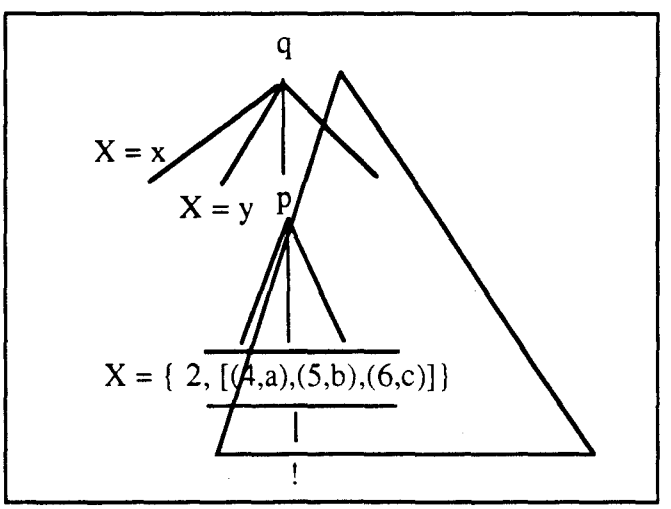


Figure (V.7) : Scope of a cut in standard resolution

We will explain how standard resolution of this program takes place.

The goal *q* is multi-unified with first clause head, resulting in the instantiation of *X* to *a*. The second alternative produces another solution. The third alternative is a rule, hence the body clauses are multi-resolved before proceeding with the following alternative.

The first subgoal is p(X). A choice point exists with three alternatives, producing a multi-instantiation X = {2,[(4,a),(5,b),(6,c)]}.

The next subgoal is a cut. The scope of this cut is to prune the brothers of the predecessor if this latter belongs to an OR node, which is the case for *p(X)*. Hence, we are faced with a problem due to the complete exploration of a choice point without knowing that a cut will be encountered further in the mutli-resolution.

What we propose is a mechanism to set partial success/failures for the solutions that should not have been computed. In our example, it means that we signal the sub-terms (5,b) and (6,c) as failing sub-terms. These failures are stored in the failures' database in terms of the date-path of each sub-term and the failing branch-location as its branch-location respectively. The failures of these examples are stored as shown in fig. (V.8).

[(2, 5)]	[]	[(2, 5)]
[(2, 6)]	[]	[(2, 6)]

Figures (V.8) : Failures' database

V.5.4 not(X)

The *not(X)* succeeds if an attempt to satisfy *X* fails, and vice versa. In the multi-resolution mode, a multi-instantiated variable *X* will propagate in the multi-resolution with all its sub-terms. Any partial success/failures are stored in the failures' database. When a *not* is encountered, a special treatment is required to process the failing values as well as the succeeding values.

The constructive not is possible to be implemented since the relational predefined predicates are treated. not(X= value) may be translated as a sequence of =\=(sub-term of X,value),

...,=\=(sub-termn,value). The results of the inequality tests are stored in the failures' database and the multi-resolution continues normally.

The case of not(subgoal) is more complex to treat. Here, a proposal may be that once a *not* is encountered, a success database is created to store the succeeding multi-terms in terms of the date-paths and the branch-locations (similar to the failures), and the multi-execution completes normally afterwards. At the multi-outputs module, this success database is converted to the global failures' database and the solutions are produced.

Example:

p(a). p(b).

For the query,

:- p(X), p(Y), not(X=Y).

X and *Y* are multi-instantiated to

X = { **1,** [(1,a), (2,b)]} , Y = { **2,** [(3,a), (4,b)]}

For the last subgoal, the success values are stored in the success database as follows,

[(1,1)]	[(2, 3)]	(0, 0)
[(1,2)]	[(2, 4)]	(0, 0)

Figures (V.9) : Success' database

At the multi-outputs module, the success' database records are appended to the failures' database records. When the combinations of *X* and *Y* are checked in the failures' database, all the sub-terms that succeeded in the not will fail to be displayed.

The role of this intermediate database is to be local to the *not* operator that created it. If several *not*s are encountered, we might need to alter the success databases several times. We care that these artificial failures will not be considered as real failures except at the multi-outputs phase.

V.6 Conclusion

In this chapter, we presented how certain predefined predicates are treated in the multi-resolution model, namely arithmetic predicates. Here, data-copying is employed. We presented the corresponding multi-unification algorithm for arithmetic operations pointing out the necessity to perform the coherency tests and the treatment of some partial success/failures. Following, we presented a brief discussion on the input/output predicates. We presented how solutions are displayed to the user. Finally, a discussion of the possibility of treating other predefined predicates such as the cut and the not is presented.

Chapter Six

Model Performance

Abstract

In this chapter, the meta-interpreter of the multi-resolution model as well as that of the standard resolution model are presented. The performance parameters upon which the comparison is based are discussed. We terminate with experimental results and comments when running different benchmarks.

VI.1 Introduction

An advantage offered by Prolog is that it allows a very concise description of its interpreter using the language itself. Here, we present two meta-interpreters: one for the multi-resolution model and the other for the standard resolution model. The first is written to observe the performance of the multi-resolution model and the second for comparison reasons.

Actually, we started writing the meta-interpreter of the multi-resolution model since the very first phase of this work, even before the model definition was concrete. It was serving us as an interactive tool to ameliorate the performance of the model. As a matter of fact, we were faced by all the problems that we discussed in chapter 2 when we ran different benchmarks that made us continuously modify our model until its the final form has evolved. This meta-interpreter may be considered as a prototype for testing out the implementation of the new features proposed by the multi-resolution model.

Both meta-interpreters are written in LPA MacProlog on an Apple Macintosh (8MRAM). A question might be why did we use this machine even with its constraining memory and speed? Just because it was the most available machine for frequent testing. Nevertheless, we kept in mind that, in future, this meta-interpreter will run on a workstation, hence we were keen to assure its portability.

First, we present a comparison between the complexity of the standard unification algorithm to that of the multi-unification algorithm, concerning the cases that handle multi-instantiations. Following, we present a discussion of both meta-interpreters together with the parameters used to compare the performance of the multi-resolution model to that of the standard resolution model. Finally, results for different benchmarks are presented.

VI.2 Complexity of the multi-unification algorithm

In chapter 3, we have discussed in detail the different cases of multi-unifying two multi-terms during the multi-resolution of a Prolog program. It is evident that by eliminating the classic deep backtracking of standard Prolog, the number of unification processes decreases significantly but with an added factor of complexity due to the presence of multi-instantiated variables. To imagine the complexity of the algorithm, the added cases to the standard unification algorithm concerning multi-instantiated variables are shown in table(VI.1).

X and*Y* are variables, *x* and *y* are atoms, x_i and y_i are variables or atoms, $\{C_i,...\}$ represents a multi-instantiation and d_i is the branch-date of a sub-term x_i. ':=' is an assignment operation, and '=' represents a multi-unification operation. The first two columns are the two multi-terms to be multi-unified, the third indicates the actions that will take place and the last two columns illustrate the complexity of the corresponding actions. The complexity is given by the product of two terms: the order of the number of times a unification (or multi-unification) operation takes place by the order of the complexity of each of these operations.

First term	Second term	Action	Complexity Standard Res.	Complexity Multi-Res.
X	{Cy,...}	X:={Cy,...}	$\geq O(n)* O(1)$	$O(1)*O(1)$
{Cx,...}	Y	Y:={...}	$\geq O(n)* O(1)$	$O(1)*O(1)$
x	$\{Cy,[(d_1,y_1),...,(d_n,y_n)]\}$	$x=y_j$ pour i=1-n	$\geq O(n)* O(1)$	$O(1)*O(n)$
$\{Cx,[(d_1,x_1),...,(d_n,x_n)]\}$	y	$y=x_j$ pour i=1-n	$\geq O(n)* O(1)$	$O(1)*O(n)$
$\{C,[(d_1,x_1),...,(d_n,x_n)]\}$	$\{C,[(d1,y_1),...,(dn,y_n)]\}$	$x_i=y_j$ pour i=1-n	$\geq O(n^2)* O(1)$	$O(1)*O(n)$
$\{Cx,[(d_1,x_1),...,(d_n,x_n)]\}$	$\{Cy,[(d_1,y_1),...,(d_m,y_m)]\}$	$x_i=y_j$ pour i=1-m et j=1-n	$\geq O(n*m)*O(1)$	$O(1)*O(n*m)$
$f(\{C,[(d_1,x_1),...(d_n,x_n)]\})$	g(t)	failure	$\geq O(n)*O(1)$	$O(1)*O(1)$

Table (VI.1): Complexity of the multi-unification algorithm

In the above table, we neglect the complexity resulting from the coherency tests and the checks in the failures' database. We discuss each of the above cases.

1- An unbound variable and a multi-instantiation:

The first two cases represent the unification process between an unbound variable and a multi-instantiated variable. Normally, in standard unification, the variable is instantiated to a value, resolution continues with this value, then, with deep backtracking, the old value is undone and the following value, respecting the leftmost rule, is bound to that variable and so on.

In multi-unification, all possible instantiations are assigned directly in one step to that variable. Hence, the number of unification operations is equal to *1* and at the same time, the complexity of the unification process is in the order of *O(1)*. In the standard case, the number of unification operations is greater than *1*, (evenmore, at least *n* times) due to the deep backtracking, with the complexity of each operation in the order of *O(1)*. Thus, we can

say that a remarkable speedup may be achieved when multi-unifying *an unbound variable to a multi-instantiated variable*. This is a very frequent case in common Prolog programs.

2- A multi-term (nonvariable) and a multi-instantiation:

We come to the next two cases when unifying any multi-term (other than a variable) with a multi-instantiated variable. Again, examining the classical case, this operation is done step by step, unifying the multi-term with one value at a time, i.e. for a multi-instantiated variable of n instantiations, at least n unification operations will take place, each having a complexity in the order $O(1)$. In the multi-resolution model, there is one multi-unification operation that invokes n multi-unifications to perform the desired operation. In this case, the complexity of the multi-unification process will be in the order of $O(1)*O(n)$.

It is worth noting that the above complexity when multi-unifying a nonvariable multi-term to a multi-instantiated variable, is the lower bound of the complexity of the unification operations occuring in a standard Prolog environment, due to the effect of deep backtracking. This could be explained as follow:

n represents the number of multi-instantiations, which may never exceed the number of alternatives at this choice point solving the subgoal. If standard resolution is considered, these n alternatives will be investigated each time a backtracking occurs. Assuming that this was the unique choice point in the program, then the number of unification operations attempted is equal to n. Now if there is at least another choice point in the program, then deep backtracking takes place and the order jumps to $O(2n)$. If more choice points exist, then the complexity increases due to the fails and deep backtracking that will occur. The order is $O(n*m)$, where m are the number of choice points, and n is the average number of alternatives.

We conclude that though the complexity in multi-unifications could be sometimes high, yet it is still the lower bound of the corresponding unification operations in the standard resolution.

3- Two multi-instantiations:

The more complex case comes when multi-unifying two multi-instantiated variables. Here, we have two classes, multi-unifying two multi-instantiated variables that belong to the same choice point or to two different choice points.

a- Same choice point:

For the former case, it is simple. It is always a single multi-unification operation between the corresponding pairs of sub-terms (same branch dates), i.e. the first element of the first multi-instantiation with the first element of the second multi-instantiation, and so on. In this case n tests are made for two multi-instantiated structures of length n, i.e, a complexity of $O(1)*O(n)$.

There are cases, that we discussed, when the two multi-instantiations are not of the same lengths. In this case, the order $O(1)*O(n)$ is the upper bound of the complexity of the multi-unification between two multi-instantiations that belong to the same choice point.

In the standard resolution, the number of unification operations is in the order of $O(n^2)$, due to deep backtracking, with a complexity of $O(1)$ for each operation. Thus, the complexity is quadratic in the case of the standard unification wheras it is linear in the multi-unification. This presents a considerable speedup, that will be shown when running the benchmarks.

b- Different choice points:

For the latter case, to respect the semantics of Prolog, the complexity of the multi-unification is more complex as a multi-unification operation is invoked between each sub-term of the first multi-instantiation structure and all the sub-terms in the second multi-instantiation. For two multi-instantiated variables of length m and n, the complexity of the multi-unification algorithm is in the order of $O(1)*O(m*n)$.

Now considering a standard resolution, we will have two choice points; one having m branches and the other having n branches. Due to fails and deep backtracking, this unification operation takes place at least $O(m*n)$ times. This is the lower bound when compared to the standard resolution. If other choice points exist, then deep backtracking will take place and the above number of unification operations are repeated.

5- Two different functors:

Here, we consider the case of multi-unifying two different functors where the argument of one is multi-instantiated. A failure is signaled due to the difference in the functors' names. The gain in the multi-resolution is that this test will take place once for all the different multi-instantiations, whereas in the standard algorithm it will be repeated several times depending on the number of backtracks, i.e. with a complexity $\geq O(n)*O(1)$ where n is the number of sub-terms in the multi-instantiation. This failure could be considered as one of the sources of

the acceleration in the multi-unification algorithm. In chapter 3, example 2, section III.3.5.1 (page III.13) discussed a similar example.

We can conclude that the multi-unification algorithm supports the multi-unification between any two multi-terms. It is always a single multi-unification operation, with a complexity that depends upon the nature of these multi-terms but is always inferior or equal to the complexity of its corresponding unification operations in the standard algorithm.

VI.3 The meta-interpreter of the multi-resolution

We present a brief description of the meta-interpreter that we built to examine the performance of the multi-resolution model. It is basically a Vanilla-interpreter. The input is a Prolog program (Edinburgh syntax) and a query. The output is the solution(s) to that query.

The two phases of the multi-resolution model (the multi-execution phase and the multi-outputs phase) are defined. The multi-unification algorithms for arithmetic and non arithmetic operations, together with the coherency checks and the failures checks are written. Moreover, some predefined predicates are treated.

To compare the performance of the multi-resolution model, we wrote the meta-interpreter of the standard model, where the unification algorithm is defined and the ensemble of solutions are displayed to the user.

Following we present the different parameters that we selected for the observation of the behaviour of the multi-resolution and how such parameters were measured.

VI.3.1 Performance parameters

Two aspects are always interesting when comparing two approaches; time and memory. To compare the performance of the multi-resolution model with respect to the standard resolution model, we chose the following parameters:

1- Speedup (S):

We define the speedup by the ratio between the time taken by standard Prolog to solve a query to the time taken by the multi-resolution model to multi-resolve the same query.

Time measurements are managed by a system clock. Each clock count represents 1/60 of a second. We read the clock before and after resolving a query for the standard resolution, and before and after each phase in the multi-resolution of the same query.

2- Theoretical Speedup (TS):

It was inspired from [24]. It represents a ratio between the number of branches of the standard resolution tree to that of multi-resolution tree that multi-resolves the same problem. It indicates the difference in the search spaces in the two models.

The search space size is computed by means of a counter that counts the number of succeeding alternatives to produce the solutions in either models.

3- Memory consumption (Mc):

It is the memory consumed to resolve (or multi-resolve) a query.

We measure the evaluation space before and after the resolution (or multi-resolution) of a query by the aid of the predicate *eval_space*. We assume that the difference is the memory consumed to produce all the solutions.

VI.3.2 Experimental results

Before discussing the speedups and memory consumption, we present some experimental results that clarify the behaviour of the different phases of the multi-resolution model. All program listings are presented in appendix A.

VI.3.2.1 Model performance

Our first objective was to understand the performance of the multi-resolution model. The time elapsed to multi-resolve a query is actually divided into two times: the time taken to multi-execute the query and the time taken to display all the solutions to the user. We were interested to observe the difference between both durations that indicates the behaviour of the model. For more details, we also compared between the times taken to display all the solutions by the two proposed algorithms that we discussed in chapter 5.

1- bits(n)

It is inspired from [28]. It is a recursive program that generates a list of *n* elements. The value of each element is either 0 or 1. Here, there are no failures.

Program	Multi-execution	Multi-outputs1	Total time1	Multi-outputs2	Total time2
bits(3)	1.28	1.23	2.51	0.75	2.03
bits(4)	1.6	2.6	4.2	1.6	3.2
bits(8)	2.18	3.7	5.88	2.6	4.78
bits(10)	4.55	290.1	294.6	157.6	162.15
bits(12)	5.6	1545.6	1551.2	1421.9	1427.5
bits(16)	8.55	5479.2	5487.75	3802.9	3811.5
bits(20)	13.1	91145.35	91158.45	63991.8	64004.9

Table (VI.2): Different time durations (in seconds) in the different phases during the multi-resolution of bits(n).

In the above table, the first column indicates the time taken to multi-execute the query, i.e. the time taken to scan the subgoals, to perform the multi-unifications, to create the multi-instantiations and to construct the multi-resolution tree. The third and fifth columns represent the time taken to display all the solutions by the two different algorithms given in chapter 3. The fourth column is the sum of the second and third columns. The sixth column is the sum of the second and the fifth columns.

It is clear that the multi-execution of the query does not consume a considerable time as that taken to display the solutions to the user. For bits(20), we find that the multi-execution time represents 0.01% of the total time!

The figures in the third column and the fifth column indicate the difference in the complexities of the 2 algorithms. When the degree of the problem is small (*bits(3), bits(4),..*), we do not really sense the difference in performance between the two algorithms. As the degree of the problem increases, the second algorithm proves to be more fast.

The above figures justify why we were motivated to define another algorithm for the display of the solutions independent of the reconstitution of the standard resolution tree.

2- bits-palindromic(n)

The previous program was a simple generator of elements in a list. *bits-palindromic(n)* adds the intelligent reverse algorithm after the generation of the *n* elements of the list. It reverses the list to obtain another symmetric list. Adding this reverse algorithm is a sort of a 'test' that will result in several failures. We observe the behaviour of the different phases as shown in table (VI.3).

Program	Multi-execution	Multi-outputs1	Total time1	Multi-outputs2	Total time2
bits-pal(8)	5.8	16.05	21.85	9.9	15.7
bits-pal(10)	8.3	47.6	55.9	18.8	27.1
bits-pal(12)	10.03	301.6	311.63	114.05	124.08
bits-pal(16)	25	882.7	907.7	317.5	342.5

Table (VI.3): Different time durations (in seconds) in the different phases during the multi-resolution of bits_palindromic(n).

Again, we observe the difference in the time consumed between the two algorithms of the display of solutions. We also notice the difference between the time taken to multi-execute a query and the time taken to display the solutions.

The time taken to produce the solutions by the reconstitution of the standard resolution tree is smaller than in the previous example since failures have occured (in the *reverse* subgoal) that reduced the size of the tree. In the multi-resolution, these failures are stored in the failures' database. During the reconstitution of the standard resolution tree, this information is very useful as it signals the algprithm to avoid the creation of the solution paths that proved to be failing.

The above runs proved that the first algorithm for the display of solutions produces the same solutions produced by standard Prolog (same order and same number).

VI.3.2.2 Speedups

Comparing the time taken by the multi-resolution to the time taken by the standard resolution model, we observe the speedups in the different programs. Since we have already two algorithms to display the solutions, we were interested to compare the worst-case speedups to the best-case speedups.

1-bits(n)

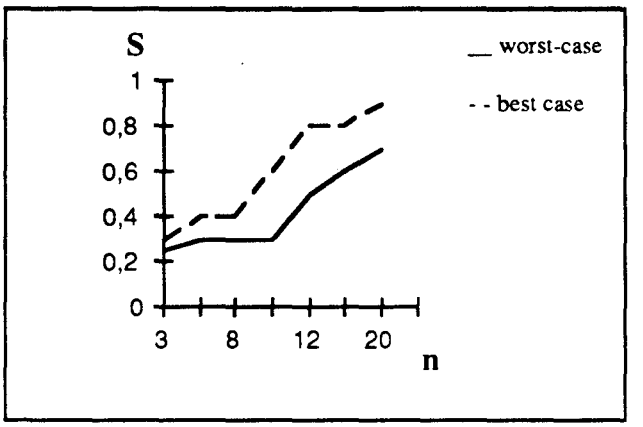


Figure (VI.1); Worst-case and best-case speedups for the multi-resolution of bits(n)

In this particular example, we remark that no speedup is attained in either cases. This is somehow reasonable: *bits(n)* is a recursive program that generates a list. A large percentage of the multi-resolution tree resembles the standard resolution tree. In other word, when the same tree is traversed in both models, the standard resolution proves to be more performant. This is due to the overhead presented by the multi-resolution model before and after each choice point. Even though afterwards, a certain number of deep backtracking take place, yet the resulting gain does not override the previous overhead.

Our first observation is that simple *generate* programs are not the most suitable problems suitable to be run in the multi-resolution model. Another example is the *permute* problem that will be discussed later in this section.

2-bits-palondromic(n)

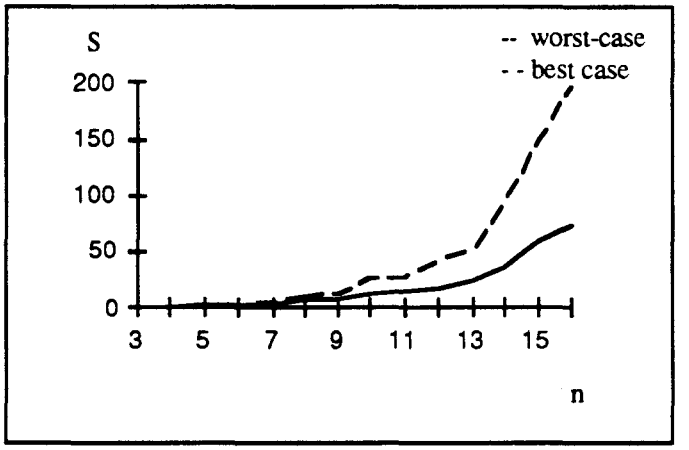


Figure (VI.2): Worst-case and best-case speedups for the multi-resolution of bits-palindromic(n)

As mentioned in the previous section, *bits-palindromic(n)* is the *bits(n)* problem with an added intelligent reverse algorithm. When multi-resolving this query, interesting speedups are achieved. To understand why, we return to the program source for the *reverse* subgoal:

```
reverse(First_list,Second_list):- reverse(First_list,[],Second_list).
reverse([First_term/Rest],Intermediate_list,Second_list):-
        reverse(Rest,[First_term/Intermediate_list],Second_list).
reverse([],List,List).
```

We will consider the case of *bits_palindromic(4)* for simplicity. Analysing more closely the call to the *reverse* subgoal in the multi-resolution, the generated list *L* to be reversed is given by:

L = [{9,[(14,0),(15,1)]},{8,[(12,0),(13,1)]},{7,[(10,0),(11,1)]},{6,[(8,0),(9,1)]}]

This is a list of 4 elements, each of which is multi-instantiated to two different values (either 0 or 1). The sub-terms are similar, i.e. same size and same structure. This makes further multi-procesing simpler. We will discuss this parameter again later in this section.

The main *reverse* subgoal will invoke **4** recursive *reverse* subgoals to reverse the list *L*. As we previously said, the tree corresponding to a recursive program is the same in both models. But, in the multi-resolution model, these **4** recursive calls are enough to reverse all the possibilities of the list *L*, whereas in the standard resolution, deep backtracking takes

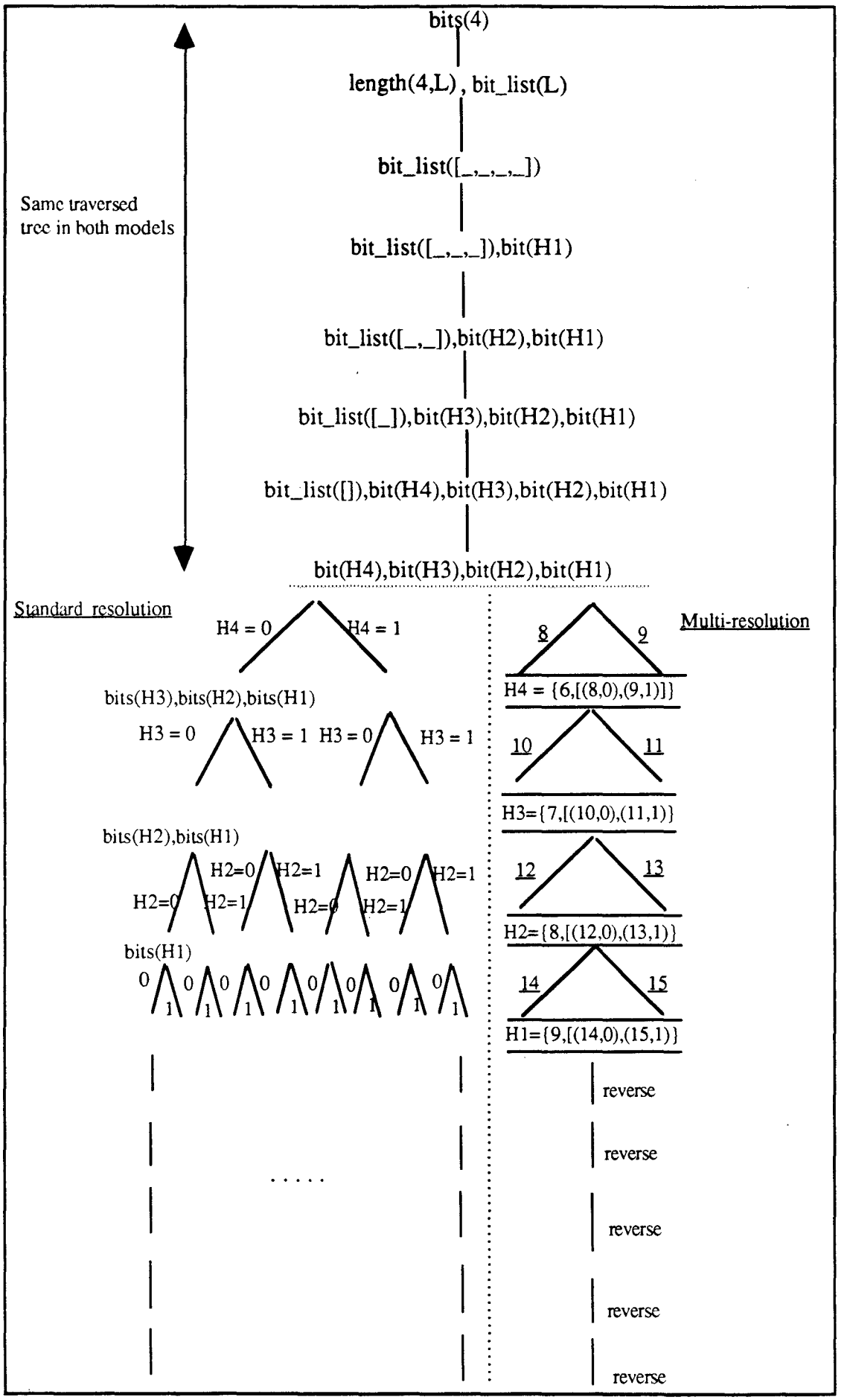


Figure (VI.3): Comparison between the multi-resolution tree and the standard resolution tree for bits-palindromic(n)

place to repeat the creation of the list *L* to be reversed. Fig.(VI.3) points out the resolution tree for both models.

Coming to the symmetry test, the corresponding subgoal is:

reverse([],[{9,[(14,0),(15,1)]},{8,[(12,0),(13,1)]},{7,[(10,0),(11,1)]},{6,[(8,0),(9,1)]}],
[{6,[(8,0),(9,1)]},{7,[(10,0),(11,1)]},{8,[(12,0),(13,1)]},{9,[(14,0),(15,1)]}])

Two lists, each having multi-instantiated elements are to be multi-unified. A multi-unification operation takes place between the corresponding elements as follows:

multi-unify({9,[(14,0),(15,1)]},{6,[(8,0),(9,1)]})
multi-unify({8,[(12,0),(13,1)]},{7,[(10,0),(11,1)]})
multi-unify({7,[(10,0),(11,1)]},{8,[(12,0),(13,1)]})
multi-unify({6,[(8,0),(9,1)]},{9,[(14,0),(15,1)]})

The first two multi-unification operations will produce all the necessary failures. It is at this step where the acceleration is attained with respect to the standard resolution. To obtain the same information (success+failures) in the standard resolution, deep backtracking takes place to undo and reinstantiate variables (*H1* to *H4*) to create the elements that will constitute the list *L* to be reversed and then at the very end (last branch in the tree) the failure takes place.

We chose this benchmark to compare the performance of our model to MultiLog[38,39,40] where this example proves to be very performant. For the problem *bits(20)*, they attained a speedup of **12**. This speedup is real as sequential MultiLog is implemented. Observing our speedup curves in both cases (worst-case and best-case), we find that even in the worst-case, we achieve a speedup of **72.9** for *bits(16)*. In the best-case measures, this jumps to **196.6**. Both speedups indeed do not result from an implementation, but, nevertheless, they are encouraging.

This example points out one of the adequate classes of programs to be run in the multi-resolution mode. They are the programs that generate a large number of multi-instantiations of simple and similar shapes followed by a test procedure that select the solutions.

3-bits-naive-palindromic(n)

When the naive reverse algorithm is employed instead of the intelligent one, more interseting speedups are obtained. The same analysis as for the previous example may be applied for reasoning.

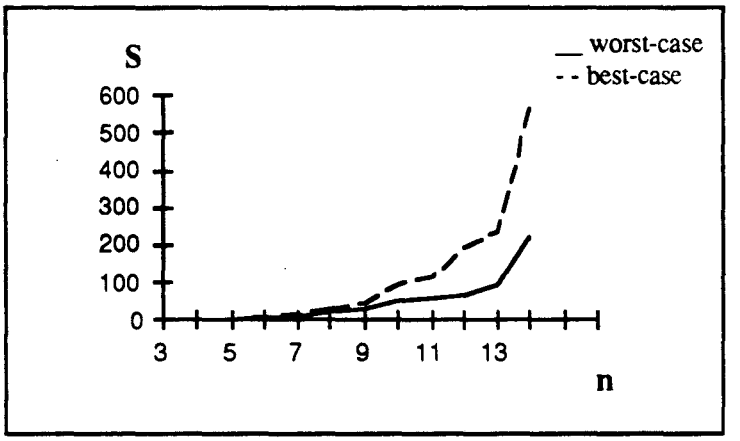


Figure (VI.4): Worst-case and best-case speedups for the multi-resolution of the bits-naive-palindromic(n)

4-permute([a1,...an],R)

An example to show that again the *generate* class of programs may not run efficiently in the multi-resolution model is the *permute* problem. Recursion is used to generate the permutations. Here, the standard search tree and the multi-resolution search tree resemble each other to a great extent making it irrelevant to adopt the multi-resolution model due to the overhead at each choice point. This explains why no speedups were produced.

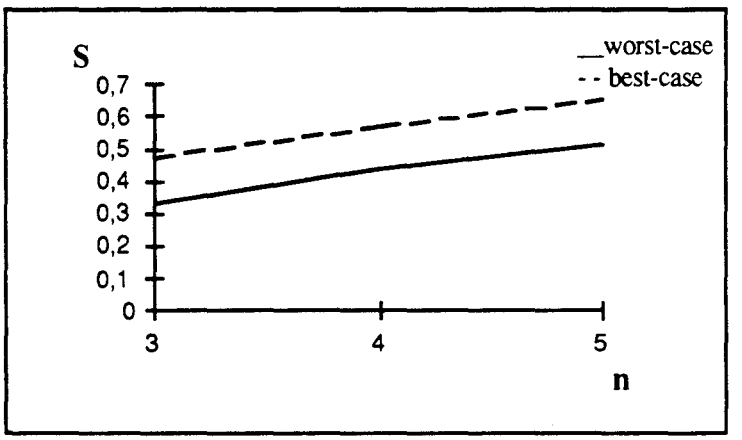


Figure (VI.5): Worst-case and best-case speedups for the multi-resolution of permute([a1,...an],R)

5- Other cases:

Here, we present a number of other benchmarks that we tried. We measure the multi-execution time only and compare it to the standard resolution time.

Algorithm	Standard-resolution	Multi-execution phase
quicksort	744.3	29.7
naive reverse	49765.93	282.55
intelligent reverse	28659.7	241
mutation	116.8	124.3

Table (VI.4): Standard resolution time and multi-execution time (in seconds)

We cannot conclude any concrete results from the above table as we are neglecting the delay resulting from the display of solutions (all the measurements in the previous section consider the display delay).

The only case where these results may be indicative is the case of the *mutation* problem. Here, no reduction of time is achieved even after the multi-execution phase. To understand why, it is important to understand how the query *mutation(M)* is multi-resolved. The first *animal* will multi-instantiate *X* to **11** animals. The second *animal* will multi-instantiate *Y* to **11** animals. Examining these multi-instantiations more closely the structure of the different sub-terms constituting the multi-instantiation. Actually, *X* will be given by:

X = {1,[(2,[a,l,l,i,g,a,t,o,r]),(3,[t,o,r,t,u,e]),(4,[c,a,r,i,b,o,u]),(5,[o,u,r,s]),
(6,[c,h,e,v,a,l]),(7,[v,a,c,h,e]),(8,[l,a,p,i,n]),(9,[p,i,n,t,a,d,e]),(10,[h,i,b,o,u]),
(11,[b,o,u,q,u,e,t,i,n]),(12,[c,h,e,v,r,e]),}

We notice that the sub-terms are not similar; though they are all lists, yet their sizes are not equal. This will reflect on the failures that will occur in the following two append subgoals. Partial success/failures will occur and multi-instantiations resulting from the same choice point will have different lengths resulting in very complicated resolution. The overhead of failures' checking is very costly in this case.The size of the failures' database is too large resulting in a longer checking procedure. Adding to this the overhead to display the solutions (which is not considered in the above table), it is clear that the *mutation* problem, though it passes in the multi-resolution model, yet no speedups are expected.

On the other hand, there are cases, when the multi-outputs phase is not invoked, but the resulting measurements are true. This is the case of clause queries, where the user

interrogates the program about the validity of a certain solution. Here, there are no variables in the query header, accordingly, the display procedure will not be invoked. We made use of this to observe the time taken to produce the first and last solutions in both models. We tested the *quicksort* algorithm in this case.

Query	Standard resolution	Multi resolution
:- sorted_list([a,a,a,a,a]).	674.7	29.3
:- sorted_list([z,z,z,z,z]).	692.7	31.75

Table (VI.5): Resolution time and multi-resolution time (in seconds)

sorted_list produces a sorted list. It invokes the creation of a list, the elements of which may have different values, then passes this list (which will be multi-instantiated in the multi-resolution) to the *quicksort* subgoal. The reader may refer to appendix A for the tested programs listings.

Speedups are attained because the elements are multi-instantiated from the very beginning. In the equality tests, the test takes place on all the multi-instantiations. In the standard resolution, the whole solution path is repeated for every combination of the 5 elements in the list *L*.

VI.3.2.3 Recapitulation

From the above discussion, it is clear that the multi-resolution accepts any program as an input program, in other words, no restriction on the programming style (recursion, base of facts, rules, arithmetic operations,..). After several runs, we observed that eliminating deep backtracking may sometimes result in more non redundant work with respect to the standard resolution model. The cases that achieved highest performance were the case when the different sub-terms of a multi-instantiations are similar: same size and same structure. Moreover, adding a test procedure after a generation of similar multi-instantiations proved to attain promising speedups. These two parameters are necessary for an optimal performance of the multi-resolution.

VI.3.2.4 Theoretical speedups

Here we compare the search space in the multi-resolution to that in the standard resolution. By eliminating backtracking, it is obvious that size of the multi-resolution tree is far less than that of the standard resolution tree. What interests us here is to show the significant drop and

at the same time to observe the increase of the search space with the increase of the complexity of the problem.We will only discuss the *bits-palindromic(n)* problem in full details.

bits-palindromic(n)

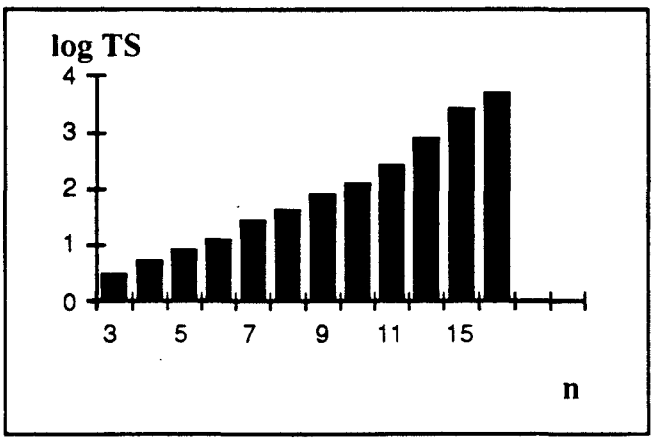


Figure (VI.6) : Theoretical speedups (logTS) for bits-palindromic(n)

Program	*Standard resolution*	*Multi-resolution*	*Theoretical Speedup (S)*
bits-pal(3)	53	16	3.31
bits-pal(4)	97	20	4.85
bits-pal(5)	205	24	8.54
bits-pal(6)	373	28	13.32
bits-pal(7)	765	32	23.9
bits-pal(8)	1421	36	39.47
bits-pal(9)	2877	40	71.93
bits-pal(10)	5469	44	124.3
bits-pal(11)	11005	48	229.27
bits-pal(13)	42749	56	763.4
bits-pal(15)	167933	66	2544.4
bits-pal(16)	332029	68	4882.8

Table (VI.6): Sizes of the standard resolution tree and the multi-resolution tree and the resulting theoretical speedups.

Examing the increase in size of the multi-resolution tree with the increase of *n*, we observe that it follows a linear behaviour. Each time *n* increases, the multi-resolution goes one step

deaper to create the elements of the list, and the resulting element is instantiated either to 0 or to 1. This will, in return, result in one more *reverse* subgoal.

This explains the increasing factor of **4** in the size of the multi-resolution tree. Whereas this increase is exponential in the case of the standard resolution. This can be proved as follows: Consider figure (VI.3), here the two trees, the standard resolution tree and the multi-resolution tree are presented for the query *bits-palindromic(4)*. The subgoal *bits_list(L)* is a recursive subgoal that terminates when there are no more elements in L, i.e function of *n*. This is the same in the multi-resolution of the same subgoal. When we come to the *bit* subgoal, the trees will start to differ. It is starting from here that the multi-resolution differs from the standard resolution. Given *n* elements in the list *L*, each having one of two different values (*0* or *1*), the standard resolution will create a sub-tree of size equal to:

$$2^1 + 2^2 + 2^3 + \ldots + 2^n$$

whereas the size of the corresponding sub-multi-resolution tree is **2*n**.

Consequently, the number of *reverse* subgoals (branches) that are called in the multi-resolution is linear (**n+1**) depending on the value of *n*, whereas the number of *reverse* subgoals (branches) invoked in the standard resolution is $(n+1)*2^n$.

Accordingly, the resulting theoretical speedups are exponential with the increase of complexity of the problem (*n*).

This demonstrates another time how the multi-resolution model reduces its search space significantly with respect to the standard resolution.

VI.3.2.5 Memory consumption

In any system, to achieve a speedup, the sacrifice is the memory space. This is the compromise to ameliorate the performance. We tried to measure the difference between the memory occupancy when we resolve a query to that when we multi-resolve the same query. In the case of standard resolution, the memory consumption is very small relative to that consumed in the multi-resolution. An example of different measures for different tested programs are shown in figs (VI.7) and (VI.8).

Analysing these measures to understand such performance, we find that standard Prolog, which follows a depth-first search occupies a memory space in the order of *O(d)* where *d* is the depth of the search tree.

In the multi-resolution model, the depth of the multi-resolution tree is the same as that of the standard resolution tree $O(d)$. Since we explore all the alternatives of a choice point, before passing to the following chopice point, the complexity of the memory space occupied in the multi-resolution is in the order of $O(b*d)$ where b is the branching factor. In a breadth-first search tree, this is in the order of $O(b^d)$.

This shows that our model lies somewhere in between the depth-first search and the breadth-first search. It follows a quasi breadth-first search without the enormous evolution of the breadth-first strategy (due to the added synchronous OR phases), which was our objective when we adopted the sharing strategy of multi-instantiations.

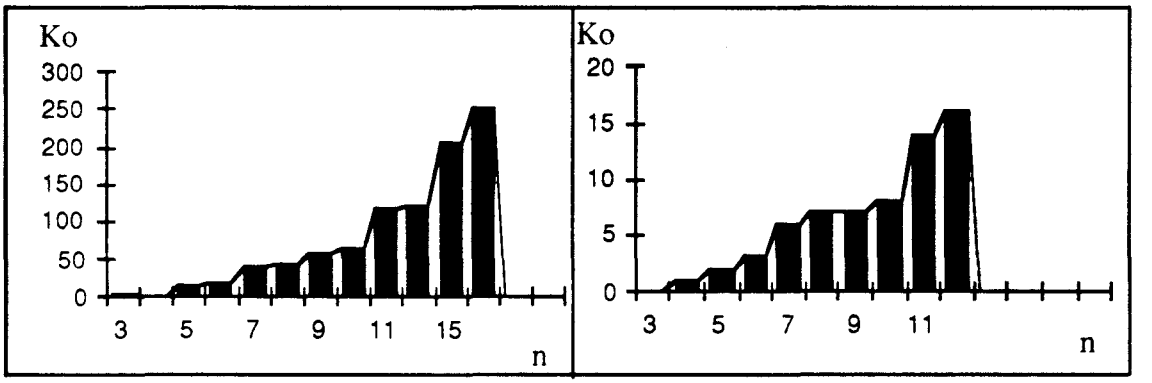


Figure (VI.7a) : Memory consumption during the multi-resolution ofbits-palindromic(n)

Figure (VI.7b) : Memory consumptionduring the standard resolution of bits-palindromic(n)

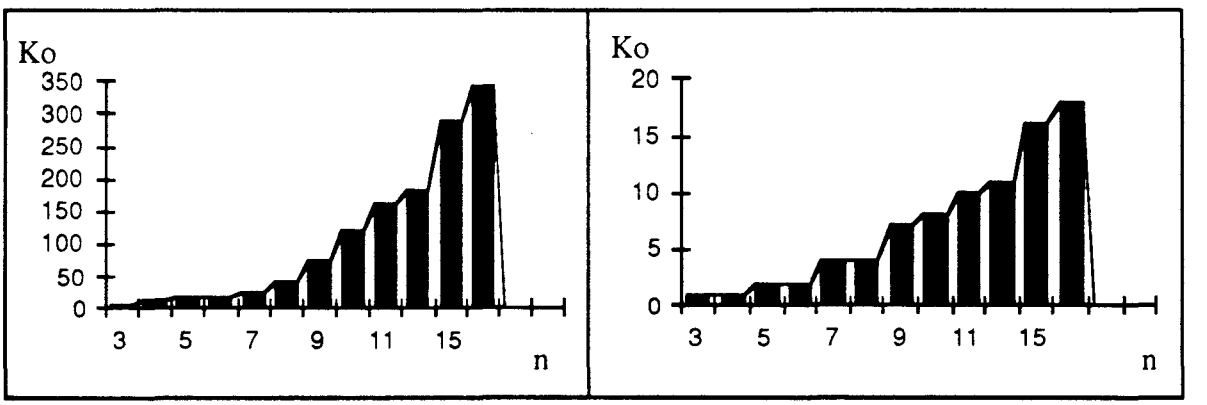


Figure (VI.8a): Memory consumption for the multi-resolution of bits-naive-palindromic(n)

Figure(VI.8b): Memory consumption during the standard resolution of bits-naive-palindromic(n)

VI.4 Conclusion

In this chapter we presented the meta-interpreter of the multi-resolution model with which we studied its performance and compared it to the standard resolution. Running different programs proved that our proposed model accepts any Prolog program irrespective of its programming style. The different runs showed that the behaviour of the multi-resolution model is very promising for the class of programs that generate many alternatives (different in value, but similar in structure) followed by a test procedure that performs a number of tests collectively on the generated multi-instantiations. Results also guided us to the case where the standard resolution is favoured which is the class of recursive programs where the traversed path is almost similar in both models. Of course, programs that do not include a significant number of deep backtracks are not expected to produce any speedups when compared to the standard resolution. More complex benchmarks may be multi-resolved by running this meta-interpreter on a more powerful machine.

Chapter Seven

Summary and Perspectives

Abstract

After having detailed the different phases of the proposed multi-resolution model in this terminating chapter we will try to situate the work presented among different related research topics. In the closing of this work, we summarise up what was done to make the point. The objectives, a brief highlight of the proposed model, and the results are concretised. We terminate with a discussion of the relevant future work.

VII.1 Introduction

In the previous chapters, we discussed clearly the multi-resolution model for the execution of logic programs written in Prolog (Edinburgh syntax). We explored the different features of the model and we showed how we substituted the deep backtracking feature, of the standard resolution model, by multi-instantiations, in the multi-resolution model.

The multi-resolution model proposed in the previous chapters aimed to reduce the search space when evaluating a Prolog program by eliminating the deep backtracking which may result in repetitive unification operations. In achieving this objective, multi-instantiations were introduced that, as we discussed in chapter 3, are based on a certain dating system. Failures are memorised aside and are processed when needed.

In the following section, we compare different aspects of the presented multi-resolution model with respect to related domains. Following, we highlight several perspectives concerning this work. As a termination, we present a general conclusion of the work presented in this thesis.

VII.2 Related work

The main objective of this work was to reduce the amount of work done to produce the solutions to a given query. In achieving this objective, the deep backtracking feature was eliminated. A combined depth-first, breadth-first strategy was defined. Synchronisation between different subgoals was introduced to assemble the different solutions in what we called a multi-instantiation. Failures' information was memorised aside.

Several aspects of this model, starting from the objectives until the dating mechanism employed to multi-instantiate the variables, can be discussed in relation with already existing research work. We discuss several points.

VII.2.1 Reducing the amount of work

This objective has been the aim of numerous research topics related to logic programming. Certain approaches tackled the problem of optimising the behaviour of the backtracking.

Other domains proposed another manner relying on constraints. Even in certain parallel models, certain techniques were defined to reduce the amount of redundant work done to achieve a certain result. Following, we highlight different related topics.

VII.2.1.1 A more intelligent backtracking

This line of research was dealing with the optimisation of the behaviour of backtracking. By optimisations, we mean related research where the objective was to reduce the number of times backtracking took place. Intelligent and semi-intelligent backtracking techniques are the most concerned in this aspect.

As mentioned in chapter one (section I.8.4), intelligent backtracking was introduced independently by Cox [19] and Perira et Porto [35,36] to reduce the number of backtracks. This approach is actually related to the occurence of failures in the standard execution model. When a failure takes place, the system analyses the cause of that failure. By 'analyses' we mean that the system detects the instantiation that caused this failure and memorises it. This is to achieve 2 objectives: first to avoid repeating the same failure another time, and secondly to result in an efficient backtracking, i.e. to the choice point that alters this instantiation. An example that illustrates this idea was presented in chapter 1, page I.21.

This idea was first implemented in Prolog interpreters that resulted in a considerable overhead. A first attempt to introduce intelligent backtracking in a Prolog complier was proposed by Lin et al. [30,31]. Codognet et al. extended the WAM architecture to include the intelligent backtracking feature. An extended unification-related instructions was introduced mainly to rememorise the source of the bindings. They proposed the DIB machine [12] then the WAMIB [13] that resulted in a more efficient implementation where speedups upto a factor of 10 for nondeterministic programs are achieved.

Semi-intelligent backtracking was proposed by Hermingildo et al.[34], where it realised intelligent backtracks in the case of independent AND subgoals. Analysis is more simple due to the fact that variable are not shared between different subgoals. This was discussed before in chapter 1, section I.8.4.2.

VII.2.1.2 Constraint Logic Programming Languages (CLP)

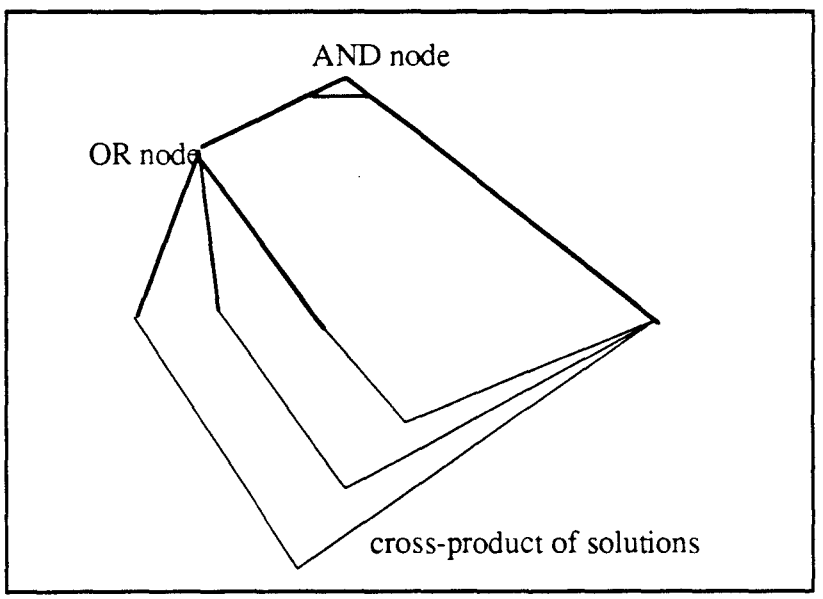
Another approach that shares our main objective is the class of constraint logic programming languages[15]. This class generalized logic programming by replacing the unification

algorithm by a general mechanism called constraint satisfaction. This set of constraint equations may be used actively to prune the search space. Given the current constraints of a rule, the resolution accumulates a number of constraints from the resolution of different subgoals. When the system fails to satisfy these constraints, a failure is signalled and backtracking takes place. CLP languages treat a wider domain of arithmetic equations than standard Prolog.

Firebird is a parallel constraint logic programming language based on finite domain constraints. Its execution model bears some similarity to our multi-resolution model. In a non-deterministic derivation step, a choice point based on any of the domain variables in the system is set up and all possible values in its domain are attempted in parallel. By this, thousands of finite domain constraints can be solved in a single step [44].

VII.2.1.3 Hybrid parallel models

These are the parallel models that comprise more than one type of parallelism. A model that shares our objective in reducing the work done to produce the solutions is the class of models that include the OR and independent AND parallelism. For an AND node as that shown in fig. (VII.1), the right hand branch is computed only once. The resulting solutions are reused instead of backtracking. This results in a reduction of the search space[6].



Figure(VII.1): Reusing already produced solutions by different OR branches, in the case of independent ANP parallelism

The transfer of solutions between the different branches results in an overhead irrespective of the binding mechanism adopted.

VII.2.1.4 DAP Prolog

DAP (Distributed Array Processor) Prolog is a data parallel logic programming language, which utilises finite domains that exploits the parallelism of an SIMD machine, namely ICL DAP. It is an extension of the Prolog language, with two new data structures; *sets* and *arrays*. In the DAP Prolog set mode, a set-oriented interpreter adopts a mixed depth-first/breadth-first search strategy in which the multiple fact branches of a conventional Prolog search tree are considered as generating bindings rather than search non determinism. DAP Prolog distributes the database over the processors of the target machine and implements unification of constants on a within processing element basis [26].

DAP Prolog necessitates a certain programming style for efficient treatment: databases are favoured than sequential list searching. This is because it is easy to execute the different alternatives of the database in parallel to create the sets of solutions.

In the multi-resolution model, both styles are allowed. There are cases when the sequential creation of lists, for example, did not lead to any speedups, but with the addition of a test procedure to this same list generator encouraging speedups were achieved, (section VI.3.2.2).

VII.2.1.5 MultiLog

MultiLog is a parallel logic programming language that runs on the MasPar MP1 with 8192 processsors, using DEC 5000 (MIPS) workstation as the front end running ULTRIX V4.2A. In MultiLog, certain goals are annotated with the unary operator *disj*. The goal *disj G* indicates that all or some subset of the solution to *G* should be collected and turned into a set of environments (disjunction of substitutions) [38,39,40].

To illustrate the difference in the environnement management between MultiLog and the multi-resolution model, consider the following example.

Example:

```
p(a).   p(b).   p(c).
:- disj(p(X)), disj(p(Y)).
```

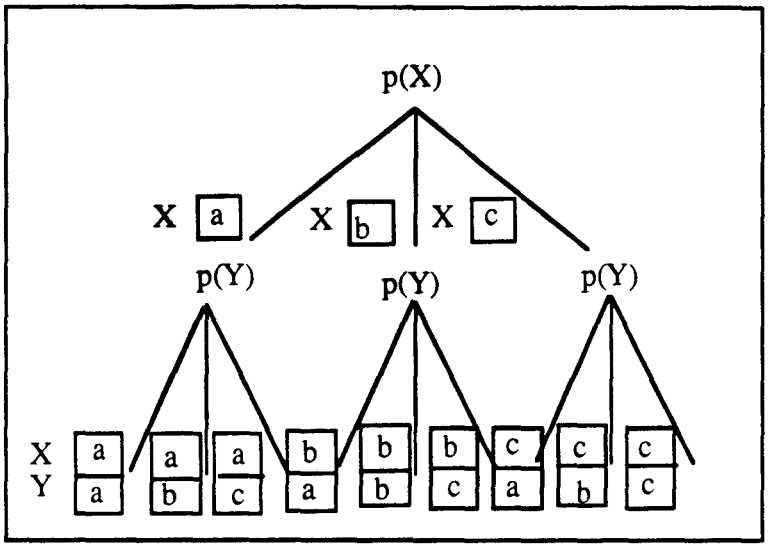


Figure (VII.2): Treatment of multiple environments in MultiLog

Here, the resolution depends on the copying strategy. Each time a new alternative (breadth-wise) is selected a copy of the already existing environment is created. In MultiLog, *X* and *Y* are multi-variables, each having **9** different instantiations. Conceptually, multi-variables in MultiLog are different from the multi-instantiations in the multi-resolution model. The latter are shared structures all throughout the multi-resolution, and include non redundant instantiations. Fig. (VII.2) points out the difference in the memory management.

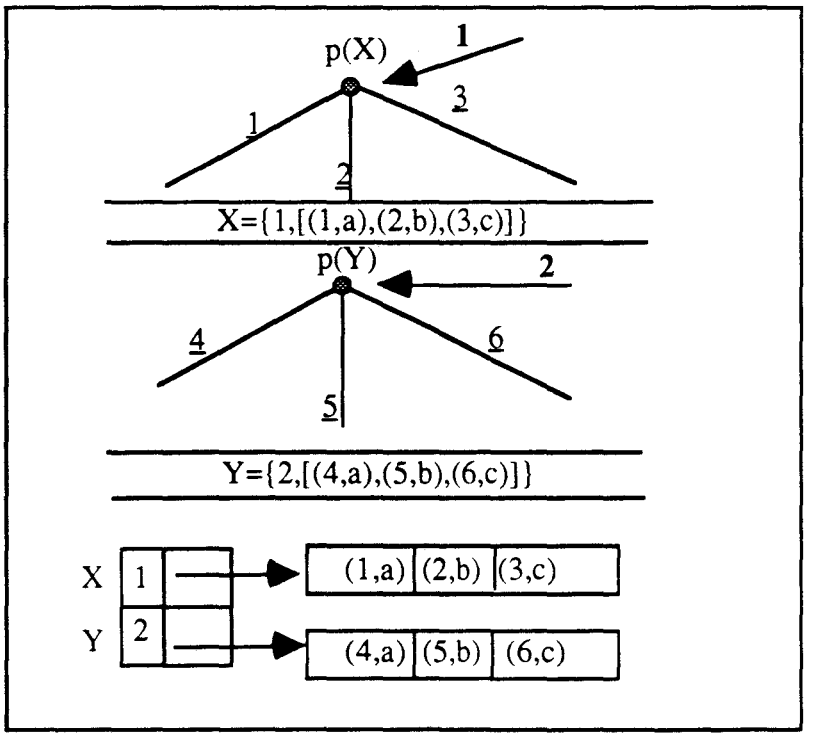


Figure (VII.3): Single environment in the multi-resolution model

In the multi-resolution model, the different solution of different alternatives are collected in the local synchronous OR phase for each choice point to create multi-instantiations. These multi-instantiations are shared among all the alternatives of the following subgoals.

In [38], the Multi-WAM architecture is defined, which is suitable for sequential execution as well as parallel execution (SIMD or MIMD target machines).

VII.2.2 Multiple Bindings of Variables

In chapter 3 (section III.2), we presented the manner by which a multi-instantiation is created. Different instantiations are tagged by the branch-date, which is a unique identifier, and the ensemble of instantiation-pairs are tagged by the choice point number that created this multi-instantiation.

It is worth noting that in the multi-resolution model, there only exists one multiple environnement, that includes mono-instantiations (i.e. normal Prolog instantiations) and multi-instantiations (definition 3.1, page III.3). In other research domains, namely dealing with parallelism, there exist multiple environnements. Similar dating mechanisms exist. We enumerate several proposed models that are analogous to our approach.

VII.2.2.1 Multi-sequential models

Multi-sequential models are models based on parallel execution, but with a finite (limited) number of resources (processors - workers). Accordingly, the execution of a Prolog program takes place by combining 2 strategies: a parallel breadth strategy and the classical sequential strategy.

A main feature of the multi-sequential models is the sharing of a part of the resolvant. If a free variable exists in this part, then a writing conflict may occur as different processes are going to bind the same variable in the shared area to different bindings. Different propositions to treat such multiple environnements are proposed, based on copying of environnements, duplicating previous computations, or sharing of environnements.

In the first two approaches, each process possesses a copy of the shared part of the resolution tree. In the third approach, only one copy of this shared part of the tree exists. Accordingly, a certain control mechanism is required that serves for 2 purposes: first to allow

the representation of different bindings of the same variable and secondly to control the coherency of the bindings accessed by each process. We are interested in the third approach which bears some similarity to our way of representing the variables. The definition 3.1, page III.3 was given to assure several aspects given that the environnement is shared in the multi-resolution model and hence a certain mechanism was required to access a binding of a variable and validate its coherency. Following is a number of different models in this domain.

1- PEPSys:

In this model, a marking technique is defined that allows to validate the bindings [49]. All bindings are tagged by a number called OBL (OR Branch Level) that corresponds to the number of choice points created by the process where this binding took place.

For a local variable, shallow binding takes place where the value of the variable is tagged by the current OBL. Each process possesses a hash-window where for a non local variable the triplet (variable, value, current OBL) is memorised.

To access the value of a local variable (dereferencing operation), the same mechanism used in the sequential model is employed. It is the case of the dereferencement of a non local variable which is more complex.

First, a validation process takes place that compares the field OBL of that binding with the age of the choice point that was created by the ancestor process that resulted in the process that is performing the dereferencing operation. If this test fails, then search takes place in the hash-window of the current process. If this latter fails too, a second search takes place all along the chain of the hash windows of the ancestor processes that might have bound that variable.

Consider the following example:

Example:

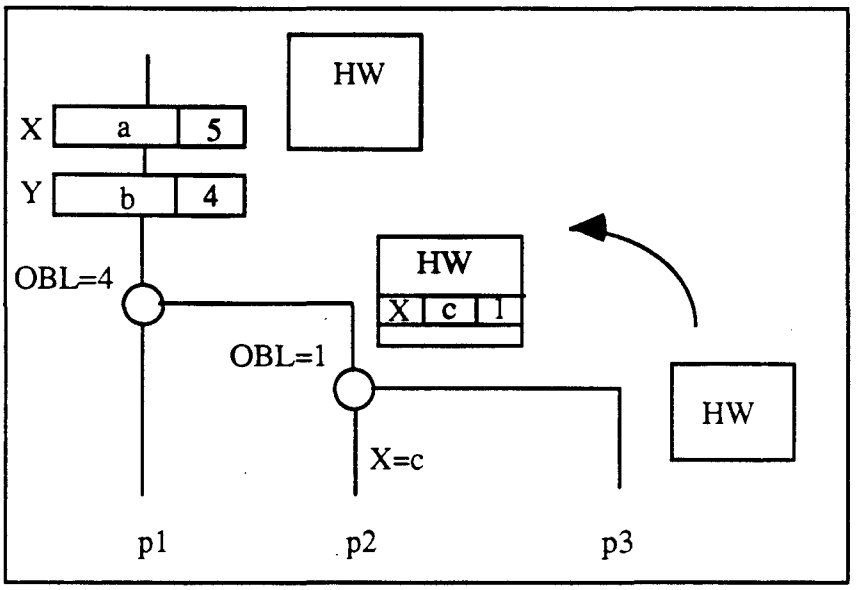


Figure (VII.4): Binding mechanism in the PEPsys model

First, the cell representing the variable is accessed. The binding X=a is not valid since the OBL that tags this value (5) is superior than the age of the choice point (4) that created the process P3. The search terminates since the chain of the hash windows of the ancestor processes is reduced to an element. Hence, the result is that the variable X is free (unbound) for the process P3.

This technique, though does not necessitate no data-copying, yet with the increase in the number of hash-windows, the chaining search becomes costly.

VII.2.2.1.2 SRI

Here, each processor possesses a *processor binding array* where all conditional bindings are stored [46,47]. A variable bound conditionally will include an index that points out its value in the different processor binding arrays.

The following example illustrates the binding procedure.

Example:

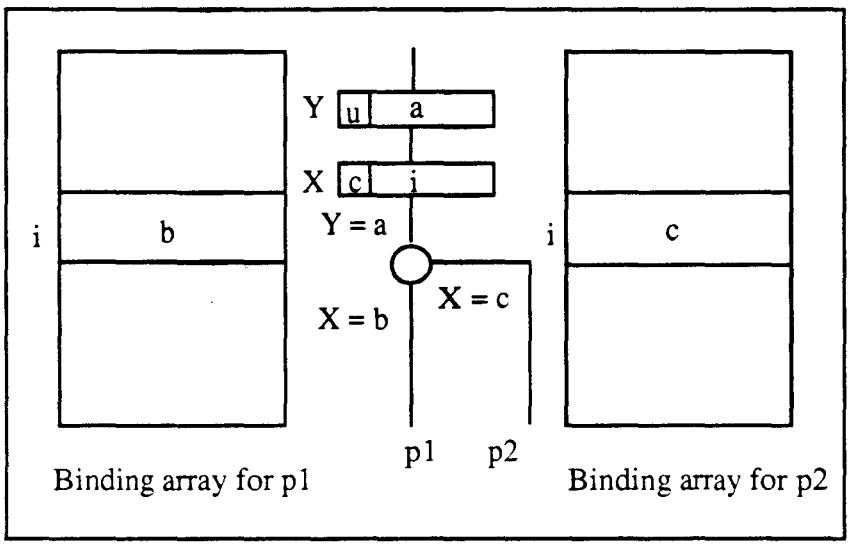


Figure (VII.5): Binding mechanism in the SRI model

In the above figure, P1 and P2 bind conditionally the variable X to b and c respectively. (Y=a is a universal binding). When the variable X is dereferenced, the index i that contains the cell that represents the variable X, is the key to the binding arrays for the dereferencement process.

The access time of a variable is constant, but the main disadvantage of this approach is the overhead of the creation of a new parallel task. Aurora [48] is an implementation of the SRI model. Actually, it offers the most efficient implementations of parallel OR existing models.

VII.2.2.1.3 Vectors Versions

It is a variant of the SRI model. Here, the different bindings of variables is represented by vectors. The length of this vector is the number of the processors in the system, and hence each cell contains the value of a binding that took place in the corresponding processor.

A variable bound conditionally is bound to a vector, the length of which is teh number of processors, and hence each cell will include a binding value.
The recopying of the binding arrays of the SRI model is translated here by the update of the vectors by the bindings that take place in the branch where an alternative is found.

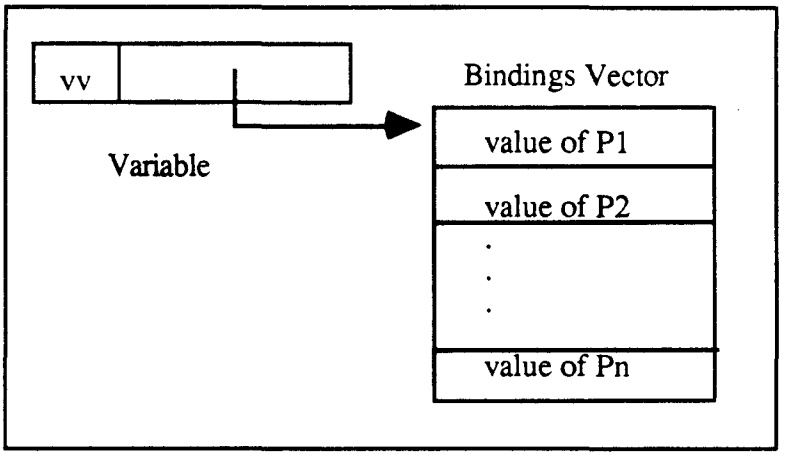


Figure (VII.6): Bindings mechanism in the versions-vector model

This technique requires a synchronisation for the creation of vectors. Actually, each vector is created dynamically by the process that binds the variable conditionally for the first time. This necessitates the synchronisation of the processes that bind simultaneously the same free variable.

VII.3 Perspectives

Generally speaking, there are two main lines for future work based on the work presented in this thesis: the first is exploiting the inherent parallelism and the second is the implementation of the sequential (and eventually the parallel) model. We briefly discuss each of these perspectives.

VII.3.1 Parallelism in the multi-resolution model

We return to our sequential multi-resolution model. We have a system that is characterised by the presence of multi-instantiated variables that replaced deep backtracking. Subgoals are traversed from left to right. Each subgoal is traversed once and only once, by attempting all its alternatives sequentially and collecting the ensemble of solutions. Afterwards, this subgoal is never reattempted.

Recalling our objective, which was a detailed study of the treatment of the multi-instantiated variables, we tend to study the influence of such variables on the execution from the point of view of data parallelism.

The promising speedup figures presented in the previous chapter, mainly resulted from the elimination of deep backtracking. We focus on the multi-unification algorithm that handles all the operations manipulating a multi-instantiated variable. The nature of the multi-unification algorithm is sequential, i.e. the invoked multi-unification operations take place sequentially in the order of the sub-terms. Our approach is to exploit the data parallelism at this phase. The idea is simple :

> *Performing a multi-unification operation between a multi-instantiation (**n** sub-terms) and another multi-term (irrespective of its nature) may invoke in parallel **n** multi-unification operations between each sub-term of the former with the latter.*

This is *data* parallelism, where the same operation (here, multi-unification) is performed on each sub-term of the multi-instantiation. In other words, the same subgoal is evaluated in parallel with different arguments. Since we succeeded in presenting the multi-instantiations in a vector-like form, we may exploit data parallelism given such a representation.

This is an appealing idea due to the fact that the model, as it is, allows the data parallelising of certain phases, without any additional modifications in the syntax, and so the already written programs may be executed in parallel. Here, no added load will bother the user as it is not his role to 'think' in a parallel way. The parallelism may be exploited in a transparent manner. By this, the corpus of Prolog programs that are already developed can be executed without any modification to the programs' source code by a parallel machine. This view confirms the formula "program = logic + control" [28].

Example :

multi_unify({**1** , [(1,a) , (2,b) , (3,c) , (4,d) , (5,e)]},c)

In chapter 3, we discussed how this multi-unification operation takes place. Since the first term is a multi-instantiated object, a multi-unification operation is invoked between each multi-instantiation of the first term and the second term, resulting in the following operations:

multi_unify({1 , [(1,a) , (2,b) , (3,c) , (4,d) , (5,e)]},c)

|_____| <----- multi-unify((1,a),c) then

|_____| <----- multi-unify((2,b),c) then

|_____| <----- multi-unify((3,c),c) then

|_____| <----- multi-unify((4,d),c) then

|_____| <----- multi-unify((5,e),c).

In our previous description of the algorithm, these operations take place in a sequential order, in the order of the sub-terms in the multi-instantiation.

Our data parallel approach is to perform the above 5 multi-unification operations in parallel since they are independent in nature. By this, a more interesting speedup is expected as we are tending to ameliorate the performance of the model after eliminating the deep backtracking and its corresponding overhead.

VII.3.2 Implementation of the multi-resolution model

Another perspective of the presented work is the implementation of the multi-resolution model. An extension of the WAM is the next step in this work to allow a real execution of the multi-resolution model. We highlight the main required modifications.

Here, the search strategy is a combined breadth-first, depth-first strategy. On the level of a subgoal, the implementation is WAM-like except for a certain number of points:

- variables may be multi-instantiated, accordingly, new unification instructions should be added to support all the different multi-unification operations, incuding the treatment of failures which will differ in the case of partial success/failures.

- environment treatment in the scope of a choice point: after the termination of an alternative (multi-unification of a clause head and multi-execution of the body clauses) the WAM continues with the following subgoal. In the proposed extension, it is required to save temporarily the different solutions resulting from the different alternatives, until no more alternatives exist.

- the synchronisation level that assembles all these temporary solution to create multi-instantiations (single environment) and releases the above temporary solutions.

On a more global level, the mechanism for treating backtracking between different subgoals is not required (at least, for the case of finite programs).

VII.4 General Conclusion

In this thesis, we have presented a new model for the multi-execution of Prolog programs, which we called the multi-resolution model. Our objective was to improve the execution time of Prolog programs. Our proposed model is characterised by the elimination of the deep backtracking feature defined in the standard model. It respects the Edinburgh syntax and hence no modification in the already existing program sources is required. It is most suitable for finite non deterministic programs when all the solutions are of interest.

The multi-resolution model follows a quasi breadth-first search; each subgoal is attempted only once, unifying it to the different clause headers sequentially (since we preserved the shallow backtracking). The body clauses of the current clause head are executed before the following clause head for the same subgoal. A synchronisation level is added after the exploration of each subgoal to assemble all the different solutions and create the multi-instantiations. Multi-resolution never returns backwards to an already attempted subgoal.

In such a model, there is one environment that is multi-instantiated. This is because a sharing strategy is adopted that is shared among the different alternatives. The only exception is the case of arithmetic operations, where data-copying is employed. To perform the multi-unification operations, we defined two multi-unification algorithms for arithmetic and non arithmetic operations. The presence of such variables also influenced on treatment of failures during the multi-resolution of a query, which we treated as well.

Two meta-interpreters were written, the first was to justify and clarify the objective of this work and the second to observe our proposed model. Both meta-interpreters are written in Prolog.

The first helped us to better understand the problem by studying closely the backtracking phenomenon in the standard model. The results of this simulator illustrated the overhead of the backtracking.

The second validated the sequential multi-resolution model. Both phases of the multi-resolution (multi-execution and multi-outputs) are defined. The multi-unification algorithm was written for arithmetic and nonarithmetic operations, and the two algorithms for the display of solutions were also included. Partial success/failures were treated and several predefined predicates were introduced.

We made use of this meta-interpreter to observe the behaviour of the model by introducing several performance parameters. We were interested in time and memory measurements. We compared the performance of our model with that of another meta-interpreter of standard Prolog.

The different results pointed out that the multi-resolution model is promising compared to the standard model. First, no prohgramming style is restricted. The multi-resolution model treats recursion, databases and arithmetic operations. There are certain classes of programs that proved to be more performant than others. This includes the *generate* and *test* programs. The *generate* subgoal creates the multi-instantiations and the *test* operates on the different sub-terms of these multi-instantiations without deep backtracking resulting in encouraging speedups. We also observed that the performance is better when the different sub-terms of a multi-instantiation are similar (same structure, same size).

On the other hand, in queries where only shallow backtracking takes place in the standard model, the standard model is normally more performant. This result was expected due to the overhead in the multi-resolution model after each choice point to create the multi-instantiations as well as the overhead of the algorithm of the display of solutions.

What we presented in this thesis is the base of the definition of the multi-resolution of Prolog programs. There are still several lines of research that are to necessary to be studied. First of all, some fine-tuning in the actual model is required. One is the proposition of a more efficient treatment of the failures' database for a more optimal performance. Another point is the introduction of the treatment of some predefined predicates such as the assert/retract, not,etc.

A second aspect is to consider the implementation of this model, by introducing the necessary extensions to the existing WAM to adopt multi-resolution in the sequential mode.

Considering the parallel aspect, we discussed the possibility of exploiting data parallelism in the multi-resolution model due to the fact that multi-instantiations are represented in a vector-like form making it possible to execute the multi-unification algorithm on a parallel target machine (SPMD machine or eventually an MIMD machine).

Given the sequential model, it is very motivating to proceed in the details of the parallel multi-resolution model together with all the implementation problems as well.

Appendix A

Tested Benchmarks

1- append:

```
append([],L,L).
append([A1|A],B,[A1|C]):- append(A,B,C).
```

2- permute:

```
permute([],[]).
permute([H|T],L):- permute(T,L1), insert(H,L1,L).

insert(A,B,[A|B]).
insert(A,[B1|B],[B1|C]):- insert(A,B,C).
```

3- quicksort:

```
quicksort([H|T],L):-
   partition(T,H,Less,More),
   quicksort(Less,L1),
   quicksort(More,M1),
   append(L1,[H|M1],L).

partition([X|Xs],Y,[X|Ls],Bs):- X≤Y, partition(Xs,Y,Ls,Bs).
partition([X|Xs],Y,Ls,[X|Bs]):- X>Y, partition(Xs,Y,Ls,Bs).
partition([],Y,[],[]).
```

4- intelligent reverse:

```
reverse1(L1,L2):- reverse1(L1,[],L2).

reverse1([H|T],A,L):- reverse1(T,[H|A],L).
reverse1([],L,L).
```

5- naive reverse:

```
reverse2([],[]).
reverse2([H|T],L):-
   reverse2(T,L1),
```

```
    append(L1,[H],L).
```

6- bits(n):

```
bits(n):- create_list(n,L), bit_list(L).

bit_list([]).
bit_list([H|T]:- bit_list(T), bit(H).

bit(0).
bit(1).
```

7- bits-pal(n):

```
bits_pal(n):- create_list(n,L), bit_list(L), reverse1(L,L).
```

8- bits-naive-pal(n):

```
bits_naive_pal(n):- create_list(n,L), bit_list(L), reverse2(L,L).
```

9- mutation(M):

```
mutation(M):- animal(X), animal(Y),
    append([A1|A2],[B1|B2],X),
    append([B1|B2],[C1|C2],Y),
    append(X,[C1|C2],M).

animal([a,l,l,i,g,a,t,o,r].
animal([t,o,r,t,u,e]).
animal([c,a,r,i,b,o,u]).
animal([o,u,r,s]).
animal([c,h,e,v,a,l]).
animal([v,a,c,h,e]).
animal([l,a,p,i,n]).
animal([p,i,n,t,a,d,e]).
animal([h,i,b,o,u]).
animal([b,o,u,q,u,e,t,i,n]).
animal([c,h,e,v,r,e]).
```

10- Sorted_list:

```
sorted_list([A,B,C,D,E]):-
    instantiate_each_term([A,B,C,D,E]), quicksort([A,B,C,D,E]).

instantiate_each_term([]).
instantiate_each_term([H|T]:- instantiate_each_term(T), letter(H).

letter(a).
letter(b).
.
.
letter(z).
```

References

[1] Barklund, J. Parallel Unification. Ph.D. thesis, Uppsala university, 1990.

[2] Boizumault, P. Prolog: L'implémentation. Masson, 1988.

[3] Bourzoufi, H. Définition et Evaluation d'une Machine Abstraite Parallèle pour un Modèle Ou-parallèle Multi-séquentiel de Prolog. PhD. thesis, LIFL, USTL, 1992.

[4] Bruynooghe, M. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters 12*, 1981.

[5] Chassin de Kergommeaux, J., Codognet, P., Robert, P. and Syre, J.C. Une programmation logique parallele : Les langages gardés. TSI, September 1989.

[6] Chassin de Kergommeaux, J. and Codognet, P. Parallel Logic Programming Systems. Technical report, INRIA, Mai 1992.

[7] Chassin de Kergommeaux, J., Codognet, P., Robert, P. and Syre, J.C. Une revue des modeles de programmation logique parallele : systemes paralleles logiques non deterministes. TSI, September 1989.

[8] Clark, K.L. and Tarnlund, S. Logic Programming. Academic Press, London, 1982.

[9] Clark, K.L. and Gregory, S. Parlog : parallel programming in logic. *Comm. of ACM*, vol. 1, pages 1-49, 1986.

[10] Clark, K.L. Parallel Logic Programming. *The computer journal,* vol. 33, no. 6 1990.

[11] Clocksin, W.F. and Mellish, C.S. Programming in Prolog. Springler-Verlag, New York, 1981.

[12] Codognet C., Codonget P. and Filé, G. Yet Another Intelligent Backtracking Method. *ICLP*, 1988.

[13] Codognet, P. and Sola, T. Extending the WAM for Intelligent Backtracking. *ICLP*, 1991.

[14] Cohen, J. A View of the Origins and the Development of Prolog. *Comm. of the ACM*, vol. 31, no. 1, Jan. 1988.

[15] Cohen, J. Constraint Logic Programming Languages. *Comm. of the ACM*, vol. 33, no. 7, July. 1990.

[16] Colmerauer et al. Un systeme de communication homme-machine en francais. Research report, Groupe intelligence artificielle, Université AIX-Marseilles II, France, 1973.

[17] Condillac, M. Prolog: Fondements et Applications. Dunod, 1986.

[18] Conery, H.S. Parallel Execution of Logic Programming. Kluwer Academic Publishers, 1987.

[19] Cox, P.T. Deduction plans, a graphical proof procedure for the first order predicate calculus. Ph.D Thesis, Dept. of Computer Science, University of Waterloo, Canada, 1977.

[20] Delahaye, J.P. Introduction a la Programmation Logique aux Systemes Experts et au Langage Prolog. Publication LIFL, France,1985.

[21] Genesereth, M.R. and Ginsberg, M.L. Logic Programming. *Comm. of the ACM*, vol. 28, no. 9, Sep. 1985.

[22] Goncalves,G., Hannequin, I., Lecouffe, P. and Toursel, B. Une Nouvelle Exécution OU-Paralléle de Prolog sur la Machine LOG-ARCH. Technical report, LIFL, France, 1991.

[23] Hannequin, I. Proposition d'un Modèle d'évaluation Parallèle de Prolog. PhD. thesis, LIFL, USTL, France,1991.

[24] Ismail,H. and Lecouffe, P. Multi-résolution de programmes Prolog. *RENPAR*, Brest, Mai 1993.

[25] Jayaraman, B. and Niar, A. Subset Logic Programming: Application and Implementation. *ICLP*, 1988.

[26] Kacsuk, P. and Bale, A. DAP Prolog: A set-oriented approach to Prolog. *The computer journal,* vol. 30, no 5, 1987.

[27] Kogge, P.M. The Architecture of Symbolic Computers. McGraw Hill, 1991.

[28] Kowalski, R. The Early Years of Logic Programming. *Comm. of the ACM*, vol. 3, no. 1, Jan. 1988.

[29] Lecouffe, P. Prolog: Traces et Parallélisme. Technical report, LIFL, France, 1991.

[30] Lin, Y-J., and Kumar, V. An intellignet backtracking scheme for Prolog. *ILPS* and *ICLP*, 1987.

[31] Lin, Y-J., and Kumar, V. A Data Dependency Based Intelligent Backtracking Scheme for Prolog. *Journal of Logic Programming*, vol. 4, 1988, pp. 165-181.

[32] Lloyd, J.W. Foundations of Logic Programming. Springler Verlag, 1984.

[33] Masuzawa, H. et al. Kabu Wake Parallel Inference Mechanism and its Evaluation. *FJCC*, IEEE, November 1986.

[34] Muthukumar, K. and Hermenegildo, H. Determination of variable dependence information through abstract interpretation. *Proceedings of North America Conference of Logic Programming,* 1989.

[35] Pereira, L.M. and Porto, A. Intelligent backtracking in horn clause programs. Technical report, Universtade Nuova de Lisboa, 1979.

[36] Pereira, L.M. and Porto, A. An interpreter of logic programs using selective backtracking. Technical report, Universtade Nuova de Lisboa, 1979.

[37] Robinson, J.A. A machine-oriented logic based on the resolution principles. *J. ACM 12*, 1965, pp. 23-41.

[38] Smith, D.A. MultiLog : Data OR-parallel Logic Programming. *ICLP*, 1993.

[39] Smith, D.A. and T.Hickey. Multi-SLD Resolution. *LPAR*, 1994.

[40] Smith, D.A. Why Multi-SLD beats SLD (even on a uniprocessor). *PLILP*, 1994.

[41] Sterling, L. and Shapiro, E. The Art of Prolog: Advanced Programming Techniques. MIT Press, 1986.

[42] Succi, G. and Marino, G. Data parallelism in Logic Programming. *ICLP* 1991.

[43] Tick, E. Parallel Logic programming. MIT press, 1991.

[44] Tong, B. and Leung, H. Concurrent Constraint Logic Programming on massively parallel SIMD computers. *ILPS*, 1993.

[45] Turk, A. Compiler Optimisations for the WAM. ICLP, 1986.

[46] Warren, D.H. An abstract Prolog Instruction Set. Technical report 309, SRI International, 1983.

[47] Warren, D.H. The SRI model for OR parallel execution of Prolog. *ILPS*, 1987.

[48] Warren, D.H. et al. The Aurora Or parallel system. *New Generation Computing*. vol 7, 1990.

[49] Westphal, H., Robert, P., Chassin de Kergommeaux, J. and Syre, JC. The PEPsys model: Combining backtracking, and and or-parallelism. *ILPS*, 1987.