Inférence parallèle et processus communicants pour les clauses de Horn : extension au premier ordre par la méthode de connexion

Maria Blanca Ibañez

THESE

Prèsentée à

L'Institut National Polytechnique de Grenoble

Pour obtenir le grade de
Docteur de l'INPG
(arrêté ministériel du 5 juillet 1984)
INFORMATIQUE

par

María Blanca IBAÑEZ

Inférence Parallèle et Processus Communicants pour les Clauses de Horn.

Extension au Premier Ordre par la Méthode de Connexion

Thèse soutenue le 12 mars 1990 devant la commission d'examen.

J. MOSSIERE	**Président**
W. BIBEL **G. COMYN**	**Rapporteurs**
R. CAFERRA **P. JORRAND**	**Examinateurs**

PROFESSEURS DES UNIVERSITES

ENSERG	BARIBAUD	Michel
ENSIEG	BARRAUD	Alain
ENSPG	BAUDELET	Bernard
INPG	BEAUFILS	Jean-Pierre
ENSERG	BLIMAN	Samuel
ENSHMG	BOIS	Philippe
ENSEEG	BONNETAIN	Lucien
ENSPG	BONNET	Guy
ENSIEG	BRISSONNEAU	Pierre
IUFA	BRUNET	Yves
ENSHMG	CAILLERI E	Denis
ENSPG	CAVAIGNAC	Jean-François
ENSPG	CHARTIER	Germain
ENSERG	CHENEVIER	Pierre
UFR PGP	CHERADAME	Hervé
ENSIEG	CHERUY	Arlette
ENSERG	CHOVET	Alain
ENSERG	COHEN	Joseph
ENSEEG	COLINET	Catherine
ENSIEG	CORNUT	Bruno
ENSIEG	COULOMB	Jean-Louis
ENSERG	COUMES	André
ENSIMAG	CROWLEY	James
ENSHMG	DARVE	Félix
ENSIMAG	DELLA DORA	Jean-François
ENSERG	DEPEY	Maurice
ENSPG	DEPORTES	Jacques
ENSEEG	DEROO	Daniel
ENSEEG	DESRE	Pierre
ENSERG	DOLMAZON	Jean-Marc
ENSEEG	DURAND	Francis
ENSPG	DURAND	Jean-Louis
ENSHMG	FAUTRELLE	Yves
ENSIEG	FOGGIA	Albert
ENSIMAG	FONLUPT	Jean
ENSIEG	FOULARD	Claude
UFR PGP	GANDINI	Alessandro
ENSPG	GAUBERT	Claude
ENSERG	GENTIL	Pierre
ENSIEG	GENTIL	Sylviane
IUFA	GREVEN	Hélène
ENSIEG	GUEGUEN	Claude
ENSERG	GUERIN	Bernard
ENSEEG	GUYO T	Pierre
ENSIEG	IVANES	Marcel
ENSIEG	JAUSSAUD	Pierre
ENSPG	JOST	Rémy
ENSPG	JOUBERT	Jean-Claude
ENSIEG	JOURDAIN	Geneviève
ENSIEG	LACOUME	Jean-Louis
ENSIEG	LADET	Pierre
ENSHMG	LESIEUR	Marcel
ENSHMG	LESPINARD	Georges
ENSPG	LONGEQUEUE	Jean-Pierre
ENSHMG	LORET	Benjamin
ENSEEG	LOUCHET	François
ENSEEG	LUCAZEAU	Guy
ENSIEG	MASSE	Philippe
ENSIEG	MASSELOT	Christian
ENSIMAG	MAZARE	Guy
ENSIMAG	MOHR	Roger
ENSHMG	MOREAU	René
ENSIEG	MORET	Roger
ENSIMAG	MOSSIERE	Jacques
ENSHMG	OBLED	Charles
ENSEEG	OZIL	Patrick
ENSEEG	PAULEAU	Yves
ENSIEG	PERRET	Robert
ENSHMG	PIAU	Jean-Michel
ENSERG	PIC	Etienne
ENSIMAG	PLATEAU	Brigitte
ENSERG	POUPOT	Christian
ENSEEG	RAMEAU	Jean-Jacque
ENSPG	REINISCH	Raymond
UFR PGP	RENAUD	Maurice
UFR PGP	ROBERT	André
ENSIMAG	ROBERT	François
ENSIEG	SABONNADIERE	Jean-Claude
ENSIMAG	SAUCIER	Gabriele
ENSPG	SCHLENKER	Claire
ENSPG	SCHLENKER	Michel
ENSERG	SERMET	Pierre
UFR PGP	SILVY	Jacques
ENSHMG	SIRIEYS	Pierre
ENSEEG	SOHM	Jean-Claude
ENSIMAG	SOLER	Jean-Louis
ENSEEG	SOUQUET	Jean-louis
ENSHMG	TROMPETTE	Philippe
ENSPG	VINCENT	Henri
ENSERG	ZADWORNY	François

PERSONNES AYANT OBTENU LE DIPLOME

d'habilitation à diriger des recherches

BECKER	M.	DANES	F.	GHIBAUDO	G.	MULLER	
BINDER	Z.	DEROO	D.	HAMAR	S.	NGUYEN TRONG	
CHASSERY	J.M.	DIARD	J.P.	HAMAR	R.	NIEZ	
CHOLLET	J.P.	DION	J.M.	LACHENAL	D.	PASTUREL	
COEY	J.	DUGARD	L.	LADET	P.	PLA	
COLINET	C.	DURAND	M.	LATOMBE	C.	ROGNON	
COMMAULT	C.	DURAND	R.	LE HUY	H.	ROUGER	
CORNUEJOLS	G.	GALERIE	A.	LE GORREC	B.	TCHUENTE	
COULOMB	J.L.	GAUTHIER	J.P.	MADAR	R.	VINCENT	
COURNIL	M.	GENTIL	S.	MEUNIER	G.	YAVARI	
DALARD	F.						

CHERCHEURS DU C.N.R.S.

1989/90

ALEMANY	Antoine
ALLIBERT	Colette
ALLIBERT	Michel
ANSARA	Ibrahim
ARMAND	Michel
AUDIER	Marc
BERNARD	Claude
BINDER	Gilbert
BONNET	Roland
BORNARD	Guy
CAILLET	Marcel
CALMET	Jacques
CARRE	René
CHATILLON	Christian
CLERMONT	Jean-Robert
COURTOIS	Bernard
DAVID	René
DION	Jean-Michel
DRIOLE	Jean
DURAND	Robert
ESCUDIER	Pierre
EUSTATHOPOULOS	Nicolas
FRUCHARD	Robert
GARNIER	Marcel
GLANGEAUD	François
GUELIN	Pierre
HOPFINGER	Emile
JORRAND	Philippe
JOUD	Jean-Charles
KAMARINOS	Georges
KLEITZ	Michel
KOFMAN	Walter
KRAKOWIAK	Sacha
LANDAU	Ioan
LEJEUNE	Gérard
LEPROVOST	Christian
MADAR	Roland
MERMET	Jean
MEUNIER	Jacques
MICHEL	Jean-Marie
NAYROLLES	Bernard
PEUZIN	Jean-Claude
PIAU	Monique
RENOUARD	Dominique
SENATEUR	Jean-Pierre
SIFAKIS	Joseph
SIMON	Jean-Paul
SUERY	Michel
TEODOSIU	Christian
VACHAUD	Georges
VAUCLIN	Michel
VENNEREAU	Pierre
VERJUS	Jean-Pierre
WACK	Bernard
YONNET	Jean-Paul

SITUATION PARTICULIERE

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSIMAG	LATOMBE	J.Claude	Détachement.....	21/10/198
ENSHMG	PIERRARD	J.Marie	Détachement.....	30/04/198
ENSIMAG	VEILLON	Gérard	Détachement.....	30/09/199
ENSIMAG	VERJUS	J.Pierre	Détachement.....	30/09/198
ENSPG	BLOCH	Daniel	Récteur à c/.....	21/12/198

SURNOMBRE

INPG	CHIAVERINA	Jean	30/09/198
ENSHMG	BOUVARD	Maurice...	30/09/199
ENSEEG	PARIAUD	J.Charles	30/09/199

.../...

PERSONNALITES AGREEES A TITRE PERMANENT A DIRIGER DES TRAVAUX DE RECHERCHE

(DECISION DU CONSEIL SCIENTIFIQUE)

ENSEEG	HAMMOU	Abdelkader
	MARTIN-GARIN	Régina
	SARRAZIN	Pierre
	SIMON	Jean-Paul
ENSERG	BOREL	Joseph
ENSIEG	DESCHIZEAUX	Pierre
	GLANGEAUD	François
	PERARD	Jacques
	REINISCH	Raymond
ENSHMG	ROWE	Alain
ENSIMAG	COURTIN	Jacques
C.E.N.G	CADET	Jean
	COEURE	Philippe
	DELHAYE	Jean-Marc
	DUPUY	Michel
	JOUVE	Hubert
	NICOLAU	Yvan
	NIFENECKER	Hervé
	PERROUD	Paul
	PEUZIN	Jean-Claude
	TAIEB	Maurice
	VINCENDON	Marc

Laboratoire extérieurs :

C.N.E.T.	DEVINE	Rodericq
	GERBER	Roland
	MERCKEL	Gérard
	PAULEAU	Yves

XXXXXXXXXXXXXXXX

Inférence Parallèle et Processus Communicants pour les Clauses de Horn Extension au Premier Ordre par la Méthode de Connexion

María-Blanca Ibáñez-Espiga

Résumé

Dans cette thèse, nous avons décrit une machine à inférence parallèle pour les Clauses de Horn qui exploite le parallélisme OU et qui utilise comme mécanisme d'inférence la résolution. Le modèle décrit pour les clauses de Horn part d'un réseau de processus qui représente la structure syntaxique du programme logique. Le fait d'avoir FP2 comme langage pour la specification des machines nous a permis d'utiliser le mécanisme de communication du langage pour réaliser l'opération de base dans l'inférence : l'unification.

L'espace de recherche de la preuve d'une formule des clauses de Horn contient uniquement les axiomes de la preuve plus la résolvante courrante. Pour prouver une formule du premier ordre, cet espace est insufisant. Nous avons presenté également une méthode correcte, fondée sur la Méthode de Connexion pour calculer les ensembles de paires de littéraux à résoudre dans formule du premier ordre. Cela représente le pas le plus difficile à franchir pour la spécification d'une machine à inférence parallèle pour la logique du premier ordre.

Abstract

In this thesis we describe an OR-parallel inference machine for Horn clause processing. This machine is formed by a network of processes ressembling the sintactic structure of the logic program and it uses resolution as its inference mechanism.

We use FP2 as specification language for describing the machine. This has allowed us to use the communication capabilities of this language for implementing the basic operation in inference : unification.

When proving a Horn clause using resolution, the search space contains the axioms of the proof and the current resolvent. However, this search space is not enough when proving a first order logic formula using Bibel's Connection Method. We have to calculate at each step the set of pairs of literals (connections) to be resolved. This set is called the spanning set of connections.

In this thesis we also present the specification of a correct method for calculating spanning sets of connections. This is the most difficult step in the specification of a parallel inference machine for First Order Logic formula processing based on Bibel's Connection Method.

Remerciements

Je tiens à remercier :

- Monsieur Jacques Mossière, Professeur à l'INPG, Directeur du LGI, de m'avoir fait l'honneur de présider le jury.

- Monsieur Wolfgang Bibel,Professeur à l'Université de Darmstadt, d'acoir accepté de juger mon travail écrit en Français.

- Monsieur Gérard Comyn, Professeur d'Université et Directeur de l'ECRC, d'avoir accepté de juger mon travail, de l'avoir fait avec si promptement et avec autant d'intêret.

- Monsieur Ricardo Caferra, Maître de Conférence à l'ENSIMAG, d'avoir accepté de participer au jury.

- Monsieur Philippe Jorrand, Directeur de recherche au CRNS, Directeur du LIFIA, de m'avoir permis de travailler dans son équipe et diriger ma thèse. Son intuition et sa confiance ont beaucoup contribué à mon travail. Il a consolidé ma formation profesionnelle et je resens pour lui une profonde admiration.

Monsieur Jorge Vidart, Directeur de l'ESLAI et Professeur invité à l'ENSIMAG m'a fait l'honneur d'assister à la soutenance de ma thèse. Monsieur J. Vidart est celui qui m'a donné les bases de ma formation profesionnèlle.

Je remercie à mon groupe de travail au LIFIA : Françoise, Jean-Michel, Mary, Mounira, Philippe, Sylvie, Xavier et Zubir pour l'intêrèt qu'ils ont porté à mon travaille. Leurs critiques m'ont beaucoup aidée. Je les remercie également pour tout ce qu'ils m'ont appris par ailleurs. Mes amis J. Michel, Pascal, Philippe, Rachid, Ramon, Sadik ont contribués à créer une ambiance de travail sympathique.

L'écriture de la thèse n'a pas été facile, heureusement M. Ph. Jorrand a été à mes côtes pour la relire, en amèliorer le style et l'orthographe. Mme. M. Bello m'a également aidée à rendre comprehensible mes idées et à corriger la plupart des fautes d'orthographe. A tous les deux un grand Merci.

Je veux aussi remercier la grande famille de la Maison Catholique de l'étudiante (MCE). En premier lieu Mlle. Jeantet qui m'a permis de rentrer dans cette communité ainsi que toute l'équipe de dames qui s'occupent du foyer; toutes les filles qui ont partagé avec moi cette maison et très spécialement à Sabine, Florence, Tracy, Akemi, Fréderique pour leur amitié et pour m'avoir permis de connaître mieux leur pays.

Je ne voudrais pas oublier "ma résidence d'été" : le CCU où j'étais accueillie par les prêtres André et François. J'y ai fait la connaissance de bons amis qui ont partagé avec moi leur force et leur enthousiasme: je pense spécialement à Michael, Geneviève et Pierre.

Je serai toujours reconnaissante à Philippe Schnoebelen pour m'avoir accueillie chez lui durant les dernières mois de ma thèse.

Je dois ensuite exprimer ma gratitude à mon groupe d'amis "latins" : Sylvie (de Normandie), Mary, Amelia, Carmen, Beatriz, Carlos R., Carlos B., Astrid et Andrea. Merci pour leur amitié, leur compréhension, leur disponibilité et leur chaleur.

Enfin, ce qui a compté beaucoup pour moi : l'appui de ceux qui me sont très chères mais qui se trouvaient loin, ma famille, mes amis de Caracas et Emilio.

Cette thèse a été possible grâce à l'aide financière du CROUS (France) et d'un crédit éducatif du CONICIT (Vénézuela).

Pour finir, je voudrais souhaiter "bon courage" à tous ceux qui préparent actuellement leur thèse au sein du LIFIA.

TABLE des MATIERES

INTRODUCTION

En guise d'introduction, nous placerons notre travail dans l'évolution des systèmes de traitement de l'information.

Jusqu'à aujourd'hui, l'histoire de l'informatique fait apparaitre quatre générations qui se distinguent par le type de technologie, l'architecture des systèmes, le mode de traitement et les langages utilisés [HwB84].

La premier génération d'ordinateurs utilisait les tubes électroniques pour réaliser l'unité de contrôle et la mémoire. Tous les programmes étaient écrits dans le langage de la machine.

La deuxième génération se caractérise par l'utilisation de transistors, circuits imprimés et mémoires à tores magnétiques. Les langages assembleurs et les premiers langages de haut niveau apparaissent : FORTRAN, ALGOL et COBOL ont été développés dans cette génération.

L'apparition des circuits de petite et moyenne intégration marque la troisième génération et réduit les dimensions et le coût du matériel. A la fin de cette génération apparaissent le temps partagé et la mémoire virtuelle.

La quatrième génération se caractérise par l'utilisation de l'intégration à grand échelle (VLSI), pour la construction des unités de traitement et des unités de mémoire et des architectures nouvelles apparaissent, ou le parallèlisme devient prépondérant. En effet, la technologie VLSI rend possible la conception d'ordinateurs avec des dizaines, des centaines et même des milliers des processeurs : Connection Machine, Stalkman machine, N-CUBE, Tansputers. De nouveaux langages doivent être crées pour exploiter les capacités de telles machines.

En même temps, la complexité des opérations grandit. Ainsi une part importante des recherches en Intelligence Artificielle est la modélisation du raisonnement. La méchanisation de la logique est une des formes. PROLOG en est un exemple, conçu pour la programmation des

machines sequentielles. Mais les performances sont decevantes et le sous-ensemble de la logique ainsi couvert (les clauses de Horn) n'est pas grand. C'est ainsi qu'une nouvelle voie fait maintenant l'objet de nombreuses études : la réalisation d'une machine à inférence parallèle.

Des nombreuses machines à inférence parallèle sont presentées dans la littérature [CiH83], [CoK81], [ISK85], [KuM86], [LiK88], [YaN84]. Elles travaillent toutes sur les clauses de Horn. Dans ces machines, deux sortes de parallélisme ont été principalement exploités : le parallélisme OU (recherche en parallèle de plusieurs alternatives pour satisfaire un but) ; le parallélisme ET (la satisfaction simultanée de plusieurs sous-buts).

Le problème principal du parallélisme OU est la manipulation des multiples liaisons de variables qui s'etablissent pendant la recherche de solutions alternatives. Les modèles basés sur le parallélisme OU essaient plutôt de décrire l'architecture des machines que de manipuler de manière efficace les données. Les derniers travaux essaient de trouver des heuristiques pour choisir les branches de l'arbre de recherche qui ont plus de possibilités de réussir.

Le problème principal du parallélisme ET est la cohérence des liaisons de variables. Ce type de parallélisme a été moins traité que le parallélisme OU.

Dans cette thèse, nous utilisons le langage FP2 (Functional Parallel Programming Langage) [Jor86] comme langage pour la description de diverses machines à inférence parallèle. En FP2, il est possible de décrire des processus communicants et de construire des réseaux de processus. Ce sont de tels réseaux qui seront utilisés pour décrire nos machines.

La méthode choisie pour décrire les machines parallèles qui exploitent le parallélisme ET et OU dans les Clauses de Horn, est en fait une méthode de traduction qui, à partir d'un ensemble de Clauses de Horn, produit une machine à inférence parallèle décrite en FP2. Il s'agit donc, pour chaque ensemble de clauses de Horn, de produire une machine particulière.

Il faut comprendre le travail décrit ici comme la première étape en vue de la réalisation effective d'une machine à inférence parallèle. En effet, une fois obtenue la description de la machine correspondant à un ensemble de clauses, il faut encore transformer cette structure de processus afin de pouvoir la "plaquer" sur une structure effective de machine parallèle comme, par exemple, un maillage rectangulaire de processus, un hypercube ou une machine Supernode. Une étude préalable de cette deuxième étape a été faite dans [Mar86] dans le cas de maillages rectangulaires.

Nous proposons aussi une façon de depasser le cadre des clauses de Horn pour aller jusqu'à la logique du premier ordre en calculant l'ensemble de résolvants dont il faut tenir compte pour prouver des formules. Pour cela nous utilisons la méthode de connexion [Bib81]. Une fois résolu le problème de savoir quelles connexions utiliser pour la preuve d'une formule,

les techniques connues pour exploiter les parallélisme OU et ET peuvent être appliquées à la logique du premier ordre.

L'organisation de cette thèse est la suivante :

Dans le premier chapitre nous rappelons les concepts liés à la programmation logique, nous présentons les deux procédures de preuve qui nous intéressent : la règle de résolution et la méthode de connexion. Nous étudions les diverses formes de parallélisme qui peuvent être exploitées dans la programmation logique et enfin nous présentons FP2, notre outil pour la description de machines à inférences parallèles.

Le deuxième chapitre contient une présentation des principaux modèles de machines à inférence qui sont connus. Dans cette presentation, nous exposons les différentes techniques existantes pour exploiter les parallélismes ET et OU.

Deux modèles de machines à inférences parallèles pour le langage des clauses de Horn sont présentés dans le troisième chapitre. Pour chaque formule à prouver, nous décrivons une machine capable de prouver la formule lorsqu'elle est valide. La structure de la machine reflète la structure syntaxique de la formule.

Enfin, nous donnons dans le quatrième chapitre une méthode pour le calcul des ensembles de résolvants à appliquer lorsque l'on veut prouver une formule en logique du premier ordre. Pour faire cela nous utilisons la méthode de connexion. Nous prouvons que l'algorithme proposé est correct.

CHAPITRE 1

Programmation Logique et Parallélisme

PROGRAMMATION LOGIQUE ET PARALLELISME

Dans ce chapitre, nous rappelons les concepts étroitement liés à la programmation logique. L'attention est centrée sur deux règles d'inférence : la résolution [Rob65] et la méthode de connexion [Bib83]. On montre les possibilités de parallélisation de ces deux règles.

Après une brève introduction historique, on donne la terminologie de base du calcul de prédicats du premier ordre, pour une étude approfondie voir [Cha73], [Man74] et [Llo84]. On présente d'abord les aspects syntaxiques (section 1.2), ensuite les aspects sémantiques (section 1.3). Notre intérêt est l'étude de systèmes de déduction (section 1.4). On approfondit notre étude sur deux règles d'inférence intéressantes pour une implémentation parallèle: la règle de résolution [Rob65] (sections 1.5 et 1.6) et la méthode de connexion [Bib83] (section 1.8). Dans la section 1.7, on introduit les notions couramment utilisées en programmation logique. Enfin, dans la section 1.9, on présente une étude sur les différentes formes de parallélisme qui peuvent être exploitées dans les deux règles d'inférence étudiées [Con87], [Bib87].

1.1. Une introduction historique.

La programmation logique est le résultat de nombreuses années de recherche dans le domaine de la demonstration automatique de théorèmes. Une des premières études sur la mécanisation du raisonnement a été faite par le mathématicien Gottfried Leibniz (XVII siècle). Deux siècles plus tard, Gottlob Frege développe le calcul de prédicats : il avait pour objectif la création d'un langage capable d'exprimer n'importe quelle pensée rationnelle de façon mathématique et systématique.

K. Gödel, J. Herbrand et T. Skolem ont prouvé, indépendamment et de manière différente, que chaque phrase valide - c'est-à-dire, phrase dont la valeur est vraie quelle que soit l'interprétation - peut être prouvée en utilisant le langage du calcul des prédicats. On est alors parvenu aux premières procédures de preuve dans le calcul des prédicats qui ont ouvert la voie

aux recherches sur la preuve automatique de théorèmes. Grâce à leurs travaux, Davis, Putnam et Prawitz, ont permis une avance significative dans ce domaine.

C'est en essayant d'améliorer les procédures de preuve connues que J.A. Robinson [Rob65] découvre le principe de résolution. La résolution est une généralisation de la règle d'inférence appelée *modus ponens*, et qui utilise l'opération l'unification.

Au début des années 70, R. Kowalski et A. Colmerauer proposent l'idée de base de la programmation logique : la logique peut être utilisée comme langage de programmation. Jusqu'alors la logique n'avait été utilisée en informatique qu'en tant qu'outil descriptif pour exprimer des spécifications.

Des travaux commencent alors sur un sous-ensemble du calcul des prédicats du premier ordre: les clauses de Horn. De là est issu le langage de programmation PROLOG (PROgrammation LOGique). R. Kowalski [Kow74] prouve qu'un tel sous-ensemble a une interprétation procédurale exploitable pour programmer. PROLOG devient ainsi un langage très utilisé en intelligence artificielle. Mais il ne faut pas penser que la programmation logique se limite à PROLOG : il est toujours possible d'utiliser des règles d'inférence différentes de la résolution, d'utiliser des sous-ensembles de la logique différents, etc. Les futurs systèmes devront poursuivre l'idéal de la programmation logique: la séparation des deux composantes de base des algorithmes, la logique et le contrôle [Kow74], de façon à ce que le programmateur s'occupe de decrire uniquement le composant logique de son algorithme sans avoir besoin d'exprimer comment son programme doit être résolu (sans donner l'ordre des clauses, l'ordre des atomes dans les clauses ou autres composants extra-logiques telles que le "cut" de PROLOG).

Un aperçu historique plus complet est donné par J.A. Robinson dans [Rob83], R.A. Kowalski raconte les premières années de la programmation logique dans [Kow88] et J.W. Lloyd présente les fondations de la programmation logique dans [Llo84].

1.2. La syntaxe du calcul des prédicats du premier ordre.

Les expressions correctes du langage des prédicats du premier ordre se forment en utilisant les symboles suivants:

- Les symboles de vérité: T et F.

• Un ensemble de symboles appelés **constantes**. Les constantes seront notées avec les lettres minuscules de l'alphabet latin, avec les nombres naturels et avec les concaténations de ces lettres et nombres. Par exemple: a, b, aab, 0, a0 etc.

• Un ensemble de symboles appelés **symboles de fonction** (ou fonctions). Les symboles de fonction seront des chaînes de caractères minuscules de l'alphabet latin telles que f, g, tree etc. Chaque symbole de fonction a une arité (ou nombre d'arguments) fixe. Les constantes sont des symboles de fonction d'arité nulle.

• Un ensemble de symboles appelés **variables**. Les variables seront notées avec les lettres majuscules de l'alphabet latin et avec les concaténations de cettes lettres. Par exemple: X, Y, UXY etc.

• Un ensemble de symboles appelés **symboles de prédicat** (ou prédicats). Ces symboles de prédicat seront des chaînes de caractères majuscules de l'alphabet latin telles que P, Q, FACT etc. Chaque symbole de prédicat a une arité (ou nombre d'arguments) fixe. Lorsque l'arité d'un prédicat est 0, le prédicat est appelé **proposition**.

• Les connecteurs logiques: ~ (négation -non-), $\wedge$ (conjonction -et-), $\vee$ (disjonction -ou-), $\rightarrow$ (implication) et $\equiv$ (équivalence).

• Les quantificateurs: $\exists$ (quantificateur existentiel), $\forall$ (quantificateur universel).

En utilisant ces symboles, on définit récursivement trois classes d'expressions: termes, atomes (ou formules atomiques) et formules bien formées.

Termes.

Une constante est un terme.

Une variable est un terme.

Si f est une fonction d'arité "n" et si $t_1,\ldots,t_n$ sont des termes alors f $(t_1,\ldots,t_n)$ est un terme.

Par exemple, tree (X, tree (a,b)) est un terme.

Atomes.

T et F sont des atomes.

Les propositions sont des atomes.

Si P est un prédicat d'arité "n" et si $t_1,\ldots,t_n$ sont des termes, alors P $(t_1,\ldots,t_n)$ est un atome.

Par exemple, FACT (0, s(0)) est un atome.

Formules bien formées, (en abrégé: f.b.f.).

Un atome est une formule bien formée.

Si S_1 et S_2 sont des formules bien formées alors $\sim S_1$, $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \rightarrow S_2$ et $S_1 \equiv S_2$ sont des formules bien formées.

Si S est une formule bien formée et X est une variable alors $\exists$ X: S et $\forall$ X: S sont des formules bien formées.

Par exemple: $\exists$ X, $\forall$ Y: ((P (X,Y) $\vee$ Q (X,Y)) – R (X)) est une formule bien formée.

Si A est un atome, alors A et ~A sont appelés des **littéraux.**

Si $\forall$ X: S est une formule bien formée, nous dirons que X dans $\forall$ X est **quantifié universellement.** De manière similaire, si $\exists$ X: S est une formule bien formée, nous dirons que X dans $\exists$ X est **quantifié existentiellement.**

Toute variable, figurant dans une f.b.f., qui est quantifiée universellement ou existentiellement est dite **liée.**

1.3. La sémantique du calcul de prédicats du premier ordre.

Nous pouvons donner une signification à chaque formule bien formée en "interprétant" les constantes, les variables, les fonctions et les prédicats de la formule. En associant différentes interprétations à une formule donnée, on obtient différentes phrases où chaque phrase est vraie ou fausse.

Une **interprétation** I d'une f.b.f. S consiste en un domaine non vide D et un ensemble d'affectations à chaque constante, variable, fonction et prédicat qui apparaît en S. Les affectations se font selon les règles suivantes :

On affecte un élément de D à chaque constante de S.

On affecte une application de D^n dans D à chaque symbole de fonction d'arité "n" ($n \geq 1$).

On affecte une application de D^n dans l'ensemble { T, F } à chaque symbole de prédicat d'arité "n" ($n \geq 1$).

Pour chaque interprétation I sur le domaine D, S peut être évaluée à T ou F selon les règles suivantes:

Si les valeurs de vérité de S_1 et S_2 sont connues alors les valeurs de vérité de $\sim S_1$, $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \rightarrow S_2$ et $S_1 \equiv S_2$ sont évaluées en utilisant les tables de vérité des connecteurs $\sim$, $\wedge$, $\vee$, $\rightarrow$ et $\equiv$ respectivement.

$\forall$ X : S est évaluée à T si et seulement si S évalue à T pour chaque affectation d'un élément de D à X. Autrement, $\forall$ X : S est évaluée à F.

$\exists$ X : S est évaluée à T si et seulement si S évalue à T pour au moins, une affectation d'un élément de D à X. Autrement, $\exists$ X : S est évaluée à F.

Si une f.b.f. S est évaluée à T dans une interprétation I on dit que I est un **modèle** de S ou que S a un modèle I.

Une f.b.f. est **valide** si et seulement si sa valeur est T selon toute interprétation. Sinon elle est **invalide**.

Une f.b.f. est **inconsistante** si et seulement si sa valeur est F quelle que soit l'interprétation. Dans le cas contraire, elle est **consistante**.

Il y a une relation importante entre les concepts de validité et d'inconsistance: Une f.b.f. S est valide si et seulement si $\sim$S est inconsistante.

Une f.b.f. qui contient "n" symboles de prédicat et qui ne contient pas de variables a 2^n interprétations. En utilisant les tables de vérité, on peut déterminer en un nombre fini d'opérations si cette formule est valide ou non, inconsistante ou non.

Pour une f.b.f. avec variables, le nombre d'interprétations est infini et il est donc impossible de proposer un algorithme capable de décider si une f.b.f. est valide ou inconsistante. Cependant, il y a des algorithmes qui pour des formules valides s'arrêteront au bout d'un nombre fini (mais non fixe) d'opérations en concluant la validité de la formule. De tels algorithmes peuvent ne jamais s'arrêter lorsque la formule n'est pas valide.

1.4. Systèmes de déduction.

Pour le calcul des prédicats du premier ordre, un système de déduction consiste en:

- Un ensemble de f.b.f. valides appelées **axiomes**. Par exemple, si S_1 et S_2 sont f.b.f. alors $S_1 \rightarrow (S_2 \rightarrow S_1)$ est un axiome.

- Un ensemble fini d'applications, de f.b.f.^{n} $\rightarrow$ f.b.f., appelées **règles d'inférence.** Par exemple, si S_1 et S_2 sont des f.b.f. alors de S_1 et $S_1 \rightarrow S_2$ on conclut S_2 (*Modus ponens*).

Une collection d'axiomes et de règles d'inférence pour le calcul de prédicats du premier ordre est donnée par Manna [Man74].

Chaque règle d'inférence prend donc une ou plusieurs f.b.f. valides en argument et retourne une f.b.f. valide. Chaque f.b.f. déduite de cette façon à partir d'axiomes est valide.

Malheureusement, la réciproque n'est pas toujours vraie et il y a des systèmes de déduction où quelques formules valides ne peuvent être déduites. Nous sommes intéressés ici par les **systèmes de déduction complets**, c'est-à-dire par les systèmes où chaque formule valide peut être déduite.

1.5. Une règle d'inférence : la résolution.

La résolution [Rob65] est une extension de la règle d'inférence du *modus ponens*, laquelle permet la déduction de S_2 à partir de S_1 et $S_1 \rightarrow S_2$.

La résolution peut travailler uniquement sur des f.b.f. écrites en forme clausale. Une f.b.f. est en **forme clausale** si elle a la forme: $\forall X_1 \forall X_2 \ldots \forall X_m [\tau_1 \wedge \tau_2 \wedge \ldots \wedge \tau_k]$ où chaque τ_i $(1 \leq i \leq k)$ est une **clause** (disjonction de littéraux) et les $X_1, X_2 \ldots X_m$ sont les variables qui apparaissent dans τ_i. La succession de quantificateurs $\forall X_1 \forall X_2 \ldots \forall X_m$ est appelée le **préfixe** de la f.b.f. La conjonction de clauses $\tau_1 \wedge \tau_2 \wedge \ldots \wedge \tau_k$ est appelée la **matrice** de la f.b.f. Toute f.b.f. peut être écrite en forme clausale [Man74] en préservant la propriété d'inconsistance.

La règle de résolution établit que pour deux clauses:

$$\tau_i: A_1 \vee \ldots \vee A_{n-1} \vee C \vee A_{n+1} \ldots$$

$$\tau_j: B_1 \vee \ldots \vee B_{m-1} \vee \sim C \vee B_{m+1} \ldots$$

où τ_i contient un littéral C et τ_j contient la négation de C, alors on peut déduire la clause:

$$\tau_{i,j}: A_1 \vee \ldots \vee A_{n-1} \vee A_{n+1} \vee \ldots \vee B_1 \vee \ldots \vee B_{m-1} \vee B_{m+1} \vee \ldots$$

La clause $\tau_{i,j}$ est appelée **la résolvante** de τ_i et τ_j.

Pour former la résolvante, il faut *unifier* les littéraux C et ~C. Deux littéraux sont unifiables si et seulement si ils sont syntaxiquement identiques ou si leurs variables peuvent être remplacées par des termes de manière à obtenir les deux littéraux identiques. Des littéraux sont identiques s'ils ont le même symbole de fonction, la même arité et si les termes correspondants sont identiques.

L'unification lie aussi les variables à des termes pendant la preuve. L'ensemble des liaisons faites pendant l'unification est appelé **substitution**. Lorsqu'une variable X est liée au terme "a" nous noterons $\{X - a\}$. Les substitutions sont dénotées par des lettres grecques, par exemple θ . Si θ est une substitution et L est un littéral, $L\theta$ est appelée une **instance** de L, résultat de l'application de la substitution θ aux variables de L. Par exemple, si L est le littéral P(X,f(Y),a) et θ est la substitution $\{ X - f(Z), Y - X \}$ alors $L\theta$ est P(f(Z),f(X),a).

Une fois introduits les concepts d'unification, de substitution et d'instance, on peut définir précisement ce qu'est une résolvante.

Supposons que nous avons les clauses

$$\tau_i: A_1 \vee \ldots \vee A_{n-1} \vee A_n \vee A_{n+1} \ldots$$

$$\tau_j: B_1 \vee \ldots \vee B_{m-1} \vee B_m \vee B_{m+1} \ldots$$

où An est un atome positif et Bm est un atome négatif. Supposons aussi qu'il existe une substitution θ telle que $A_n\theta = B_m\theta$. La résolvante de τ_i et τ_j est:

$$\tau_{i,j}: A_1\theta \vee \ldots \vee A_{n-1}\theta \vee A_{n+1}\theta \vee \ldots \vee B_1\theta \vee \ldots \vee B_{m-1}\theta \vee B_{m+1}\theta \ldots$$

Le principle de résolution est complet par réfutation. Au lieu de déterminer la validité d'une formule directement, la résolution détermine l'inconsistance de sa négation. Le principe de résolution établit qu'un ensemble de clauses C est inconsistant si et seulement si la méthode de résolution dérive la clause vide à partir de C. Décrivons maintenant comment faire une telle preuve.

Preuve par résolution.[Rob65]

Soit A un ensemble d'axiomes.

Soit τ la formule à prouver.

Constituer C, ensemble de f.b.f. écrites en forme clausale, pour chaque f.b.f. de A.

Ajouter à l'ensemble C la forme clausale de $\sim\tau$.

Tant que la clause vide n'est pas dans C *faire*:

- Choisir deux clauses différentes de C.

• *Si* elles ont des résolvantes *alors* en choisir une et l'ajouter à C.

Il faut remarquer que dans cet algorithme de résolution, nous ne précisons pas quelles sont les clauses à choisir ni quels sont les littéraux à résoudre. Dans la prochaine section, nous parlerons des améliorations possibles de la procédure de réfutation.

Exemple 1.1 :

Supposons que nous avons les axiomes suivants:

$$P\ (0, X, X)$$

$$\sim P\ (X, Y, Z)\ \nu\ P\ (s(X), Y, s(Z))$$

où les variables des axiomes sont quantifiées universellement.

Supposons que nous voulons prouver:

$$\exists\ W: P\ (s(0), s(0), W)$$

C est formé par les axiomes et la négation de la formule à prouver :

$$\tau_1: P\ (0, X, X)$$

$$\tau_2: \sim P\ (X, Y, Z)\ \nu\ P\ (s(X), Y, s(Z))$$

$$\tau_3: \sim P\ (s(0), s(0), W)$$

Le mécanisme de la preuve est le suivant :

Il y a une résolvante des clauses τ_2 et τ_3 en utilisant la substitution $\theta_1 = \{ X - 0, Y - s(0), W - s(Z) \}$, la résolvante est:

$$\tau_{23}: \sim P\ (0, s(0), Z)$$

C contient maintenant les clauses τ_1, τ_2, τ_3 et τ_{23}.

Il y a une résolvante des clauses τ_{23} et τ_1 en utilisant la substitution $\theta_2 = \{ X - s(0), Z - s(0) \}$. La résolvante de ces clauses est la clause vide. On a donc prouvé l'inconsistance des formules de C, et on a donc prouvé : $\exists$ W: P (s(0), s(0), W).

L'inconsistance d'une formule est garantie par la déduction de la clause vide. Cependant, du point de vue de la programmation, nous nous intéressons aux valeurs liées aux variables par l'unification.

La valeur de la variable W dans notre exemple est obtenue par la composition des substitutions θ_1 et θ_2, la valeur de W est donc s(s(0)).

fin de l'exemple

1.6. Des stratégies de résolution.

La règle de résolution présente certains désavantages, à savoir:

- elle produit des résultats redondants;

- elle permet de dériver des résolvantes qui ne conduisent à aucun résultat.

Supposons que nous voulons utiliser la résolution pour prouver que l'ensemble de clauses: { P(X) v Q(Y), ~P(b) v R(a), ~R(a), ~Q(a) } est inconsistant.

Le graphe ci-dessous, dit **graphe de dérivation** représente toutes les dérivations possibles à partir de l'ensemble initial de clauses. Une dérivation correspond à l'application d'une étape de résolution.

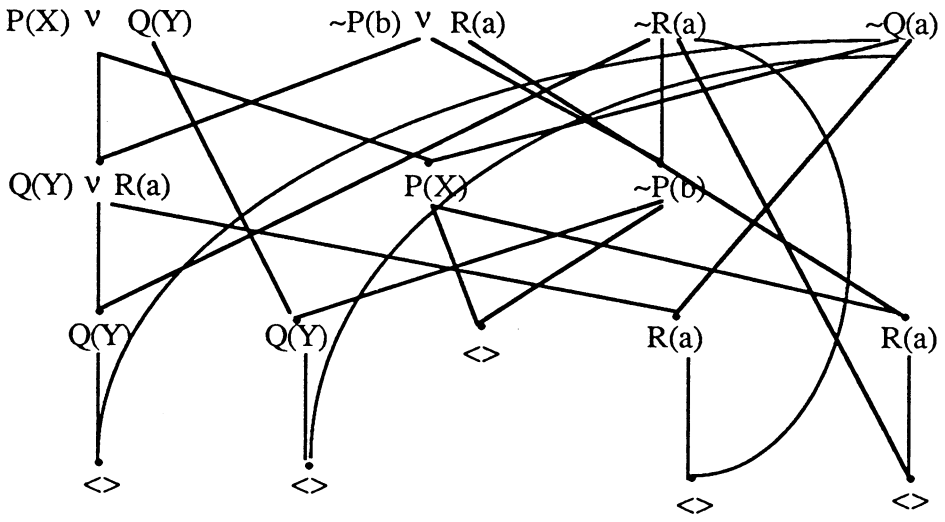


Figure 1.1

On observe qu'il y a des résultats redondants, par exemple, R(a) est dérivé de deux manières différentes. Après chaque étape de résolution, l'espace de recherche est agrandi et de ce fait pose des problèmes d'espace et de temps à une implémentation éventuelle de la résolution.

A partir d'un graphe de dérivation, on peut extraire plusieurs **graphes de réfutation** tels que:

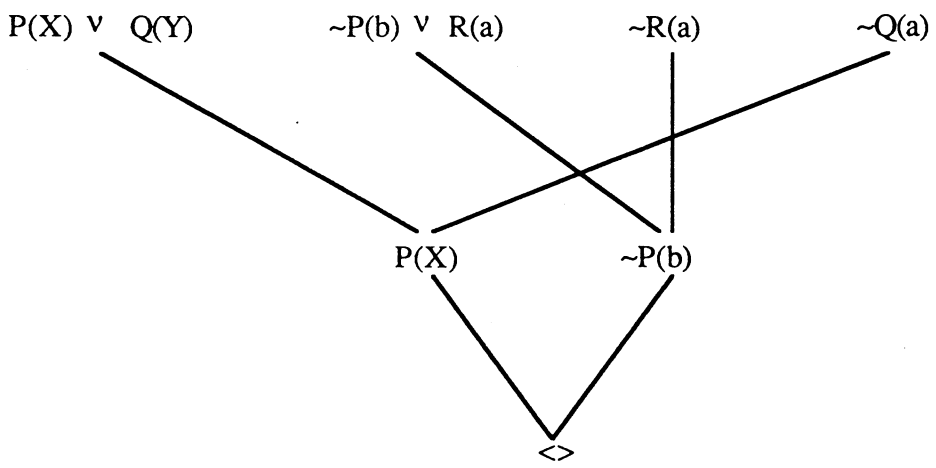


Figure 1.2

Selon la manière dont on choisit les clauses (et les littéraux de ces clauses) à résoudre, on peut définir diverses stratégies de résolution. Une telle stratégie restreint ou ordonne les choix de clauses et de littéraux sur lesquels sera tentée la résolution, selon des critères qui lui sont propres. Les stratégies ont pour but d'éliminer (dans la mesure du possible) le travail inutile en restreignant l'espace de recherche.

Une stratégie de résolution sera qualifiée de **complète** si, lorsqu'il est possible de prouver par résolution l'inconsistance d'une formule, on arrive a prouver l'inconsistence de la formule en utilisant cette stratégie[1].

Une grande partie des efforts réalisés dans la preuve automatique de théorèmes a été le développement de stratégies de contrôle et de raffinements de la méthode de résolution. Des études approfondies sur ces stratégies sont présentées dans [Cha73], [Sti86]. On présente ici, comme exemple de stratégie de résolution, la résolution linéaire ("linear résolution").

Résolution linéaire ("linear resolution").

Une partie de la redondance de la résolution sans restrictions provient du fait que l'on résoud des résolvantes intermédiaires avec d'autres résolvantes intermédiaires. La résolution linéaire évite quelques inférences inutiles en utilisant pour les déductions seulement les ancêtres d'une clause choisie *a priori* et les axiomes de la preuve.

Soit C un ensemble de clauses et soit τ_0 une clause de C. Toutes les résolvantes autorisées auront τ_0 pour ancêtre. Une telle stratégie de résolution peut être exprimée par l'arbre suivant:

---

[1] La complétude d'une stratégie de résolution ne doit pas être confondue avec la complétude du principe de résolution.

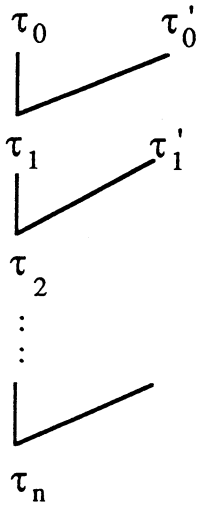


Figure 1.3

où les τ'i appartient à C u { τ'1 ,...,τ'j } j < i.

Supposons que nous voulons prouver l'inconsistance de la formule PvQ ∧ ~PvQ ∧ Pv~Q ∧ ~Pv~Q en utilisant la stratégie de résolution linéaire.

Le graphe ci-dessous montre la résolution linéaire de la formule en prenant comme τ_0 la clause PvQ.

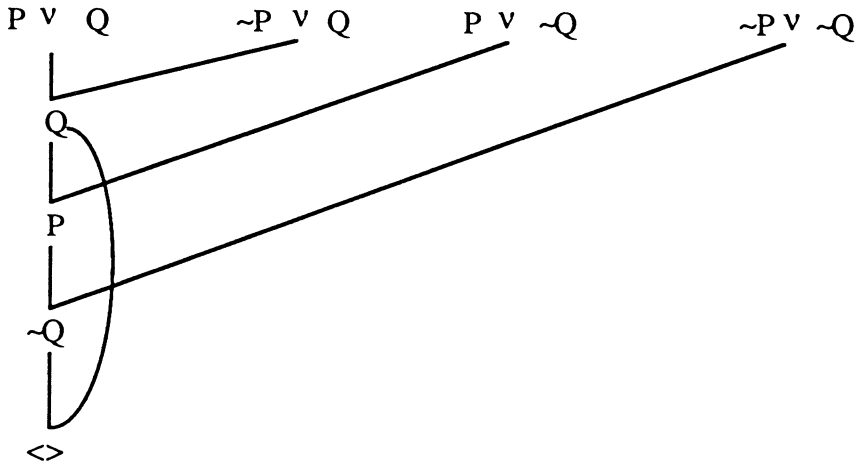


Figure 1.4

1.7. Systèmes de programmation logique.

Un sous-ensemble du calcul de prédicats du premier ordre, *les clauses de Horn*, est employé en programmation logique.

Un **programme logique** est une conjonction de clauses écrites comme une disjunction de littéraux.

Une clause de la forme : $A_1 \vee \ldots \vee A_m \vee \sim B_1 \vee \ldots \vee \sim B_n$ est écrite: $A_1 \ldots A_m$ <- $B_1 \ldots B_n$. $A_1 \ldots A_m$ est la **tête** de la clause et $B_1 \ldots B_n$ est le **corps** de la clause. On dira que : B_1 et ... et B_n implique A_1 ou ... ou A_m.

Dans le langage des **clauses de Horn**, chaque clause a au plus un littéral positif, il y a donc quatre types de clauses:

A <- B_1 et ... et B_n	Clause conditionnelle
A.	Clause d'assertion.
<- B_1 et ... et B_n	Clause de buts.
<>	Clause vide.

Les clauses sont regroupées en procédures. Une **procédure** est un ensemble de clauses qui ont dans leurs têtes le même symbole de prédicat avec la même arité.

Dans un système de programmation logique, un programme est l'ensemble de clauses conditionnelles plus l'ensemble de clauses d'assertion et la clause à prouver qui est la clause de buts. Ce sous-ensemble de la logique est un véritable langage de programmation, il a une interprétation procédurale [Kow74]. Une clause conditionnelle peut être considérée comme la définition d'une procédure, dans une clause de buts comme <- B_1 et ... et B_n, où les B_i sont des invocations de procédures. Une étape d'exécution consiste à choisir un littéral B de la clause de buts et ensuite, à trouver une clause dont la tête s'unifie avec B. La résolvante obtenue devient la nouvelle clause de buts. L'unification devient un mécanisme pour le passage de paramètres, pour la sélection et la construction de données.

Il faut encore préciser pour cette stratégie :

- Quel littéral choisir dans la clause de buts (règle de calcul);

- Quelle clause choisir parmi celles dont la tête s'unifie avec le littéral selectionné (règle de recherche).

Dans le système de programmation logique PROLOG, la règle de calcul choisit le littéral le plus à gauche de la clause de buts et la règle de recherche consiste à chercher en profondeur ("depth-first search) en prenant les clauses dans l'ordre où elles sont écrites, en utilisant le "backtracking" en cas d'échec. Cette stratégie est efficace (si on la compare avec d'autres stratégies séquentielles), mais elle n'est pas complète.

En effet, il y a des formules du langage des clauses de Horn qui sont inconsistantes mais qui ne peuvent pas être prouvées en PROLOG. Par exemple, le programme:

P <- P,Q.

P <- Q.

Q.

<- P.

produit un calcul infini.

Les étapes d'exécution d'un programme logique peuvent être représentées sur un **arbre de recherche**. Un arbre de recherche est un arbre où chaque nœud est une clause de buts. Les fils d'un nœud τ sont les nœuds qui représentent toutes les clauses de buts dérivables à partir de τ en une étape d'inférence. La racine d'un tel arbre est la clause de buts donnée par le programmeur et les feuilles de l'arbre sont ou bien la clause vide (succès), ou bien des nœuds d'échec, lorsque ces feuilles échouent, l'arbre peut aussi avoir des branches infinies.

Des règles de calcul différentes produisent, en général, des arbres de recherche différents. Le nombre de chemins qui finissent avec succès est indépendant de la règle de calcul utilisée [Llo84]. Cependant, la règle de calcul affecte le nombre de chemins qui échouent.

Exemple 1.2 :

Soit le programme suivant:

P(X,Z) <- P(Y,Z), Q(X,Y).

P(X,X).

Q(a,b).

<- P(X,b).

L'arbre de recherche (règle "plus à droite") est:

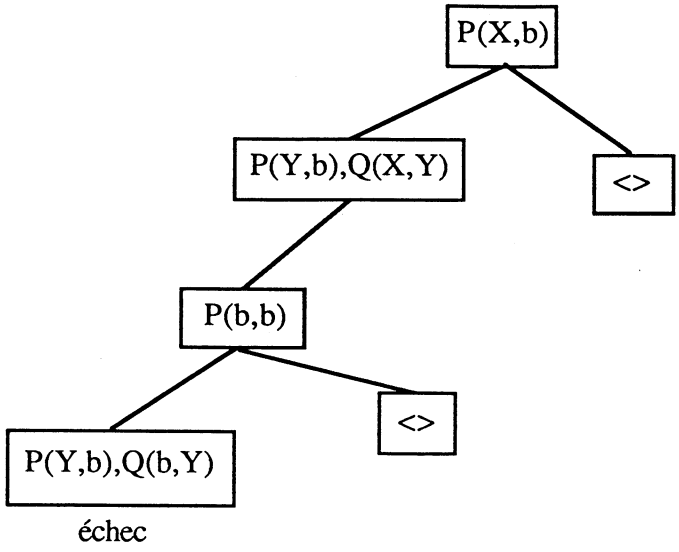


Figure 1.5

L'arbre de recherche montre qu'il y a deux façons différentes de prouver P(X,b), il contient aussi un chemin fini d'échec.

L'arbre de recherche (règle "plus à gauche") est:

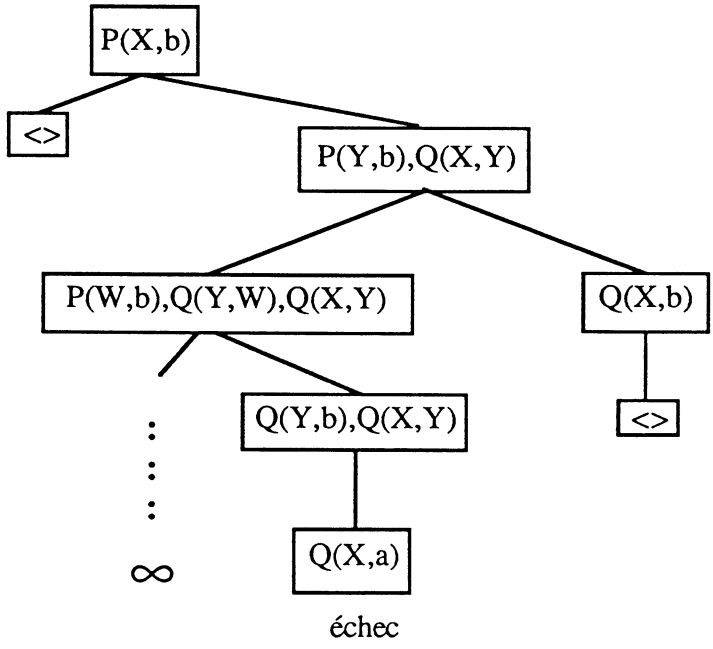


Figure 1.6

L'arbre de recherche montre deux façons différentes de prouver P(X,b) qui coïncident avec celles trouvées dans la règle de recherche plus à droite. L'arbre contient aussi le chemin fini d'échec qu'on avait trouvé dans l'arbre précédent mais cet arbre montre, en plus, un échec par chemin infini de dérivations.

1.8. La méthode de connexion.

La méthode de connexion est une procédure de preuve complète pour les formules bien formées. La méthode n'est pas une forme de la résolution, ni une méthode de réfutation.

Bien que la méthode de connexion n'exige pas que les formules soient en forme clausale, nous travaillerons avec la forme clausale pour simplifier la présentation de la méthode.

Supposons que nous voulions prouver la formule:

$$\sim P(f(f(a))) \vee \exists Y\,(\sim Q(Y)) \vee \exists X\,(P(f(X)) \wedge Q(X) \wedge \sim P(X)) \vee P(a)$$

Dans la formule ci-dessus, il y a des paires de littéraux avec le même symbole de prédicat, la même arité, de signe contraire dont les arguments sont unifiables (exemple, ~P(f(f(a))) et P(f(X)) avec la substitution { X -> f(a) }. Ces littéraux sont appelés **littéraux complémentaires**. Les littéraux complémentaires sont liés par arcs (appelés **connexions**) pour former la **matrice de connexions** (désignée par M). La formule peut en effet mise sous la forme d'une matrice qui a une colonne par clause et qui, en plus, contient toutes les connexions.

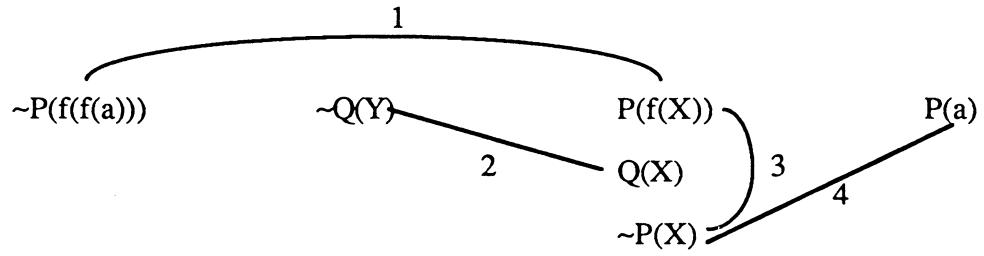


Figure 1.7

Un **chemin** à travers une telle matrice est une séquence de littéraux, en prenant un littéral dans chaque colonne. Les trois chemins de notre matrice sont: [~P(f(f(a))), ~Q(Y), P(f(X)), P(a)]; [~P(f(f(a))), ~Q(Y), Q(X), P(a)] et [~P(f(f(a))), ~Q(Y), ~P(X), P(a)].

Un **chemin** est **complémentaire** s'il contient, au moins, une paire de littéraux complémentaires. Pour le calcul de prédicats sans variables, la méthode de connexion établit qu'une formule est valide si et seulement si tous les chemins de la matrice sont complémentaires.

Cependant la vérification de la propriété de complémentarité des chemins de la matrice prend beaucoup de temps. Un **spanning set** S est un ensemble de connexions tel que chaque chemin de la matrice a, au moins, une connexion de S. Un spanning set peut être vu comme un ensemble de connexions qui participent à tous les chemins de la matrice.

Lorsqu'on utilise une règle d'inférence basée sur la résolution, chaque connexion est susceptible d'être utilisée dans la dérivation. Avec la méthode de connexion, chaque spanning set donne l'ensemble de connexions nécessaires pour prouver la formule, le spanning set représentant un schéma d'exécution.

Un **spanning set unifiable** est un spanning set pour lequel il existe un unificateur le plus général des littéraux de ses connexions.

Etant donné que les variables de la formule sont quantifiées de manière existentielle, les clauses peuvent être utilisées plusieurs fois. La méthode de connexion se fonde donc sur le principe suivante :

Une formule bien formée est valide si et seulement si il y a un nombre fini de copies de ses clauses possédant un spanning set unifiable.

Dans notre exemple, pour trouver un spanning set unifiable, on doit dupliquer les deuxième et troisième clauses.

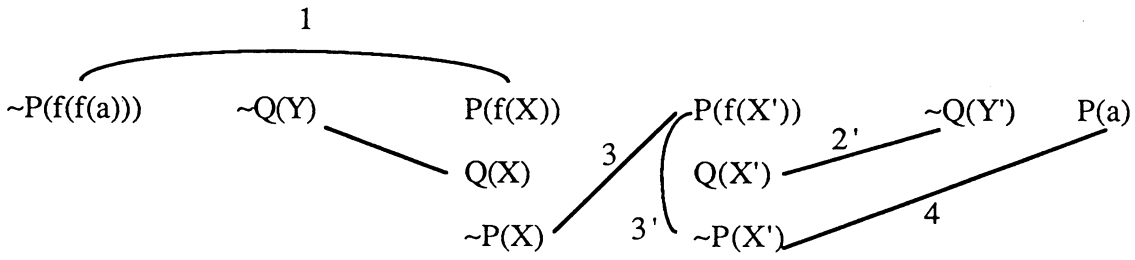


Figure 1.8

La substitution { X -> f(a); X' -> a; Y -> f(a); Y' -> a } est la substitution la plus générale du spanning set { 1,2,3,2',4 }.

1.9. Parallélisme dans le calcul de prédicats du premier ordre.

Traditionnellement, les modèles de machines d'inférence pour les clauses de Horn utilisent comme règle d'inférence la résolution. Dans cette section, on étudie les différentes sources de parallélisme dans chaque règle d'inférence qui nous intéresse: la résolution et la méthode de connexion. L'étude s'effectue sur le calcul des prédicats du premier ordre.

1.9.1. Parallélisme dans la résolution.

Rappelons qu'on utilise la résolution comme une procédure de réfutation. Lorsqu'on se sert de la résolution pour prouver l'inconsistance d'une formule, l'objectif est de construire (à partir de l'ensemble initial de clauses) la clause vide.

Selon la règle de résolution, si nous avons les clauses $\tau_1=A_{1,1},\ldots,A_{1,i1},\ldots,A_{1,k1}$, $\tau_2=A_{2,1},\ldots,A_{2,i2},\ldots,A_{2,k2}$ et s'il existe une substitution θ tel que $A_{1,i1}\theta = A_{2,i2}\theta$, où $A_{1,i1}$ et $A_{2,i2}$ sont des signes opposés, alors nous pouvons construire une nouvelle résolvante :

$$\tau_{1,2}=[A_{1,1},\ldots,A_{1,i1-1},A_{1,i1+1},\ldots,A_{1,k1},A_{2,1},\ldots,A_{2,i2-1},A_{2,i2+1},\ldots,A_{2,k2}]\theta.$$

Si $A_{1,i1}$ est complémentaire aux littéraux $A_{2,i2}$, $A_{3,i3}$,..., $A_{n,in}$ appartenant aux clauses $\tau_2,\tau_3,\ldots,\tau_n$ respectivement, on peut créer n-1 résolvantes:

$$\tau_{1,2}=[A_{1,1},\ldots,A_{1,i1-1},A_{1,i1+1},\ldots,A_{1,k1},A_{2,1},\ldots,A_{2,i2-1},A_{2,i2+1},\ldots,A_{2,k2}]\theta 2$$

$$\tau_{1,3}=[A_{1,1},\ldots,A_{1,i1-1},A_{1,i1+1},\ldots,A_{1,k1},A_{3,1},\ldots,A_{3,i3-1},A_{2,i3+1},\ldots,A_{3,k3}]\theta 3$$

$$\ldots$$

$$\tau_{1,n}=[A_{1,1},\ldots,A_{1,i1-1},A_{1,i1+1},\ldots,A_{1,k1},A_{n,1},\ldots,A_{n,in-1},A_{n,in+1},\ldots,A_{n,kn}]\theta n$$

Chacune des résolvantes $\tau_{1,2},\tau_{1,3},\ldots,\tau_{1,n}$ représente une façon différente de résoudre $A_{1,i1}$, c'est-à-dire chacune de ces résolvantes représente une solution distincte. Le fait de pouvoir résoudre $A_{1,i1}$ de n-1 façons signifie que les variables de $A_{1,i1}$ peuvent être liées de n-1 façons différentes. La création de $\tau_{1,2},\tau_{1,3}\ldots,\tau_{1,n}$ peut être faite en parallèle, et c'est ce qu'on appelle le **parallélisme OU**.

Dans l'exemple 1.2, P(X,b) a deux façons d'être résolu: la résolvante P(Y,b),Q(X,Y) qui lie X au terme "a" et la résolvante <> qui lie la variable X au terme "b", deux manières de lier la variable X.

Le problème inhérent au parallélisme OU est de savoir comment les liaisons des variables peuvent être représentées et stockées pour assurer un accès rapide et correct à leurs valeurs.

Supposons que les littéraux $A_{1,1},\ldots,A_{1,i1},\ldots,A_{1,k1}$ (de la clause τ_1) sont complémentaires aux littéraux $A'_{1,j1},\ldots,A'_{i1,ji1},\ldots,A'_{k1,jk1}$ des clauses $\tau'_1,\ldots,\tau'_{i1},\ldots,\tau'_{k1}$ respectivement. Supposons aussi que l'unificateur plus général de $A_{1,p}$ (p=1,...,i1,...,k1) et $A'_{p,q}$ (q=j1,...,ji1,...,jk1) est $\theta'p$, il est possible alors de créer la résolvante :

$$\tau'= [\ \tau'_1\backslash A'_{1,j1},\ldots,\tau'_{i1}\backslash A'_{i1,ji1},\ldots,\tau'_{k1}\backslash A'_{k1,jk1}\ ]\ \theta'1\circ\ldots\circ\theta'i1\circ\ldots\circ\theta'k1$$

où la notation $\tau i\backslash Ai,j$ ($\tau i=Ai,1,\ldots,Ai,j,\ldots,Ai,k$) signifie qu'on enlève le littéral Ai,j de la clause τi, $\tau i\backslash Ai,j=Ai,1,\ldots,Ai,j-1,Ai,j+1,\ldots,Ai,k$.

La création de τ' peut être faite en parallèle, et c'est ce qu'on appelle le **parallélisme ET**. Lorsqu'on implémente cette forme de parallélisme, le problème à résoudre est celui de gérer les variables qui apparaissent dans plus d'un littéral à l'intérieur d'une clause.

Dans la clause : P(X),Q(X); la variable X apparaît dans les deux littéraux. Pour résoudre cette clause, on doit trouver des valeurs de X qui satisfont P et Q.

Bien entendu, les parallélismes ET et OU peuvent aussi être combinés.

Au niveau des opérations à réaliser, il y a aussi différentes formes de parallélisme. Lorsqu'on veut satisfaire un littéral L, on doit chercher tous ses littéraux complémentaires, et cette opération peut être réalisée en parallèle: **parallélisme de recherche**. L'opération d'unification peut être réalisée en parallèle: c'est ce qu'on appele **le parallélisme de l'unification**, ce dernier type de parallélisme présente évidemment les mêmes problèmes que le parallélisme ET.

Pour un système réel de preuves en parallèle, on doit définir les types de parallélisme à employer et la façon de choisir les clauses qui interviendront dans chaque étape de la résolution. De manière classique, dans les systèmes pour les clauses de Horn, la résolution commence à être faite à partir de la clause de buts. Dans les systèmes de calcul de prédicats du premier ordre, les clauses initiales jointes aux résolvantes déjà obtenues peuvent se combiner entre elles de manière à construire le graphe de dérivation en parallèle, en l'absence d'une stratégie de résolution plus précise.

1.9.2. Parallélisme dans la méthode de connexion.

La méthode de connexion travaille avec la représentation matricielle des formules bien formées. Dans une telle matrice, tout littéral est relié à ses littéraux complémentaires. Chaque paire de littéraux complémentaires peut être trouvée de façon indépendante; les liaisons entre les littéraux peuvent donc être établies en parallèle.

Pour le calcul de prédicats sans variables, la validité d'une formule F est prouvée lorsque tous les chemins de la matrice associée à F sont complémentaires. La vérification de la complémentarité de ces chemins peut être réalisée en parallèle.

Lorsqu'il s'agit de prouver une formule F du calcul de prédicats, il est préférable (pour raison d'efficacité) de calculer les spanning sets de connexions de F plutôt que de travailler de manière explicite avec tous les chemins de la matrice associée à F.

Chaque connexion δ appartenant à un spanning set S, représente une famille de chemins complémentaires: tous les chemins qui contiennent la connexion δ. Pendant la construction de S, on choisit des nouvelles connexions δ' qui permettent d'élargir le nombre de chemins complémentaires considérés : le but est d'arriver à considérer tous les chemins de la matrice. La construction des spanning sets peut être réalisée de deux manières différentes, indépendamment ou non d'une séparation eventuelle des deux étapes qui la composent :

- L'étape de construction des spanning sets proprement dite;

- L'étape où l'on vérifie qu'un spanning set donné est un spanning set unifiable.

Si l'on sépare ces deux étapes, il faut alors pour vérifier qu'un spanning set S est unifiable que:

- Pour chaque connexion $\delta_i = <L_i, L'_i>$ appartenant à S, il existe une substitution θ_i telle que $L_i\theta_i = L'_i\theta_i$;

- Si une variable X a été liée par deux substitutions θ_i, θ_j ($i \neq j$) aux termes t_i et t_j respectivement, alors t_i et t_j doivent être unifiables.

Les actions que l'on vient de décrire peuvent être réalisées en parallèle et correspondent à ce que l'on a présenté comme étant le parallélisme ET.

Supposons le cas contraire, c'est-à-dire que les spanning sets sont vérifiés comme étant unifiables au fur et à mesure de leur construction. Dans ce cas, chaque fois qu'une connexion - susceptible d'être inclu dans un spanning set- est incidente à un littéral qui a plusieurs connexions incidentes, on peut exploiter le parallélisme OU.

1.10. FP2 : Langage de spécification de machines à inférence paralléles.

FP2 est un langage de programmation parallèle et fonctionnel développé par Ph. Jorrand et ses collaborateurs au le laboratoire LIFIA (Grenoble). FP2 est fondé sur des bases théoriques telles que les systèmes de transition, les systèmes de réécriture de termes et les spécifications algébriques. Ces bases théoriques permettent de prouver des propriétés sur le comportement des programmes décrits, de comparer la spécification d'un système avec son implémentation etc. FP2 est donc un langage de spécification. C'est grâce à la capacité du langage à exprimer le parallélisme et pour ses caractéristiques de langage de spécification qu'il a été choisi dans le projet ESPRIT 415 comme langage pour la description de machines à inférence parallèles.

Dans cette section on décrit FP2 de manière informelle, comme langage de programmation parallèle, on ne prend de FP2 que ce qui sera directement utile pour l'utilisation que nous allons en faire. En particulier on subordonne l'aspect fonctionnel du langage à la partie qui nous intéresse le plus : sa capacité d'exprimer le parallélisme. Une description plus formelle et plus complète du langage peut être trouvée dans [Jor86].

Processus : Unité de base d'un programme.

Un programme FP2 est formé d'un réseau de processus communicants. Un processus est une unité de calcul qui peut établir des communications avec d'autres processus. Un processus peut être vu comme "une machine" qui à chaque instant et dans un état donné peut soit évoluer vers un nouvel état en choisissant parmis plusieurs successeurs possibles, soit rester bloqué lorsqu'il n'y a pas d'autres possibilités.

Le comportement interne d'un processus est à la fois séquentiel et non-déterministe. Les capacités parallèles du langage sur la composition de plusieurs processus pour former une machine parallèle.

Caracteristique N° 1 : Les processus ont un comportement non-déterministe.

Pour illustrer le changement d'état d'un processus ainsi que son non-déterminisme, nous proposons de décrire un exemple simulant un parcours dans un labyrinthe.

Dans un premier temps, on peut caractériser un processus comme une machine qui a un nom, un ensemble d'états et un ensemble de règles de transition qui permettent de définir son comportement comme une sucession d'états. Une règle peut être de deux types :

• Règle initiale, de la forme :

==> <état>

• Règle de transition, de la forme :

<état> : ==> <état>

Chaque processus a au moins une règle initiale. Lorsqu'un processus commence à s'exécuter, son état initial est choisi de manière non-déterministe parmi un des états qui apparaissent dans ses règles initiales.

Une fois qu'un processus est dans un état S, si cet état n'apparait dans la partie gauche d'aucune règle alors le processus s'arrête, autrement le processus choisit de manière non-déterministe entre une des règles où S apparait comme partie gauche. Soit S : ==> T la règle choisie, le processus passe donc à l'état T.

Exemple 1.3 :

Soit le labyrinthe montré dans la figure suivante :

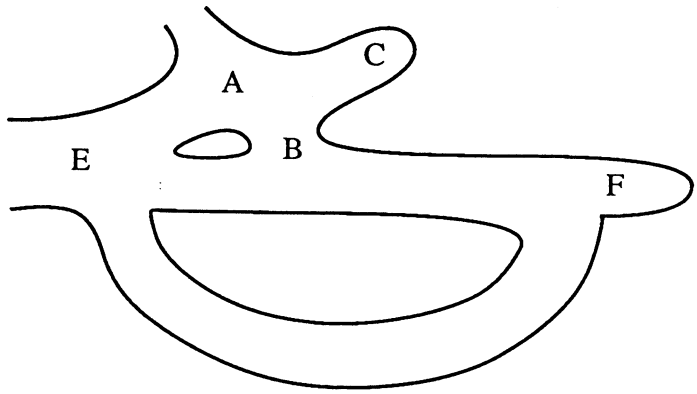


Figure 1.9

On peut abstraire sa structure par le graphe ci-dessous :

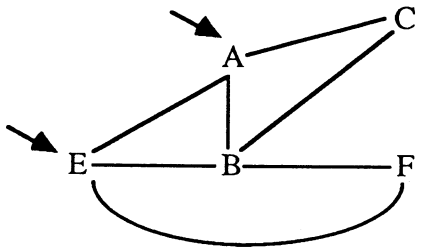


Figure 1.10

Le processus labyrinthe est :

process labyrinthe

states

;; Les états correspondent aux endroits clés du labyrinthe.

E,A,B,C,F : -

rules

;; Règles initiales

;; On peut rentrer au labyrinthe soit par la porte E, soit par la porte A.

==> E ;

==> A ;

;; Règles de transition

;; Une fois que l'on se trouve dans l'endroit E, on peut aller soit à A, soit à B, soit à F.

```
E:      ==>     A;
E:      ==>     B;
E:      ==>     F;

A:      ==>     E;
A:      ==>     B;
A:      ==>     C;

B:      ==>     E;
B:      ==>     A;
B:      ==>     F;
B:      ==>     C;

C:      ==>     A;
C:      ==>     B;
```

endprocess

On peut commencer le parcours du labyrinthe en rentrant par la porte E ou par la porte A (E et A sont les deux états initiaux de notre processus), notre but est d'arriver à l'endroit F (état final) où il y a un prix.

Supposons que l'on rentre par A (le processus choisit ==> A comme règle initiale), dans ce cas on a trois options : soit aller à E, soit aller à B, soit aller à C (trois règles différentes : A : ==> E; A : ==>B et A : ==>C). On n'a aucune manière de savoir quelle est la meilleure option à prendre : choisissons d'aller à B, une fois en B on peut aller à A de nouveau (on ne se rappelle pas que l'on vient de A), on peut aller soit à E, soit à C, soit à F (on ne sait pas que F est un état final), le processus choisit de nouveau de façon non-déterministe une des trois règles etc...

Fin de l'exemple

Caracteristique N° 2 : Un processus est une machine de transition d'états fondée sur la réecriture de termes.

Les états des processus que l'on vient de décrire n'offrent guère de possibilités de calcul. En générale les **états d'un processus FP2** ont des termes associés et le changement d'un état à un autre fait que ces termes sont réécrits. La forme générale d'un état est :

$$Q\ (u_1,\ldots,u_n)$$

où Q est un constructeur d'état auquel est associé le domaine $Type_1$ x ... x $Type_n$. Les $u_1,\ldots,u_n$ sont des termes respectivement de type $Type_1$, ... , $Type_n$.

Pour simplifier la présentation du langage, nous restreignons la définition de terme à la définition que nous en avons donnée précédemment pour la logique. De manière informelle nous établissons le type d'un terme de la façon suivante :

• Toute variable est associée à un type.

• Toute constante est associée à un type.

• Si f $(t_1,\ldots,t_n)$ est un terme où $\forall$ i = 1...n : t_i est de type T_i et le symbole de fonction f est de type T_1 x ... x T_n -> T, alors f $(t_1,\ldots,t_n)$ est de type T.

Examinons maintenant le comportement d'un processus. Supposons un processus dans un état $S(t_1,\ldots,t_n)$. Parmi ses règles, ce processus a les "p" règles ci-dessous, qui ont le symbole S dans leur partie gauche :

$S(t_{11},\ldots,t_{1k_1})$: ==> $T1(u_{11},\ldots,u_{1l_1})$;

$S(t_{21},\ldots,t_{2k_2})$: ==> $T2(u_{21},\ldots,u_{2l_2})$;

...

$S(t_{p1},\ldots,t_{pk_p})$: ==> $Tp(u_{p1},\ldots,u_{pl_p})$;

où les t_{ij}, u_{ij} sont des termes avec variables. Si $S(t_1,\ldots,t_n)$ et $S(t_{i1},\ldots,t_{ik_i})$ sont comparables (opération de filtrage), on dit que la règle :

$S(t_{i1},\ldots,t_{ik_i})$: ==> $Ti(u_{i1},\ldots,u_{il_i})$;

est applicable. Le processus choisit une des règles applicables soit cette règle la suivante :

$S(t_{j1},\ldots,t_{jk_j})$: ==> $Tj(u_{j1},\ldots,u_{jl_j})$;

Le prochain état du processus est $\mu(Tj(u_{j1},\ldots,u_{jl_j}))$, ($\mu$ la substitution qui a permi de filtrer $S(t_1,\ldots,t_n)$ par $S(t_{j1},\ldots,t_{jk_j})$).

Continuons avec notre exemple. Supposons qu'une souris rentre dans le labyrinthe en cherchent un morceau de fromage qui se trouve en F. A la sortie du labyrinthe, la souris nous dira quel est le chemin qu'elle a parcouru.

Pour résoudre ce nouveau problème, nous donnons une nouvelle version du processus labyrinthe. Les états du processus permettent de construire une séquence des endroits parcourus

dans le labyrinthe. Pour cela, les états ont un type associé, le type "path". Un "path" est une séquence formée par les caractères suivantes : "a", "e", "b", "c", "q". Où "a" signifie, par exemple, que la souris a passé par l'endroit nommé "a" dans le labyrinthe.

Exemple 1.4 :

process labyrinthe

states

E,A,B,C,F : path

Q : -

vars

info : path

rules

;; Règles initiales

==> E (e);

==> A (a);

;; Règles de transition

E (info): ==> A (a.info);

E (info): ==> B (b.info);

E (info): ==> F (info);

A (info): ==> E (e.info);

A (info): ==> B (b.info);

A (info): ==> C (c.info);

B (info): ==> E (e.info);

B (info): ==> A (a.info);

B (info): ==> F (info);

B (info): ==> C (c.info);

C (info): ==> A (a.info);

C (info): ==> B (b.info);

F (info) : ==> Q(rev(info))

endprocess

Une séquence d'états possibles pour cette nouvelle version du processus labyrinthe est la suivante :

A(a.nil) -> B(b.a.nil) -> E(e.b.a.nil) -> B(b.e.b.a.nil) -> F(b.e.b.a.nil) -> Q(a.b.e.b.nil)

fin de l'exemple

Caracteristique N° 3 : Un processus peut communiquer avec son environnement.

Jusqu'ici les processus décrits ne communiquent pas avec leur environnement. La **forme générale d'une règle** est :

<pré-condition> : <événement> ==> <post-condition>

où la pré-condition et la post-condition sont des termes d'états pouvant contenir des variables.

Une **règle initiale** a la forme :

$$==> T(u_1,\ldots,u_k)$$

Une **règle de transition** a la forme :

$$S(t_1,\ldots,t_p) : \quad k_1(v_1) \ldots k_n(v_n) \quad ==> T(u_1,\ldots,u_k)$$

Les $k_1,\ldots,k_n$ sont des **connecteurs** associés à des termes de type $T_1,\ldots,T_n$ respectivement. Par le connecteur k_i ne peuvent circuler que des termes (messages) de type T_i. Les t_i, v_i et u_i sont des termes pouvant contenir des variables.

Le **comportement d'un processus** est décrit dans [Jor86] comme suit :

• Initialement, on choisit de manière non-déterministe une des règles initiales du processus. La post-condition de cette règle devient l'état courant du processus.

• L'état courant $S(s_1,\ldots s_p)$ est comparé avec les pré-conditions des règles (opération de filtrage). Une règle :

$$S(t_1,\ldots,t_p) : \quad k_1(v_1) \ldots k_n(v_n) \quad ==> T(u_1,\ldots,u_k)$$

est **pré-applicable** si il existe une substitution μ telle que $\mu(S(t_1,\ldots,t_p)) = S(s_1,\ldots s_p)$. Si il n'y a pas de règles pré-applicables, le processus s'arrête.

• Soient $m_1,\ldots,m_n$ les messages prêts à être communiqués à travers les connecteurs $k_1,\ldots,k_n$. La règle est **applicable** s'il existe une substitution σ telle que $\forall\ i = 1\ldots n : \sigma(\mu(v_i)) = m_i$.

• Le processus choisit de manière non-déterministe une des règles applicables et l'applique. L'état actuel du processus devient :

$$\sigma(\mu(T(u_1,\ldots,u_k))).$$

Exemple 1.5 :

Supposons que nous voulons avoir un processus ADD qui reçoit une succession de naturels : $n_1,\ldots,n_k$ et qui chaque fois que l'on lui envoie un signal, ADD donne l'addition des naturels qu'il a lus depuis le dernier signal. La figure ci-dessous est la représentation graphique du processus ADD.

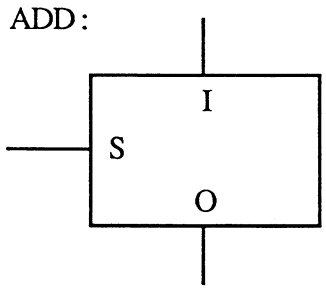


Figure 1.11

La définition en FP2 du processus ADD est :

process ADD

connecteurs

I,O : **Nat**

S : **Signal**

states

A : **Nat**

vars

n,p : **Nat**

rules

		==> A(0)
A (p) :	I(n)	==> A(p+n)
A (p) :	S(s) O(p)	==> A(0)

endprocess

Les déclarations des connecteurs nous indiquent que I,O et S sont des connecteurs tels que ne peuvent circuler à travers I et O que des nombres naturels et à travers de S que des valeurs de type Signal.

Voyons ensuite comment travaille le processus ADD. Supposons que nous allons lui donner les valeurs: 1 puis 3, puis un signal. Le processus commence son exécution dans l'état A(0). Ensuite, il y a deux règles pré-applicables :

A (p) : I(n) ==> A(p+n)

A (p) : S(signal) O(p) ==> A(0)

Lorsqu'il reçoit le nombre 1 la seule règle applicable est :

A (p) : I(n) ==> A(p+n)

Le processus arrive a l'état A(1), il reçoit le nombre 3, ADD passe à l'état A(4), les deux règles :

A (p) : I(n) ==> A(p+n)

A (p) : S(s) O(p) ==> A(0)

sont pré-applicables, le signale arrive alors, et maintenant est applicable la règle :

A (p) : S(s) O(p) ==> A(0)

Par O sort alors le nombre 4 et le processus arrive à l'état A(0).

Fin de l'exemple

Caracteristique N° 4 : L'environnement d'un processus est constitué d'autres processus.

Plusieurs processus peuvent être liés entre eux par l'intermédiaire des connecteurs et ces processus peuvent alors communiquer et travailler en parallèle. Les processus sont ainsi composés grâce a des opérateurs dont nous ne donnerons que ceux qui nous serons utiles : || (mise en parallèle), + (connexion).

FP2 propose en effet un ensemble d'opérateurs qui permettent la combinaison de processus. Comme résultat de ces combinaisons on obtient un nouveau processus, c'est-à-dire un ensemble de règles de transition qui decrivent le comportement de la machine parallèle qui a été construite. On ne décrit ici qu'un sous-ensemble d'opérateurs.

Opérateur composition parallèle : ||

Soient P et Q deux processus. Lorsqu'un événement a lieu dans le processus P || Q, un événement a lieu dans P alors que Q est inactif, ou un événement a lieu dans Q alors que P est inactif ou un événement a lieu dans P et un événement a lieu dans Q.

Opérateur de connexion : +

Soient P et Q deux processus qui ont été composés pour donner un processus R. Soient A et B deux connecteurs de P et Q respectivement tels que A et B peuvent communiquer des messages du même type. Dans ces conditions, on peut écrire : R + A.B. Le processus R + A.B se comporte comme le processus R et lorsqu'un événement qui concerne à la fois à A et à B se produit, un événement peut également se produire dans R + A.B, où les messages m_1 et m_2 qui circulent à travers de A et B respectivement sont unifiés.

Exemple 1.6 :

Pour illustrer le comportement d'un réseau de processus, considérons le problème suivant : Une plaque carrée en acier a une température initiale de 50°C et elle est touchée sur un de ses bords par une surface qui est à une température de 0°C, les autres bords de la plaque sont touchées par des surfaces qui ont une température de 100°C. Le haut et le bas de la plaque sont entourés d'une couverture isolante. Le problème est de calculer l'évolution de la température dans les 100 points de la plaque (voir la figure ci-dessous).

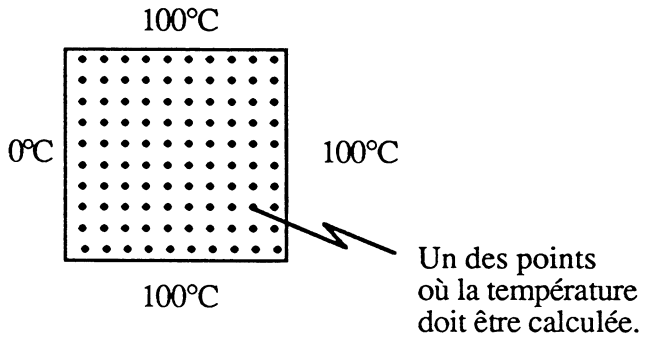


Figure 1.12

Pour résoudre ce problème de façon parallèle, supposons que chaque point de la plaque soit représenté par un processus comme celui que l'on montre dans la figure ci-dessous.

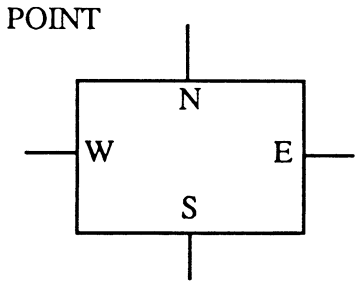


Figure 1.13

La température du point qui occupe la position <x,y> dans la plaque est noté $\phi_{x,y}$ et calculée en fonction de la température de ses points voisins de la manière suivante :

$$\phi_{x,y} = (\phi_{x-1,y} + \phi_{x,y-1} + \phi_{x+1,y} + \phi_{x,y+1}) \div 4$$

On représente la plaque par le réseau suivant :

PLAQUE:

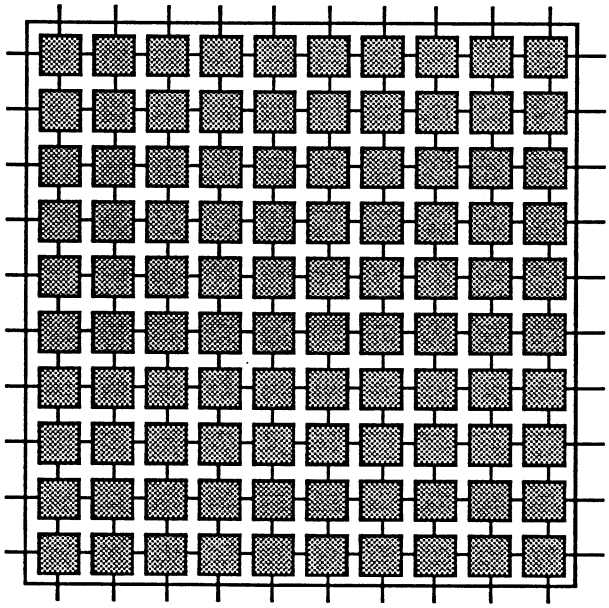


Figure 1.14

Les définitions des processus peuvent être paramétrées. Lorsqu'un processus a été défini, il peut être instancié avec des paramètres effectifs. Ainsi, la définition du processus PLAQUE contient un paramètre formel qui permet de créer des PLAQUEs de dimensions différentes. Pour notre exemple, PLAQUE sera instantié avec le paramètre 10.

PLAQUE [10] contient 100 processus POINT groupés en 10 lignes. Il est donc nécessaire de pouvoir distinguer entre les POINTs de la PLAQUE. FP2 indexe les processus POINT de la manière suivante :

POINT_1_1 POINT_1_2 ...

POINT_2_1 POINT_2_2 ...

: :

: :

les indices des processus sont hérités par les connecteurs des processus. Ainsi POINT_m_n possède les connecteurs : N_m_n ; S_m_n ; E_m_n ; W_m_n.

L'expression FP2 associée est :

process PLAQUE [n : **Nat**] **is**

|| { ROW [i,n] | i = 1...n }

+ { S_i_j . N_(i+1)_j | i = 1... n-1, j = 1...n }

process ROW [i,n : **Nat**] **is**

|| { POINT [i,j] | j = 1...n }

+ { W_i_(j+1) . E_i_j | j = 1...n - 1 }

Nous classifions les processus POINT comme pairs et impairs. Les quatre voisins d'un processus pair sont impairs et réciproquement. Supposons que le processus qui se trouve dans la position <1,1> soit pair.

Initialement les POINTs impairs calculent leur température et ensuite les POINTs pairs calculent leur température et ce cycle se répète indéfiniment. Cet algorithme est appelé "odd-even ordering with Chebyshev acceleration" [Qui87].

Process POINT [i,j : **Nat**]

connecteurs

N,S,E,W : **Real**

states

B : **Nat**

S : -

T : **Real**

vars

V1,V2,V3,V4, Temp : **Real**

rules

==> B (**if** pair(i+j) **then** even **else** odd)

B (odd) : ==> S

B (even) : ==> T (50)

;; Reçoit la température de ses voisins

S : N (V1) S (V2) E (V3) W (V4) ==> T ((V1 + V2 + V3 + V4) ÷ 4)

;; Donne sa température à ses voisins

T (Temp) : N (Temp) S (Temp) E (Temp) W (Temp) ==> S

endprocess

On ajoute au réseau quatre nouveaux processus qui représentent les bords de la plaque, le réseau final a la forme :

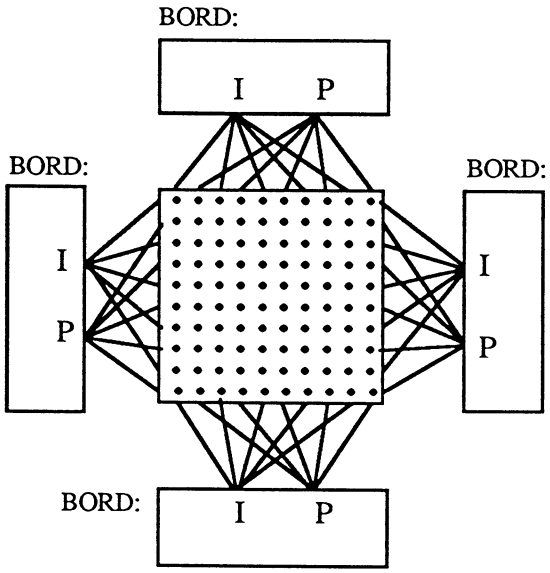


Figure 1.15

Le processus BORD est défini de la manière suivante :

processus BORD [Temp : **Real**]

connecteurs

I, P : **Real**

states

S : -

vars

Temp : **Real**

rules

==> S

S : I (Temp) ==> S

S : P (Temp) ==> S

end

L'expression FP2 qui représente le système est :

process NETW [n : **Nat** , Temp : **Real**] **is**

|| { BORD_i [Temp] | i = 1...4 } || PLAQUE [n]

;; BORD_1

+ { (**if** pair (i) **then** P_1 **else** I_1) . N_1_i | i = 1...n }

;; BORD_2

+ { (**if** pair (i) **then** P_2 **else** I_2) . E_i_1 | i = 1...n }

;; BORD_3

+ { (**if** pair (i) **then** P_3 **else** I_3) . S_n_i | i = 1...n }

;; BORD_4

+ { (**if** pair (i) **then** P_4 **else** I_4) . W_1_n | i = 1...n }

La définition actuelle du langage exige que le nombre de processus d'un réseau soit fixé statiquement. Cependant, pour des applications telles que la spécification de machines d'inférence parallèles, il est plus commode d'avoir la possibilité de créer des processus au fur et à mesure que la machine en a besoin.

On peut enrichir simplement FP2 en introduisant la possibilité de créer des processus dynamiquement de la manière décrite ci-dessous.

Les processus qui peuvent être créés dynamiquement ont des **règles initiales** de la forme :

$$k_1(v_1),\ldots,k_n(v_n) \quad ==> \quad T\,(u_1,\ldots,u_q)$$

et des **règles de transition** de la forme :

$$S\,(t_1,\ldots,t_p) : \; k_1(v_1),\ldots,k_n(v_n) \quad ==> \quad T\,(u_1,\ldots,u_q)$$

Les $k_1,\ldots,k_n$ sont des **connecteurs** associés à des termes du type $\tau_1,\ldots,\tau_n$ respectivement. Par le connecteur k_i ne peuvent circuler que des termes (messages) de type τ_i. Les t_i, v_i et u_i sont des termes qui peuvent contenir des variables.

Soit P un processus qui peut être créé dynamiquement. Le **comportement** de P est le suivant :

• Initiallement P se trouve dans un état d'inactivité S_i, S_i est un état qui n'appartient pas aux états définis dans P.

• Toutes les règles initiales de P sont pré-applicables.

• Soient $m_1,\ldots,m_n$ les messages prêts à être communiqués à travers les connecteurs $k_1,\ldots,k_n$. Une règle initiale est **applicable** s'il existe une substitution σ telle que $\forall$ i, i = 1...n : σ (μ (v_i)) = m_i. Le processus P devient un **processus actif.**

• Le processus choisit de manière non-déterministe une des règles initiales applicables et l'applique. L'état courant du processus devient :

$$\sigma (\mu (T (u_1,\ldots,u_q)))$$

A partir de ce moment, le comportement de P est comme celui des processus de FP2 standard.

Exemple 1.7 :

Nous proposons de calculer la factorielle d'un nombre "n" en utilisant un réseau infini de processus FACT. Un processus FACT a la forme suivante :

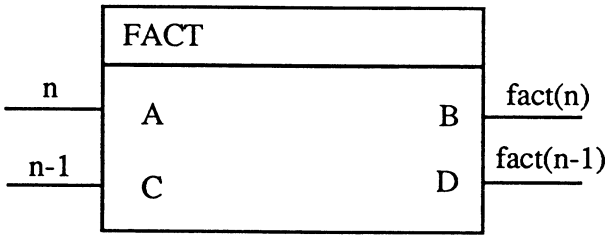


Figure 1.16

Un processus FACT a quatre connecteurs : A,B,C,D. Le connecteur A reçoit le nombre "n", FACT calcule la factorielle de "n" et envoie le résultat à travers le connecteur B. Pour calculer la factorielle de "n", le processus FACT réalise les actions suivantes :

• Si n = 0 alors la factorielle de "n" est 1, 1 est envoyé par B ;

• Si n > 0 alors

  • FACT envoie le nombre "n-1" à travers le connecteur C ;

  • FACT reçoit la factorielle de "n-1" à travers le connecteur D, et envoie par B :

$$n * fact (n-1)$$

La définition du processus FACT en FP2 avec création dynamique de processus est la suivante :

process FACT

connecteurs

A,B,C,D : **Nat**

states

T : -

S,W : **Nat**

vars

m,n : **Nat**

rules

A (n) ==> S (n)

;; fact (0) = 1

S (0) : B (1) ==> T

*;; fact (n+1) = (n+1) * fact (n)*

S (n+1) : C (n) ==> W (n+1)

W (n) : D(m) B (n*m) ==> T

endprocess

Le réseau de processus capable de calculer la factorielle d'un nombre est :

processus reseau **is**

|| { FACT_i | i = 1... } + { C_i.A_i+1 | i = 1... } + { D_i.B_i+1 | i = 1... }

endprocess

Une représentation du réseau est la suivante :

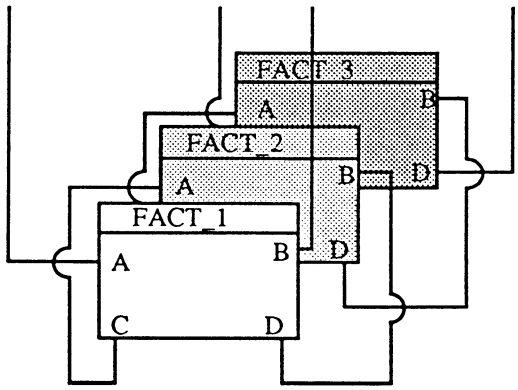


Figure 1.17

Supposons que nous voulons calculer la factorielle de 3. Nous communiquons le nombre 3 et un processus FACT : FACT_1 est activé. L'histoire de FACT_1 est :

$$S_i \xrightarrow{A\_1(3)} S(3) \xrightarrow{C\_1(2)} W(3)$$

Lorsque le nombre 2 est communiqué à travers le connecteur C_1, un nouveau processus FACT est activé : FACT_2. L'histoire de FACT_2 est :

$$S_i \xrightarrow{A\_2(2)} S(2) \xrightarrow{C\_2(1)} W(2)$$

De manière analogue le processus FACT_3 est activé et son histoire est :

$$S_i \xrightarrow{A\_3(1)} S(1) \xrightarrow{C\_3(0)} W(1)$$

Enfin, un dernièr processus est activé : FACT_4 et son histoire est :

$$S_i \xrightarrow{A\_4(0)} S(0)$$

Les processus FACT_1, FACT_2, FACT_3 et FACT_4 se synchronisent e B_1 communique la factorielle de 3 : 6

Fin de l'exemple

Selon ce principe de création dynamique, les processus sont donc créés "par nécessité" sur une structure de réseau préalablement definie.

CHAPITRE 2

Machines à Inférence Parallèlles pour le Langage des clauses de Horn

MACHINES A INFERENCE PARALLELE POUR LE LANGAGE DES CLAUSES DE HORN

Dans ce chapitre, nous présentons divers modèles de machines à inférence parallèle qui exploitent les parallélismes OU et ET. Tous les modèles présentés travaillent sur les clauses de Horn et utilisent comme méthode d'inférence la résolution. L'objectif du chapitre est de familiariser le lecteur avec les principales difficultés que l'on trouve dans le domaine et les différentes techniques employées pour les résoudre.

2.1. Modèles qui exploitent le parallélisme OU

Nous rappelons que le parallélisme OU permet de rechercher simultanément plusieurs solutions d'un but.

Pour étudier les machines à inférence qui exploitent le parallélisme OU, nous distinguons deux parties principales dans les machines:

- Le controle de l'exécution du programme ;

- La gestion de la mémoire.

Le controle d'exécution du programme peut être effectué de deux manières différentes:

La première consiste à comprendre l'exécution du programme comme l'ensemble de résolvants qui se forment. Les différents processus de la machine se chargent de travailler avec un ou plusieurs de ces résolvants. Les modèles qui suivent ce schéma sont présentés dans la section intitulée "Modèles à grosse granularité".

Sous le titre "Modèles à fine granularité", nous groupons les modèles qui contrôlent l'exécution en fonction de la structure syntaxique du programme et, parfois, en fonction d'opérations telles que l'unification.

2.1.1. Modèles à grosse granularité

Les modèles à grosse granularité se basent sur l'implémentation d'arbres de recherche parallèle. Les processus se chargent de résoudre des parties de l'arbre de recherche du programme. Ces parties de l'arbre sont divisées de telle sorte que les processus sont presque indépendants les uns des autres. Cela se traduit par un nombre réduit de communication entre les processus.

Dans cette section, nous présentons deux modèles. Celui proposé par A. Ciepielewski et S. Haridi [CiH83] est le modèle plus représentatif parmi les modèles de grosse granularité pour le parallélisme OU. Dans ce modèle, l'accent est spécialement mis sur le problème de la gestion efficace de la mémoire. La méthode du Kabu-Wake [SoS85] montre une façon simple et efficace d'affecter des parties de l'arbre de recherche à un nombre réduit de processus.

Modèle de A. Ciepielewski et S. Haridi.

Dans le modèle de A. Ciepielewski et S. Haridi, chaque résolvant est représenté par un processus appelé Interp. Chaque processus Interp a donc une liste de buts à satisfaire. Soit P un tel processus et soit L la liste de ses buts. Soit $A \in L$ le sous-but à satisfaire dans la prochaine étape d'exécution. Supposons que le programme contienne les clauses:

$$A_1 \text{ <- } B_{11},\ldots,B_{1k1}$$

...

$$A_n \text{ <- } B_{n1},\ldots,B_{nkn}$$

et supposons qu'il y ait des substitutions $\theta_1,\ldots,\theta_n$ telles que $A\theta_1 = A_1\theta_1,\ldots,A\theta_n = A_n\theta_n$. Le processus P active "n" nouveaux processus P1,...,Pn dont les résolvants respectifs sont

$$[\, L - \{A\} \cup \{B_{11},\ldots,B_{1k1}\} \,]\,\theta_1$$

...

$$[\, L - \{A\} \cup \{B_{n1},\ldots,B_{nkn}\} \,]\,\theta_n$$

Après cette opération, le processus P finit son travail.

Pour illustrer le fonctionnement du modèle nous présentons l'exemple suivant :

Exemple 2.1 :

```
G   <- A,B,C
G   <- D,E
A   <- A1,A2
D   <- D11,D12
D   <- D21
    :
    :
    <- G
```

Initialement on active le processus P1 qui doit satisfaire le but "G". Il y a deux façons différentes de satisfaire "G" (clauses G <- A,B,C et G <- D,E): deux nouveaux processus sont activés P2 et P3; ils ont comme résolvants "A,B,C" et "D,E" respectivement. Une fois que P2 et P3 sont activés, P1 finit son travail. La figure ci-dessous montre trois générations de processus pour l'exemple 2.1.

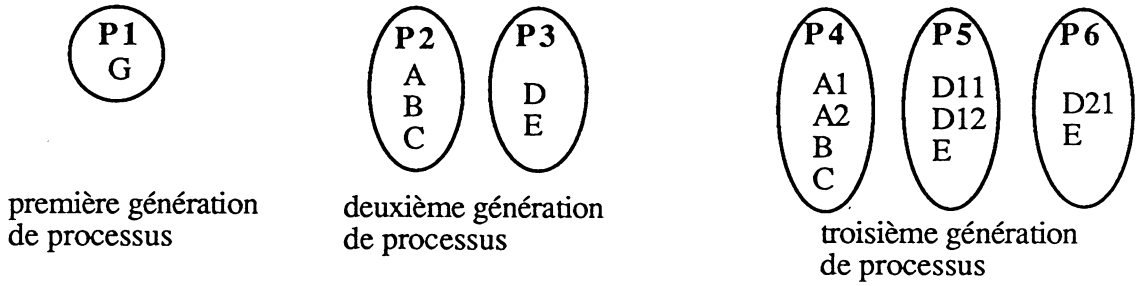


Figure 2.1

Fin de l'exemple

Méthode du KABU-WAKE

La méthode du Kabu-Wake [SoS85] est présentée comme un mécanisme d'inférence basé sur le procédé séquentiel. L'inférence est réalisée grâce à plusieurs processeurs identiques (PE). Chacun de ces processeurs travaille de manière séquentielle sur un arbre de recherche qui représente une partie de l'exécution du programme, l'arbre est parcouru suivant la technique "depth first search". A la demande d'un PE libre, l'arbre de recherche d'un PE actif sera coupé en deux parties égales (à un sous-arbre près lorsque le nombre de fils est impair). La division de l'arbre se produit dans l'alternative OU qui se trouve la plus proche de la racine de l'arbre de recherche du PE actif. La partie qui se trouve à gauche du point de coupure est l'arbre de

recherche gauche et l'autre partie est l'arbre de recherche droite. Chaque PE a donc une partie de l'arbre de recherche du programme qui lui permet de calculer des solutions différentes.

Exemple 2.2 :

Supposons que nous devons exécuter le programme suivant :

A <- B1, C1

A <- B2, C2

A <- B3, C3

B1 <- D

B1 <- E

B1 <- F

:

:

<- A

Initialement un PE (PE0) est activé pour satisfaire "A", PE0 crée un arbre de recherche dont le seul nœud est "A". "A" peut être satisfait de trois manières différentes, l'arbre de recherche de PE0 devient :

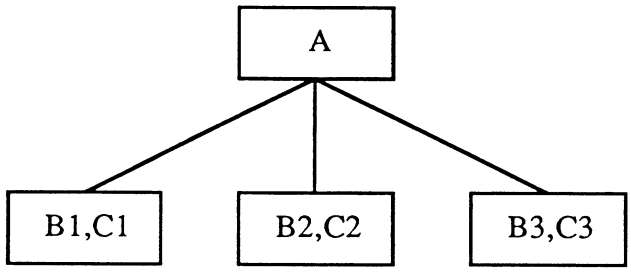


Figure 2.2

Le processeur PE1 est libre et demande à PE0 une partie de l'arbre. PE0 coupe son arbre en deux et donne à PE1 une des parties de son arbre.

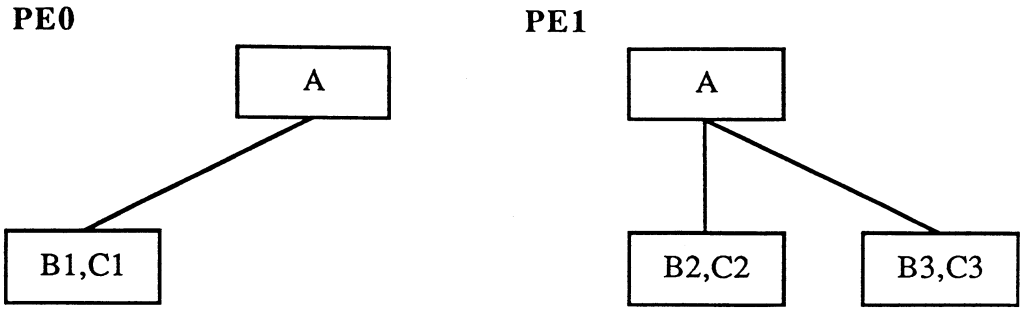


Figure 2.3

PE0 travaille avec la résolvante "B1,C1" pendant que PE1 travaille d'abord avec la résolvante "B2,C2" et enfin avec la résolvante "B3,C3".

PE0 continue son travail et son arbre de recherche devient :

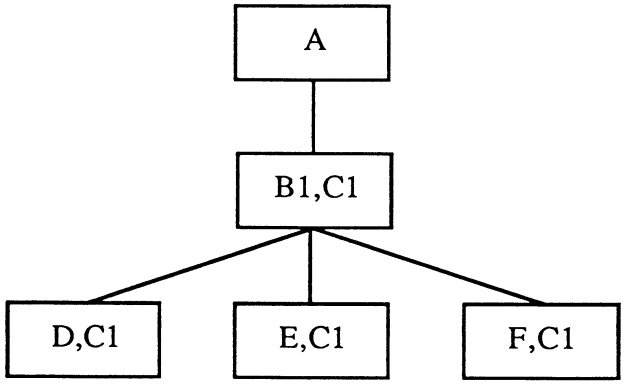


Figure 2.4

Le processeur PE2 est libre et demande à PE0 une partie de l'arbre. Les processeurs PE0 et PE2 ont donc les arbres suivants :

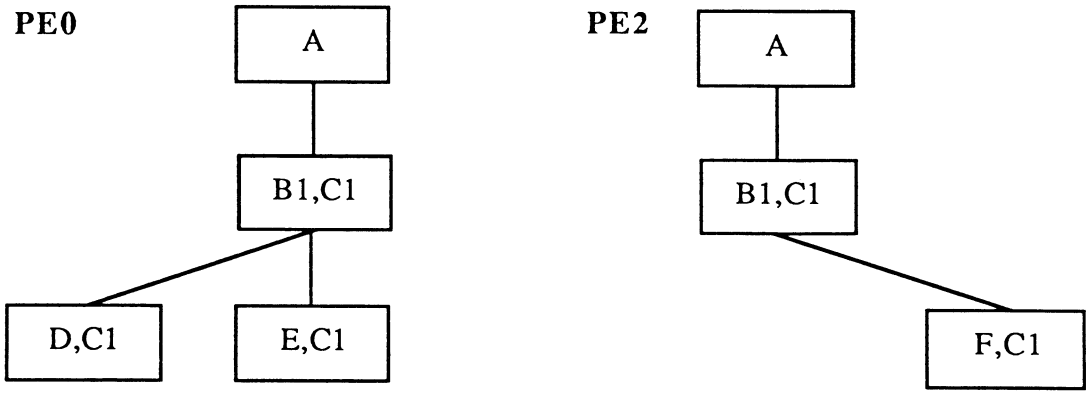


Figure 2.5

Le travail continue soit jusqu'à ce que l'on trouve la preuve, soit jusqu'à ce que l'on ne puisse pas continuer parce que la formule que le programme représente est non valide.

Fin de l'exemple

Lorsque l'on travaille avec des variables, au moment de découper un arbre de recherche en deux, les variables de l'arbre de recherche droit sont traitées comme si l'objectif à gauche du point de coupure avait échoué (rappelons que la méthode se base sur l'exécution en PROLOG séquentiel).

Le KABU_WAKE a été testé sur une architecture organisée comme la figure ci-dessous :

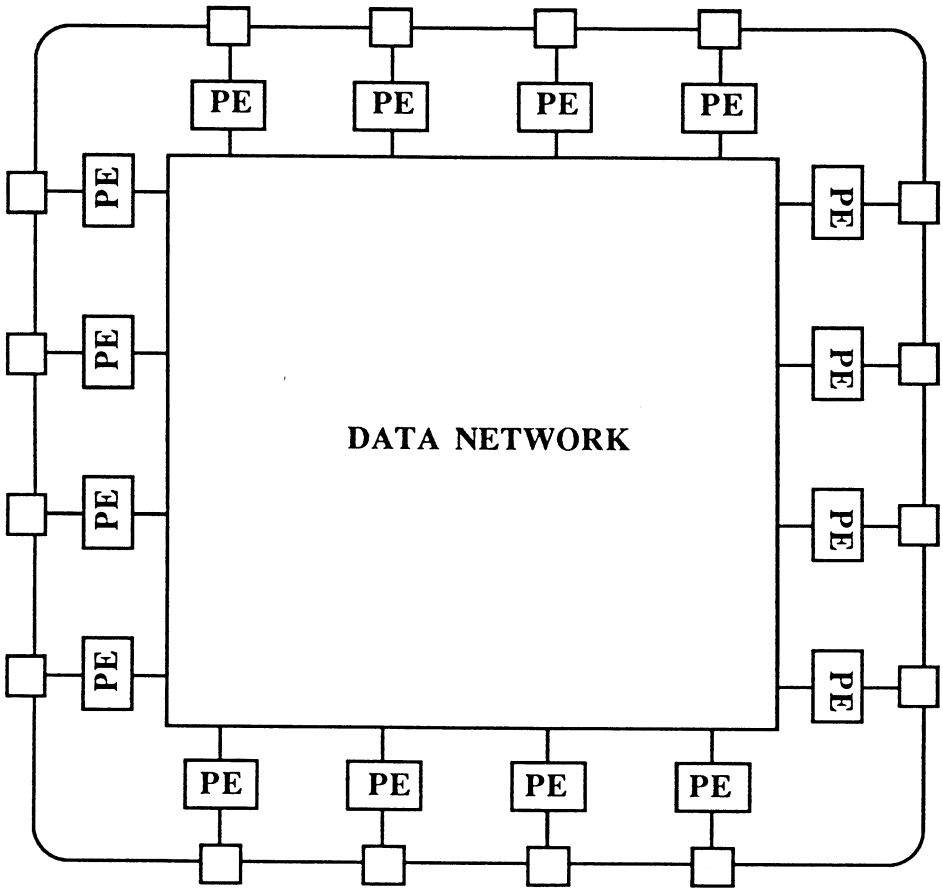


Figure 2.6

Les processus PE inactifs font leurs demandes de travail à travers le réseau CONT-NETWORK. Les arbres de recherche circulent à travers le réseau DATA-NETWORK.

Les avantages de cette méthode sont les suivants:

- Les PE's font peu de travail additionnel pour permettre le parallélisme OU.

- Il y a peu de communication entre les PE's.

L'inconvénient de la méthode : elle doit utiliser beaucoup d'espace pour prévoir une coupure de l'arbre de recherche.

Les résultats expérimentaux montrent de bonnes performances pour cette méthode.

La technique du KABU-WAKE a été utilisée dans le système ORBIT [YaN84] et en [KMI86]. D'autres travaux sont faits en utilisant comme modèle séquentiel de base la WAM ("Warren Abstract Machine") [War83]. La WAM est un des modèles d'exécution les plus efficaces en Prolog séquentiel. L'idée d'utiliser la WAM comme modèle d'exécution de base est celle de combiner les advantages du parallélisme avec une implémentation efficace de Prolog.

Méthodes pour limiter la quantité de parallélisme.

Les processus de la famille de modèles que l'on vient de présenter doivent exécuter une ou plusieurs branches de l'arbre de recherche. Le nombre de branches de l'arbre peut être supérieur au nombre de processeurs de la machine sur laquelle on travaille. Il est donc nécessaire d'établir des critères pour assigner les processus aux processeurs disponibles. Plusieurs critères peuvent être appliqués :

1.- Chercher les meilleurs chemins ;

2.- Effacer certaines branches de l'arbre ;

3.- Activer des nouveaux processus seulement quand la machine possède des processeurs disponibles.

Bien entendu, il est toujours possible de faire une assignation aveugle des processus aux processeurs.

Les méthodes qui cherchent les meilleurs chemins utilisent des heuristiques pour choisir les branches de l'arbre qui ont les probabilités les plus grandes de réussite. L'information heuristique est modifiée légèrement après chaque exécution de programmes. Certaines méthodes utilisent seulement une stratégie fixe, c'est-à-dire une heuristique qui ne varie pas au cours de l'exécution du programme [LiW85], d'autres utilisent une stratégie adaptative, c'est-à-dire une heuristique qui peut évoluer pendant l'exécution du programme [LiH85].

La technique qui consiste à effacer certaines branches de l'arbre a été proposée par A. Ciepielewski et S. Haridi dans [CiH84]. Son idée de base : dans les programmes logiques apparaissent des prédicats dont la solution est unique. Pour ces prédicats, que l'on appellera *buts fonctionnels*, il est nécessaire de considérer une seule branche de l'arbre. Comme exemples de buts fonctionnels on peut citer:

1.- Les relations définies comme fonctions pour certains profils de paramètres, par exemple : FACT(N,M) ;

2.- Les relations définies pour vérifier si une condition est satisfaite ou non, par exemple : GREATERZERO(N).

Lorsqu'une solution pour un but fonctionnel est trouvée, les autres processus qui cherchent des solutions pour ce but sont détruits.

Enfin, le dernier critère énoncé qui consiste à activer des nouveaux processus seulement quand la machine a des processeurs disponibles est la technique utilisée par le KABU-WAKE.

2.1.2. Modèles à fine granularité.

Dans les modèles à fine granularité, la structure du réseau de processus soit résulte d'une partition physique du programme, soit met en évidence des opérations pour exécuter le programme. Le nombre de processus nécessaires pour ces modèles est beaucoup plus grand que le nombre de processus dans les modèles à grosse granularité. Cela a principalement deux conséquences : les modèles à granularité fine sont plus modulaires mais les processus sont inter-dépendants. L'inter-dépendance des processus se traduit en un grand "overhead" de communication. D'autre part, la modularité fait que ces modèles sont plus aptes à contrôler la quantité de parallélisme.

Dans cette section, nous présentons deux modèles, le modèle de J. Conery et D. Kibler [CoK81] où la structure du réseau reflète une partition physique du programme et le modèle de S. Umeyama et K. Tamura [UmT83] où la structure du réseau reflète les opérations à réaliser pour exécuter le programme.

Modèle de J. Conery et D. Kibler

J. Conery et D. Kibler présentent un modèle pour l'interprétation parallèle OU de programmes logiques [CoK81]. Le modèle consiste en un ensemble de processus indépendants qui communiquent à travers des messages. Les processus sont activés pour résoudre des petits morceaux des programmes logiques. Un processus termine son travail lorsque toutes les solutions de son sous-problème ont été trouvées.

Il y a deux types de processus: processus OR et processus AND. Un processus OR se charge de résoudre un but en particulier. La fonction d'un processus AND est de produire une solution pour une conjonction de buts. Les processus OR et AND sont organisés comme un arbre de recherche.

Il y a trois types de messages: "success", "fail" et "redo". "Fail" et "success" sont toujours envoyés de fils au père, et "redo" est toujours envoyé de père au fils. Lorsqu'un processus contient une solution, il l'envoie à son père avec le message "success". Si un processus ne trouve pas de solutions, il envoie le message "fail" à son père et cesse son activité. Un message "redo" est envoyé de père au fils pour demander une nouvelle solution.

Lorsqu'un processus OR est activé pour résoudre le but L, il essaie de résoudre L de la manière suivante :

- Il cherche les clauses du programme dont les têtes s'unifient avec L. Si il n'y a pas de telles clauses, le processus OU envoie le message "fail" à son père et termine ;
- Si L s'unifie avec une assertion, le processus OR peut envoyer à son père un message "success". Par exemple, si L est P(a,X) et le programme contient l'assertion P(Z,b), le processus OR envoie à son père le message : success(P(a,b)).
- Si L s'unifie avec la tête d'une clause conditionnelle, le processus OR active un processus AND comme descendant pour résoudre le corps de la clause. Par exemple, si L est P(a,X,Y), le programme contient la clause conditionnelle P(V,W,c) <- Q(V) $\wedge$ R(W) et le processus AND envoie le message success(Q(a)$\wedge$R(b)) alors le processus OR envoie à son père le message success(P(a,b,c)).

Pour chaque clause dont la tête s'unifie avec L, le processus OR active un processus AND pour résoudre le corps de la clause. Lorsqu'un processus OR reçoit un résultat d'un de ses processus AND, il envoie à son père ce résultat. A partir de ce moment, le processus OR n'envoie plus de résultats à son père, sauf si celui-ci lui envoie un message "redo".

Les processus AND n'exploitent pas le parallélisme ET, ils résolvent leurs conjonctions de gauche à droite de façon séquentielle. Le processus AND qui doit résoudre une conjonction, active un processus OR pour chaque littéral de la conjonction. Lorsqu'un littéral est satisfait, les liaisons de ses variables sont prises en compte pour le littéral suivant. Si le dernier littéral est satisfait alors une solution a été obtenue et le processus AND envoie à son père un message "success". Si un littéral ne peut pas être résolu, le littéral précédent doit être résolu d'une façon différente (les liaisons de variables doivent être refaites), un message "redo" est envoyé au processus OR correspondant.

Exemple 2.3 :

Supposons le programme suivant :

G1 (Y) <- A (Y)

G1 (Z) <- B (Z)

A (a)

B (b)

G2 (b)

<- G1 (X), G2 (X)

Initialement, un processus AND est activé pour résoudre la conjonction des littéraux : G1(X)∧G2(X). Le processus AND active un processus OR pour satisfaire le but G1(X).

Le processus OR trouve deux clauses dont ses têtes s'unifient avec G1(X) et il active deux processus AND pour résoudre les corps de ces clauses.

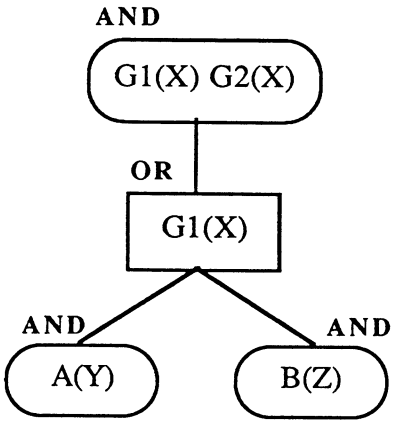


Figure 2.7

Les deux processus AND qui viennent d'être activés par le processus OR travaillent en parallèle, chacun active un processus OR qui résoud son littéral. Des messages "success" vont de ces processus OR à leurs AND respectifs et d'eux mêmes au OR qui doit satisfaire G1(X).

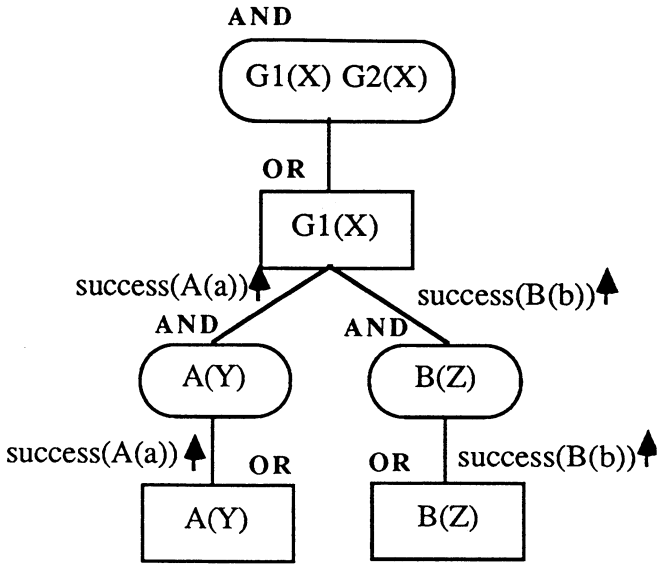


Figure 2.8

Si le OR qui doit résoudre G1(X) reçoit d'abord le message success(A(a)), il envoie le message à son processus AND père qui essaie de satisfaire G2(a) et échoue. Le AND envoie alors un message "redo" a son processus OR fils qui a un nouveau message à lui envoyer : success(B(b)). Le travail continue et la conjonction G1(X)∧G2(X) est résolue en liant X à "b".

Fin de l'exemple

Le parallélisme ET a été aussi introduit dans ce modèle par [CoK85]. Le modèle étendu sera montré dans la section 2.2.2.

Modèles basées sur les machines Data-flow

S. Umeyama et K. Tamura présentent un modèle "data flow" qui exploite le parallélisme OU de programmes logiques [UmT83].

Les clauses d'un programme logique sont traduites par des graphes "data flow" indépendants. Ces graphes sont formés à partir de nœuds et d'arêtes. Les arêtes permettent le transport de "tokens" d'un nœud à un autre. Un "token" est un message qui circule entre les nœuds des graphes "data flow" du modèle. Un "token" contient l'information d'un terme plus l'information qui permet l'identification du "token" et sa circulation à travers le graphe.

L'information d'un terme comprend le terme tel qu'il apparaît dans le programme, ainsi que les valeurs acquises par ses variables pendant l'exécution. Par exemple, F(X,Y) est représenté ainsi : [F(X,Y),<X,undef>,<Y,undef>]. Supposons que par unification, X soit liée à 0, alors F(X,Y) est représenté de la manière suivante : [F(X,Y),<X,0>,<Y,undef>].

Chaque clause d'un programme est mise sous la forme d'un sous-graphe "data flow" qui a comme composants les cinq types d'unités que l'on montre dans la figure ci-dessous. Un point sur chaque port représente un "token" placé sur le port.

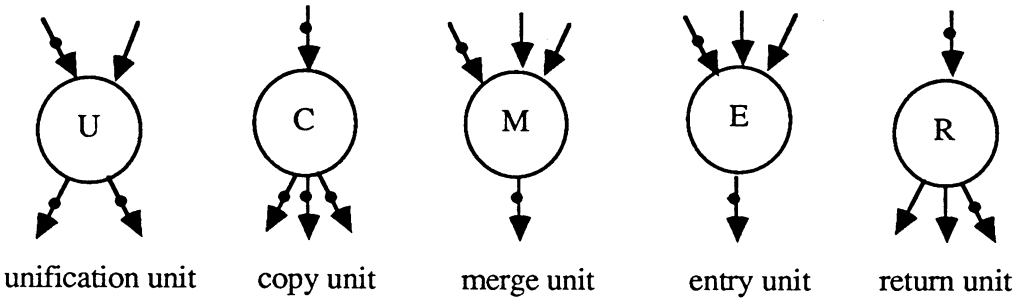


Figure 2.9

Les différents sous-graphes sont liés entre eux suivant la structure syntaxique du programme.

Ensuite nous décrivons les fonctions des nœuds présentés dans la figure 2.9.

Unification Unit.

Ce type de nœud a deux ports d'entrée ("trigger port", "store port") et deux ports de sortie.

Lorsqu'un "token" arrive par le "store port", il est stocké dans la "unification unit".

Lorsqu'un "token" t arrive au "trigger port", on unifie t avec une copie du "token" qui se trouve dans la "unification unit". Si l'unification réussit, le "token" qui résulte de l'unification est placé dans les ports de sortie de la "unification unit", sinon les "tokens" sont détruits.

Exemple 2.4:

Supposons que le "token" : [F(X,Y),<X,0>,<Y,1>] arrive au "store port" d'une "unification unit" et que le "token" : [F(X,Y),<X,0>,<Y,undef>] arrive au "trigger port" de la même "unification unit". Comme conséquence, par les deux ports de sortie de cette unité sortent deux "tokens" de la forme suivante : [F(X,Y),<X,0>,<Y,1>].

Fin de l'exemple

Copy unit.

Lorsqu'un "token" arrive par le port d'entrée, il est copié et envoyé à chaque port de sortie.

Merge unit.

Chaque "token" qui arrive par un port d'entrée est copié dans le port de sortie.

Return unit.

Le dernier nœud de chaque sous-graphe "data flow" est un "return unit". Lorsqu'un "token" arrive au port d'entrée d'un nœud "return", il est envoyé par le port de sortie, au graphe qui avait déclanché l'appel.

Entry unit

Le premier nœud d'un sous-graphe "data flow" est une "entry unit", il reçoit les "query tokens" provenant des ports de sortie de différents "return units" et les envoie vers la "unification unit" correspondante de son sous-réseau.

La figure ci-dessous montre le sous-graphe "data flow" pour une clause de la forme:

$$P <- Q1,Q2,\ldots,Qn$$

Un port qui finit sur une marque "x" veut dire que le "token" placé sur le port est détruit.

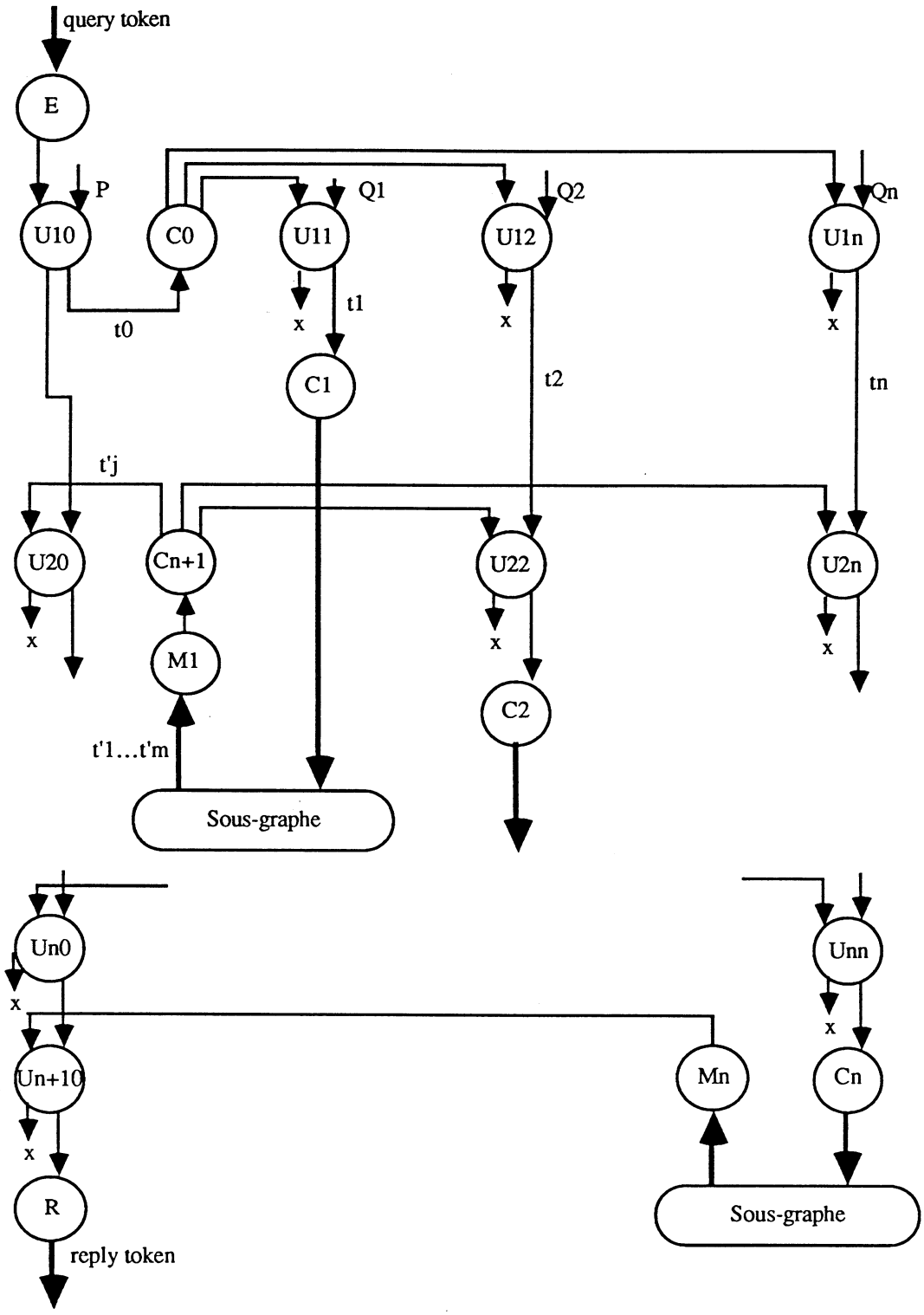


Figure 2.10

Un "query token" est placé dans le port d'entrée de la "entry unit" du sous-graphe et des "reply tokens" s'acheminent de la "return unit" de ce sous-graphe vers le sous-graphe qui a envoyé le "query token".

Lorsqu'un "query token" est placé dans le port d'entrée de la "entry unit" du sous-graphe, la "unification unit" U10 unifie ce "token" avec P, le résultat ("token" t0) est stocké dans U20 et s'achemine vers les "unification units" U11,U12,U1n. Les "unification units" U1i ($1 \leq i \leq n$) unifient t0 avec Qi.

Le "token" t1 est envoyé comme "query token" vers les sous-graphes qui représentent les clauses dont sa tête est Q1 et la "merge unit" reçoit les "reply tokens" t'j. Chacun de ces "reply tokens" sont acheminés vers les "unification units" U20, U22, U23,...,U2n.

La "unification unit" U20 unifie chaque t'j avec le "token" produit par U10. Cela veut dire que chaque "token" produit par U20 est le résultat de l'unification de t'j avec le terme qui représente le littéral P unifié avec le "query token".

Les "unification units" U2i ($2 \leq i \leq n$) unifient chaque t'j avec le "token" qui représente le littéral Qi où sur ses variables ont été appliquées les substitutions dues à l'unification du "query token" avec P.

Les autres "unification units" travaillent de manière similaire.

Exemple 2.5 :

Le programme suivant représente la relation familier : "grand-père" [UmT83] :

GP(X,Y) <- P(X,Z), P(Z,Y)

P(aiko,kazuo)

P(aiko,yooko)

P(kazuo,emiko)

<- GP(aiko,Q)

la figure suivante montre le graphe "data flow" du programme et les "tokens" qui parcourent le graphe. Les "tokens" de la figure ne montrent que l'information qui permet de suivre l'exemple plus facilement, par exemple, le "token" t_2 contient aussi la valeur de la variable Q (undef).

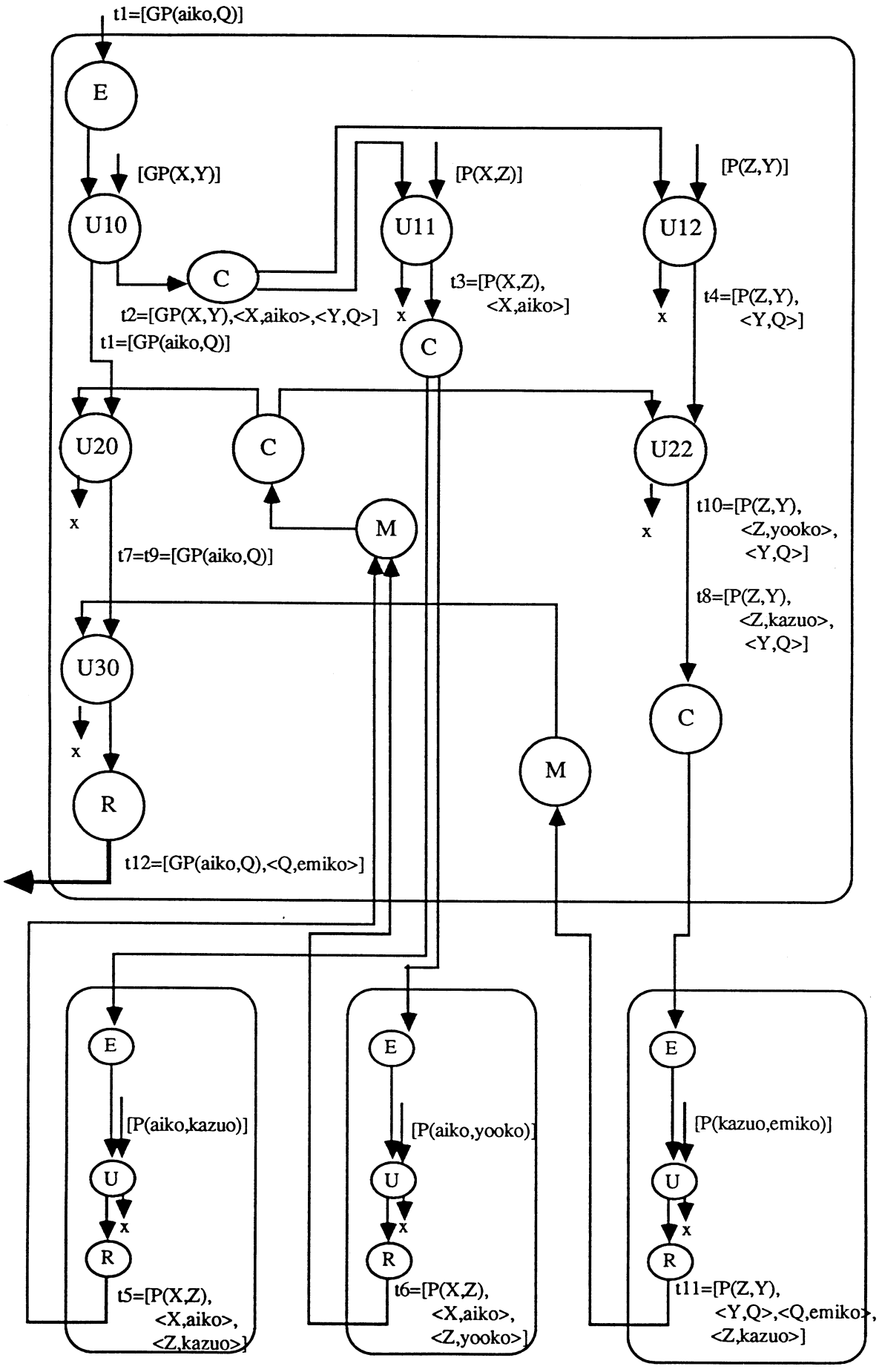

t1=[GP(aiko,Q)]
E
[GP(X,Y)]
U10
C
t2=[GP(X,Y),<X,aiko>,<Y,Q>]
t1=[GP(aiko,Q)]
[P(X,Z)]
U11
x
t3=[P(X,Z),
<X,aiko>]
C
[P(Z,Y)]
U12
x
t4=[P(Z,Y),
<Y,Q>]
U20
x
C
M
t7=t9=[GP(aiko,Q)]
U22
x
t10=[P(Z,Y),
<Z,yooko>,
<Y,Q>]
t8=[P(Z,Y),
<Z,kazuo>,
<Y,Q>]
U30
x
R
t12=[GP(aiko,Q),<Q,emiko>]
C
M
E
[P(aiko,kazuo)]
U
x
R
t5=[P(X,Z),
<X,aiko>,
<Z,kazuo>]
E
[P(aiko,yooko)]
U
x
R
t6=[P(X,Z),
<X,aiko>,
<Z,yooko>]
E
[P(kazuo,emiko)]
U
x
R
t11=[P(Z,Y),
<Y,Q>,<Q,emiko>,
<Z,kazuo>]

Figure 2.11

Fin de l'exemple

Dans ce modèle, chaque clause du programme est traduite par un sous-graphe, ces sous-graphes sont liés entre eux selon la structure syntaxique du programme. Les unités d'un sous-graphe doivent travailler avec des "tokens" qui viennent des différents appels a la même clause et avec ceux qui représentent différentes solutions (OR-solutions) d'un but. Un "token" stocke donc l'information qui lui permet d'identifier à quelle activation d'une clause appartient le "token". Le "token" contient aussi l'information nécessaire pour le différencier des autres OR-solutions.

Lorsqu'un "token" arrive à une "unification unit", il faut l'unifier avec le "token" qui appartient à la même OR-solution.

B. Schwinn et G. Barth ont proposé une extension de ce modèle de façon à considérer aussi le parallélisme ET [ScB86].

2.1.2.1. Méthodes pour limiter la quantité de parallélisme.

Pour les modèles à fine granularité, la quantité de parallélisme peut aller d'un système totalement séquentiel au degré maximal de parallélisme.

Dans le modèle de K. Furukawa, N. Nitta et Y. Matsumoto [FNM82], on observe un parallélisme "goutte-à-goutte". L'idée est que, pendant qu'un processus P travaille avec un résultat donné par un de ses fils P', P' continue avec son travail afin de trouver une autre solution, et seulement une autre. Si la solution est prête, mais que P travaille encore sur l'ancienne solution, P' se bloque. P' continue son travail pour être prêt à donner une solution.

Le degré maximal de parallélisme se trouve dans le modèle de G. Lindstrom et P. Panangaden [LiP84] où chaque processus produit toutes les solutions possibles. Les processus AND doivent accepter et travailler avec toutes les solutions que leurs processus OR produisent.

Le modèle de J.S. Conery et D.F. Kibler peut être considéré comme de parallélisme moyen puisque ses processus OR pourront produire plusieurs solutions, mais ses processus AND travailleront sur chacune de ces solutions séparément.

2.1.3. Méthodes de gestion de mémoire

Etant donné un but "G" et "n" clauses capables de le satisfaire, le parallélisme OU permet de trouver "n" solutions alternatives à "G" en essayant les "n" clauses en parallèle. Mais quelques variables de "G" peuvent être liées à différentes valeurs comme résultat de l'unification avec les têtes de clauses. Si une des "n" clauses doit satisfaire les buts de son corps, n'importe quelle variable de "G" peut être liée à plusieurs valeurs.

Le problème du parallélisme OU consiste à trouver une manière correcte et efficace de stocker les valeurs des variables. Une façon de résoudre le problème est en utilisant des contextes. Un **contexte** est un ensemble de faits <variable,terme> où les termes sont les valeurs auxquelles les variables d'une clause sont liées lorsque celle-ci est invoquée.

Pour chaque clause que l'on utilise pour satisfaire un but "G", un nouveau contexte est nécessaire. Par ailleurs, une fois qu'une variable est liée à une valeur par unification, sa valeur ne peut pas être modifiée. Cela veut dire qu'une variable liée peut être partagée par plusieurs processus.

Dans cette section, nous montrons différentes méthodes de gestion de mémoire. Cependant, il faut tenir compte du fait que ce sont les modèles à grosse granularité qui ont développé les méthodes de gestion de mémoire les plus efficaces.

Les méthodes que nous décrivons ici, travaillent toutes sur un modèle à grosse granularité où chaque processus représente une résolvante. Les méthodes peuvent être modifiées à pouvoir travailler sur un autre type de modèle.

Méthode des répertoires.

Cette méthode a été développée par A. Ciepielewski et S. Haridi en [CiH83]. Dans leur modèle de machine d'inférence pour les clauses de Horn, chaque processus P a un répertoire associé. Le répertoire dispose des adresses des contextes avec lesquels P peut travailler. Les contextes qui ont toutes leurs variables liées peuvent être partagés par plusieurs processus. Les contextes qui ont des variables non liées doivent être dupliqués.

Lorsque un processus P active les processus P1,...,Pn, il active aussi les répertoires de P1,...,Pn et il détruit son propre répertoire. Les contextes référencés par le répertoire de P doivent être scrutés pour savoir lesquels doivent être partagés et lesquels doivent être copiés pour les processus fils.

Pour illustrer la méthode, nous allons travailler avec l'exemple suivant :

Exemple 2.6 :

G (F(a)) <- G1 (F(a))

G (b) <- G2 (b,Z)

G1 (U) <- G11 (U)

G1 (U) <- G12 (U,U)

G2 (b,c) <- G21 (W)

G2 (b,d) <- G22 (W,W)

<- G (X)

Initialement le processus qui représente la clause de but initiale (G (X)) est activé et un répertoire lui est associé. Ce répertoire contient l'adresse du contexte qui stocke la valeur de X.

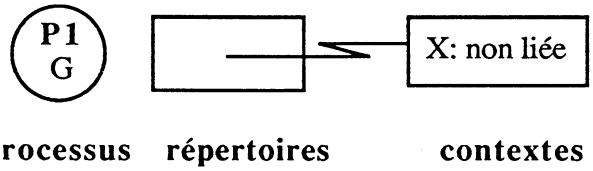


Figure 2.12

L'objectif "G" a deux manières différentes de se satisfaire :

G (F(a)) <- G1 (F(a))

G (b) <- G2 (b,Z)

Deux processus différents sont donc activés : P2 et P3, et avec eux, les répertoires associés. Etant donné que P1 a un contexte associé où il y a une variable non liée, P2 et P3 ne peuvent pas partager ce contexte. La situation obtenue est montrée dans la figure suivante :

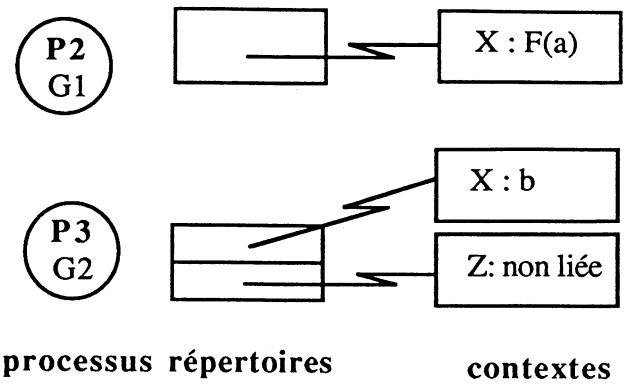


Figure 2.13

Les processus P2 et P3 travaillent alors de manière indépendante. Le contexte associé à P2 a toutes ses variables liées, ce contexte est donc partagé par les processus P4 et P5 qui se substitueront au processus P2.

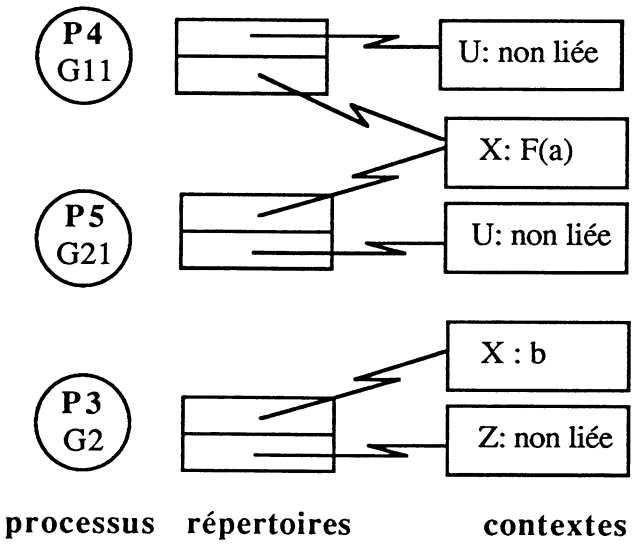


Figure 2.14

Fin de l'exemple

L'inconvénient de cette méthode : le parcours des répertoires est nécessaire pour trouver les contextes à dupliquer pour les processus fils.

Méthode des arbres de répertoires.

Un autre modèle de gestion de mémoire est proposé par A. Ciepielewski et S. Haridi dans [CiH83]. Dans ce modèle amélioré, il y a un arbre de répertoires qui est créé pendant l'exécution du programme. A chaque processus est associée une feuille de cet arbre.

Lorsqu'un processus P est activé, son répertoire est créé, et devient une nouvelle feuille de l'arbre, descendante du répertoire de son processus père. Le seul contexte créé est celui que P introduit. Les autres entrées du répertoire n'ont aucun contexte associé.

Pour avoir accès à une variable, on la cherche dans les contextes associés au répertoire courant. En cas d'échec, la variable est cherchée dans les contextes associés au répertoire père. La recherche continue jusqu'à ce que l'on trouve la variable. Une fois que la variable est trouvée dans un contexte C, si le contexte a toutes ses variables liées alors l'adresse de C est copiée dans tous les répertoires parcourus sinon, une copie de C est faite comme dans la méthode précédente et son adresse est copiée dans tous les répertoires parcourus.

Dans l'exemple développé pour la méthode précédente, le processus qui représente la clause de buts initiale (G(X)) est activé et un répertoire lui est associé de la même manière que pour la méthode précedente.

"G" a deux manières différentes d'être atteint. Deux processus différents sont donc activés : P2 et P3, et avec eux les deux répertoires associés ainsi que le montre la figure ci-dessous.

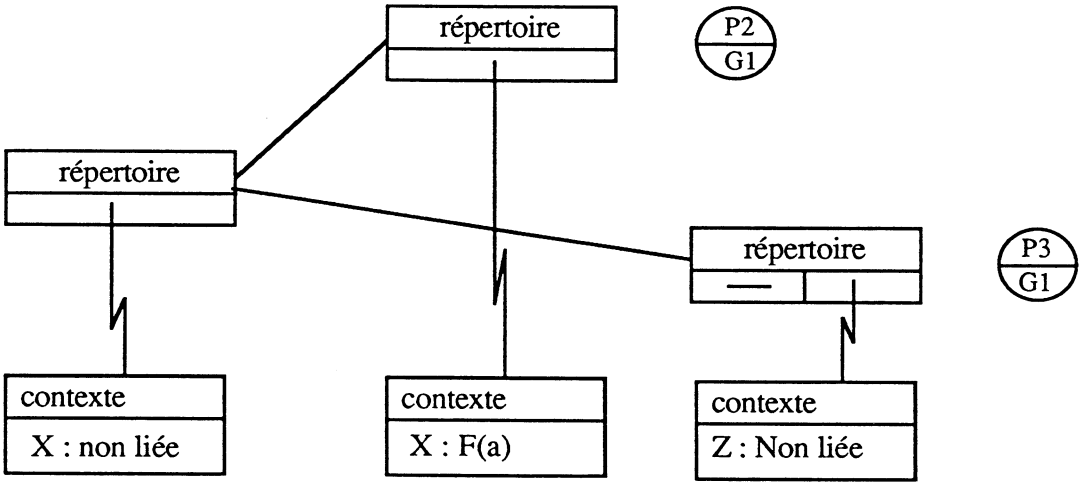


Figure 2.15

Lorsque P3 doit faire l'unification de "b" avec "X" (non liée) qui se trouve dans le contexte associé au répertoire père du répertoire associé à P3, un nouveau contexte est créé et son adresse est copiée dans le répertoire associé à P3.

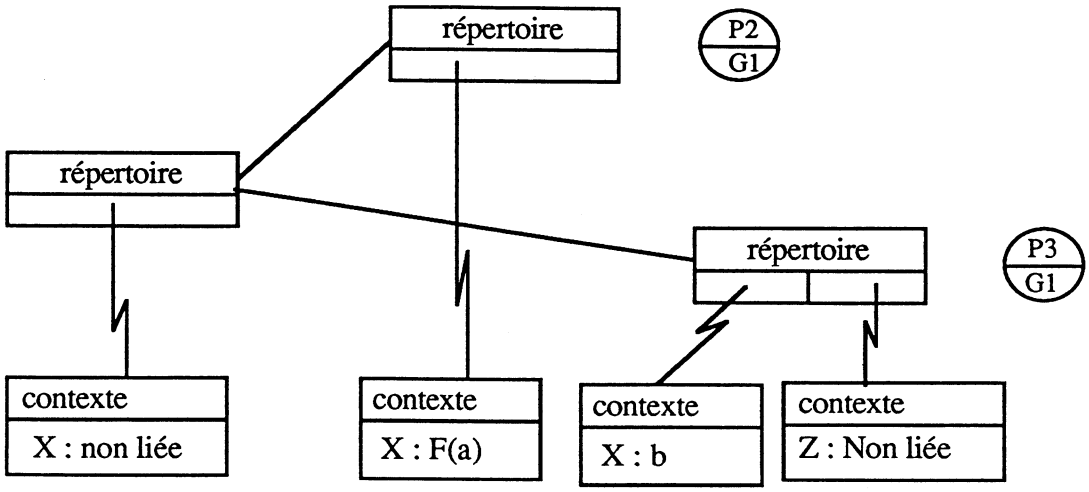


Figure 2.16

L'inconvénient de cette méthode est que le premier accès à un contexte est plus cher que celui de la méthode précédente. Ne copier un contexte que lorsqu'il est nécessaire d'avoir accès à une de ses variables, empêche la création superflue de contextes et il en résulte un gain de temps et de mémoire.

Méthode des fenêtres.

Ce mécanisme de gestion de mémoire se base sur le fait que l'unification du but à satisfaire avec la tête d'une clause ne lie généralement qu'un nombre réduit de variables des contextes ancêtres [Bor84].

L'idée consiste à stocker les liaisons de variables (faites par l'unification) dans des fenêtres. Ces fenêtres sont accessibles par le processus qui représente la clause invoquée et par ses descendants.

Les informations suivantes sont associées à chaque processus P au moment de sa création:

- un contexte local avec toutes les variables de la clause représentée par P ;

- une fenêtre locale où sont placées toutes les liaisons faites sur les variables des contextes ancêtres de P.

Si le résultat de l'invocation de la clause est une unification qui lie une variable appartenant à un contexte ancêtre, alors l'adresse de la variable et sa liaison sont placées dans la fenêtre locale. Cela veut dire que les différentes liaisons qu'une variable peut avoir en raison du parallélisme OU, sont stockées dans les fenêtres des processus qui représentent les alternatives OU.

Exemple 2.7 :

Soit le programme suivant :

G (a,U) <- G1 (U,U)

G (b,V) <- G2 (V)

<- G (X,Y)

Initialement le processus P1 est activé, il représente la clause de buts : G (X,Y). Le contexte C1 stocke les valeurs associées aux variables de la clause de buts (les deux variables sont non liées). La fenêtre F1 ne contient aucune valeur.

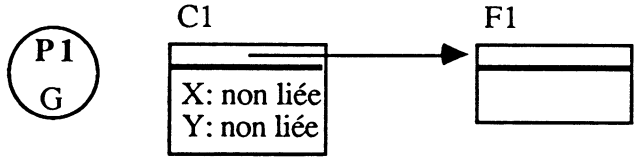


Figure 2.17

Le prochain pas consiste à activer deux processus P2 et P3 qui représentent les clauses de buts : G1(U,U) et G2(V) respectivement. La fenêtre F2 associée à P2 contient l'adresse de X et sa valeur associée : "a". Le contexte C2 associé à P2 contient U et sa valeur associée: "non liée".

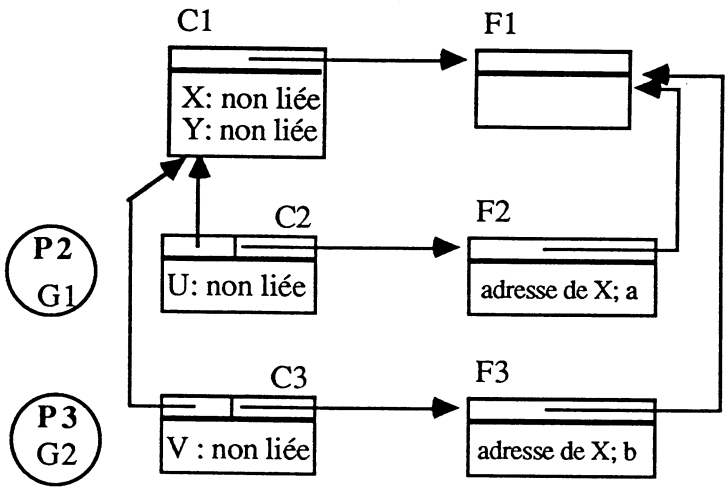


Figure 2.18

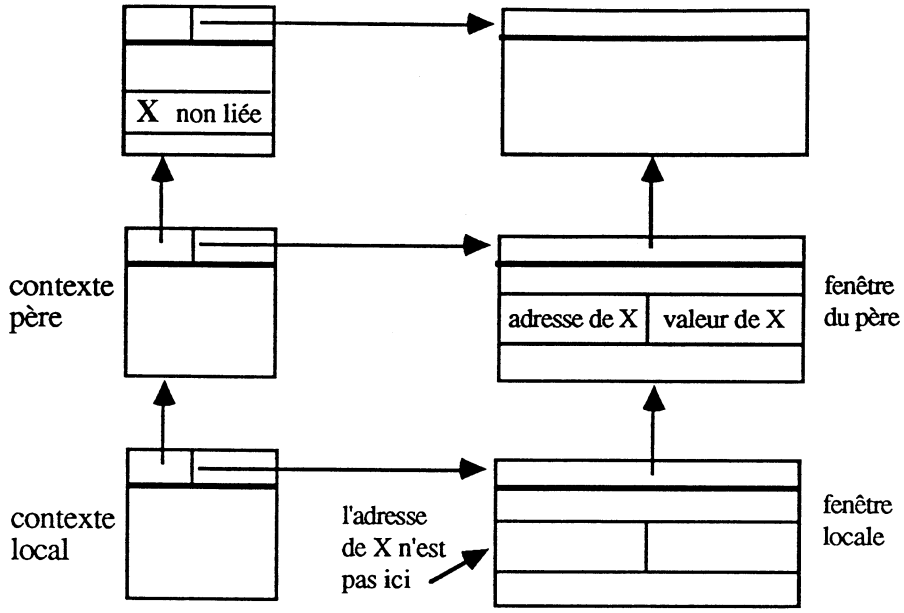
Fin de l'exemple

Pour avoir accès à une variable V, on suit la démarche suivante :

V est cherchée dans le contexte local, si elle n'y se trouve pas alors V est cherchée dans le contexte père. La recherche continue jusqu'à ce que V soit trouvée.

Si V est liée, on utilise sa valeur.

Si V est non-liée, on cherche son adresse dans la fenêtre locale. Si elle n'est pas trouvée, on la cherche dans la fenêtre de son processus père, la recherche continue jusqu'à ce que l'on ait trouvé l'adresse de V (et avec l'adresse, sa valeur), ou jusqu'à l'épuisement de toutes les possibilités. Dans ce dernier cas, V appartiendra à la fenêtre locale et sa valeur sera "non-liée".



Recherche de la variable X

Figure 2.19

Une variable non liée qui n'appartient pas au contexte local est copiée dans la fenêtre associée au contexte la premier fois que ce contexte la demande. Les accès ultérieurs à cette variable dans ce contexte sont comparables à ceux réalisés sur les variables du contexte. Cela permet de copier seulement les variables qui sont nécessaires (pas tout un contexte) et la copie se réalise lorsque la variables est demandée.

Méthode par importation de variables.

Cette technique de gestion de mémoire [Lin84] se base sur le fait que l'unification ne peut lier que les variables non-liées du contexte ancêtre. L'idée de cette technique est d'importer dans le contexte local toutes les variables non liées du contexte ancêtre.

Lorsqu'une clause est invoquée, un processus P est activé. A chaque processus P est associé un bloc B. Un bloc est un ensemble de positions de mémoire qui contiennent :

• l'identification des variables qui appartiennent à la clause considerée ;

• la référence au bloc associé au processus père de P (processus qui active le processus P);

• les références à trois vecteurs : vecteur de variables V_v, vecteur d'importation V_i et vecteur d'exportation V_e. L'information associée à ces trois vecteurs sera précisée dans la suite.

Nous notons $<B,V_v,V_i,V_e>$ l'information associée au processus P et $<B',V'_v,V'_i,V'_e>$ l'information associée au processus père de P.

Ce vecteur de variables V_V est un ensemble de positions de mémoire qui contiennent la valeur des variables qui appartiennent à la clause plus la valeur d'autres variables.

Le vecteur d'importation V_i est une copie de V'_V. Pour chaque variable non liée dans V'_V, une position est ajoutée dans le vecteur de variables V_V. Chaque position dans V_i qui correspond à une variable non liée du bloc B' contient alors une référence au vecteur de variables V_V. Cette référence indique la position de la valeur de la variable dans le vecteur de variables V_V.

Pour illustrer la méthode d'importation de variables, nous travaillerons dans cette section avec le programme suivant :

```
G ( U,V,W,S )  <- G1 (c,d,V )

G1 ( X, Z,e )  <- G2 ( X,Y,Z )

                  :

                  :

                  <- G ( a,V1,b,V2 )
```

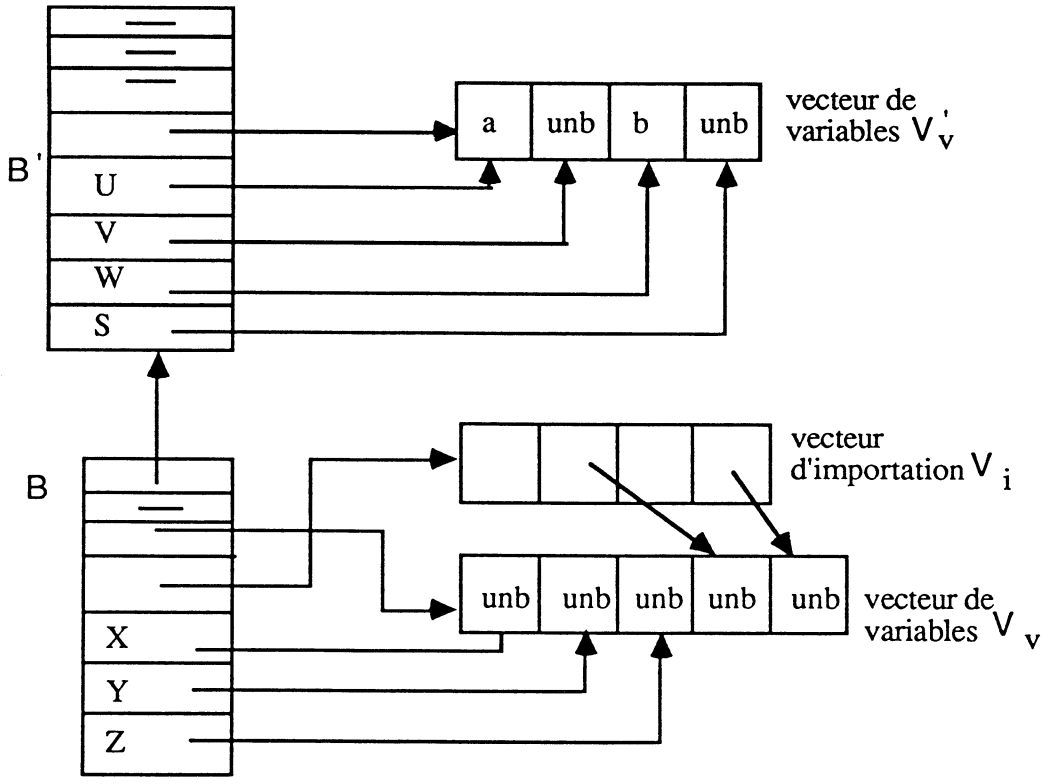


Figure 2.20

Pour avoir accès à une variable V, on suit la démarche suivante :

Supposons que l'on se trouve dans le bloc B. V est cherchée dans le contexte local. Si elle s'y trouve, on travaille avec sa valeur. Si elle ne s'y trouve pas alors V est cherchée dans le contexte père. La recherche continue jusqu'à ce que V soit trouvée. Soit B' le bloc dans lequel V est trouvée.

Si la variable V est liée, on utilise sa valeur.

Si la variable V est non liée, cela veut dire qu'elle a été importée. Connaissant sa position dans le vecteur de variables de B', en examinant alors le vecteur d'importation du bloc fils de B', si V est liée dans le vecteur de variables associé au bloc fils de B', la recherche est terminée. Sinon, il faut continuer la recherche à travers les vecteurs d'importation des descendants ultérieurs de B', qui sont tous ancêtres de B.

Remarquons que si le bloc fait une nouvelle demande de V cette recherche doit être entièrement reprise.

Lorsqu'un processus P (avec <B,V_v,V_i,V_e>) termine l'exécution de tous les buts de sa clause, avant de satisfaire le prochain but, un nouveau bloc père est créé. Ce nouveau bloc reçoit les valeurs de toutes les variables de V_i avec leurs nouvelles valeurs et les place dans son vecteur de variables. Toutes les variables non liées de V_v sont exportées, au moyen d'un vecteur d'exportation, vers le vecteur de variables de B. Nous dirons que le bloc B a été exporté.

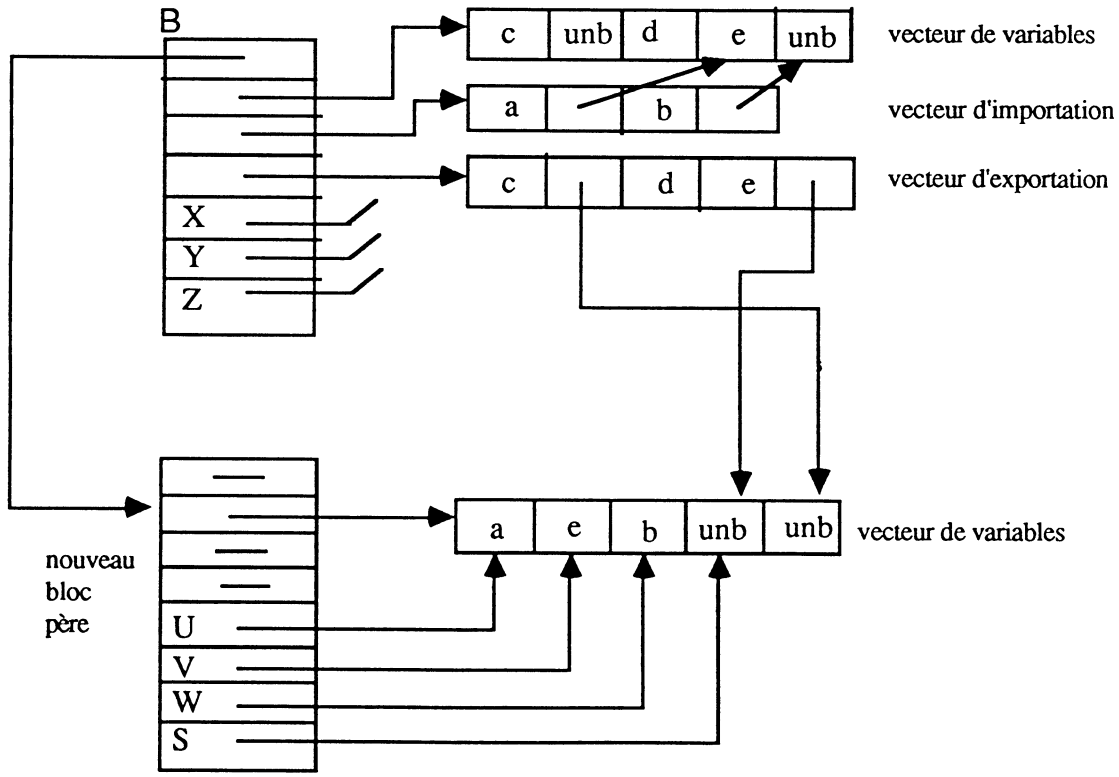


Figure 2.21

La façon de trouver une variable comporte donc aussi la règle suivante : Si elle est non liée et elle se trouve dans un bloc B qui a été exporté, sa valeur est examinée dans le bloc père de B. La recherche continue jusqu'à ce que l'on trouve une valeur pour la variable ou jusqu'à ce que l'on arrive à un bloc qui n'a pas été exporté.

Cette technique est plus complexe que les techniques précédentes et elle est chère en mémoire. Mais elle dépense moins de temps que les autres pour faire l'unification. Cela grâce au fait que les variables qui doivent être liées (celles du processus P ou celles du père de P) appartiennent au bloc associé à P.

Ces trois techniques de gestion de mémoire pour le parallélisme OU ont été comparées dans [Cra85]. La comparaison a été faite pour une architecture avec mémoire partagée et un "pool" de processus. Le modèle d'exécution utilisé est celui que nous avons présenté dans la section 1.1.1. Le résultat de cette expérience : la méthode des fenêtres permet une exécution plus rapide des programmes logiques et utilise moins de mémoire.

Ciepielewski et Hausman présentent une étude similaire en [CiH86]. L'étude est réalisée sur les méthodes d'arbres de répertoires et sur des variations inspirées de la méthode d'arbre de répertoires et de celle de fenêtres. Selon ces résultats, les techniques qui s'inspirent des fenêtres donnent de meilleures performances.

2.2. Modèles qui exploitent le parallélisme ET

Le parallélisme ET consiste en l'exécution en parallèle de plusieurs littéraux qui appartiennent (en général) à la même clause. Dans cette section nous centrons notre attention sur le problème intrinsèque du parallélisme ET : comment garantir que les différentes valeurs acquises par une variable soient unifiables.

Une réalisation du parallélisme ET doit considérer les remarques suivantes :

1.- Les variables partagées par plusieurs littéraux doivent être liées à des valeurs unifiables entre elles. Par exemple, si l'on résout en parallèle les littéraux: P(X,Y), Q(Y,Z) et R(Z,X), on doit trouver des valeurs pour X, Y et Z qui satisfont en même temps P, Q et R. Les solutions P(1,2), Q(3,4) et R(5,6) peuvent être indépendamment correctes, mais elles ne satisfont pas P, Q et R en même temps.

2.- Pour limiter l'espace de recherche, il peut être convenable de satisfaire certains littéraux avant d'autres. Regardons par exemple le programme suivant:

P(a,a)

P(b,b)

P(c,c)

Q(a,a)

Q(b,b)

Q(a,c)

R(a)

<- P(X,Y), Q(X,Z), R(X).

La seule substitution qui permette de satisfaire R est X <- a, les solutions de P et Q qui ne lient pas X au terme "a" ne conduisent pas à une solution.

Il peut être difficile, voire impossible, de connaître le nombre de solutions qu'un littéral aura en exécution. Cependant, il est possible de connaître le nombre de solutions pour les littéraux dans des applications en bases de données.

3.- Parfois, des prédicats doivent instancier certaines de leurs variables pour être satisfaits. Par exemple:

<- P(X), Y is 2 * X

La multiplication ne peut pas s'effectuer si X n'est pas liée à un valeur numérique. Le littéral P(X) doit produire une valeur pour la variable X avant que le calcul de la multiplication.

Ce problème survient pour des prédicats spécifiés équationnellement.

La première remarque montre que l'on peut, soit établir un ordre d'exécution entre les littéraux, soit vérifier la consistance des solutions obtenues.

La deuxième peut fort fien ne pas être considerée dans une première approche d'un modèle qui exploite le parallélisme ET.

La troisième exige d'établir un ordre d'exécution entre les littéraux d'une clause.

La politique la plus répandue est de décider de l'ordre d'exécution des littéraux en établissant quels littéraux doivent être résolus séquentiellement et quels autres peuvent l'être en parallèle.

Il y a deux manières de décider l'ordre d'exécution des littéraux d'une clause :

. Avec l'aide du programmeur

. De manière automatique par utilisation d'heuristiques.

L'ordre d'exécution des littéraux d'une clause peut être donné par le programmeur à travers des primitives de contrôle utilisées en programmation logique qui font la distinction entre des littéraux producteurs et des littéraux consommateurs de variables. Tel est le cas de langages comme Parlog [ClG81], IC-Prolog [ClM79], Concurrent Prolog [Sha83], GHC [Ued85] et certaines versions de Prolog qui permettent la primitive "mode". La sémantique de ces langages de programmation s'éloignant de la logique, ces langages ne seront pas étudiés ici.

Pour inférer de manière automatique l'ordre d'exécution des littéraux dans une clause, il faut déterminer les variables susceptibles de produire des conflits. Examinons les situations où ces conflits peuvent apparaître :

CAS 1

Soit la clause :

P(X,Y) <- Q(X,Z), R(Z,Y).

Les littéraux Q(X,Z) et R(Z,Y) ne peuvent être satisfaits en parallèle parce qu'ils ont la variable Z en commun, ce sera soit Q soit R qui sera nommé producteur de Z, et celui qui sera choisi comme producteur de Z sera le premier à être satisfait.

CAS 2

Soit la clause :

P(X) :- Q(X), R(X).

Q(X) et R(X) ont une variable en commun, en principe on suppose donc que Q(X) et R(X) ne doivent en principe pas être satisfaits en parallèle. Cependant, si P(X) lie X à un terme sans variables pendant l'exécution alors Q(X) et R(X) peuvent être exécutés en parallèle (mais leurs valeurs doivent être unifiables).

Dans ce type de situation la parallélisation de plusieurs littéraux peut être décidée au cours de l'exécution.

CAS 3

Soit la clause:

P(X,Y) :- Q(X),R(Y).

Q(X) et R(Y) ne partagent pas de variables mais P(X,Y) peut lier X et Y aux termes qui ont des variables en commun (ex X <- Z et Y <- Z) ; dans ce cas Q(X) et R(Y) ne peuvent pas être satisfaits en parallèle.

Différentes heuristiques ont été proposées pour établir l'ordre d'exécution des littéraux dans une clause. Par exemple :

- La tête de la clause est productrice de variables qui sont liées au moment d'invocation de la clause. Cette information est acquise au moment d'exécution. Des modèles comme celui de J.S. Conery et D.F. Kibler [CoK85] utilisent (entre autres) cette heuristique.

- La règle de connexion ("connection rule") [CoK85] : l'idée de cette règle est qu'un littéral peut se résoudre plus facilement lorsqu'il faut chercher un sous-ensemble de toutes ses solutions, c'est-à-dire, lorsque quelques-unes de ses variables ont été liées.

  Supposons que l'on veuille trouver un littéral générateur de la variable X, on prend alors un littéral L qui est consommateur d'au moins une variable et qui contient X. La règle de connexion choisit L comme générateur de X.

  Exemple 2.8 :

  Soit la clause : G0(X,Y,Z) <- G1(X,Y), G2(Z), G3(Y,Z).

  Supposons que comme conséquence de l'invocation de la clause, Z soit liée au terme "a", c'est-à-dire, le littéral G0(X,Y,Z) est générateur de Z. On doit trouver des littéraux générateurs pour X et Y. Etant donné que G3(Y,Z) est déjà consommateur de Z, il est choisi par la "connection rule" comme le générateur de Y. De manière similaire, G1(X,Y) est designé comme générateur de X.

  Fin de l'exemple

Cette règle est appliquée au moment de l'exécution et elle est coûteuse. Elle est utilisée dans le modèle de J.S. Conery et D.F. Kibler [CoK85].

• Règle "plus à gauche" ("leftmost rule").

Le littéral générateur d'une variable X est le littéral qui contient cette variable et qui se trouve dans la position la plus à gauche dans la clause.

La règle se base sur un critère simplement syntaxique, sans prétention à donner la meilleure solution. L'application de cette règle n'est pas coûteuse. Le modèle de Yow-Jian Lin et Vipin Kumar [LiK88] l'utilise comme unique règle. Cette règle est utilisée aussi par les modèles de J.S. Conery et D.F. Kibler [CoK85] et celui de Li et Martin [LiM86].

• Choisir comme générateur d'une variable le littéral qui réduit l'espace de recherche. On a déjà discuté les avantages et inconvénients de cette politique au début de cette section.

On trouve plusieurs façons d'aborder ce problème d'exploitation du parallélisme ET. Quelques-unes se basent sur des nouvelles extensions des clauses de Horn [ClG81], [ClM79], [Sha83], [Ued85], les autres sur la méthode dite générateurs/consommateurs de variables [DeG84], [CoK85], [LiK88] ,[LiM86]. Nous présentons dans la suite deux modèles représentatifs de la méthode générateurs/consommateurs de variables : la méthode de D. DeGroot [DeG84] qui prévoit les possibilités du parallélisme ET grâce à une analyse statique des clauses et la méthode de J. Conery et D. Kibler [CoK85] qui réalise l'analyse des conflits de liaisons de variables pendant l'exécution du programme.

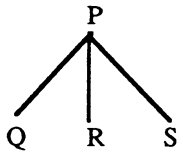
2.2.1. Modèle de D. DeGroot.

Le modèle de D. DeGroot [DeG84] consiste à faire une analyse statique des clauses du programme. Comme résultat de cette analyse, une expression décrivant le graphe d'exécution de chaque clause du programme est construit. Chaque expression (graphe) exprime le parallélisme possible de la clause associée. Pendant l'exécution du programme, de simples vérifications sur les variables de l'expression d'une clause permettent de décider un ordre d'exécution des littéraux de la clause.

Pour une clause telle que P(X) <- Q(X),R(X),S(X), il y a trois manières différentes de réaliser l'exécution :

Si X est liée à une valeur constante lorsque la clause est invoquée

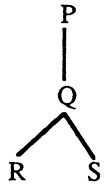
alors Q, R et S peuvent s'exécuter en parallèle



sinon Q sera satisfait avant R et S

Si comme résultat de la satisfaction de Q, la variable X est liée à une valeur constante

alors R et S peuvent s'exécuter en parallèle



sinon Q, R et S s'exécutent de manière séquentielle

Les directives d'exécution que nous avons décrites ci-dessus peuvent être exprimées par les expressions du modèle. Au fur et à mesure que nous avancerons dans la description du modèle, nous donnerons l'ensemble des expressions du modèle.

Pour commencer, nous dirons qu'il y a deux types d'expressions :

G

et (GPAR($X_1 \ldots X_k$) $E_1 \ldots E_n$)

où G est un littéral et où l'expression GPAR indique que si tous les termes X_i sont des termes constants alors les expressions $E_1 \ldots E_n$ peuvent s'exécuter en parallèle, sinon les E_i doivent s'exécuter de façon séquentielle de gauche à droite.

L'expression associée à la clause P(X) <- Q(X),R(X),S(X) est donc :

P(X) = (GPAR(X)

Q(X)

(GPAR(X) R(X) S(X)))

L'évaluation de cette expression permet la réalisation des trois exécutions possibles de la clause qui ont été montrées précédemment.

Pour savoir si un terme est ou non un terme constant, le modèle dispose d'un algorithme pour déterminer le type des termes du programme. A chaque terme du programme est associé un des types suivants :

TC :	Terme constant, pour les termes qui n'ont pas de variable.
TV :	Variable, pour les termes qui sont des variables non liées.
TNVNC :	Termes différents des variables qui ne sont pas des termes constants (ils contient donc des variables).

Comme résultat de la compilation du programme, un type est associé à chaque terme. Dans un terme structuré (terme de la forme $f(t_1,\ldots,t_n)$), un type est associé au terme $f(t_1,\ldots,t_n)$ et à chacun de ses composants $(t_1,\ldots,t_n)$ et cela à tous les niveaux du terme.

Pendant l'exécution, les types des termes sont déterminés par un algorithme qui ne parcout pas toute la structure des termes à chaque invocation ou terminaison d'une clause. L'algorithme travaille seulement avec le type des sous-termes $t_1,\ldots,t_n$ d'un terme $f(t_1,\ldots,t_n)$.

L'algorithme nous donne une manière rapide, mais parfois incorrecte, de déterminer le type des termes. Voyons l'exemple suivant :

G0(X) <- G1(X,V), V=nil

G1(F(U,G(V)),V)

<- G0(F(a,G(Y)))

Dans la première clause, les termes X et V reçoivent le type TV. Dans la deuxième clause, U et V reçoivent le type TV et G(V) et F(U,G(V)) le type TNVNC. Dans la clause de buts, les termes F(a,G(Y)) et G(Y) reçoivent le type TNVNC et les termes a et Y reçoivent les types TC et TV respectivement.

Lorsque la première clause est invoquée, le type TNVNC est associé au terme X. Ensuite à l'invocation du but G1(X,V), la variable Y du terme F(a,G(Y)) est liée à la variable V qui,

ensuite est liée à la constante nil. La variable X est donc liée au terme F(a,G(nil)). Etant donné que le sous-terme G(nil) ne change pas son type, le terme X a encore le type TNVNC.

Bien que l'algorithme ne calcule pas correctement le type des termes dans tous les cas, son utilisation permet la détection d'une grande partie du parallélisme ET dans le modèle.

Pour des clauses comme P(X,Y) <- Q(X), R(Y), l'exécution est différente selon que les variables X et Y sont liées ou non à des termes indépendants. Deux termes sont considérés comme indépendants s'ils ne partagent aucune variable. Par exemple, les termes X et Y sont indépendants mais si X est liée à F(U) et Y est liée à G(U,a), alors X et Y ne sont pas indépendants.

Si lorsque la clause P(X,Y) <- Q(X),R(Y) est invoquée les termes X et Y sont indépendants: Q(X) et R(Y) peuvent s'exécuter en parallèle, autrement Q(X) et R(Y) doivent s'exécuter séquentiellement.

Une autre expression est nécessaire :

$$(\text{IPAR}(X_1,\ldots,X_k)\ E_1,\ldots,E_n)$$

L'expression IPAR indique que si tous les termes X_i sont indépendants (voir algorithme ci-dessous) alors les expressions $E_1,\ldots,E_n$ peuvent s'exécuter en parallèle. Par exemple, pour la clause :

P(X,Y) <- Q(X), R(Y)

le graphe d'exécution est :

(GPAR(X,Y)

(IPAR(X,Y) Q(X), R(Y)))

L'algorithme pour vérifier l'indépendance de deux termes [DeG84] est le suivant :

IF TYPE(ARG1) = TC OR TYPE(ARG2) = TC

THEN INDEPENDENT

ELSE IF TYPE(ARG1) = TYPE(ARG2) = TV AND ADDRESS(ARG1) ¬= ADDRESSE(ARG2)

THEN INDEPENDENT

ELSE (ASSUME) DEPENDENT;

La vérification de l'indépendance de "n" termes consomme $O(n^2)$ vérifications. Heureusement le nombre de termes qui apparaissent dans la tête des clauses n'est pas grand en général.

Dans le graphe d'exécution peuvent aussi apparaître les expressions suivantes :

(SEQ E_1,...,E_n)

(PAR E_1,...,E_n)

(IF E_1 E_2 E_3)

Une expression SEQ indique que les expressions E_1,...,E_n doivent s'exécuter séquentiellement. Une expression PAR indique que les expressions E_1,...,E_n doivent s'exécuter parallèlement. L'expression IF réalise l'alternative en fonction du résultat de l'expression booléanne E1.

Voyons le programme suivant en son expression associée [DeG84] :

```
qksort(L,SL) <-
   partition (L,L1,L2) Ÿ
   qksort (L1,SL1) Ÿ
   qksort (L2,SL2) Ÿ
   append (SL1,SL2,SL).


(SEQ
   partition (L,L1,L2)
   (  IPAR (L1,L2)
      qksort (L1,SL1)
      qksort (L2,SL2) )
   append (SL1,SL2,SL) ).
```

Cette méthode n'arrive pas à exprimer totalement le parallélisme possible des programmes logiques en raison de l'affectation des types aux termes. Cependant, le modèle permet l'obtention des expressions qui expriment le parallélisme ET d'une bonne gamme de

programmes logiques sans dépenser trop de temps dans la construction et l'évaluation des expressions qui décrivent le graphe du programme.

M. Hermenegildo présente une machine parallèle qui est une extension de la WAM en utilisant la méthode de D. DeGroot [Her86].

2.2.2. La méthode de J. Conery et D. Kibler pour le parallélisme ET.

Il s'agit d'une extension du modèle de J. Conery et D. Kibler présenté dans la section 2.1.2. Le nouveau modèle exploite les parallélismes OU et ET [CoK85]. Dans le modèle étendu, les processus AND satisfont en parallèle les littéraux de la clause qu'ils représentent. Le "backtracking" sur ces littéraux ne suit pas nécessairement l'ordre syntaxique des littéraux de la clause.

Dans la méthode de J. Conery et D. Kibler pour le parallélisme ET, lorsque plusieurs littéraux d'une clause partagent une variable V non liée, un des littéraux est désigné comme **générateur** de V et les autres littéraux comme **consommateurs** de V. Le littéral générateur doit être résolu avant les consommateurs. Chaque variable d'une clause doit avoir son littéral générateur. Par exemple, dans la clause <- Q(X),R(Y),S(X,Y), le littéral Q(X) peut être désigné comme le générateur de la variable X et S(X,Y) comme un consommateur de X; le littéral R(Y) peut être designé comme le générateur de la variable Y et S(X,Y) comme son consommateur. Les littéraux Q(X) et R(Y) peuvent donc être satisfaits en parallèle et lorsque les deux sont satisfaits, S(X,Y) peut être exécuté.

Pour la gestion du parallélisme ET on distingue deux parties principales : la première partie connue comme "foreward execution algorithm", choisit les générateurs des variables partagées par plusieurs littéraux d'une clause ; la deuxième partie -"backward execution algorithm"-, choisit les littéraux à satisfaire de nouveau lorsqu'un littéral L échoue.

Chaque processus AND travaille avec un graphe "data flow" où sont présentées les relations de générateurs/consommateurs des variables de la clause concernée. Ce graphe est construit pendant l'exécution du programme. Dans un tel graphe il y a un nœud pour chaque littéral de la clause, et un ensemble d'arcs dirigés pour chaque variable. Les arcs vont du générateur de chaque variable vers les consommateurs de cette variable. Un prédécesseur immédiat d'un littéral L est un générateur d'une des variables de L. Un prédécesseur est soit un prédécesseur immédiat, soit le prédécesseur d'un prédécesseur immédiat. Les relations : successeur et successeur immédiat sont définies de manière similaire. Par exemple, le graphe "data flow" de la clause <- Q(X),R(Y),S(X,Y) est le suivant :

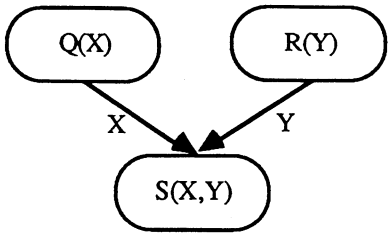


Figure 2.22

La tête d'une clause correspond au littéral L à résoudre pour le processus OR père, ce littéral est inclu dans le graphe "data flow". Ce littéral est le générateur de chaque variable qui apparait dans la tête de la clause et qui est lié lorsque la clause est invoquée. Dans son rôle de générateur ce littéral apparaît dans le graphe "data flow" étiqueté : HG. Le littéral est consommateur de toutes les variables non liées au moment de l'invocation de la clause, dans son rôle de consommateur ce littéral apparaît dans le graphe data flow étiqueté : HC.

L'ordonnancement des littéraux des clauses se fait en appliquant les règles suivantes :

• La tête de la clause est génératrice de toutes les variables liées au moment de l'invocation de la clause ;

• La "connection rule", définie précédemment ;

• La "leftmost rule", définie précédemment.

Exemple 2.9:

L'exemple suivant est pris de [CoK87]. Soit le programme suivant :

```
C1 : PAPER(P,D,I)    <- DATE(P,D), AUTHOR(P,A), LOC(A,I,D)
                     :
                     :
Cn :                    <- PAPER(X,1978,uci).
```

Comme conséquence de l'invocation de la clause C1, le littéral PAPER(P,D,I) est le générateur des variables D et I. Par la "connection rule", DATE(P,D) et LOC(A,I,D) sont consommateurs de D et LOC(A,I,D) est consommateur de I. La "connection rule" établit que LOC(A,I,D) est le générateur de la variable A et que AUTHOR(P,A) est son consommateur. Enfin, par la règle "plus à gauche", DATE(P,D) devient le générateur de la variable P et PAPER(P,D,I), AUTHOR(P,A) ses consommateurs. Le graphe "data flow" est donc:

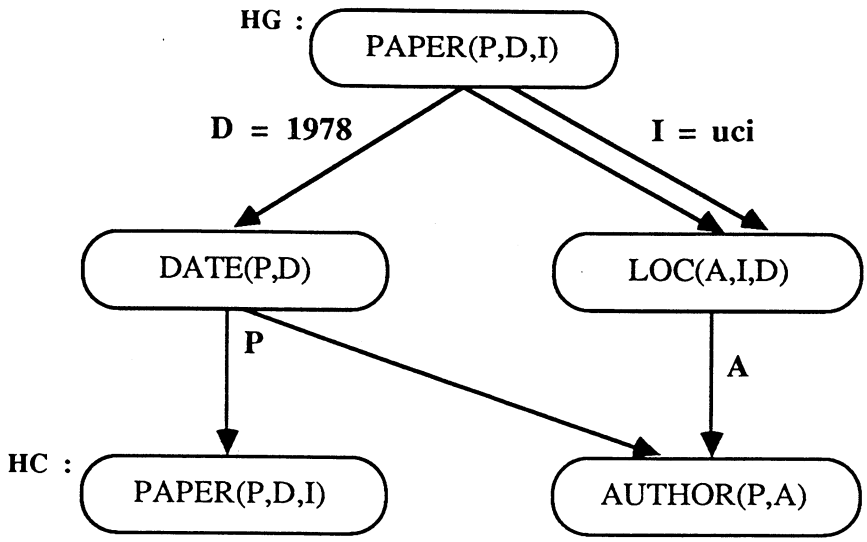


Figure 2.23

Le graphe "data flow" d'une clause est construit pendant l'exécution du programme, et les relations générateurs/consommateurs définies dans le graphe peuvent être modifiées pendant l'exécution. Pour illustrer une telle situation voyons l'exemple suivant :

Exemple 2.10 :

Soit la clause suivante : <- P(X),Q(X),R(X). Le graphe "data flow" initial est :

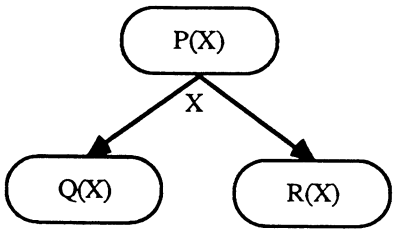


Figure 2.24

si P(X) lie la variable X à la constant "a", le graphe "data flow" n'est pas modifié. Si P(X) lie la variable X au terme non constante "F(Y)", le graphe "data flow" devient :

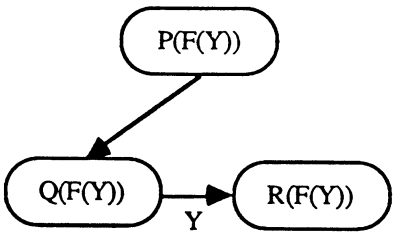


Figure 2.25

Fin de l'exemple

Le "forward execution algorithm" décrit le comportement des processus AND. L'algorithme établit que lorsqu'un processus AND est activé, le premier pas à accomplir est la construction d'un graphe "data flow" pour les littéraux de la clause qui vient d'être invoquée. Le graphe est construit en suivant les règles que nous avons données précédemment. Puis le graphe est parcouru en cherchant les littéraux dont tous les prédécesseurs immédiats ont déjà été satisfaits (les prédécesseurs ont généré toutes les variables qu'un tel littéral consomme). Pour chacun de ces littéraux un processus OR est activé. Lorsqu'un littéral est résolu, le graphe "data-flow" est modifié. Le graphe est parcouru en cherchant de nouveaux littéraux dont les prédécesseurs immédiats on déjà été satisfaits. Un processus OR est activé pour satisfaire chacun de ces littéraux. Si un processus OR envoie un message "fail" à son processus AND père, le "backward exécution algorithm" est invoqué. Lorsque tous les littéraux de la clause ont été résolus, le processus AND envoie un message "success" à son processus père.

Le "backward execution algorithm" coordonne les actions des générateurs après la réception d'un message "fail" ou "redo" par un processus AND. L'algorithme coordonne aussi les actions des littéraux consommateurs des variables produites par les générateurs des variables du processus AND.

Le parcours du graphe "data flow" en profondeur établit un ordre des nœuds du graphe. Le "backward execution algorithm" a une liste L qui reflète cet ordre. Si un littéral L consomme une variable V, L apparaît dans L après tous les générateurs de V.

Supposons que le "backward execution algorithm" soit invoqué lorsque le littéral L a échoué. L'algorithme suit les pas suivants :

• Les prédécesseurs immédiats de L dans le graphe (générateurs des variables de L) sont étiquetés avec L.

• Un message "redo" est envoyé au processus OR représentant le littéral G qui satisfait la propriété suivante : G est étiqueté soit par L, soit par un successeur de L et occupe la position la plus à droite dans L. L'exemple suivant montre comment travaille le "backward execution algorithm" et pourquoi il est nécessaire tenir compte des successeurs de L.

Exemple 2.11 :

Soit la clause suivante :

$$\text{COLOR(A,B,C,D,E)} \leftarrow \text{NEXT(A,B)} \wedge \text{NEXT(C,D)} \wedge \text{NEXT(A,C)} \wedge \text{NEXT(A,D)} \wedge \text{NEXT(B,C)} \wedge \text{NEXT(B,E)} \wedge \text{NEXT(C,E)} \wedge \text{NEXT(D,E)}$$

son graphe "data flow" est :

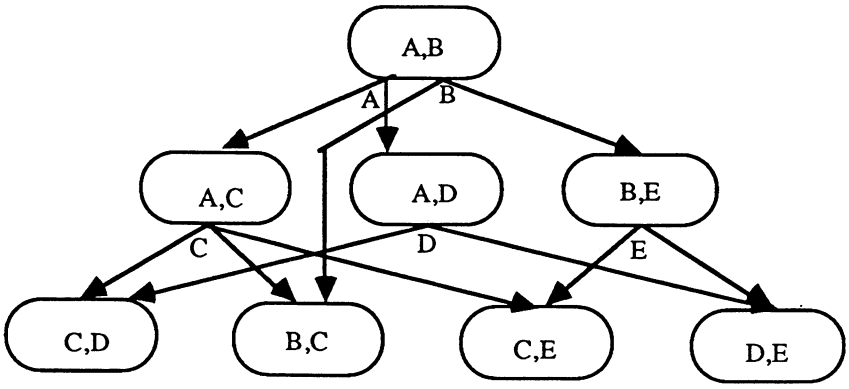


Figure 2.26

la liste L est : [(A,B), (A,C), (A,D), (B,E), (C,D), (B,C), (C,E), (D,E)].

Supposons que le processus qui représente le littéral NEXT(D,E) échoue, l'algorithme marque les nœuds (A,D) et (B,E) avec (D,E). Supposons que le processus qui représente le générateur (B,E) échoue aussi; dans ce cas, avant d'essayer de satisfaire les prédécesseurs immédiats de (B,E) qui sont marqués par (B,E) (nœud (A,B)), il est nécessaire d'essayer l'autre prédécesseur de (D,E) (nœud (A,D)).

Fin de l'exemple

• Les littéraux qui sont générateurs des variables de n'importe quel successeur de G, doivent renvoyer les solutions qui ont été déjà calculées.

• Enfin, tout processus OR qui consomme des variables modifiées par G, doit finir son exécution courante, annuler l'exécution de ses descendants et commencer à nouveau son travail en utilisant les nouvelles solutions.

La méthode de J.S. Conery et D. Kibler permet d'exploiter un niveau de parallélisme plus élevé que celui de la méthode de D. DeGroot mais la construction et la modification du graphe "data flow" pendant l'exécution introduisent un "overhead" considérable.

Des améliorations ont été faites sur ce modèle [LiK88] et [CoK85]. Ces travaux ont amélioré les performances et ils ont éliminé une erreur (voir [LiK88]) dans le "backward execution algorithm".

CHAPITRE 3

Une proposition de machine à Inférence Parallèlle pour le Langage des clauses de Horn

UNE PROPOSITION DE MACHINE À INFÉRENCE PARALLELE POUR LE LANGAGE DES CLAUSES DE HORN QUI EXPLOITE LE PARALLÉLISME OU

Dans ce chapitre nous décrivons un modèle d'inférence parallèle pour les clauses de Horn qui exploite le parallélisme OU. Nous utilisons FP2 [Jor84] comme outil de spécification.

Le modèle que nous présentons est de granularité fine. Les programmes sont traduits en un réseau de processus et ce réseau reflète la structure syntaxique du programme.

Dans le modèle que nous présentons, la quantité de parallélisme OU est la plus grande possible, c'est-à-dire que toutes les solutions d'un but sont générées et utilisées immédiatement. Le modèle a une modularité qui lui permet la possibilité de restreindre le parallélisme jusqu'au niveau souhaité.

Les termes du programme sont distribués entre un sous-réseau de processus qui réalisent la gestion de la mémoire de la machine. Les processus de ce sous-réseau stockent et manipulent les termes du programme. L'unification est effectuée par les processus qui s'occupent de la gestion de la mémoire et utilisent le mécanisme de communication de FP2.

Nous présentons d'abord la spécification en FP2 de la machine pour la logique propositionnelle. Nous commençons par la logique propositionnelle parce que elle nous permet d'aborder dans un cas simple, la structure d'ensemble de la machine qui sera reprise par les clauses de Horn. Nous décrivons ensuite son comportement pour les clauses de Horn. Une nouvelle machine est construite à partir de cette machine pour la logique propositionnelle à laquelle nous connectons un sous-réseau de processus pour stocker et manipuler les termes du programme.

3.1. Le modèle pour la logique propositionnelle.

Nous commençons par définir dans cette section des arbres AND/OR qui permettent de représenter l'exécution de programmes de la logique propositionnelle. Ces arbres AND/OR reflètent la structure syntaxique du programme à exécuter. L'exécution d'un programme commence par la racine de l'arbre et l'exécution est réalisée par des transformations sur l'arbre. Nous décrivons en FP2 des réseaux des processus FP2 capables de simuler la construction et l'utilisation de ces arbres.

Les figures 3.1 et 3.2 illustrent des conventions de dessin qui seront employées dans ce chapitre.

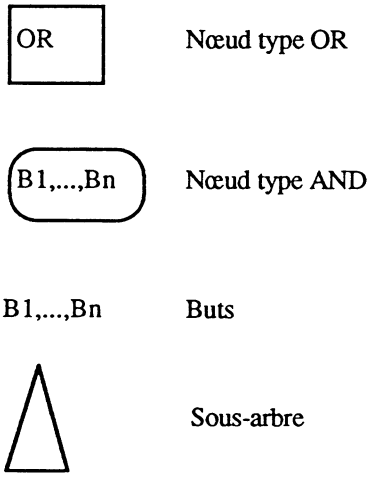


Figure 3.1

La figure 3.1 montre les différents composants des arbres AND/OR introduits dans la section 3.1.1.

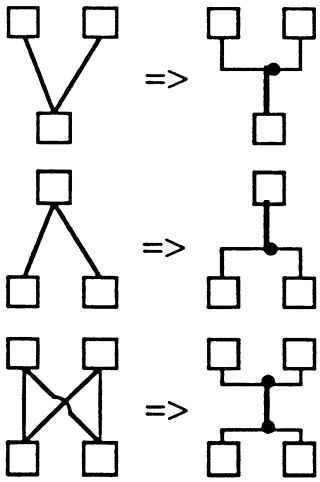


Figure 3.2

Pour simplifier nos dessins, nous utiliserons les conventions montrées dans la figure 3.2, les rectangles représentent soit des nœuds, soit des processus et les lignes représentent les liaisons entre les nœuds ou les processus. La colonne de gauche montre les liaisons réeles entre les processus et la colonne de droite montre comment ces liaisons seront représentées dans nos dessins.

3.1.1. Traduction d'un programme logique en un arbre AND/OR.

Les arbres AND/OR sont formés à partir de sous-arbres qui reflètent la structure syntaxique du programme. Ces sous-arbres sont définis ci-dessous :

L'ensemble des clauses qui forment une procédure (une définition) telle que :

A <- B11,...,B1n1

A <- B21,...,B2n2

...

A <- Bm1,...,Bmnm

est représenté par le sous-arbre :

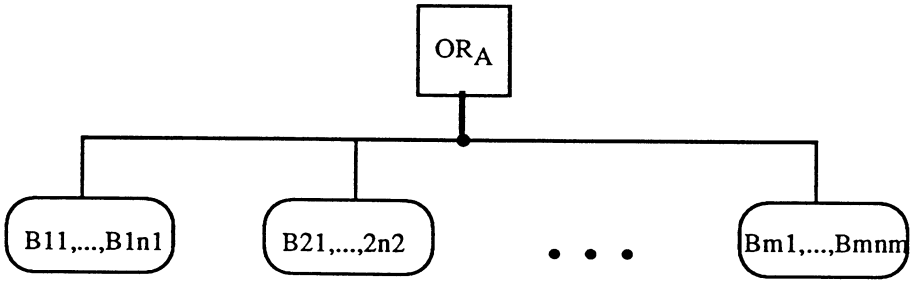


Figure 3.3

la racine de ce sous-arbre est un nœud de type OR et ses fils sont des nœuds de type AND.

Une clause d'assertion, de la forme "A" est une clause sans corps. Le corps de cette clause est représenté par le nœud de type AND noté :

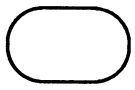


Figure 3.4

A la clause de buts, de la forme : <- B1 ∧ B2 ∧ ... ∧ Bn est associé le sous-arbre suivant :

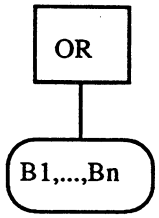


Figure 3.5

La racine de l'arbre AND/OR que nous proposons est un nœud de type OR qui a comme fils un nœud de type AND qui représente la clause de buts. Nous dirons que la racine se trouve au niveau 0 de l'arbre AND/OR. Les niveaux impairs de l'arbre sont tous des nœuds de type AND et les niveaux pairs sont tous des nœuds de type OR.

3.1.2. Transformations sur l'arbre AND/OR.

L'exécution du programme peut être vue comme une suite de transformations sur l'arbre AND/OR. Initialement l'arbre représente la clause de buts. Les transformations qui peuvent être réalisées sur ces arbres AND/OR sont les suivantes :

• Choisir une feuille (ensemble de littéraux) dans l'arbre et un littéral A dans cette feuille. Attacher à A le sous-arbre dont la racine est OR_A. Etant donné que nous ne sommes pas intéressés par le parallélisme ET, dans un nœud AND nous choisissons un seul littéral à satisfaire à la fois, la règle de calcul étant "de gauche à droite".

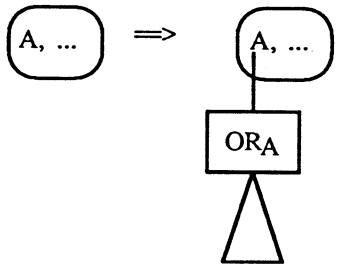


Figure 3.6

• Lorsque l'un des fils d'un nœud de type OR est satisfait, le littéral père du nœud OR est satisfait.

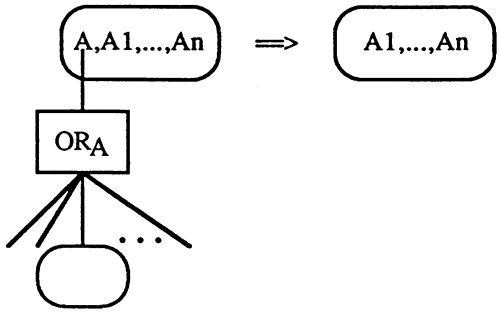


Figure 3.7

Etant donné que nous sommes intéressés par le parallélisme OU, plusieurs nœuds peuvent être transformés à la fois.

Pour illustrer comment les arbres AND/OR que nous avons décrits représentent l'exécution des programmes, considérons l'exemple suivant :

3.1.3. Exemple 3.1

Soit le programme :

C1: P <- P,Q,P.

C2: P <- R,Q.

C3: Q <- R.

C4: R.

C5: <- P.

Initialement, l'arbre contient la racine (un nœud OR) et une feuille representant la clause de buts : <P>. Une seule transformation est applicable, l'arbre qui résulte est :

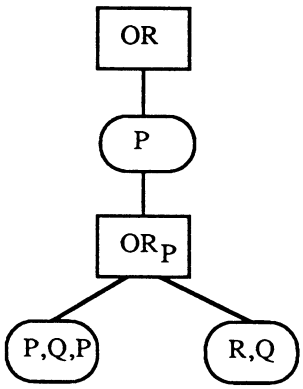


Figure 3.8

Deux transformations peuvent être réalisées sur l'arbre une transformation sur chaque feuille de l'arbre. L'arbre se transforme en :

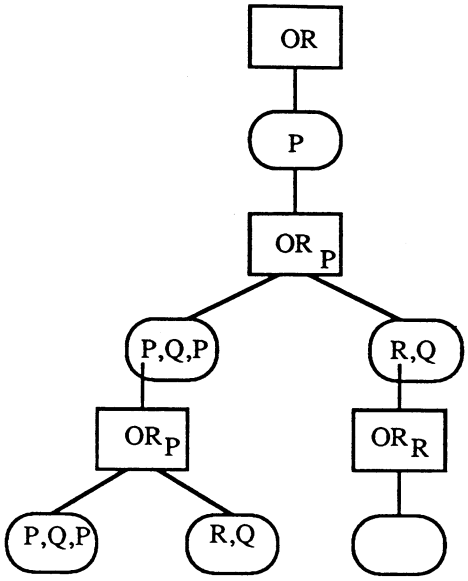


Figure 3.9

A chaque branche terminale de l'arbre correspond une résolvante différente. Ainsi dans cet arbre figurent trois résolvantes : [P,Q,P,Q,P], [R,Q,Q,P] et [Q].

Trois transformations peuvent être réalisées sur l'arbre obtenu. Nous montrons ici seulement la transformation de la branche la plus à droite de l'arbre où le littéral R vient d'être résolu, les trois prochaines transformations dans cette branche permettent de résoudre le littéral Q :

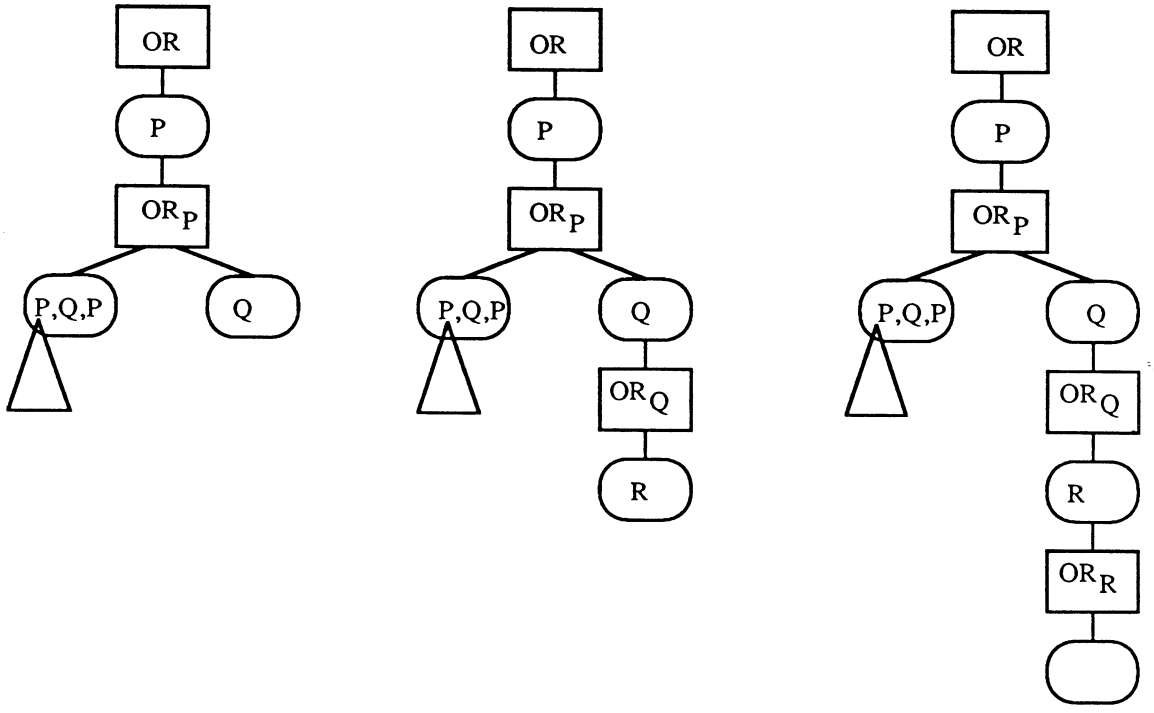


Figure 3.10

Les trois transformations suivantes sur la branche droite de l'arbre permettent de prouver la clause de buts :

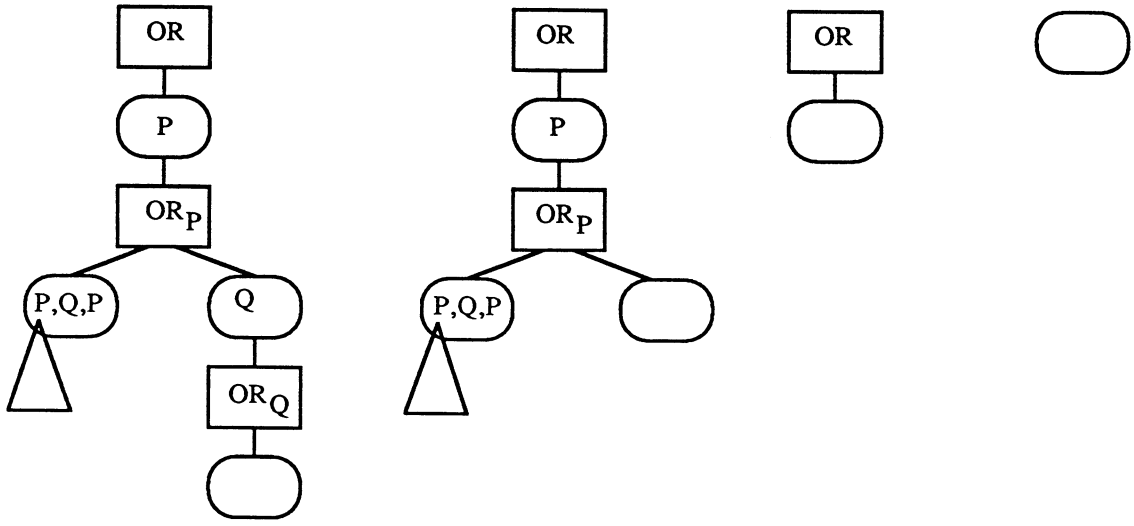


Figure 3.11

Etant donné que la racine de l'arbre est : <>, la clause de buts a donc été prouvée.

Fin de l'exemple

3.1.4. Traduction d'un programme logique en un réseau de processus FP2.

Notre modèle de processus FP2 suit l'idée de la construction des arbres AND/OR, de manière analogue à ce que nous avons fait avec le modèle d'arbre. Voyons maintenant comment traduire les clauses d'un programme en sous-réseaux de processus FP2.

Dans la suite, les processus FP2 sont représentés par des rectangles, ces rectangles sont divisés par une ligne horizontale en deux parties. Dans la partie supérieure du rectangle apparait le nom du processus et dans la partie inférieure apparaissent soit les éléments syntaxiques du programme logique sur lesquels le processus travaille, soit le sous-réseau des processus qui constituent le processus qui nous occuppe. Les lignes qui vont d'un processus à un autre représentent les connexions entre les processus et les types de messages qui circulent à travers ces connexions seront placés à côté de ces lignes.

L'ensemble des clauses qui forment une procédure (une définition) telle que :

A <- B11,...,B1n1

A <- B21,...,B2n2

...

A <- Bm1,...,Bmnm

sont représentées par le sous-réseau ci-dessous.

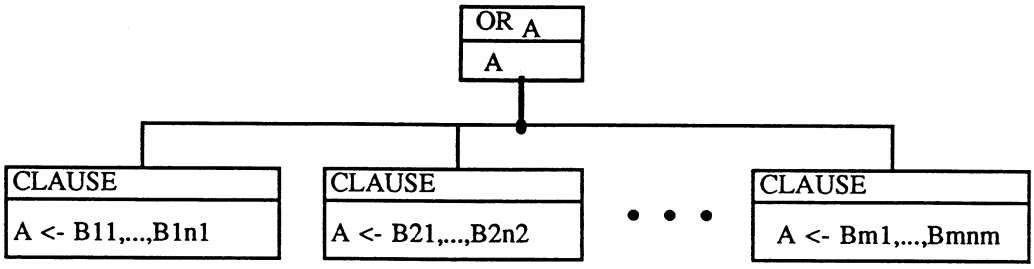


Figure 3.12

Une clause d'assertion de la forme "A" est représentée par un processus ASSERTION.

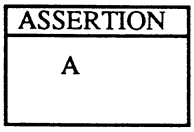


Figure 3.13

Etant donné que parmi les clauses définissant un littéral il peut exister une clause assertion, le sous-réseau qui représente la définition de ce littéral a un processus ASSERTION pour représenter la clause correspondant à la place d'un processus CLAUSE.

A une clause de buts, de la forme : <- B1 ∧ B2 ∧ ... ∧ Bn est associé le sous-réseau suivant :

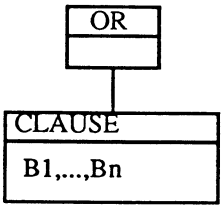


Figure 3.14

Lorsqu'un but "A" apparaît dans le corps d'une clause et que sa définition n'apparaît pas dans le programme logique, nous représentons la définition de "A" par un processus PFALSE :

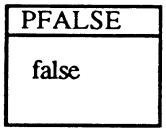


Figure 3.15

Chaque sous-réseau qui représente une définition reçoit le nom de processus DEFINITION. Chaque processus DEFINITION est donc construit à partir d'un sous-réseau de

processus qui contient soit un processus OR auquel sont associés autant de processus CLAUSE qu'il existe de clauses dans la définition correspondante, soit un processus PFALSE.

Etant donné que pendant l'exécution du programme une clause peut être invoquée plusieurs fois, nous avons besoin d'un nombre indéfini de chaque sous-réseau DEFINITION défini pour le programme. Chaque fois qu'une clause doit être invoquée, le sous-réseau correspondant sera activé.

Pour le programme de l'exemple 3.1, nous avons les familles de processus DEFINITION suivantes:

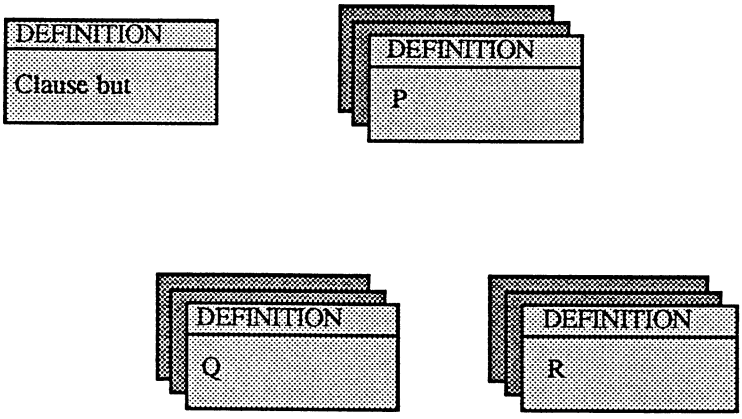


Figure 3.16

Lorsqu'un littéral L doit être satisfait, un processus DEFINITION qui représente la définition de L est activé, le processus OR correspondant envoie un signal ("buzz") qui permet d'activer en parallèle chacun de ses processus CLAUSE et ASSERTION (ou processus PFALSE). Le littéral est satisfait lorsqu'un des processus CLAUSE envoie la valeur "true" au processus OR qui représente le littéral et auquel sont liés ces processus.

La définition en FP2 des processus OR est donc la suivante :

process OR [ncl : **Nat**]

connecteurs

OX : **Nat x Nat | Nat x Nat x Bool**

OY : **Signal | Bool**

states

S : **Nat x Nat**

T : **Nat x Nat x Nat**

F : -

vars

Idport, Idgoal, n-answers, i : **Nat**

rules

;; Activation du processus OR

OX (Idport, Idgoal) ==> S (Idport, Idgoal)

;; Le processus OR active tous ses processus HEADs en parallèle.

S (Idport, Idgoal) : { OY_i (buzz) | i = 1...ncl } ==> T (Idport, Idgoal,ncl - 1)

;; Le processus OR reçoit les réponses de ses HEADs.

{ T (Idport, Idgoal,n-answers) : OY_i (true) OX (Idport, Idgoal,true)

==> F | i = 1... ncl }

{ T (Idport, Idgoal,n-answers + 1) : OY_i (false)

==> T (Idport, Idgoal,n-answers) | i = 1 ... ncl }

T (Idport, Idgoal,0) : OX (Idport, Idgoal,false) ==> F

endprocess

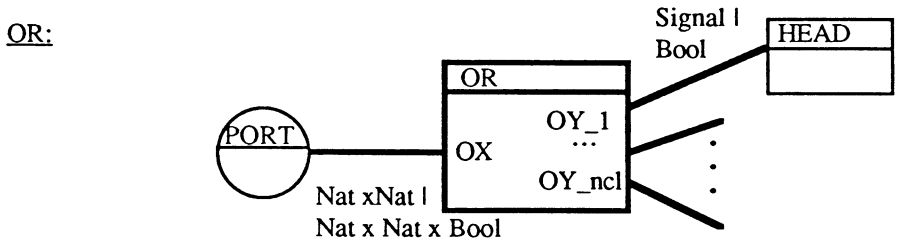


Figure 3.17

Dans la spécification des processus OR, il est nécessaire de préciser comment se réalise son activation. L'activation d'un processus OR se réalise lorsque le processus DEFINITION qui doit satisfaire un de ses buts envoie au OR deux naturels (Idport et Idgoal) qui identifient le but dans ce processus DEFINITION. Ces deux naturels permettent au processus OR d'acheminer la réponse obtenue à partir des résultats envoyés par les processus HEAD vers le

but du processus DEFINITION qui en a effectué la demande. Nous verrons plus tard pourquoi il est nécessaire d'avoir deux naturels pour l'identification d'un but dans une DEFINITION.

Voyons plus en détail les processus CLAUSE. Ils reflètent la structure de la clause qu'ils représentent. Un processus CLAUSE est construit à partir d'un sous-réseau de processus qui contient un processus HEAD et autant de processus GOALs que le nombre de buts de la clause. La figure ci-dessous montre comment sont liés entre eux les processus qui représentent une clause de la forme : A←B11,...,B1n1 .

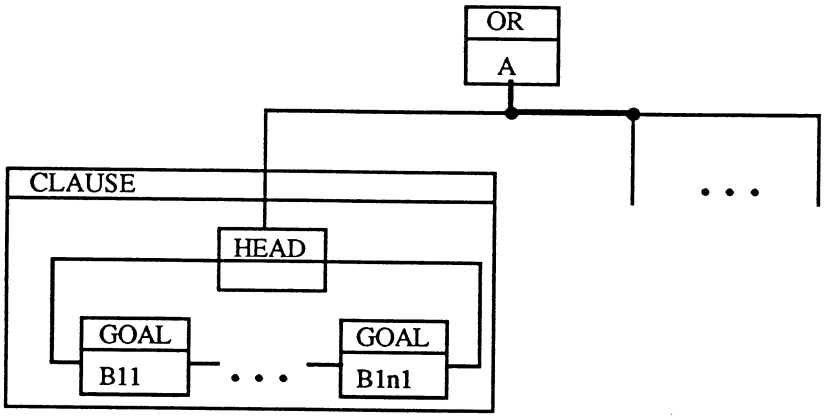


Figure 3.18

HEAD représente la tête de la clause. Il est activé par le processus OR auquel HEAD est lié, l'activation se réalise lorsque le processus OR envoie à HEAD le signal "buzz". L'activation des processus GOALs se fait de gauche à droite (d'abord B11, puis B12, enfin B1n1), le GOAL B11 est activé par HEAD. L'activation se réalise ainsi : le processus HEAD envoie le signal "buzz" au GOAL B11. Lorsque le but B11 est satisfait, c'est-à-dire lorsque le processus GOAL B11 reçoit une solution (message "true" ou "false"), celle-ci permet d'activer le GOAL B12. Les solutions obtenues par GOAL B1n1 sont envoyées au processus HEAD qui les achemine vers le processus OR. La spécification en FP2 du processus HEAD est la suivante :

process HEAD

connecteurs

HX : **Signal** | **Bool**

HY : **Signal**

HZ : **Bool**

states

S,T,F : -

vars

answer : **Bool**

rules

;; Le processus HEAD est activé.

HX (buzz) ==> S

;; Le processus HEAD active à son premier GOAL ou processus PTRUE.

S : HY(buzz) ==> T

;; Le processus HEAD reçoit la réponse (true ou false) soit de son dernier GOAL, soit de son PTRUE et

;; envoie cette réponse à son processus OR

T : HZ (answer) HX (answer) ==> F

endprocess

HEAD:

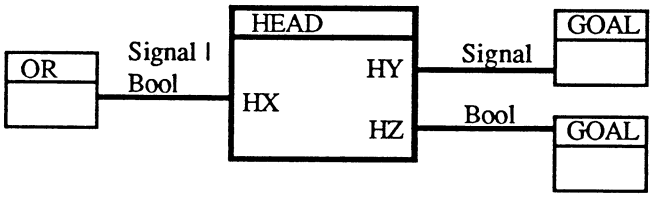


Figure 3.19

Un processus GOAL doit envoyer un message qui permet d'activer un processus DEFINITION de la famille appropriée. Par exemple, le GOAL B11 de la figure précédente doit envoyer un message qui permet d'activer un processus DEFINITION B11. Celui-ci produit alors la solution (on travaille sur le calcul propositionnel) qui est envoyée au GOAL B11.

La spécification en FP2 du processus GOAL est la suivante :

process GOAL [ngoal : **Nat**]

connecteurs

GX : **Signal | Bool**

GY : **Bool**

GZ : **Nat | Nat x Bool**

states

S, T, F : -

vars

answer : **Bool**

rules

;; Le processus GOAL est activé.

GX (buzz) ==> S

GX (true) ==> S

;; Le processus GOAL active la DEFINITION associée à lui à travers son processus PORT

S : GZ (ngoal) ==> T

;; Le processus GOAL reçoit la réponse de la DEFINITION à travers son processus PORT.

T : GZ (ngoal, true) GY(true) ==> F

T : GZ (ngoal, false) GY(false) ==> F

;; Si le processus GOAL [ngoal - 1] a échoué, le processus GOAL ngoal est activé mais

;; le GOAL [ngoal] n'essayera pas de satisfaire la DEFINITION associée.

GX (false) GY (false) ==> F

endprocess

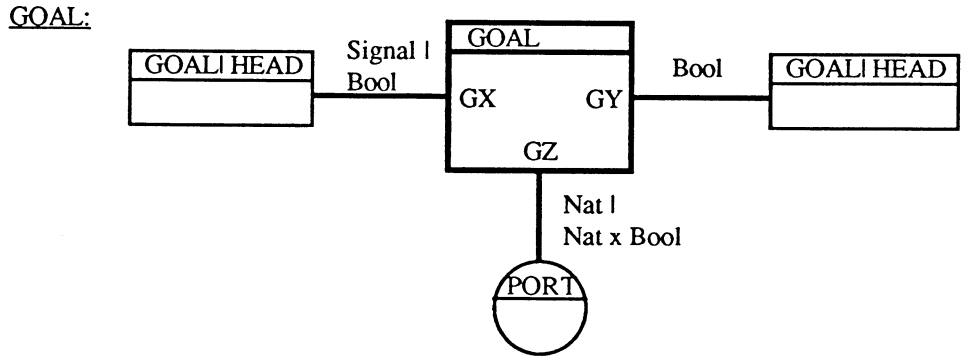


Figure 3.20

Le premier processus GOAL est activé par son processus HEAD lorsqu'il reçoit le signal "buzz". Chaque processus GOAL present dans un processus DEFINITION est identifié par un

entier naturel que nous nommerons : "ngoal". Cet entier "ngoal" possède deux utilisations dans le processus GOAL appartenant à la DEFINITION d :

• Comme message qui permet à GOAL d'activer une DEFINITION.

• Comme filtre pour recevoir les messages envoyés par le processus DEFINITION qui ont été activés par les différentes processus GOAL appartenant à la DEFINITION d.

Les processus ASSERTION se construisent à partir d'un sous-réseau de processus qui contient un processus HEAD et un processus PTRUE comme le montre la figure ci-dessous :

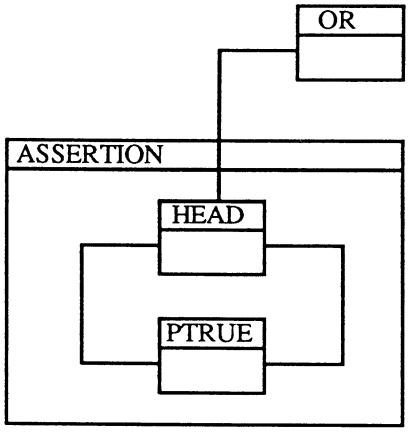


Figure 3.21

Le processus ASSERTION représente une clause assertion du programme logique, la tête de la clause est représentée par un processus HEAD et son corps vide est représenté par un processus PTRUE.

La description du processus PTRUE en FP2 est la suivante :

process PTRUE

connecteurs

TX : **Signal**

TY : **Bool**

states

S, F : -

rules

;; Ce processus représente une assertion, il envoie comme réponse : true.

	TX (buzz)	==>	S
S :	TY (true)	==>	F

endprocess

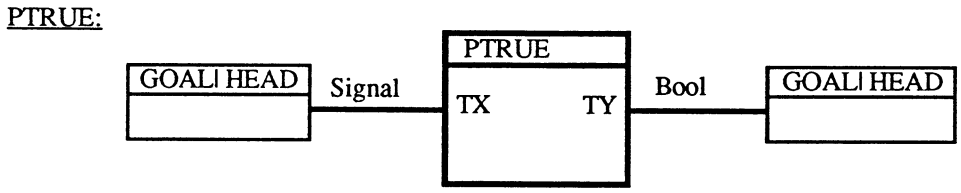


Figure 3.22

L'activité du processus PTRUE est très simple : une fois qu'il est activé, il communique le message : "true".

Le processus ASSERTION est activé lorsque le processus OR auquel est lié envoie le signal : "buzz" au HEAD du processus ASSERTION. Comme conséquence de l'intéraction des processus HEAD et PTRUE, le processus ASSERTION envoie au processus OR auquel est lié et à travers son processus HEAD, le message "true".

Pour les buts qui apparaissent dans le corps d'une clause et dont les définitions n'apparaissent pas dans le programme logique, nous avons le processus PFALSE. La spécification en FP2 de PFALSE est :

process PFALSE

connecteurs

FX : **Nat | Nat x Bool**

states

S : **Nat**

F : -

vars

Id : **Nat**

rules

;; Ce processus représente l'échec d'un but, il envoie comme réponse : false.

	FX (Id)	==>	S (Id)
S (Id) :	FX (Id,false)	==>	F

endprocess

PFALSE:

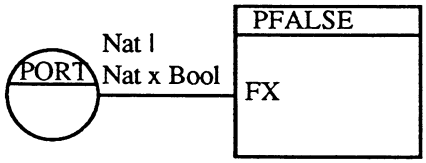


Figure 3.23

Pour réduire le nombre de connexions dans le réseau nous allons introduire dans chaque processus DEFINITION associé au littéral L autant de processus PORT que de buts différents apparaissant dans le corps des clauses qui forment la définition de L. Les processus PORT agissent comme intermédiaires entre les processus GOAL et les processus DEFINITION.

Les sous-réseaux de la famille DEFINITION P de notre exemple ont donc la forme:

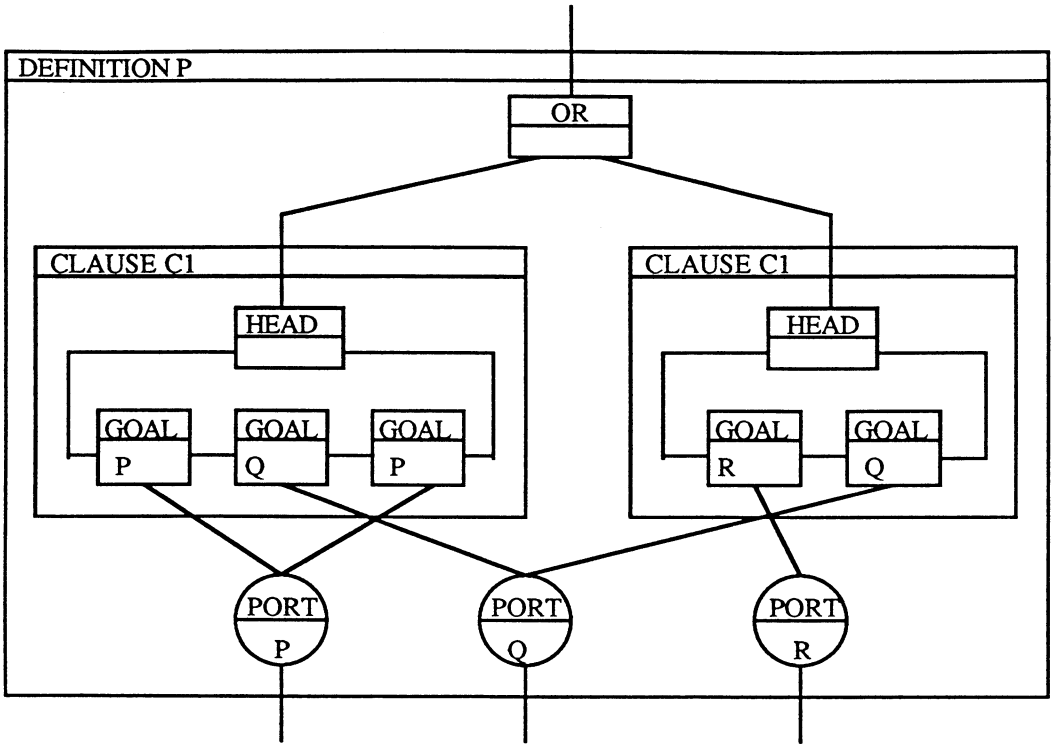


Figure 3.24

Chaque processus PORT dans le réseau doit avoir une identification unique. Cette identification est un naturel produit par le processus X. La spécification en FP2 du processus X est la suivante :

process X

connecteurs

B : **Nat**

states

S : **Nat**

vars

I : **Nat**

rules

==> S (1)

S (I): B (I) ==> S (I + 1)

endprocess

La spécification en FP2 des processus PORT est la suivante :

process PORT

connecteurs

PX : **Nat** | **Nat x Bool**

PY : **Nat x Nat** | **Nat x Nat x Bool**

A : **Nat**

states

S : **Nat x Nat**

T : **Nat**

vars

answer : **Bool**

Idgoal, Idport : **Nat**

rules

;; Le processus PORT s'active en recevant l'identification du processus GOAL

;; qui demande l'activation d'une DEFINITION et en demandant une identification

;; pour lui au processus X.

PX (Idgoal) A (Idport) ==> S (Idport,Idgoal)

;; Le processus PORT active une DEFINITION

S (Idport,Idgoal) : PY (Idport,Idgoal) ==> T (Idport)

T (Idport) : PX (Idgoal) ==> S (Idport,Idgoal)

;; Le processus PORT reçoit la réponse d'une des DEFINITION qui lui a activé

;; et envoie cette réponse au processus GOAL qui avait demandé

;; l'activation de la DEFINITION.

T (Idport) : PY (Idport,Idgoal,answer) PX (Idgoal,answer) ==> T (Idport)

endprocess

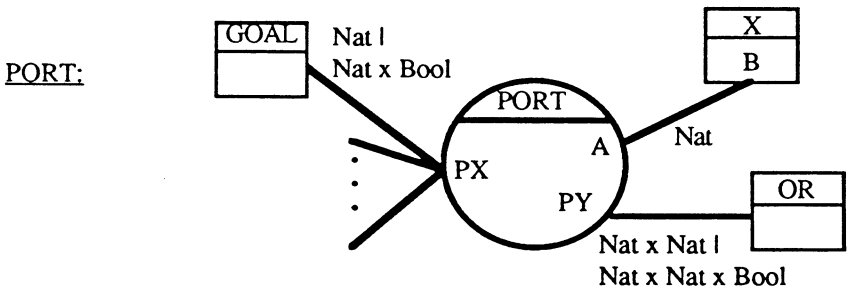


Figure 3.25

Les sous-réseaux sont liés entre eux de la manière suivante : chaque processus PORT qui représente le littéral L est lié à la famille de sous-réseaux de processus qui représente la définition de L.

La machine à inférence parallèle du programme logique :

C1: P <- P,Q,P.

C2: P <- R,Q.

C3: Q <- R.

C4: R.

C5: <- P.

est définie en FP2 de la manière suivante :

```
process CLAUSE_1 is
   HEAD_1 || GOAL_1 [1] || GOAL_2 [2] || GOAL_3 [3]
;; HEAD - GOAL
   + HY_1.GX_1 + HZ_1.GY_3
;; GOAL - GOAL
   + GY_1.GX_2 + GY_2.GX_3
endprocess
process CLAUSE_2 is
   HEAD_2 || GOAL_4 [4] || GOAL_5 [5]
;; HEAD - GOAL
   + HY_2.GX_4 + HZ_2.GY_5
;; GOAL - GOAL
   + GY_4.GX_5
endprocess
process DEFINITIONP is
   CLAUSE_1 || CLAUSE_2 [1] || OR_1 [2] || PORT_1 || PORT_2 || PORT_3
;; OR - CLAUSE
   + OY_1_1.HX_1 + OY_1_2.HX_2
;; CLAUSE - PORT
   + GZ_1.PX_1 + GZ_3.PX_1 + GZ_2.PX_2 + GZ_5.PX_2+ GZ_4.PX_3
endprocess
```

```
process CLAUSE_3 is
   HEAD_3 || GOAL_6 [6]
;; HEAD - GOAL
   + HY_3.GX_6 + HZ_3.GY_6
endprocess
process DEFINITIONQ is
   OR_2 [1] || CLAUSE_3 || PORT_4
;; OR - CLAUSE
   + OY_2_1.HX_3
;; CLAUSE - PORT
   + GZ_6.PX_4
endprocess
process ASSERTION_1 is
   HEAD_4 || PTRUE_1
   + HY_4.TX_1 + HZ_4.TY_1
endprocess
process DEFINITIONR is
   OR_3 [1] || ASSERTION_1
;; OR - ASSERTION
   + OY_3_1.HX_4
endprocess
process CLAUSE_5 is
   HEAD_5 || GOAL_7 [7]
;; HEAD - GOAL
   + HY_5.GX_7 + HZ_5.GY_7
endprocess
```

process DEFINITION **is**

OR_4 [1] || CLAUSE_5 || PORT_5

;; OR - CLAUSE

+ OY_4_1.HX_5

;; CLAUSE - PORT

+ GZ_7.PX_5

endprocess

process MIP **is**

X || DEFINITION + A_5.B

|| { DEFINITIONP_i + A_i_1.B + A_i_2.B + A_i_3.B | i = 1 ... }

|| { DEFINITIONQ_i + A_i_4.B | i = 1... } || { DEFINITIONR_i | i = 1 ... }

;; DEFINITION - DEFINITIONP

+ { PY_i_5.OX_j_1 | i = 1..., j = 1... }

;;DEFINITIONP - DEFINITIONP

+ { PY_i_1.OX_j_1 | i = 1..., j = 1... }

;;DEFINITIONP - DEFINITIONQ

+ { PY_i_2.OX_j_2 | i = 1..., j = 1... }

;;DEFINITIONP - DEFINITIONR

+ { PY_i_3.OX_j_3 | i = 1..., j = 1... }

;;DEFINITIONQ - DEFINITIONR

+ { PY_i_4.OX_j_3 | i = 1..., j = 1... }

endprocess

Notre exemple devient:

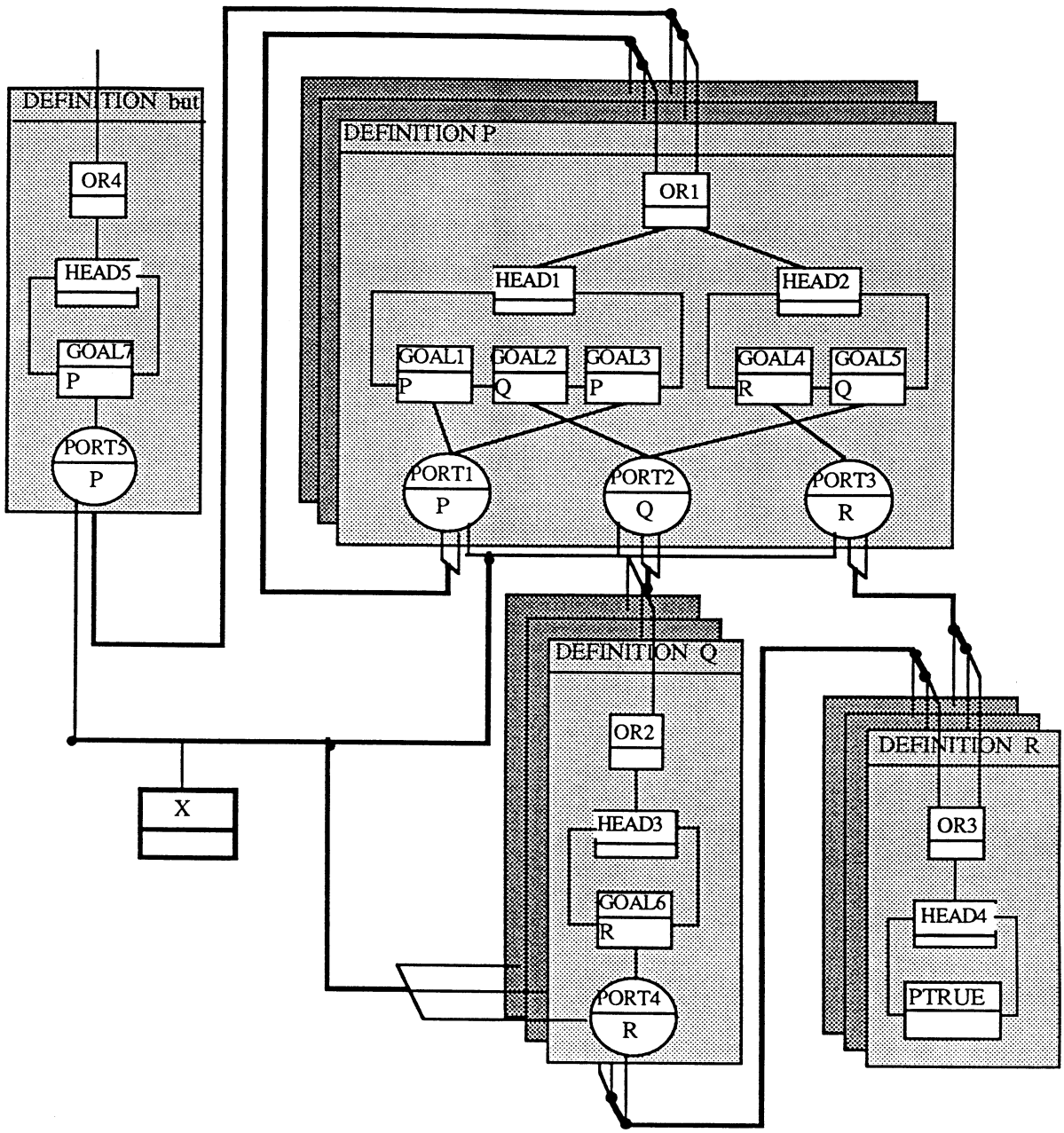


Figure 3.26

Il faut remarquer que cette description en FP2 peut être obtenue par traduction systématique à partir de l'ensemble des clauses. La machine commence à travailler lorsqu'elle reçoit, à travers le connecteur OX_4, deux natureles quelconque (0 et 0 par exemple). Lorsque la machine obtient une solution, celle-ci nous est communiquée à travers le même connecteur. Nous présentons maintenant une exécution du programme logique qui nous occuppe dans la machine construite pour ce programme.

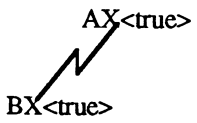
L'exécution du programme est décrite grâce à une liste de lignes qui contient :

• L'identification de la ligne.

• Le nom du processus dont le comportement est décrit dans la ligne.

• Le comportement du processus décrit dans la ligne. Ce comportement est décrit comme un système de transition d'états, c'est-à-dire la succession d'états par lesquels le processus passe. La transition entre états est notée par le symbole "==>", au dessus duquel apparaît l'ensemble de connecteurs et de valeurs qui circulent à travers les connecteurs pour effectuer la transition.

Nous avons aussi utilisé la convention du dessin suivante :



Le message "true" est transmis à travers AX.BX

Figure 3.27

L'exécution du programme est décrite dans la figure ci-dessous.

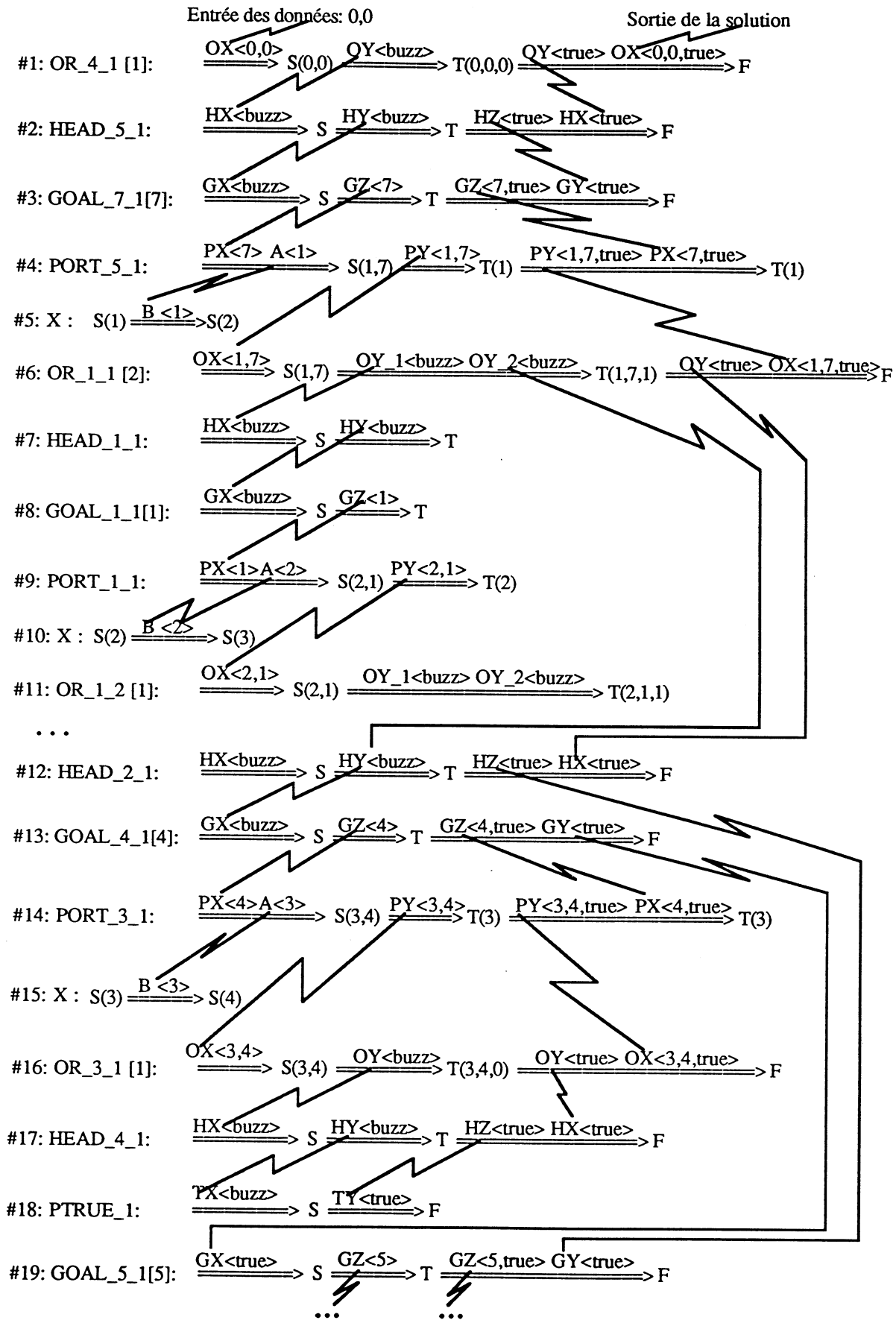

Entrée des données: 0,0
Sortie de la solution
#1: OR_4_1 [1]:
OX<0,0> S(0,0) QY<buzz> T(0,0,0) QY<true> OX<0,0,true> F
#2: HEAD_5_1:
HX<buzz> S HY<buzz> T HZ<true> HX<true> F
#3: GOAL_7_1[7]:
GX<buzz> S GZ<7> T GZ<7,true> GY<true> F
#4: PORT_5_1:
PX<7> A<1> S(1,7) PY<1,7> T(1) PY<1,7,true> PX<7,true> T(1)
#5: X : S(1) B <1> S(2)
#6: OR_1_1 [2]:
OX<1,7> S(1,7) OY_1<buzz> OY_2<buzz> T(1,7,1) QY<true> OX<1,7,true> F
#7: HEAD_1_1:
HX<buzz> S HY<buzz> T
#8: GOAL_1_1[1]:
GX<buzz> S GZ<1> T
#9: PORT_1_1:
PX<1>A<2> S(2,1) PY<2,1> T(2)
#10: X : S(2) B <2> S(3)
#11: OR_1_2 [1]:
OX<2,1> S(2,1) OY_1<buzz> OY_2<buzz> T(2,1,1)
• • •
#12: HEAD_2_1:
HX<buzz> S HY<buzz> T HZ<true> HX<true> F
#13: GOAL_4_1[4]:
GX<buzz> S GZ<4> T GZ<4,true> GY<true> F
#14: PORT_3_1:
PX<4>A<3> S(3,4) PY<3,4> T(3) PY<3,4,true> PX<4,true> T(3)
#15: X : S(3) B <3> S(4)
#16: OR_3_1 [1]:
OX<3,4> S(3,4) OY<buzz> T(3,4,0) OY<true> OX<3,4,true> F
#17: HEAD_4_1:
HX<buzz> S HY<buzz> T HZ<true> HX<true> F
#18: PTRUE_1:
TX<buzz> S TY<true> F
#19: GOAL_5_1[5]:
GX<true> S GZ<5> T GZ<5,true> GY<true> F
• • •
• • •

Figure 3.28

Initialement, le processus OR_4_1 (ligne N°1) reçoit les valeurs 0,0 à travers son connecteur OX, et en conséquence, le processus passe à l'état S(0,0). Puis il réalise la communication OY(buzz) qui active le processus HEAD_5_1. Puis HEAD_5_1 passe à l'état S pendant que OR_4_1 reste dans l'état S(0,0). De cette manière on peut lire l'exécution jusqu'à la ligne N°6. Lorsque le processus OR_1_1 se trouve dans l'état S(1,7), la communication OY_1(buzz), OY_2(buzz) se réalise, comme conséquence de cette communication les processus HEAD_1_1 (ligne N°7) et HEAD_2_1 (ligne N°12) sont activés. L'exécution à partir de HEAD_1_1 est un cycle infini et l'exécution de HEAD_2_1 aboutit à une solution.

Voyons ici le chemin d'exécution qui passe par le processus HEAD_2_1,le pas intéressant est l'intéraction des processus HEAD_4_1 (ligne N°17) et PTRUE_1 (ligne N°18). Lorsque le processus HEAD_4_1 est dans l'état S, il réalise la communication HY(buzz) qui active le processus PTRUE_1 et fait que HEAD_4_1 passe à l'état T et que PTRUE_1 passe à l'état S. PTRUE_1 et HEAD_4_1 sont donc prêts à se synchroniser et à transmettre la valeur "true". Le processus HEAD_4_1 ne peut réaliser la communication HZ(true) HX(true) que si le processus OR_3_1 intervient aussi dans cette communication. Par conséquent, les processus PTRUE_1, HEAD_4_1, OR_3_1,PORT_3_1, GOAL_4_1 se synchronisent avec le processus GOAL_5_1 (ligne 19). L'inférence continue et enfin les processus GOAL_5_1, HEAD_2_1, OR_1_1,..., OR_4_1 se synchronisent pour donner la solution du programme "true".

3.2. Le modèle pour les clauses de Horn.

Dans le modèle présenté précédemment, un programme logique est résolu grâce à l'inter-relation entre plusieurs processus qui communiquent entre eux par messages. Chaque processus se charge de résoudre une partie du programme. Par exemple, un processus CLAUSE doit résoudre l'ensemble des littéraux qui forment le corps d'une clause du programme et un processus OR doit résoudre un littéral en particulier.

Dans cette section nous présentons une extension de ce modèle qui permet de travailler avec les clauses de Horn. Le modèle étendu garde les caractéristiques énumérées ci-dessus mais le comportement des processus est différent. Le nouveau modèle permet le traitement des termes et la possibilité de calculer toutes les solutions possibles. Les activités des processus sont décrites ensuite.

Par simplicité, nous considérons d'abord le réseau comme composé de : processus OR et processus CLAUSE. Le comportement général des processus OR et CLAUSE peut être résumé ainsi :

Lorsqu'un littéral L doit être résolu, un processus OR reçoit L. Le littéral doit être résolu de toutes les manières possibles. Pour cela, L doit être unifié avec toutes les têtes de clauses du programme qui ont en tête le même symbole de prédicat que L et qui sont représentées par les

processus CLAUSE liés au processus OR. Le OR envoie le littéral L à tous ses processus CLAUSE. Chaque processus CLAUSE unifie L avec la tête de la clause que ce processus représente. Si l'unification échoue, CLAUSE envoie au OR le message "false", sinon les littéraux du corps de la clause doivent être satisfaits. Toutes les solutions (liaisons des variables d'une clause) obtenues par les processus CLAUSE sont envoyées au processus père du processus OR. Si toutes les processus CLAUSE échouent, le OR envoie un message "false" à son processus CLAUSE père.

Un processus CLAUSE doit résoudre tous les littéraux qui appartiennent au corps de la clause qu'il représente. Chacun de ces littéraux est résolu par un processus OR que CLAUSE doit activer. Les littéraux de la clause sont résolus de gauche à droite et les processus OR chargés de cette tâche travaillent de la manière suivante : lorsqu'un processus OR satisfait un littéral L, les liaisons des variables obtenues sont envoyées comme message au processus CLAUSE. La destination du message dépend de la position qu'occupe L dans la clause. Si L se trouve dans la position extrême droite dans la clause le message est envoyée au processus OR père de CLAUSE, sinon le message est envoyé au processus OR qui se charge de satisfaire le littéral à la droite de L dans la clause. Le processus OR qui avait satisfait L peut continuer la recherche d'autres manières de satisfaire le littéral.

La machine à inférence parallèle pour les clauses de Horn doit :

- stocker les termes du programme ;
- manipuler ces termes.

Dans notre machine, les deux activités citées ci-dessus sont réalisées par un sous-réseau qui a un nombre non limité de processus MEMORY plus un processus N.

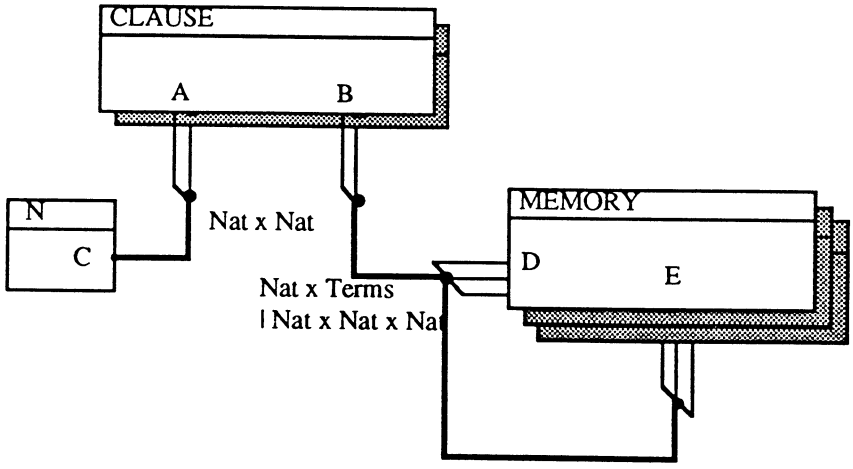


Figure 3.29

Le processus N se charge de générer des nombres naturels qui identifieront les processus MEMORY. Chaque fois qu'un processus MEMORY est activé, il doit recevoir une

identification. Cette identification permet aux processus OR et CLAUSE de savoir où sont les termes du programme. La spécification en FP2 de N est la suivante :

process N

connecteurs

C : **Nat** x **Nat**

states

S : **Nat**

vars

n, I : **Nat**

rules

==> S(0)

S(I) : C (n,I) ==> S (I+n)

endprocess

Un processus MEMORY se charge de stocker et de manipuler les termes d'une clause (un terme qui représente le littéral à la tête de la clause suivi des termes qui représentent les littéraux dans le corps de la clause).

Un processus MEMORY peut être activé soit par un processus CLAUSE, soit par un autre processus MEMORY.

Lorsqu'un processus CLAUSE active un nouveau processus MEMORY, trois processus se synchronisent :

- Le processus CLAUSE ;
- Le processus MEMORY ;
- Le processus N.

Lors de cet événement synchone, le processus MEMORY reçoit deux informations : les termes qu'il doit stocker et son identification.

La règle dans le processus CLAUSE qui permet l'activation d'un processus MEMORY a la forme suivante :

T(...,List-of-terms) : A(1,Id) B(Id,List-of-terms) ==> U(...,List-of-terms,Id)

Par le connecteur A, le processus CLAUSE communique avec N : CLAUSE demande une identification pour un processus MEMORY. Par le connecteur B, le processus CLAUSE active un nouveau processus MEMORY, en lui envoyant son identification et une liste de termes. Le processus CLAUSE retient la même liste de termes et l'identification du processus MEMORY.

Un processus CLAUSE peut aussi se charger d'initier ce que nous appelons la copie d'un processus MEMORY. La copie d'un processus MEMORY est un nouveau processus MEMORY, avec une nouvelle identification, contenant les mêmes termes que le processus copié. Pour réaliser la copie d'un processus (identifié avec le naturel Id1), trois processus se synchronisent :

- Le processus CLAUSE ;
- Le processus MEMORY identifié avec Id1 ;
- Le processus N.

Comme conséquence de cette synchronisation, le processus MEMORY identifié avec Id1 doit activer "n" autres processus MEMORY qui stockent ses termes.

La règle dans le processus CLAUSE qui déclenche la copie à la forme suivante :

T (..., Id1,List-of-terms) : A (n, Id) B (Id1, Id, n) ==> U (..., Id1,List-of-terms)

Par le connecteur A, le processus CLAUSE communique avec N : CLAUSE demande "n" identifications pour "n" processus MEMORY. Par le connecteur B, le processus CLAUSE demande au processus MEMORY identifié avec le naturel Id1, de faire "n" copies de lui même. L'identification de ces copies est : Id, Id+1,...,Id+n-1. Le processus CLAUSE retient la même liste de termes el l'identification du processus MEMORY copié : Id1.

Nous présentons ci-dessous les règles du processus MEMORY qui permettent :

- L'activation de ce processus MEMORY :

D (Id, List-of-terms) ==> S (Id, List-of-terms)

Le processus MEMORY est activé en recevant son identification : Id et la liste de termes à stocker.

- La reception de l'ordre de copie :

T (..., Id1, List-of-terms) : D (Id1, Id, n) ==> T'(n, Id, Id1,List-of-terms)

Le processus MEMORY identifié avec le naturel Id1, reçoit par le connecteur D le nombre de copies à faire : "n" et l'identification de la premier copie : Id.

•L'activation de "n" nouveaux processus MEMORY contenant une copie des termes :

T'(n+1, Id,Id1, List-of-terms) : E (Id, List-of-terms)

==> T' (n, Id+1,Id1, List-of-terms)

T' (0,...) ==> U(...)

Résolution d'un littéral.

Lorsqu'un littéral L appartenant à une clause τ doit être résolu, le processus OR chargé de résoudre L : OR_L effectue les actions suivantes :

• Il reçoit comme message du processus CLAUSE τ, l'identification F du processus MEMORY F où se trouvent stockés les termes de la clause τ.

• Soit "n" le nombre de processus CLAUSE (ou ASSERTION) associés au processus OR_L. Le processus OR_L se synchronise avec le processus MEMORY F pour le copier "n" fois. Soient $F_1,\ldots,F_n$ les identifications des processus MEMORY ainsi activés.

• OR_L envoie aux processus CLAUSE_i (i=1...n) qui dependent de lui (ou aux processus ASSERTION_i (i= 1...n)) l'identification du processus MEMORY F_i.

La figure ci-dessous montre cette situation :

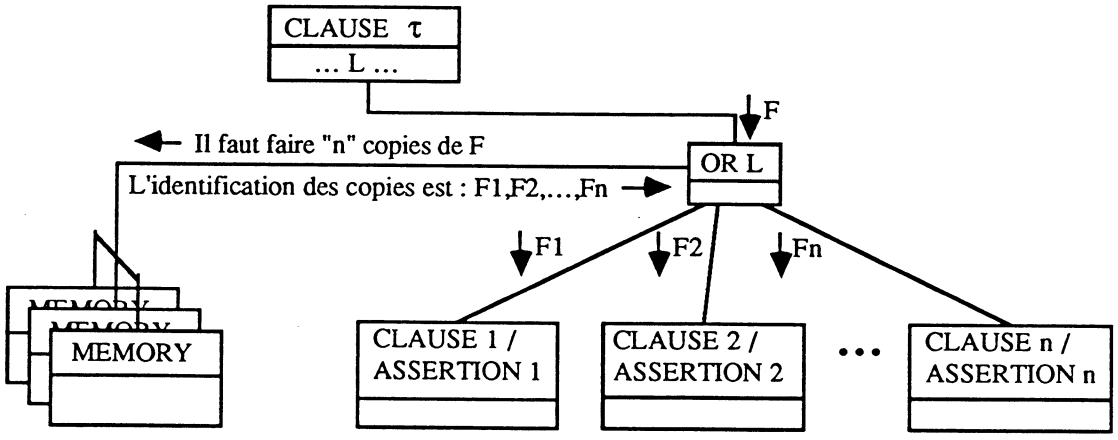


Figure 3.30

• Si l'identification de F_i est reçue par le processus ASSERTION_i qui représente l'assertion τ_i du programme, ce processus se charge de :

• Activer un processus MEMORY, soit F'_i l'identification de ce processus, qui contient les termes de l'assertion τ_i.

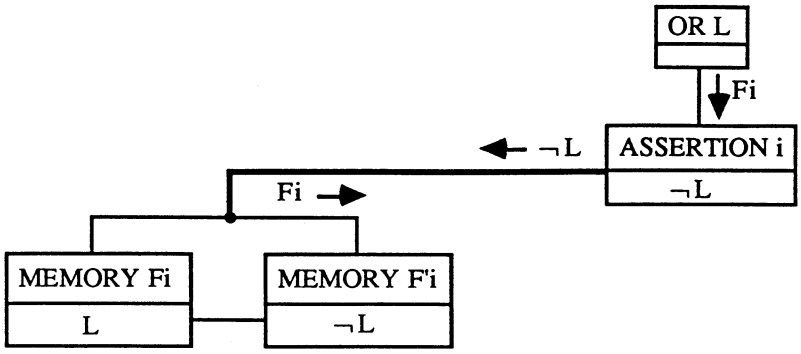


Figure 3.31

• Demander aux processus MEMORY identifiés avec F_i et F'_i de réaliser l'unification des littéraux L et ¬L qui sont stockés dans MEMORY F_i et MEMORY F'_i respectivement. F_i et F'_i se synchronisent et réalisent l'unification de ces littéraux. Nous verrons dans la section 3.2.4 comment se effectue l'unification.

• Si l'unification réussit, MEMORY F_i (ou MEMORY F'_i) envoie à ASSERTION_i le message "succès", ASSERTION_i envoie à OR_L l'identification F_i et OR_L envoie à CLAUSE τ l'identification F_i. Par la façon comme l'unification se réalise, les processus MEMORY F_i et MEMORY F'_i ne peuvent pas signaler quand l'unification échoue (voir section 3.2.4).

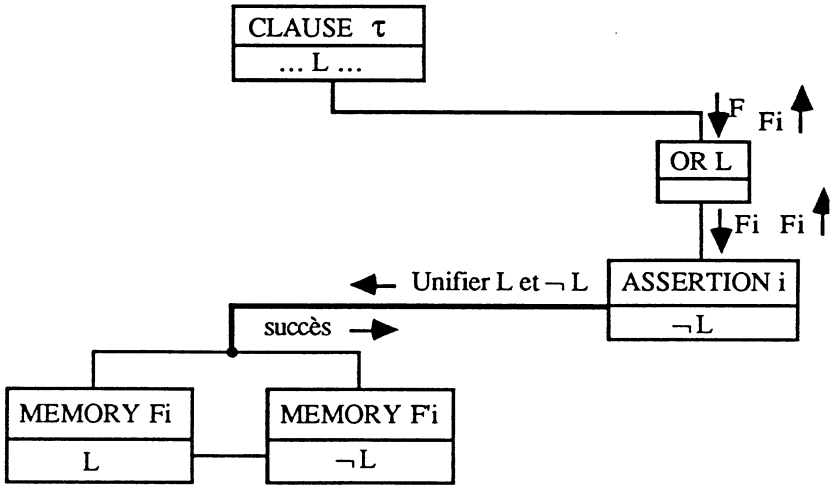


Figure 3.32

• Si l'identification F_i est reçue par le processus CLAUSE_i qui représente la clause τ_i du programme, le processus CLAUSE_i se charge de :

• Activer un processus MEMORY, soit F'_i l'identification de ce processus qui stocke les termes de la clause τ_i.

• Demander aux processus MEMORY identifiés avec F_i et F'_i de réaliser l'unification des littéraux L et $\neg$L qui sont stockés dans MEMORY F_i et MEMORY F'_i respectivement. F_i et F'_i se synchronisent et réalisent l'unification de ces littéraux.

• Si l'unification réussit, MEMORY F_i (ou MEMORY F'_i) envoie à CLAUSE_i le message "succès".

• L'identification de F'_i est envoyée par OR_L au processus OR_{L1} qui entête la définition du premier littéral L1 du corps de la clause τ_i.

• Le processus OR_{L1} se charge de satisfaire L1 et envoie à la CLAUSE_i toutes les solutions obtenues. Ces solutions sont envoyées au processus OR_{L2} qui se charge de satisfaire L2 et ainsi de suite jusqu'à ce que le dernier littéral de la CLAUSE_i soit satisfait.

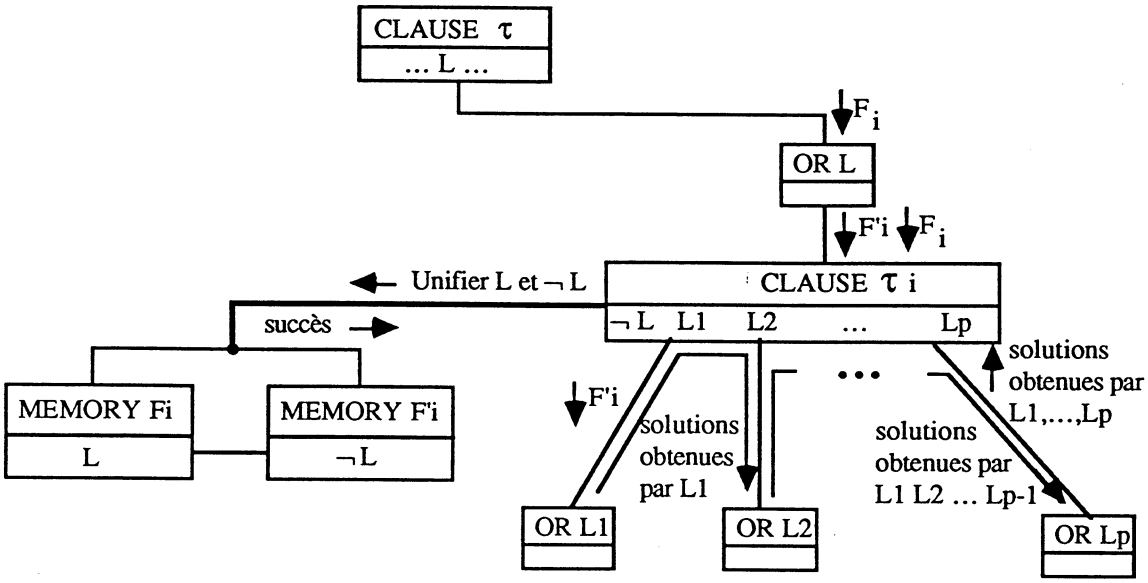


Figure 3.33

Rappelons qu'une conséquence de l'unification de L (dans MEMORY F_i) avec $\neg$L (dans MEMORY F'_i) est la liaison des variables de F_i (F'_i) à celles de F'_i (F_i).

• Les variables des solutions obtenues par $L_1,\ldots,L_p$ sont liées aux termes de F_i mais la réciproque est fausse. Il est donc nécessaire d'activer autant de copies de F_i que des solutions obtenues par $L_1,\ldots,L_p$ et associer à chacune de ces copies une des solutions. Nous verrons dans la suite comment effectuer cette association.

• Le processus OR_L reçoit de CLAUSE τ_i l'identification de ces copies et les envoie au processus CLAUSE τ.

Un processus CLAUSE doit résoudre tous les littéraux qui appartiennent au corps de la clause qu'il représente. Nous avons vu qu'un processus CLAUSE est formé par un processus HEAD et autant de processus GOAL que de littéraux dans le corps de cette clause. Le processus HEAD se charge de commander l'unification entre le littéral à satisfaire et la tête de la clause qu'il représente, chaque processus GOAL se charge de satisfaire le littéral qu'il représente. Les solutions obtenues par les processus GOAL sont acheminées par le processus HEAD vers le processus OR qui groupe toutes les clauses qui appartiennent à la même définition que HEAD.

Le comportement des processus OR, HEAD et GOAL est le suivant :

Processus OR

• Reçoit du processus DEFINITION (en fait, d'un GOAL qui appartient à un processus CLAUSE d'un processus DEFINITION) qui l'appelle:

L'identification (F) d'un processus du sous-réseau MEMORY et la position (p) qu'occupe le littéral à satisfaire dans le processus <F,p>.

Actions:

Supposons que le processus OR soit lié à "n" processus HEAD.

Le processus OR se synchronise avec le processus MEMORY dont l'identification est F pour faire "n" copies de F. Soient F1,...,Fn les identifications des processus MEMORY ainsi activés.

• Envoie à chaque HEADi

L'identification d'un des processus MEMORY qui viennent d'être activés et la position où se trouve le terme à unifier. C'est-à-dire, le processus OR envoie à $HEAD_i$ ($i \varepsilon 1...n$) : $<F_i,p>$.

• Reçoit de chaque HEADi

L'identification de k_i processus MEMORY: $F_{i1},...,F_{iki}$. Ces k_i processus stockent les termes de la clause où se trouve L et ils représentent les différentes manières de résoudre L à partir de la clause dont la tête est représentée par HEAD_i.

• Envoi au processus DEFINITION (en fait, d'un GOAL qui appartient à un processus CLAUSE d'un processus DEFINITION) qui l'a appelé

Toutes les identifications des processus MEMORY qui ont été envoyées par ses processus HEAD's:

$F_{11},\ldots,F_{1k1}$

$F_{21},\ldots,F_{2k2}$

:

$F_{n1},\ldots,F_{nkn}$

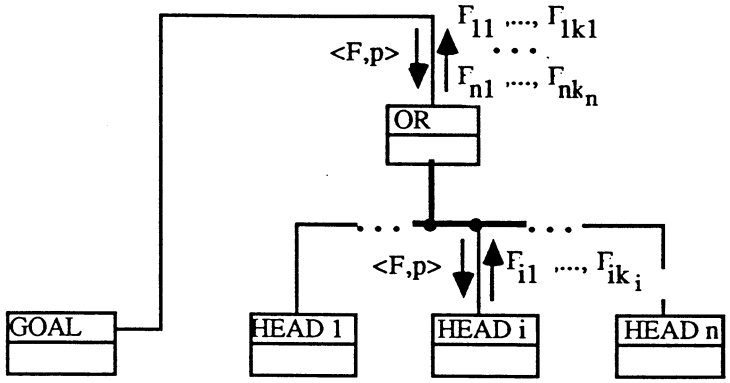


Figure 3.34

Processus HEAD

- Reçoit du processus OR qui l'a appelé

  L'identification d'un processus du sous-réseau MEMORY et la position d'un terme à unifier: <F,p>

  Actions:

  HEAD active un processus du sous-réseau MEMORY où sont stockés les termes de la clause que HEAD représente. Soit F' l'identification de ce processus.

  HEAD envoie un message aux processus MEMORY identifiés avec F et F' en leur demandant l'unification du terme qui se trouve dans la position p de F avec le terme qui se trouve dans la position 1 de F'.

  Si l'unification échoue le processus HEAD envoie le message "fail" à son processus OR père et finit son travail.

  Si l'unification réussit, le processus HEAD poursuit son travail.

- Envoie au premier GOAL

  L'identification du processus F'.

- Reçoit de son dernier GOAL

  L'identification de plusieurs processus MEMORY F_i.

  Rappelons que le processus HEAD a demandé l'unification des termes qui se trouvent dans les positions <F,p> et <F',1>, et que $F',F'_1,\ldots,F'_k$ contient les termes d'une même clause.

Les variables de $F', F'_1, \ldots, F'_k$ sont liées aux termes de F tandis que les variables de F ne peuvent être liées qu'aux termes d'un seul processus F'. Il est donc nécessaire activer "k" copies de F : $F_1, \ldots, F_k$ et réaliser les changements suivants :

Pour chaque variable de F qui est liée à un terme de F', la variable correspondante de F_i sera liée au terme correspondant de F'_i et vice-versa.

- Envoi au processus OR qui l'a appelé

  Les identifications F_i's.

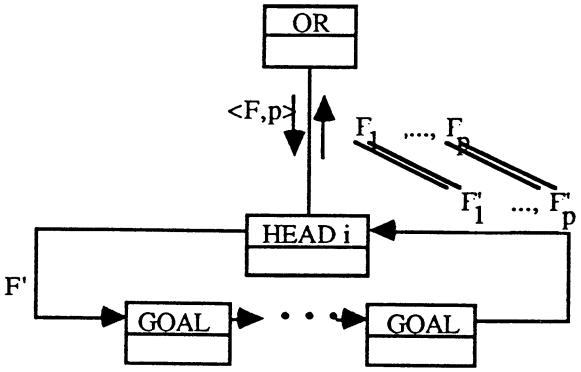


Figure 3.35

Processus GOAL

- Reçoit du processus HEAD ou de son frère gauche

  L'identification F d'un processus du sous-réseau MEMORY.

- Envoie à une DEFINITION

  L'identification F d'un processus du sous-réseau MEMORY et la position i du terme dans F avec lequel effectuer l'unification.

- Reçoit de la DEFINITION

  L'identification des processus copies de chaque processus identifié avec F : F1,...,Fq. Ces processus copies représentent les différentes solutions obtenues à partir du F envoyé.

- Envoie au GOAL à sa droite ou à HEAD

  Les F_i (i=1...q) qu'il reçoit.

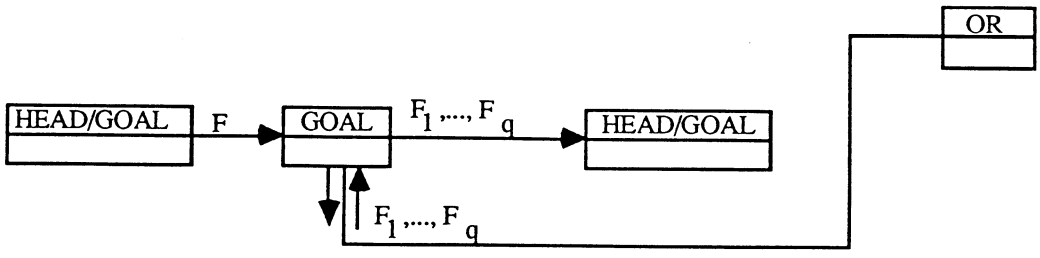


Figure 3.36

L'exemple ci-dessous permet de décrire le comportement de la machine pour les clauses de Horn.

Exemple 3.2 :

Soit le programme logique suivante :

P(X) <- Q(X), R(X)

Q(a)

Q(b)

R(a)

R(b)

<- P(Z)

La configuration du réseau de base de ce programme (sans tenir compte des processus qui gèrent la mémoire) est la suivante :

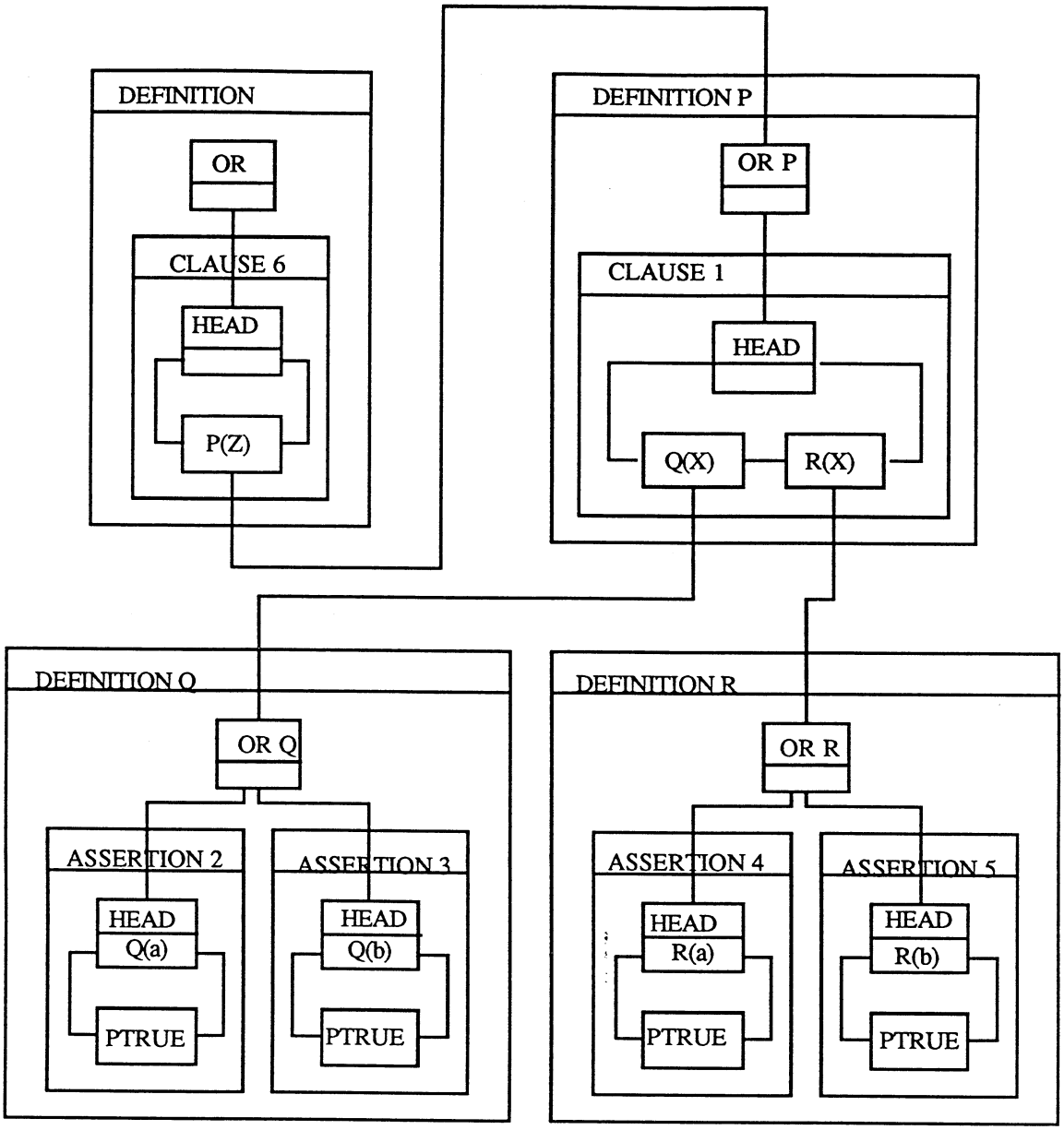
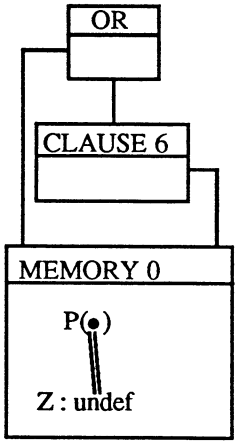

DEFINITION
OR
CLAUSE 6
HEAD
P(Z)
DEFINITION P
OR P
CLAUSE 1
HEAD
Q(X)
R(X)
DEFINITION Q
OR Q
ASSERTION 2
HEAD
Q(a)
PTRUE
ASSERTION 3
HEAD
Q(b)
PTRUE
DEFINITION R
OR R
ASSERTION 4
HEAD
R(a)
PTRUE
ASSERTION 5
HEAD
R(b)
PTRUE

Figure 3.37

Initialement le réseau a deux processus actifs : les deux processus qui correspondent à la DEFINITION de la clause de buts : un processus OR et son processus CLAUSE associé : CLAUSE 6, le processus CLAUSE 6 contient le terme P(Z). CLAUSE 6 active un processus MEMORY pour ranger le terme de la clause de buts : P(Z). Soit 0 l'identification de ce processus : P(Z) est rangé dans MEMORY 0. La valeur de Z est non-liée (undef) (figure 38).



La double ligne indique la liaison de la variable de P représentée par un point avec la valeur de la variable Z ("undef")

Figure 3.38

CLAUSE 6 envoie au sous-réseau DEFINITION$_P$ (processus OR$_P$) l'identification du processus MEMORY où se trouve le terme à satisfaire : P(Z), c'est-à-dire processus MEMORY 0. Chaque clause dont la tête est P peut résoudre P(Z) d'une manière différente. Il est donc nécessaire de faire autant de copies de MEMORY 0 que de clauses dont la tête est P. Pour assurer un traitement uniforme, cette copie est faite même lorsque P(Z) peut être résolu d'une seule manière. Le processus OR$_P$ de la DEFINITION$_P$ reçoit l'identification 0 et se synchronise avec MEMORY 0 pour activer une copie de MEMORY 0 : soit 1 l'identification de cette copie. Le processus OR$_P$ envoie l'identification 1 à son processus CLAUSE 1. CLAUSE 1 active un processus MEMORY identifié par 2 où sont rangés les termes : P(X), Q(X), R(X). CLAUSE 1 demande aux processus MEMORY identifiés avec les nombres 1 et 2 que P(Z) de MEMORY 1 et P(X) de MEMORY 2 soient unifiés, MEMORY 1 et MEMORY 2 se synchronisent et font cette unification. On verra dans la section 3.2.4 comment se réalise l'unification.

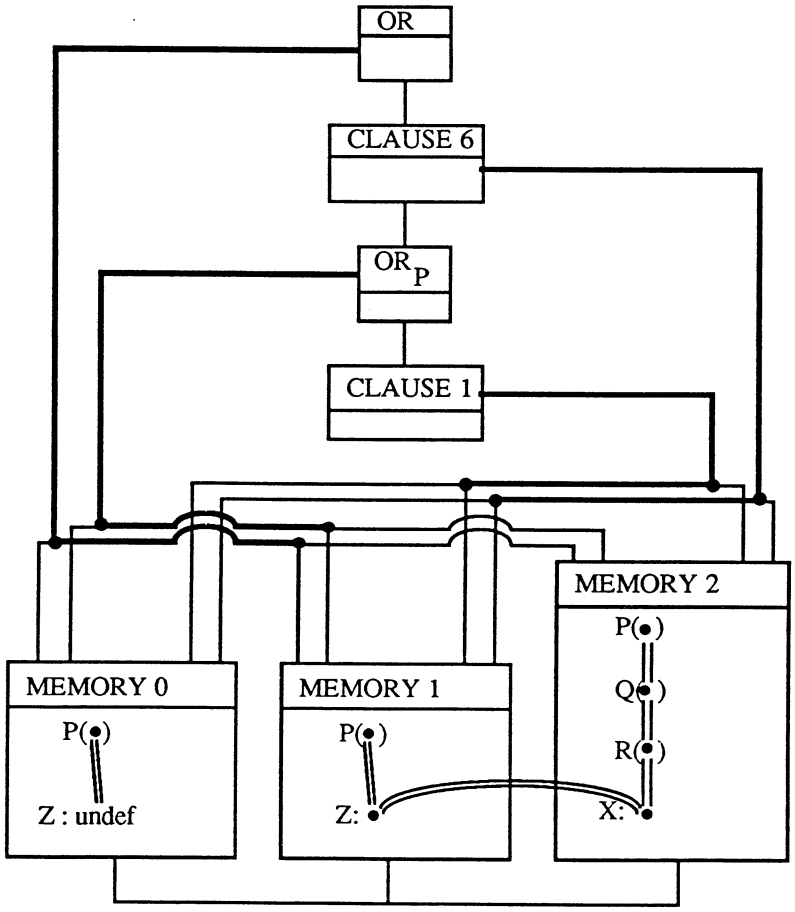


Figure 3.39

Un processus OR$_Q$ est activé par le processus CLAUSE 1. OR$_Q$ reçoit l'identification du processus MEMORY 2 et doit satisfaire le littéral Q(X) qui se trouve rangé dans ce processus. OR$_Q$ se synchronise avec MEMORY 2 pour demander deux copies de ce processus. MEMORY 2 active deux processus MEMORY identifiés par 3 et 4 qui contiennent les termes de MEMORY 2. Le processus OR$_Q$ a deux processus ASSERTION associés : ASSERTION 2 et

ASSERTION 3, OR_Q envoie à ASSERTION 2 l'identification MEMORY 3 et envoie à ASSERTION 3 le nombre 4. ASSERTION 2 et ASSERTION 3 activent deux processus MEMORY identifiés par 5 et 6 respectivement. Le processus MEMORY identifié par 5 reçoit le terme de la clause 2 : Q(a) et le processus identifié avec 6 reçoit le terme de la clause 3 : Q(b). L'unification du terme Q(X) de MEMORY 3 avec le terme Q(a) de MEMORY 5 a lieu, ainsi que celle du terme Q(X) de MEMORY 4 avec le terme Q(b) de MEMORY 6. Voir figure 40.

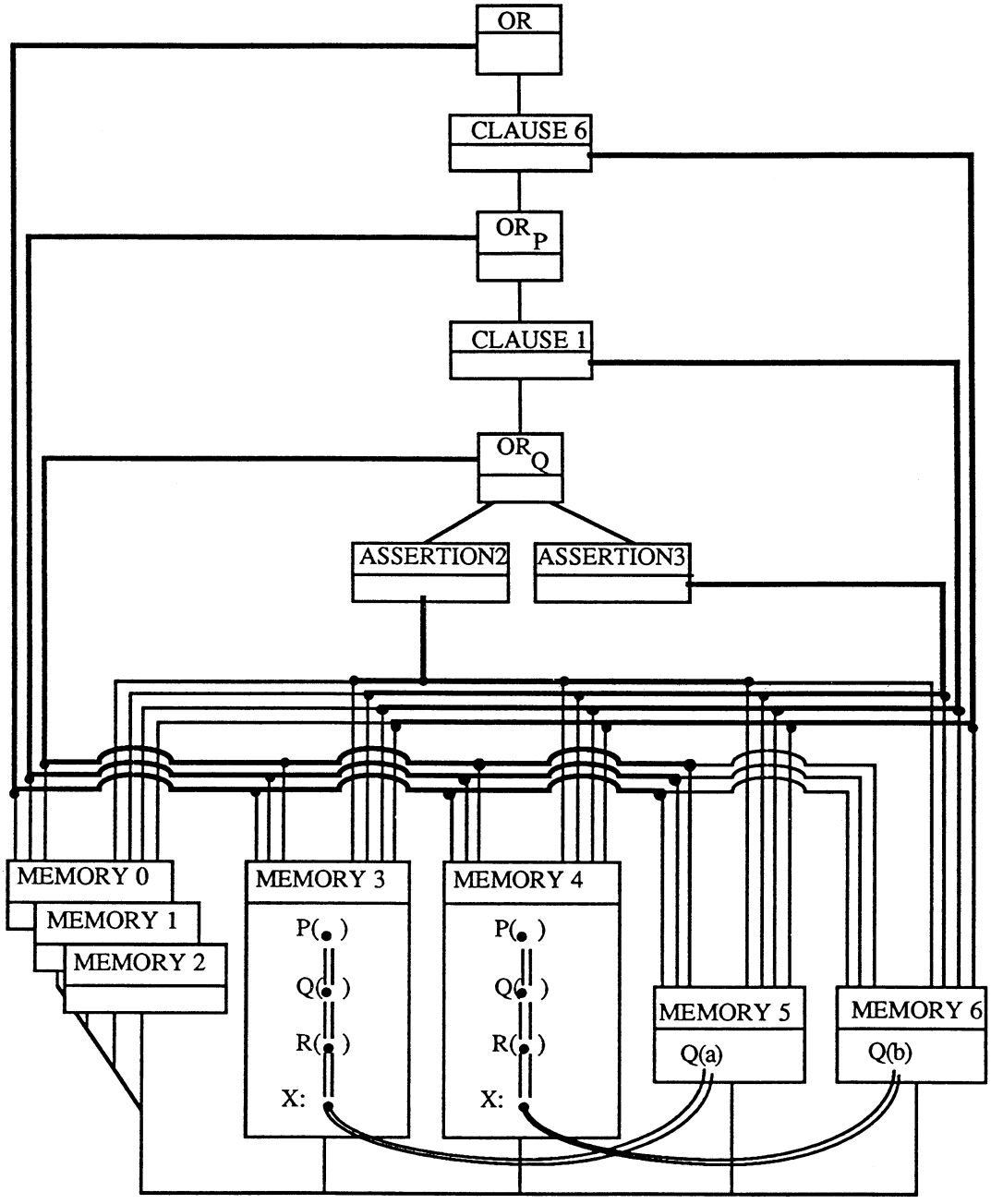


Figure 3.40

Supposons que ASSERTION 3 arrive à obtenir le résultat de l'unification avant ASSERTION 2. Dans ce cas le processus CLAUSE 1 active une définition de R et OR_R doit satisfaire le littéral R(X) qui se trouve dans MEMORY 4. Le processus OR_R se synchronise avec MEMORY 4 pour activer deux copies de MEMORY 4 identifiées avec les nombres 7 et 8. La figure 41 illustre cette situation.

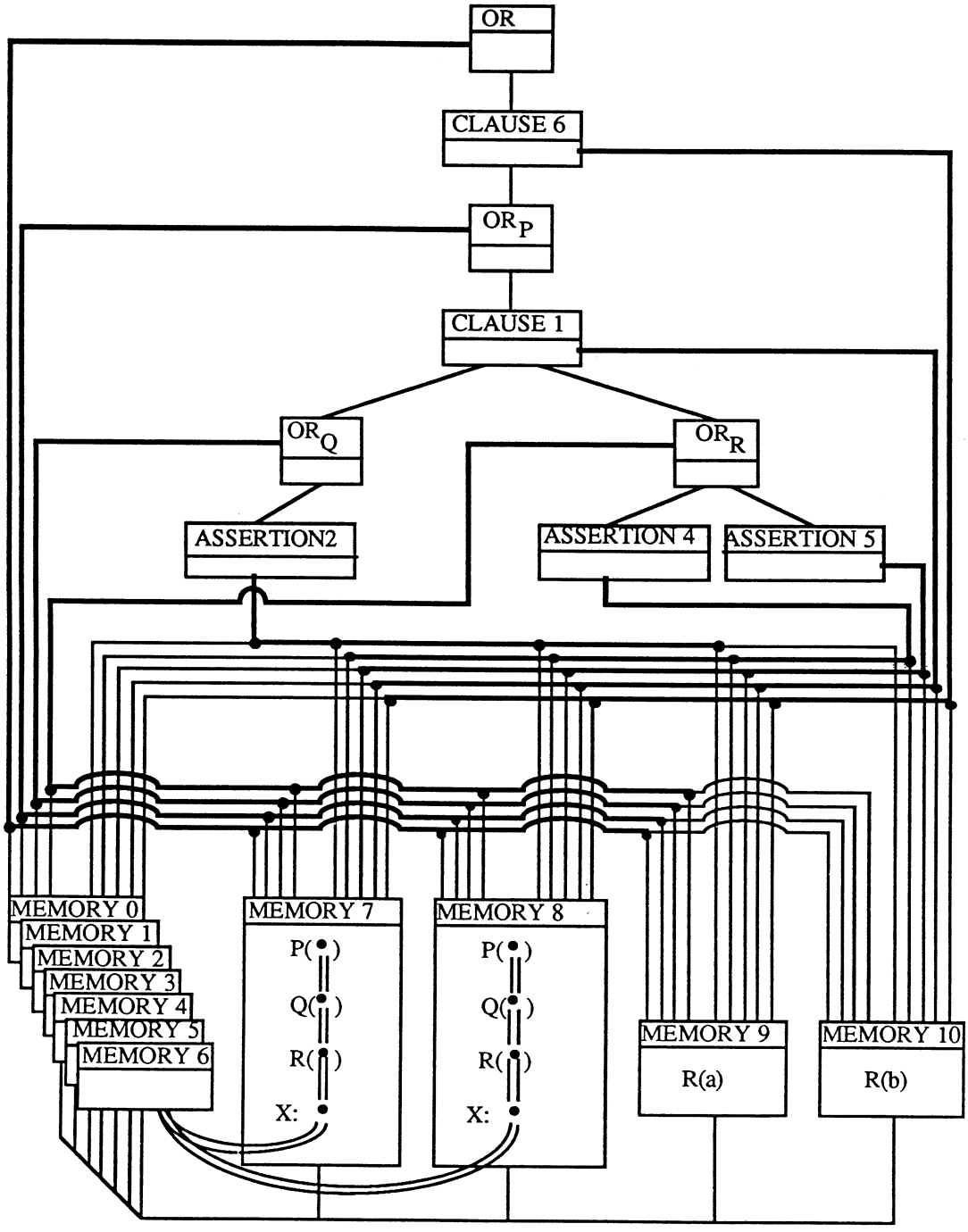


Figure 3.41

L'unification de R(X) et R(b) des processus MEMORY 7 et MEMORY 10 respectivement réussit. MEMORY 7 représente une solution de P(X), l'identification de cette solution est

envoyée à CLAUSE 1. CLAUSE 1 avait demandé l'unification de P(Z) (MEMORY 1) avec P(X) (MEMORY 2). CLAUSE 1 se synchronise avec MEMORY 1 pour demander une copie de MEMORY 1, le processus ainsi activé est identifié par le nombre 11. La figure 3.42 montre les processus MEMORY 6, MEMORY 7 et MEMORY 11.

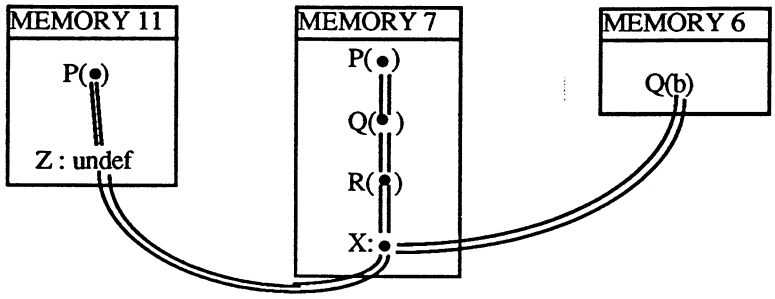


Figure 3.42

Fin de l'exemple

3.2.1. Représentation des termes dans la mémoire.

Dans la section précédente nous avons décrit de manière générale la machine à inférence parallèle pour les Clauses de Horn. Dans le modèle que nous avons décrit, chaque fois qu'un littéral L doit être résolu, il faut dupliquer l'information de la clause qui contient L autant de fois qu'il y a de clauses dans la définition de L. L'information d'une clause se trouve dans un processus du sous-réseau MEMORY, les processus-copies appartiennent aussi au sous-réseau MEMORY. Nous présentons dans cette section la façon dont les termes sont représentés dans la mémoire.

Le sous-réseau MEMORY est composé de deux types de processus : processus M-CLAUSE (pour stocker la partie non variable des termes des clauses) et processus FRAME (pour stocker le valeur des variables des clauses).

La figure ci-dessous montre le sous-ensemble de liaisons des processus M-CLAUSE et FRAME avec le reste de la machine qui permet que se réalisent les opérations d'activation et de copie.

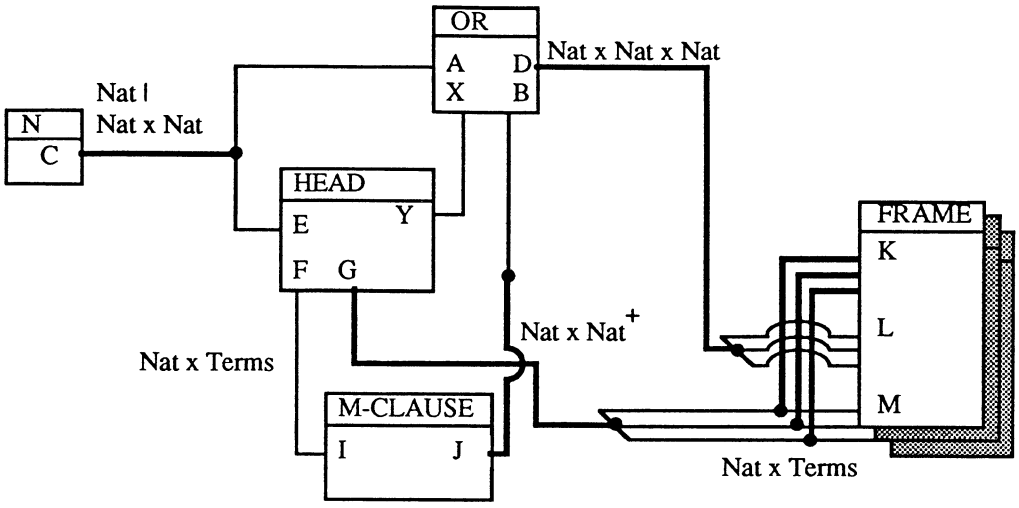


Figure 3.43

Pour stocker les termes d'une clause dans le sous-réseau qui se charge de la gestion de la mémoire, il faut activer d'une part un processus M-CLAUSE où on range la partie non-variable des termes de la clause et il faut activer d'autre part un processus FRAME où on stocke la valeur des variables de la clause.

L'activation des processus M-CLAUSE et FRAME est réalisée par un processus HEAD de la manière suivante : le processus HEAD se synchronise avec le processus N, un processus M-CLAUSE et un processus FRAME. Comme conséquence de cette synchronisation :

• Le processus N génère un identificateur pour M-CLAUSE et un autre pour FRAME ;

• M-CLAUSE et FRAME sont activés et stockent la partie non-variable des termes de la clause et la valeur des variables de la clause respectivement.

La spécification en FP2 de l'opération d'activation est la suivante :

process HEAD

...

;; HEAD active un processus M-CLAUSE et un processus FRAME.

T (...) : E (2,Id1) F (Id1, List-of-terms-fixe) G(Id1+1,List-of-terms-var) ==> U (...)

...

endprocess

Par le connecteur E, le processus HEAD communique avec N et demande deux identifications : la premier Id1 pour un M-CLAUSE et la deuxième Id1+1 pour un FRAME. Par le connecteur F, HEAD active un nouveau processus M-CLAUSE, en lui envoyant son

identification et la partie non-variable des termes de la clause. Par le connecteur G, HEAD active un nouveau processus FRAME, en lui envoyant son identification et la partie variable de la clause.

process M-CLAUSE

...

;; M-CLAUSE est activé

I (Id,List-of-terms) ==> T ([Id],List-of-terms)

...

endprocess

process FRAME

...

;; FRAME est activé

K (Id,List-of-terms) ==> T (Id,List-of-terms)

...

endprocess

Pour réaliser plusieurs copies d'une clause, il faut réaliser la copie du processus FRAME qui contient la valeur des variables des termes de la clause et il faut envoyer un message au processus M-CLAUSE qui stocke la partie non-variable des termes de la clause en indiquant l'identification des processus FRAME qui sont activés comme conséquence de cette opération de copie. Les processus suivants se synchronisent pour réaliser la copie des termes d'une clause:

• Le processus OR qui demande la copie ;

• Le processus N qui génère l'identification des processus FRAME qui vont s'activer pour stocker la valeur des variables de la clause ;

• Le processus M-CLAUSE associé à la clause pour stocker l'identification des nouveaux processus FRAME et

• Le processus FRAME à copier qui reçoit un message pour activer des processus FRAME pour stocker ses termes.

La spécification en FP2 de l'opération de copie est la suivante :

processus OR

...

;; OR demande "n" copies de la clause qui se trouve stockée dans

;; le processus M-CLAUSE M et le processus FRAME F.

T (...) : A (n,Id1) B (M,[Id1,...,Id1+n]) D (F,Id1,n) `==> U (...)

...

endprocess

Par le connecteur A, le processus OR communique avec N : OR demande "n" identifications. Par le connecteur D, OR communique avec le processus FRAME F qu'il doit activer. Par le connecteur B, OR communique avec le processus M-CLAUSE M et lui envoie l'identification des "n" processus FRAME qui lui sont associés.

process M-CLAUSE

...

;; Comme conséquence d'une opération de copie,

;; de processus M-CLAUSE a de nouveaux processus FRAME associés.

T (Sons,List-of-terms,...): J (M,Sons1) ==> U(append(Sons,Sons1),List-of-terms,...)

...

endprocess

process FRAME

...

;; FRAME reçoit l'ordre de copier ses termes dans "n" processus FRAME.

T (F,List-of-terms,...) : L (F, Id1, n) ==> T' (F,n, Id1,List-of-terms,...)

;; FRAME réalise la copie

T' (F,0,...) ==> U (...)

T' (F,n+1, Id, List-of-terms,...) : M (Id, List-of-terms)

==> T' (n, Id+1, List-of-terms)

endprocess

L'exemple ci-dessous illustre comment l'information des clauses est distribuée dans le sous-réseau MEMORY. Puis nous définissons comment sont représentés les termes dans les processus M-CLAUSE et FRAME.

Exemple 3.3 :

Soit le programme suivant :

CL1 : P(X,f(Y,Z)) <- Q(Y,Z)

CL2 : Q(W,b)

CL3 : Q(b,W)

La configuration de réseau de base de ce programme (sans tenir compte des processus qui gèrent la mémoire) est la suivante :

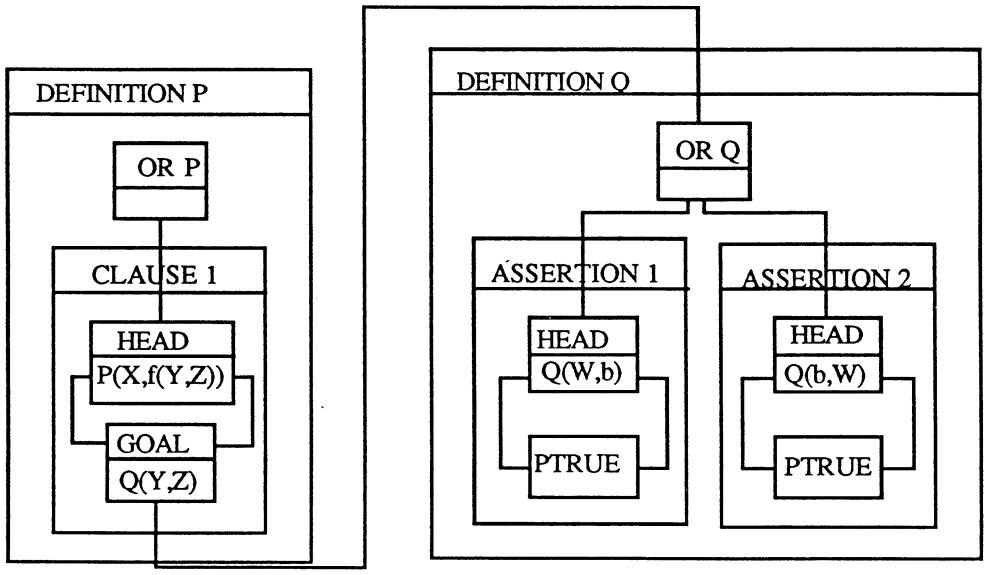


Figure 3.44

Supposons que la clause 1 soit invoquée et que comme conséquence de cette invocation, la variable X soit liée au terme "a", le processus HEAD qui représente la clause 1 active un processus M-CLAUSE et un processus FRAME du sous-réseau MEMORY pour stocker l'information de clause 1.

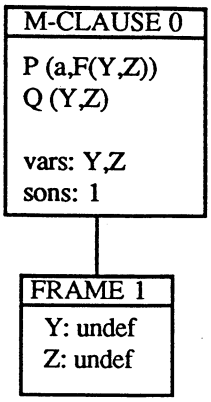


Figure 3.45

La prochaine étape d'exécution consiste à résoudre le but Q(Y,Z). La définition de Q permet de résoudre ce but de deux façons différentes définies par le clauses clause 2 et clause 3. Le processus OR chargé de résoudre le but Q(Y,Z) active deux processus FRAME pour stocker les variables de la clause 1 (processus FRAME qui sont identifiés avec 3 et 4 dans la figure ci-dessous) :

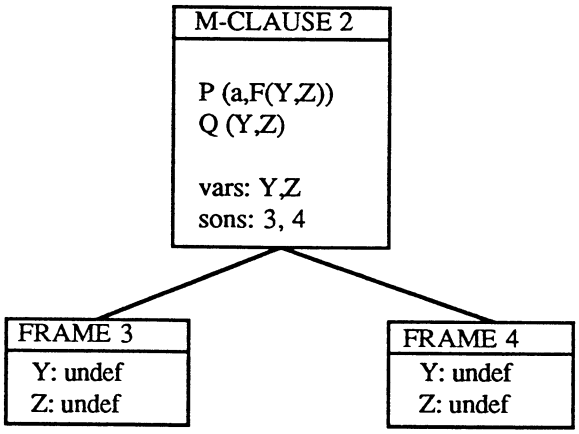


Figure 3.46

Le processus DEFINITIONQ a deux processus HEAD, un de ces processus travaille avec M-CLAUSE 2 et FRAME 3 et l'autre travaille avec M-CLAUSE 2 et FRAME 4. Ces processus HEAD activent des processus du sous-réseau MEMORY où sont stockés les termes des clauses 2 et 3. Les processus 5-6, et 7-8 stockent les termes des clauses 2 et 3 respectivement.

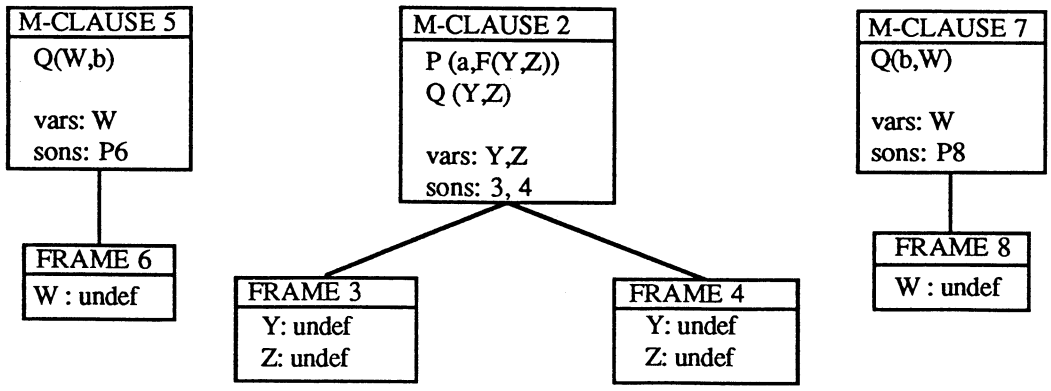


Figure 3.47

Comme conséquence de l'unification de Q(Y,Z) avec Q(W,b) et de l'unification de Q(Y,Z) avec Q(b,W) nous avons la configuration de mémoire suivante :

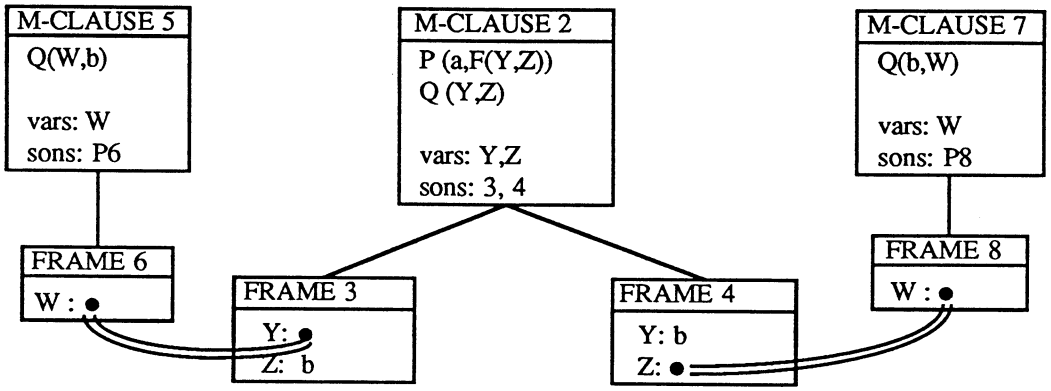


Figure 3.48

Les étapes ultérieures de l'exécution n'apportent pas d'information intéressante pour le propos de cet exemple.

Fin de l'exemple

Nous rappelons que le sous-processus MEMORY est formé par deux types de processus : M-CLAUSE et FRAME. Les termes d'une clause se stockent dans les processus M-CLAUSE et les valeurs des variables de ces termes se stockent dans les processus FRAME.

La représentation d'un terme dans le sous-réseau MEMORY suit les règles suivantes :

1. Tous les littéraux d'une clause sont stockés dans les processus M-CLAUSE sous la forme de termes[1].

1.1. Pour une clause de la forme :

$$A <- B_1,\ldots,Bn$$

[1] Entre un littéral et un terme il y a une différence sémantique mais non syntaxique.

le terme qui représente le littéral A occupe la position 1 dans le processus et le terme qui représente le littéral B_i i ε [1,n] occupe la position i+1.

1.2. Pour une clause de la forme :

A

le terme qui représente le littéral A occupe la position 1.

1.3. Pour une clause de la forme :

<- $B_1,\ldots,Bn$

le terme qui représente le littéral B_i i ε [1,n] occupe la position i.

2. Les valeurs des variables des termes stockés dans les processus M-CLAUSE sont stockés dans le processus FRAME.

Chaque processus du sous-réseau MEMORY est identifié par un entier naturel.

L'**adresse** d'un terme dans le sous-réseau MEMORY est une 2-uplet : <d,π> où d ε Nat et note l'identification d'un processus du sous-réseau MEMORY (M-CLAUSE ou FRAME) et π ε Nat* est la position que le terme occupe dans le processus.

La **position** que le terme occupe dans un processus est notée par π, π_1,... et elle est définie de la manière suivante :

- La position de la racine du terme est un naturel.
- Si le terme $F(t_1,\ldots,t_n)$ occupe la position π, le terme t_i occupe la position π.i.

Un terme t est représenté (rep(t)) dans les processus M-CLAUSE et FRAME avec les conventions suivantes :

- rep(c) (où c est une constante) est c. Les constantes peuvent apparaître dans les processus M-CLAUSE et dans les processus FRAME.
- rep(v) (où v est une variable) est
  - dans les processus M-CLAUSE : v
  - dans le processus FRAME :

    Si v est non liée : undef

    Si v est liée : ad(d,π) où ad(d,π) est l'adresse du terme auquel la variable est liée.

- rep($F(t_1,\ldots,t_n)$) (où F est soit un symbole de fonction, soit un symbole de prédicat et $t_1,\ldots,t_n$ sont des termes) est $F(rep(t_1),\ldots,rep(t_n))$. Les termes sans variables peuvent apparaître dans les processus M-CLAUSE et dans les processus FRAME, les termes avec variables peuvent apparaître seulement dans les processus M-CLAUSE, la valeur de ses variables apparaîtront dans les processus FRAME.

Exemple 3.4 :

Supposons que nous ayons le programme suivant :

CL1 : P(X,Y) <- Q(X,Z), R(G(X,Y))

CL2 : <- P(F(b),W)

La clause CL2 est stockée dans la mémoire de la manière suivante :

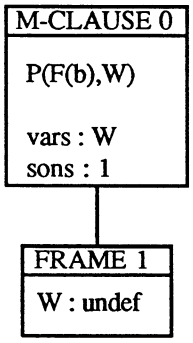


Figure 3.49

Le littéral P(F(b),W) est stocké dans la position 1 du processus M-CLAUSE identifié avec le naturel 0, il est donc à l'adresse <0,1>. Le terme F(b) se trouve à l'adresse <0,1.1>, la constante "b" se trouve à l'adresse <0,1.1.1> et la variable W qui apparait dans le processus M-CLAUSE et dans le processus FRAME, est considerée comme étant à l'adresse <1,1>.

La clause CL1 est stockée dans la mémoire de la manière suivante :

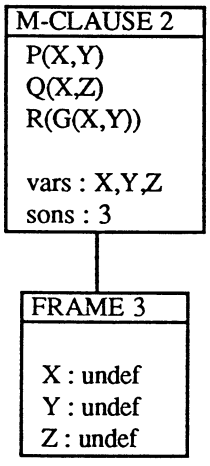


Figure 3.50

Lorsque les littéraux P(F(b)) de CL2 et P(X,Y) de CL1 s'unifient la configuration de la mémoire devient :

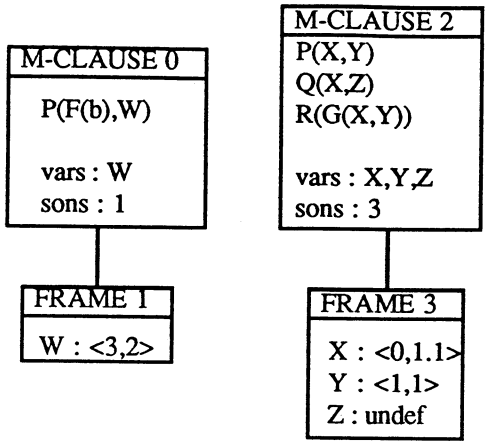


Figure 3.51

Fin de l'exemple

3.2.2. L'accès aux termes stockés dans la mémoire.

Les processus M-CLAUSE et FRAME se chargent de stocker et de manipuler les termes du programme.

Un processus M-CLAUSE peut être père de plusieurs processus FRAME. Un processus M-CLAUSE associé à chacun de ses processus FRAME fils forme une solution de la clause dont ce couple de processus contient les termes.

L'opération la plus importante à réaliser sur les termes d'une clause est l'unification. Lorsqu'un terme doit intervenir dans une opération d'unification, les processus M-CLAUSE et FRAME qui stockent le terme doivent d'abord reconstituer le terme en prenant la partie fixe du processus M-CLAUSE et la valeur de ses variables dans le processus FRAME correspondant puis cooordonner les actions nécessaires pour faire l'unification. Etant donné qu'un processus M-CLAUSE participe en général à solutions et qu'il intervient dans la coordinnation de l'unification de ses termes, lorsqu'un processus M-CLAUSE commande l'unification d'un de ses termes, l'unification d'autres termes du même M-CLAUSE reste en attente.

Cependant le parallélisme OU établit que plusieurs solutions d'un même but peuvent être formées en parallèle : cela signifie que plusieurs termes associés à un même processus M-CLAUSE doivent intervenir dans des opérations d'unification.

Un problème de séquentialité se pose, il est dû à deux faits :

- Plusieurs solutions partagent un même processus M-CLAUSE.

- Plusieurs manipulations sur les termes doivent être faites à la fois pour un même processus M-CLAUSE.

Nous proposons comme solution à ce problème de sequentialité que les processus FRAME gardent le contrôle sur la manipulation des termes. Pour n'importe quel terme t d'une clause dont la partie fixe se trouve dans un processus M-CLAUSE M, dont la valeur des variables se trouve dans un processus FRAME F, c'est F qui se charge de la manipulation de t. Lorsque t doit être manipulé les actions suivantes se produisent :

- Le processus FRAME F demande le terme t au processus M-CLAUSE M;
- M-CLAUSE M envoie t au FRAME F ;
- FRAME F active un nouveau processus FRAME F' et t est stocké dans FRAME F'.

Toute opération qui doit être faite sur le terme t sera commandée par FRAME F'.

3.2.3. L'unification des termes dans les programmes logiques.

Dans notre machine, les termes d'un programme logique sont rangés dans deux types de processus : M-CLAUSE et FRAME. L'objectif de cette section est de montrer comment les processus FRAME se chargent d'effectuer l'unification des termes des programmes logiques en utilisant le mécanisme de communication des processus FP2.

Exemple 3.5 :

Supposons que nous ayons un programme qui contient les clauses :

CL1 : P (X) <- Q (F(a,X)), R (X,Z).

CL2 : Q (F(Y,F(a,a))).

Nous proposons d'avoir deux processus FP2 : P et Q tels que comme résultat de leur interaction (par le mécanisme de communication de processus de FP2) le but Q (F(a,X)) dans CL1 soit satisfait. Les processus P et Q ne correspondent à aucun type de processus de notre machine à inférence parallèle, ils vont simplement nous permettre d'introduire une technique de programmation en FP2 pour réaliser l'unification des termes d'un programme logique en utilisant le mécanisme de communication de processus en FP2. Nous ne montrerons que les parties de P et Q qui nous intéressent.

proc P

:

S : Q1 (F(a,X)) => T (X)

T (X) : R (X,Z) => U (X,Z)

:

end

proc Q

:

V : Q2 (F(Y,F(a,a))) => W

:

:

end

Construisons un nouveau processus R en composant les processus P et Q:

$$P \,||\, Q \mathrel{++} Q1.Q2$$

Comme résultat de la communication entre P et Q à travers Q1.Q2, les termes F(a,X) et F(Y,F(a,a)) unifient pour former le terme F(a,F(a,a)) et la variable X est liée au terme "F(a,a)", ainsi le but Q(F(a,X)) est satisfait et le processus P peut essayer de résoudre le but R (X,Z). Si les termes n'étaient pas unifiables, P et Q sont peut être restés bloqués. Nous nous occuperons de ce problème plus tard.

Notre situation est légèrement différente parce que les termes à unifier se trouvent, en général, dispersés sur plusieurs processus.

Dans la figure ci-dessous, nous montrons des processus du sous-réseau MEMORY avec les termes que ces processus contiennent. Les processus sont représentés par rectangles et les termes par de triangles. Lorsqu'une des variables d'un terme t est liée à un terme t', le triangle qui représente t dans la figure contient l'adresse où se trouve t'.

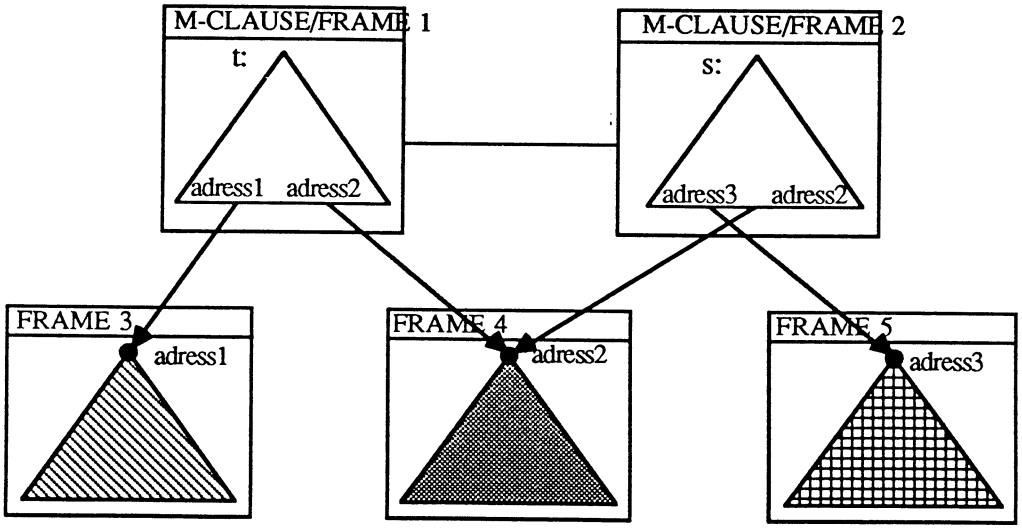


Figure 3.52

Les adresses telles que adress1, adress2 ou adrees3 ont la forme : ad(p,π), (p ε Nat, π ε Nat^+) où p indique l'identification d'un processus de la mémoire et π la position du terme dans un tel processus. Ces adresses sont donc des termes constantes pour FP2 et l'unification de t et s échoue même si t et s de la figure ci-dessous sont unifiables :

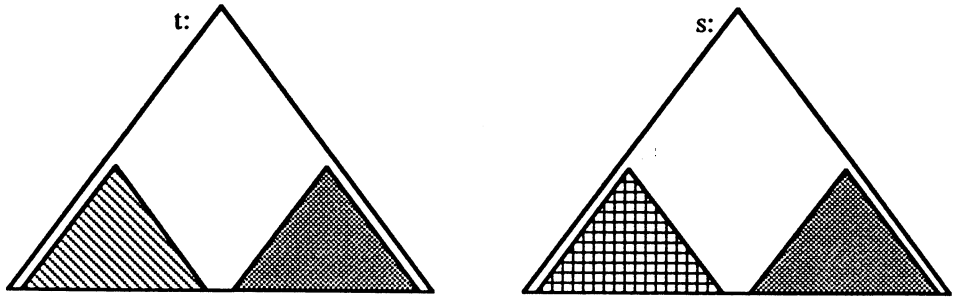


Figure 3.53

Nos objectifs sont donc les suivants :

• Profiter du mécanisme de communication des processus FP2 pour unifier les termes du programme.

• Maintenir la représentation des termes définie plus haut.

Pour y parvenir, l'unification de deux termes t et s est effectuée comme suit :

• Le processus FRAME F1 (FRAME F2) construit un terme t'(s') à partir de t(s) en remplaçant les adresses de la forme : ad(p,π) par variables des FP2 qui n'apparaissent pas dans les termes t(s). Ces variables que l'on introduit sont nommées : "**var-adress**".

• FRAME F1 et FRAME F2 se synchronisent pour unifier t' et s'.

• FRAME F1 (FRAME F2) continue l'unification avec les termes dont les var-adress ont été liées. Pour savoir où sont ces termes, FRAME F1 (FRAME F2) examine les positions de t' et s' où les var-adress ont été introduites.

Pour notre exemple, P1 et P2 peuvent unifier t' et s' en utilisant la communication de FP2, mais pour garantir que l'unification de t et s réussit, il faut encore vérifier que :

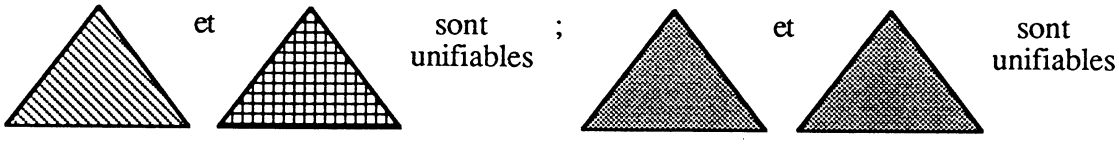


Figure 3.54

Comme conséquence de l'unification de t' et s', une var-adress peut être liée à:

i) Une variable non liée du terme : var-nonliée ;

ii) Une variable introduite pour nous : var-adress, qui substitue l'adresse à laquelle une variable du terme original a été liée ;

iii) Un terme différent d'une variable.

Supposons que les termes t et s sont aux les positions π_1 et π_2 des processus P et Q respectivement. Supposons, sans perte de généralité, que la var-adress appartienne à t et qu'elle soit à la position $\pi_1.\pi$ dans P. Cette variable est donc liée au terme qui est à la position $\pi_2.\pi$ dans Q.

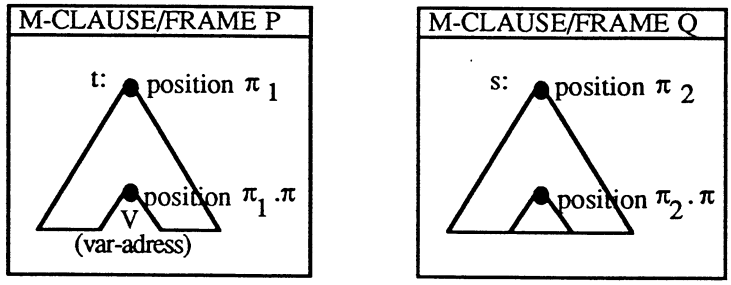


Figure 3.55

Dans le cas i), la var-nonliée qui se trouve dans la position $\pi_2.\pi$ est liée à l'adresse :

$$< \text{Id-processusP}, \pi_1.\pi >.$$

Dans le cas ii), l'unification de t et s réussit si et seulement si l'unification des termes qui se trouvent dans les adresses représentées par les deux var-adress réussit.

Dans le cas iii), l'unification de t et s réussit si et seulement si l'unification des termes qui se trouvent dans:

- l'adresse indiquée par la var-adress V (position $\pi_1.\pi$ de P) ;

- l'adresse $<$ Id-processusQ, $\pi_2.\pi >$

réussit.

Avant de présenter comment l'unification est réalisée par les processus FRAME, nous présentons de quelle manière les processus FRAME et M-CLAUSE sont liés entre eux.

• Tous les processus FRAME sont liés entre eux.

• Chaque processus M-CLAUSE est lié à tous les processus FRAME.

La figure suivante montre le sous-ensemble de connexions entre les processus M-CLAUSE et FRAME qui permettent les communications pour l'unification.

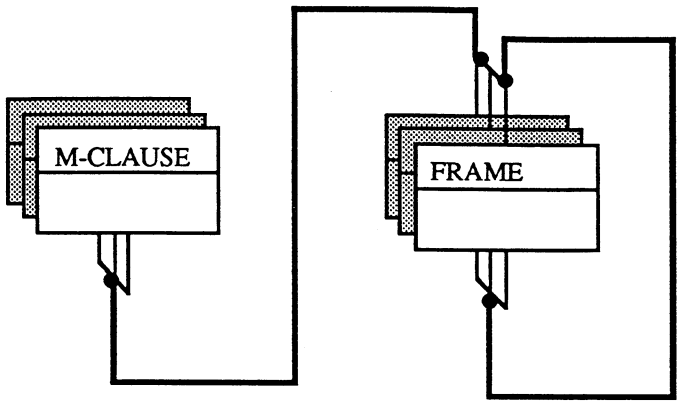


Figure 3.56

Un FRAME réalise l'opération d'unification lorsqu'il reçoit l'information suivante:

- L'identification du processus qui demande l'unification (Id-father) ;
- L'adresse du terme t à unifier: <F,π_1> ;
- L'adresse du terme s avec lequel FRAME F doit unifier le terme t : <F',π_2>.

Deux cas peuvent se présenter :

CAS 1 : La racine du terme est dans le processus M-CLAUSE identifié avec M, le processus FRAME F n'a que les valeurs des variables du terme ;

- FRAME F active un nouveau FRAME F".
- FRAME F établit la communication avec M-CLAUSE M en lui demandant d'envoyer le terme stocké dans la position π_1 de FRAME F".
- Le processus M-CLAUSE M construit un nouveau terme à partir du terme rangé à la position π_1 de M où chaque variable non liée est remplacée par une adresse qui indique la position à laquelle se trouve la valeur de la variable dans FRAME F. M envoie à F" ce nouveau terme.
- F" reçoit le terme dans sa position π_1 (adresse du terme <F",π_1>).
- FRAME F envoie au FRAME F" l'information suivante :
  - Id-father ;
  - L'adresse du terme à unifier : <F",π_1> ;
  - L'adresse du terme s avec qui FRAME F" doit unifier le terme t : <F',π_2>.

Maintenant, FRAME F" a la racine du terme à unifier. F" est donc dans le CAS 2 :

CAS 2 : La racine du terme se trouve dans FRAME F.

L'unification se fait en trois étapes:

- L'étape de préparation ;
- L'étape de communication ;
- L'étape d'analyse des résultats.

• L'étape de préparation

FRAME F construit un terme t' à partir de t où les sous-termes qui représentent les adresses en t (sous-termes de la forme : ad(p,π)) sont substitués par des variables FP2 (var-adress) notées par : V1, V2 ....

Exemple 3.7 :

Soit t = F (X,ad(1,1),G (Y,X,ad(1,2.3))).

Le terme que l'on construit à partir de t est : t' = F(X,V1,G (Y,X,V2)).

Fin de l'exemple

FRAME F construit une liste l où sont rangés les positions de t' où il y a soit des variables du terme t (exemple : X, Y), soit des variables var-adress introduites dans le pas précédent. Là où il y a des var-adress, on range aussi l'adresse representée par ces var-adress.

La liste l est formée par des éléments de la forme :

$$< \text{Position}^{+}, \text{Valeur} >$$

où "valeur" est 0 lorsque la variable de t' représentée dans l est du type var-nonliée et "valeur" est une position lorsque la variable de t' représentée dans l est du type var-adress.

Pour notre exemple, la liste l est :

l = [< [1.1, 1.3.2], 0 >, < [1.2], <1,1> >, < [1.3.1], 0 >, < [1.3.3], <1,2.3> >]

Le premier élément de l nous indique que la variable X du type var-nonliée occupe les positions 1.1 et 1.3.2 dans le terme t.

Le dernier élément de l nous indique que dans la position 1.3.3 il y a une var-adress (V2) qui représente l'adress : <1,2.3>.

• L'étape de communication

Supposons que le FRAME F doive unifier son terme t avec le terme s du FRAME F'. Supposons, sans perte de généralité, l'identification du FRAME F < l'identification du FRAME F'. Le FRAME F a construit t' et lt à partir de t et il se trouve prêt à communiquer avec F'.

De manière analogue, le FRAME F' a construit s' et ls à partir de s et il se trouve prêt à communiquer avec F.

La synchronisation de F et F' se réalise de la manière suivante :

• F envoie comme message : <F,F',t',lt,ls>. Les deux premiers paramètres permettent que F communique avec F' et pas avec un autre FRAME, ls est une variable FP2 non liée où F aura, comme conséquence de la communication, la liste qui indique où sont les variables de s.

• De manière analogue, F' envoie comme message : <F,F',s',lt,ls>

Lorsque la communication se réalise, t' et s' sont unifiés à un même terme, F (F') reçoit la liste ls (lt) construite par F' (F).

Exemple 3.8 :

Soient t' = F (X,G(a),a) et s' = F(Y,V1,Y)

lt = [< [1.1],0 >] et ls = [< [1.1,1.3],0 >, <[1.2], <fr,pos> >]

Comme résultat de l'unification,

s' = t' = F(a,G(a),a)

Fin de l'exemple

• L'étape d'analyse des résultats de l'unification

Les listes lt et ls stockent les adresses des var-nonliée et var-adress en t' et s'. Après l'unification, on regarde dans ces positions de t' et s' et on obtient les termes dont les variables ont été liées.

Une var-nonliée ou une var-adress peut avoir été liée à un des types de termes suivants:

- var-adress
- var-nonliée
- terme constant
- terme non constant et différent d'une variable

CAS 1 : Une var-adress est liée à :

- une autre var-adress,
- un terme constant ou
- un terme non constant mais différent d'une var-adress, différent d'une var-non liée

le résultat de l'unification dépend du fait que soient unifiables:

- le terme qui se trouve dans l'adresse représentée par l'var-adress

et

-le terme dont l'var-adress a été liée.

Lorsqu'une var-adress est liée à une var-adress d'un autre FRAME, le FRAME dont son identification est plus grand, envoie un message à l'autre FRAME en lui demandant que soit faite l'unification des termes qui se trouvent dans les positions indiquées par les var-adress.

CAS 2 : Une var-nonliée est liée à une var-adress

Le terme t (ou s) correspondant doit substituer la variable par l'adresse représentée par l'var-adress.

CAS 3 : Une var-nonliée est liée à un terme constant

Le terme t (ou s) correspondant doit substituer la variable par le terme constant.

CAS 4 : Une var-nonliée est liée à une autre var-nonliée

Supposons que l'identification de FRAME F > L'identification de FRAME F'.

Dans le terme t (ou s) du FRAME F', ces variables sont substituées par les adresses des variables correspondants du FRAME F.

CAS 5 : Une variable normale est liée à un terme non constant

Supposons que l'identification de FRAME F > L'identification de FRAME F'.

Le FRAME F choisit la position du terme t (ou s) où le terme non constant sera stocké. Dans les autres positions où apparaît la var-nonliée, sa valeur est substituée par l'adresse choisie.

3.3. Conclusion

Le modèle présenté construit un réseau de processus pour chaque programme écrit dans le langage des clauses de Horn. Chacun de ces réseaux reflète la structure syntaxique du programme que le réseau représente. En effet, le processus qui représente le but L du corps d'une clause est lié à un processus OR qui représente les clauses qui ont L comme tête. Lorsqu'un but L du corps d'une clause doit être satisfait, pour activer les clauses qui peuvent résoudre L, il suffit d'envoyer un message du processus qui représente L vers le processus OR auquel il est lié. Nous n'avons donc pas besoin de chercher ces clauses dans des bases de données,comme il est nécessaire dans les modèles séquentiels classiques, tout est préparé par les connexions du réseau.

On travaille avec des réseaux qui ont un grand nombre de processus et où chaque processus ne realise que peu d'activité. En général, les modèles qui ont cette caractéristique dépensent beaucoup de temps en communication. Pour minimiser le nombre de communications dans notre modèle, la partie de la machine qui manipule les termes se charge des communications "lourdes" : envoyer et recevoir des termes et le reste de la machine communique des messages plus petits : l'identification des processus.

Dans les modèles de fine granularité peu d'efforts ont été faits pour une bonne gestion de la mémoire. Notre modèle a des processus spéciaux qui s'occupent de la gestion de la mémoire :

- Des processus qui stockent la partie constante des termes d'une clause (M-CLAUSE) ;
- Des processus qui stockent la valeur des variables des termes d'une clause (FRAME.)

L'information d'un processus M-CLAUSE est partagée par plusieurs processus FRAME.

Pour le problème de gestion de la mémoire, notre modèle donne une solution intermédiaire entre les modèles de fine granularité fine et ceux de grosse granularité.

L'unification des termes est réalisée par les processus FRAME en utilisant le mécanisme de communication entre processus de FP2. Etant donné que les termes se trouvent dispersés dans le réseau, il n'est pas possible, en général, de réaliser l'unification avec une seule communication et il faut dépenser une certaine quantité de temps à préparer la communication et à analyser le résultat de l'unification. Cette analyse dit si de nouvelles unifications sont nécessaires. Cette mise en oeuvre de l'unification permet de réaliser moins de communications entre les processus.

CHAPITRE 4

Vers une Machine à Inférence Parallèlle pour la Logique de Premier ordre baseé sur la Méthode de Connexion

VERS UNE MACHINE A INFÉRENCE PARALLELE POUR LA LOGIQUE DE PREMIER ORDRE BASÉE SUR LA MÉTHODE DE CONNEXION

Nous avons vu (chapitre 1) que lorsque l'on utilise comme règle d'inférence la résolution, la preuve d'une formule consiste à :

• Choisir un littéral L' du résolvant courrant.

• Choisir un littéral L pour satisfaire L'. L peut appartenir à un des axiomes ou à un résolvant précédent.

Etant donné que dans le langage des clauses de Horn les clauses ont au plus un littéral positif, les résolvants sont formés par des littéraux négatifs. Les littéraux L se trouvent donc toujours dans l'ensemble des axiomes.

Lorsque l'on travaille avec toute la logique du premier ordre, l'espace de recherche grandit car les littéraux L peuvent se trouver soit dans les axiomes, soit dans les résolvants formés précédemment.

Dans la Methode de Connexion (voir chapitre 1, section 1.8), un Spanning Set nous donne un ensemble de connexions de la matrice qui permet de prouver la formule (si cela c'est possible). En travaillant avec un sous-ensemble de connexions de la matrice, l'espace de recherche est réduit.

L'objectif principal de ce chapitre est de donner une méthode pour le calcul des Spanning Sets.

4.1. Aperçu historique

L'utilisation de la méthode de connexion comme règle d'inférence pour les machines d'inférence est récente et c'est dans le cadre du projet ESPRIT 415-F qu'ont été fait la plupart des travaux sur ce sujet. Cependant, hors du projet ESPRIT, G.V. Wilson a fait l'implémentation séquentielle d'un démonstrateur de théorèmes pour la logique modale S5 basée

sur la méthode de connexion [Wil86] ; S.H. Lavington, Y.J. Jiang et M. Azmoodeh travaillent aussi sur une implémentation parallèle de la méthode [LJA88].

Quelques modèles de machines à inférence parallèles basés sur la méthode de connexion ont été proposés par le groupe d'Intelligence Artificielle de L'Université Technologique de Munich et par P. Jorrand au LIFIA. Ces modèles appartiennent à l'une des catégories suivantes :

- Modèles Top-down.
- Modèles Bottom-up.

Le modèle top-down qui a été proposé [Asp86] pour le langage des clauses de Horn simule l'arbre de dérivation AND-OR des formules.

Deux modèles bottom-up apparaissent dans [AsB85] et [Jor87], ces modèles travaillent sur le calcul propositionnel. L'idée consiste à construire tous les chemins de longueur 1,2,...n ("n" est le nombre de clauses de la formule) qui ne sont pas complémentaires. La formule est valide si et seulement si la machine ne peut pas construire un chemin de longueur "n" qui n'est pas complémentaire.

L'architecture proposée par K. Aspetsberger et S. Bayerl est un "pipeline" de processus:

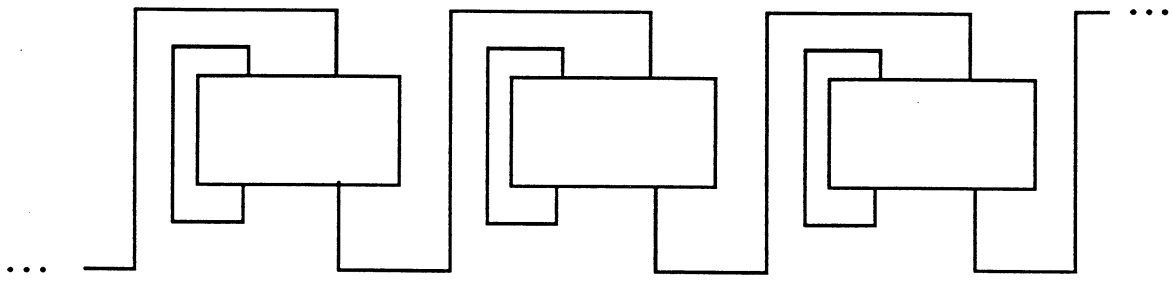


Figure 4.1

Ph. Jorrand propose comme architecture un tableau systolique triangulaire :

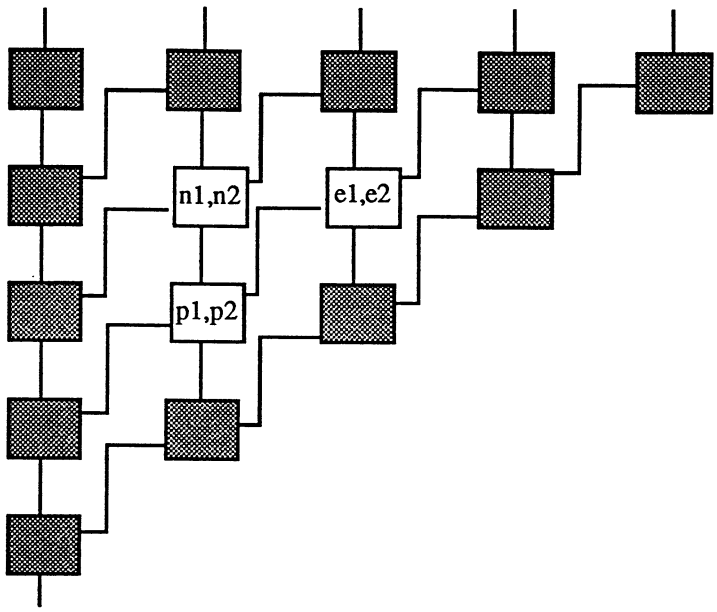


Figure 4.2

Le modèle proposé par K. Aspetsberger et S. Bayerl a été étendu aux clauses de Horn par K. Aspetsberger dans [Asp86].

Ces modèles donnent un premier pas vers l'utilisation de la méthode de connexion pour la preuve de théorèmes. Cependant, le fait d'examiner les chemins de la matrice exhaustivement rend ces modèles inefficaces.

W. Bibel, F. Kurfeβ et d'autres proposent un modèle [BiK87] qui utilise le concept de Spanning Sets. Le modèle travaille pour les clauses de Horn et il est divisé en trois étapes :

- L'étape de compilation ;
- L'étape préparatoire ;
- L'étape principale.

Dans l'étape de compilation se fait le calcul des connexions de la formule (ici n'intervient pas l'unification) et se construit un DAG à partir de la formule et des connexions calculées.

L'étape préparatoire consiste à intégrer les requêtes du programme dans la représentation que l'on a de la formule.

Dans l'étape principale se fait le calcul des spanning sets de manière top-down et se vérifie l'unificabilité des spanning sets.

La distinction logique de ces trois étapes n'implique pas le fait qu'elles ne puissent pas être superposées dans le temps.

Exemple 4.1 :

Soit le programme suivant :

FATHER (F(U), U).

GRANDFATHER (X, Y) <- FATHER (X, Z), FATHER (Z, Y).

<- GRANDFATHER (V, W).

La matrice de connexions associée est :

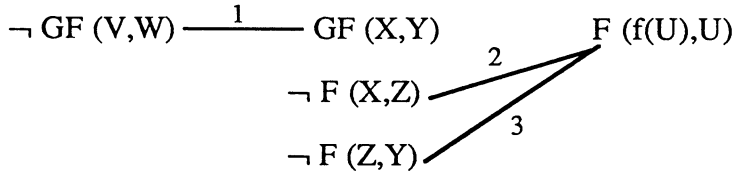


Figure 4.3

Fin de l'exemple

La figure 4.4 montre un DAG qui est la représentation des connexions de la matrice de la figure 4.3. Le sous-DAG, dont la racine est le symbole moins (-) contient des littéraux négatifs. De manière analogue, le sous-DAG, dont la racine est le symbole plus (+), a des littéraux positifs. Les littéraux touchés par la n-ième connexion sont les n-ièmes descendants immédiats des deux sous-DAG's.

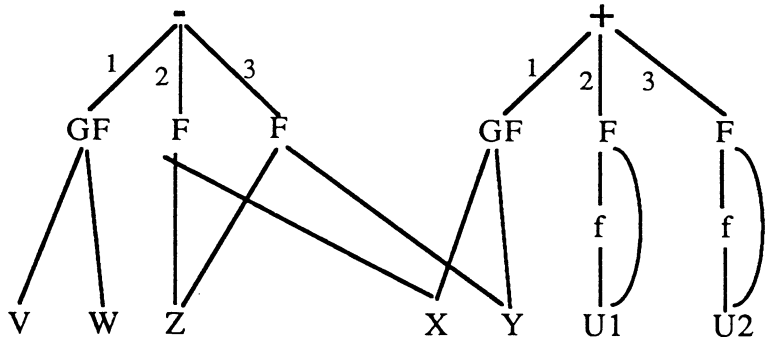


Figure 4.4

4.2. Le modèle proposé

Un premier pas vers la construction de la machine pour la logique du premier ordre est de scinder le problème en trois étapes et de construire un sous-réseau de processus pour chacune de ces étapes. Les trois étapes que l'on propose sont :

- Obtenir les connexions de la matrice ;
- Calculer les candidats Spanning Sets ;
- Vérifier que les candidats Spanning Sets obtenus dans l'étape précédente sont des véritables Spanning Sets.

Chacune de ces trois étapes peut être réalisée en parallèle. Une fois que l'on sait comment calculer les candidats à Spanning Sets et comment ils doivent être utilisés pour la preuve de la formule, nous pouvons envisager de réaliser ces deux étapes en parallèle.

Afin d'obtenir les connexions de la matrice en parallèle, nous proposons un réseau de processus où chaque processus représente un littéral de la formule et où chaque processus est lié à tous les autres processus de ce réseau.

L'information stockée par un processus est :

- Le signe du littéral qu'il représente ;
- Le nom du littéral ;
- Les coordonnées du littéral (numéro de la clause à laquelle le littéral appartient et position occupée par le littéral dans la clause).

Pour la formule :

$$A \vee \neg A \vee \neg A \wedge B \vee \neg B$$

le réseau est :

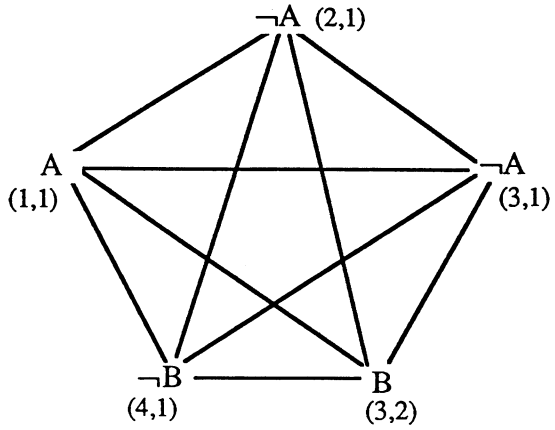


Figure 4.5

La prochaine étape consiste à calculer les Spanning Sets. Ce chapitre est consacré au développement d'une méthode pour procéder à ce calcul. Nous développons la méthode proposée, nous la décrivons formellement avec un DAG. A partir du DAG, nous développons la machine parallèle. Enfin, nous prouvons que chaque DAG ainsi construit à partir d'une machine de connexions M représente un candidat Spanning Set (voir paragraphe ci-dessous) de M. La méthode permet de calculer tous les candidats Spanning Sets de M.

Il faut préciser maintenant ce que nous appelons un candidat à Spanning Set. Un candidat à Spanning Set est un ensemble de connexions qui satisfait la définition de Spanning Set lorsque l'on travaille sur une formule sans variable, c'est-à-dire lorsque l'on ne cherche pas à trouver l'unificateur le plus général des littéraux touchés par les connexions du Spanning Set.

Par exemple, soit la formule suivante :

$$\neg P(F(F(a))) \vee \exists Y\,(P(Y) \wedge \neg Q(Y)) \vee \exists X\,(Q(F(X)) \wedge \neg P(X)) \vee Q(a)$$

Les candidats Spanning Set peuvent être calculés sur la matrice de connexions suivante :

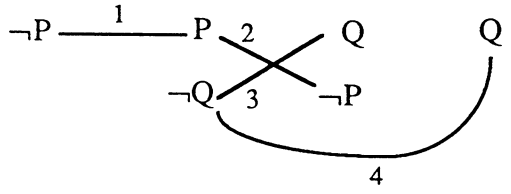


Figure 4.6

et ils sont : {1,4} et {1,2,3,4}.

Cependant, ni {1,4} ni {1,2,3,4} ne sont pas des Spanning Sets pour la formule. Si nous réécrivons la matrice comme suit :

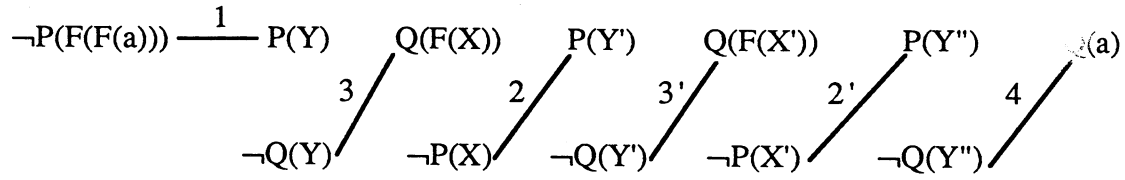


Figure 4.7

alors, {1,3,2,3',2',4} est un Spanning Set (Spanning Set unifiable).

En utilisant deux fois les connexions 2 et 3 du candidat à Spanning Set nous arrivons à prouver la formule.

Par abus de langage, nous parlerons par la suite de "Spanning Set" à la place de "candidat à Spanning Set".

L'utilisation des candidats à Spanning Sets pour la preuve des formules ne sera pas traitée dans cette thèse.

4.2.1. Méthode de calcul des Spanning Sets

Nous allons tout d'abord présenter la méthode proposée à l'aide d'un exemple.

Soit la matrice de connexions suivante:

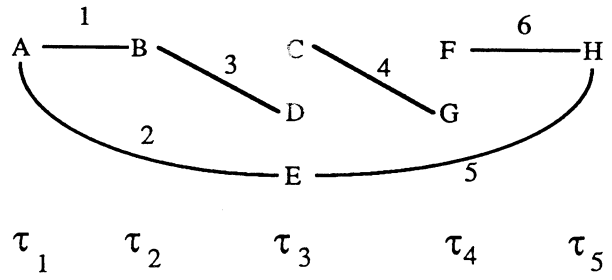


Figure 4.8

Selon la définition de Spanning Set, nous voulons trouver un ensemble de connexions S de la matrice M. Tout chemin de la matrice doit avoir au moins deux littéraux liés par une connexion de S.

Prenons une clause quelconque, par exemple $\tau 4$. Si l'on réunit tous les chemins qui passent par le littéral F (P_F) avec ceux qui passent par le littéral G (P_G), on aura tous les chemins de la matrice. Notre problème peut donc se résumer ainsi:

Nous voulons trouver un ensemble de connexions S de la matrice M. Tous les chemins de P_F et tous ceux de P_G doivent avoir deux littéraux liés par une connexion de S.

Tout chemin passant par "F" a la forme: $X_1X_2X_3FH$[1]. Si l'on ajoute la connexion 6 à S, tous les chemins P_F ont deux littéraux - F et H - liés par une connexion de S -6-.

Les chemins qui passent par "G" ont la forme:

$$X_1X_2CGX_5$$

ou $X_1X_2DGX_5$

ou $X_1X_2EGX_5$

ceux du premier groupe sont complémentaires grâce à la connexion 4, donc S = { 4,6 }.

Les littéraux "D" et "G" ne sont pas complémentaires, les littéraux "E" et "G" non plus. On peut donc poser le problème de la façon suivante :

Trouver un ensemble de connexions S' tel que tous les chemins P_D et tous les chemins P_E aient au moins deux littéraux liés par une connexion de S'.

Tous les chemins qui passent par "D" passent aussi par "B". "B" et "D" sont complémentaires (ils sont liés par la connexion 3). Donc, si l'on ajoute la connexion 3 à S, tout chemin passant par "D" aura deux littéraux liés par une connexion de S.

Le lecteur remarquera que tous les chemins qui passent par "E" sont complémentaires soit par la connexion 2, soit par la connexion 5.

Nous avons donc deux Spanning Sets possibles:

{ 2,3,4,6 } et { 3,4,5,6 }

Le Spanning Set obtenu dépend de la clause choisie : si l'on prend la clause τ_2, on obtiendra les ensembles de connexions suivants: { 1 }, { 2,3,4,6 } et { 3,4,5,6 }.

Pour être sûr d'avoir trouvé tous les Spanning Sets possibles, il est nécessaire d'appliquer la méthode sur toutes les clauses de la matrice.

4.2.2. Le DAG: une structure qui permet de formaliser la méthode

Nous avons besoin d'une structure formelle pour pouvoir spécifier notre méthode. Voyons comment la méthode peut être décrite à l'aide d'un arbre.

---

[1] $X_1 \varepsilon \tau_1$, $X_2 \varepsilon \tau_2$, $X_3 \varepsilon \tau_3$.

Par exemple, si la méthode commence à s'appliquer sur la clause τ_4 de l'exemple précédent les arbres qui représentent les Spanning Sets sont:

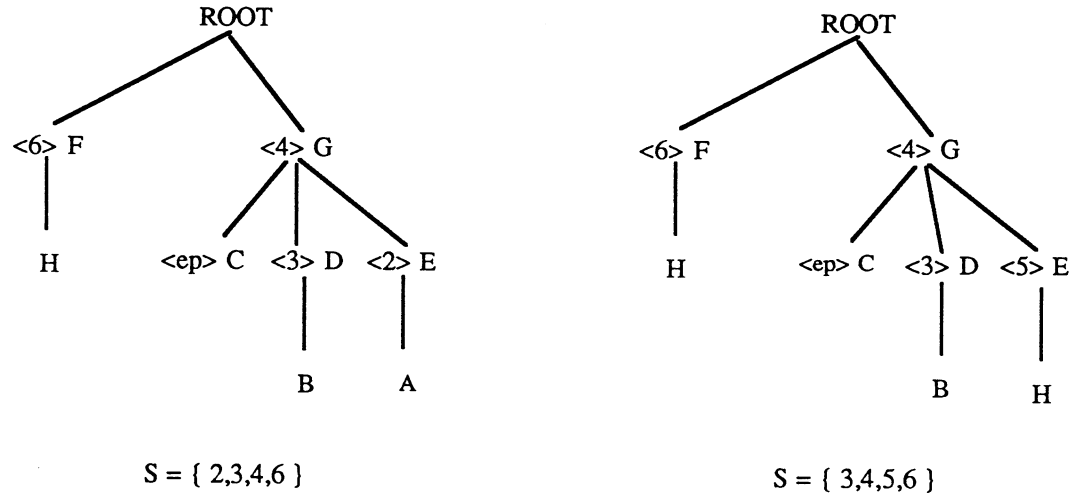


Figure 4.9

Dans ces arbres apparaissent trois types de nœuds:

1. Nœud type **root**

   C'est la racine de l'arbre. Sa forme générale est: **root**. Ses fils sont les littéraux de la clause où la méthode commence à s'appliquer. La racine peut avoir l'étiquette "satisfaid" lorsque tous ses fils ont été satisfaits.

2. Nœud type **leaf**

   Ce sont les feuilles de l'arbre, ils sont la forme: **L** où L est un littéral. Lorsque une feuille est introduite dans l'arbre, cela signifie que tous les chemins qui passent par son nœud père sont complémentaires.

3. Nœuds type **intern**

   Ils sont de la forme: **L** ou L est un littéral. Si $L_1,\ldots,L_n$ sont les fils d'un noeud interne cela signifie que dans la matrice il y a deux clauses de la forme:

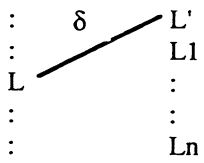


Figure 4.10

" L " et " L' " sont deux littéraux complémentaires et dans la matrice, ce fait est indiqué par la connexion "δ".

Un nœud interne L peut avoir trois étiquettes:

- δ Un nœud interne est toujours étiqueté avec la connexion qui lui permette d'être satisfait.

- "entry-point" Lorsque δ = < père(L), L >

- "satisfied" Si le nœud est déjà satisfait.

Pour éviter la répétition du travail, on construit des DAGs[1] au lieu de construire des arbres.

Soit la matrice de connexions suivante:

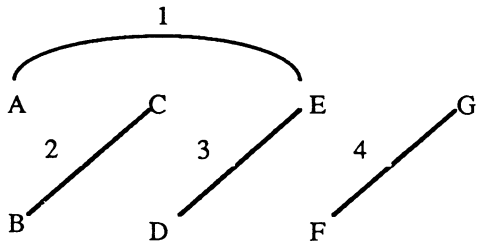


Figure 4.11

Lorsque la méthode s'applique sur la deuxième clause de la matrice, le DAG qui en résulte est:

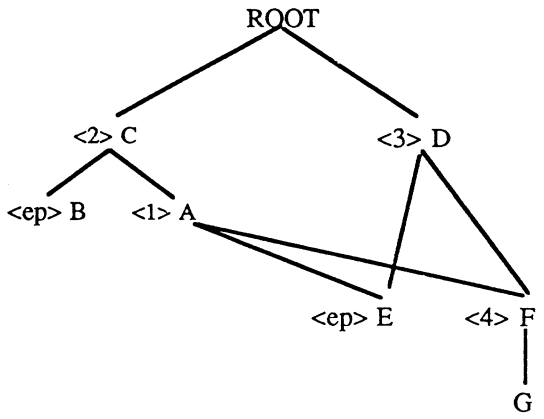


Figure 4.12

4.2.3. La construction du DAG à partir d'une matrice de connexions

On va calculer les Spannings sets d'une matrice de connexions en nous aidant des DAGs. L'idée de l'algorithme est la suivante:

---

[1] *Direct Acyclic Graph*

L'algorithme construit un Dag pour chaque clause de la matrice - fonction Compute-Spanning-Sets - ;

Les actions à réaliser sur chaque Dags sont:

- Choisir un littéral L à satisfaire (et la connexion δ avec laquelle L sera satisfait) - fonction css -.
- Ajouter la connexion δ au Dag et ajouter les nouveaux littéraux à satisfaire comme fils de L - fonction put-in -.
- Déterminer quels littéraux ont été satisfaits avec la connexion δ - fonction satisfy-dag -.

Ces actions se répètent jusqu'à trouver un Spanning Set - fonction css - ou jusqu'à détecter l'impossibilité de construire un Spanning Set à partir du Dag donné - fonctions css et put-in -.

L ' a l g o r i t h m e

V a r i a b l e s

M	Matrice de connexions.
D	Dag.
i,m,k_i	Nat
INTERN	intern
LEAF	leaf
ROOT	root
X, X_1,...,X_n	intern | leaf | root
L,L',L_{ij}	Literal
δ	Connection
τ,τ'	Clause

On donne ensuite les fonctions principales qui décrivent la méthode proposée, (les fonctions secondaires se trouvent dans l'annexe1).

Les fonctions principales

Compute-Spanning-Sets: Connection-Matrix – { Seq-Connexion }$^+$ | fail

A partir d'une matrice de connexions, la fonction donne les spanning sets de la matrice si cela est possible, sinon le résultat est "fail".

Si la matrice de connexions a "m" clauses, alors "m" Dags sont construits, (un Dag pour chaque clause).

Compute-Spanning-Sets (M) **is**

$$\frac{\text{number-of-clauses (M)} = m \wedge \forall i = 1...m\text{: clause-p (M,i)} = \langle L_{i1},...,L_{iki}\rangle}{\forall i = 1...m : \; ||\ \text{css (M, link-sons (ROOT,ROOT, } \langle L_{i1},...,L_{iki} \rangle \text{), <>,<>)}}$$

css: Connection-Matrix x Dag x Literal x Connection – { Seq-Connexion }$^+$ | fail

La fonction css ("Compute Spanning Sets") se charge de:

- *Décider la terminaison (avec succès ou échec) de la construction du Dag ;*
- *Continuer avec la construction du Dag:*

  en choisissant le littéral à satisfaire

  ou

  en invoquant la fonction put-in.

css(M, D, L, δ) **is**

Terminaison avec succès

$$\frac{\exists\ \text{ROOT} \in D\text{: is (D,ROOT,root)} \wedge \text{satisfied (D,ROOT)}}{\text{solution (D)}}$$

Terminaison avec échec

$$\frac{\exists\, ROOT \in D:\ is(D,ROOT,root) \wedge \text{not satisfied}(D,ROOT) \wedge \forall X \in D:\ is(D,X,intern) \wedge links(D,X) \neq \langle\rangle}{destroy(D)}$$

Il faut choisir un littéral du Dag et le satisfaire

$$\frac{L = \langle\rangle \wedge \exists X \in D:\ is(D,X,intern) \wedge links(D,X) = \langle\rangle}{\forall i = 1 \ldots card(incident(M,X)):\ ||\ css(M,D,X,i\text{-}esime(incident(M,X),i))}$$

Le littéral L sera satisfait avec la connexion δ.

$$\frac{L \neq \langle\rangle}{put\text{-}in(M,D,L,\delta)}$$

put-in: Connection-Matrix x Dag x Literal x Connection – { Seq-Connexion }$^+$ | fail

Le littéral "L" est satisfait avec la connexion "δ = <L,L'>"

put-in (M,D,L,< L,L'>) **is**

L' appartient à une clause unitaire

$$\frac{unit\text{-}clause(clause\text{-}of\text{-}literals(M, L'))}{satisfy\text{-}dag(M, link\text{-}sons(put\text{-}label(D,L,L'), L, \langle L'\rangle))}$$

L' appartient à une clause τ' qui n'est pas dans le Dag

$$\frac{assign(\tau', clause\text{-}of\text{-}literals(M, L')) \wedge \text{not } unit\text{-}clause(\tau') \wedge \text{not } in\text{-}dag(D, \tau')}{satisfy\text{-}dag(M, put\text{-}label(link\text{-}sons(put\text{-}label(D,L,L'), L, \tau'), L', \text{"entry-point"}))}$$

L' appartient à une clause τ' qui est déjà dans le Dag. L' est le point d'entrée.

assign (τ',clause-of-literals (M,L')) $\wedge$
not unit-clause (τ') $\wedge$ in-dag (D,τ') $\wedge$ entry-point (D, L')

---

satisfy-dag (M, link-sons (put-label (D,L,L'), L, τ'))

L' appartient à une clause τ' qui est déjà dans le Dag. L' n'est pas le point d'entrée. Ce cas sera considéré dans la section 4.3.2.

assign (τ',clause-of-literals (M,L')) $\wedge$
not unit-clause (τ') $\wedge$ in-dag (D,τ')$\wedge$ not entry-point (D, L')

---

fail

satisfy-dag: Connection-Matrix x Dag – { Seq-Connexion }$^+$ | fail

Cette fonction se charge d'étiqueter par "satisfied" les nœuds qui viennent d'être satisfaits.

satisfy-dag (M, D) **is**

$\forall$ X ε D: son (D, X) = <X_1,...,X_n> $\wedge$
$\forall$i = 1...n: ((is(D,X_i,intern) $\wedge$ satisfied(D,X_i)) v is(D,X_i,leaf))

---

css (M, put-label (D, X, "satisfied"), <>, <>)

Dans notre DAG, les littéraux d'une clause apparaissent tout au plus une fois. Si pour satisfaire un littéral L, nous avons besoin d'un littéral L' de la clause $\tau_j = L_{j1} \wedge \ldots \wedge L_{j\,t-1} \wedge L' \wedge L_{j\,t+1} \wedge \ldots \wedge L_{j\,kj}$ et si les littéraux de τ' n'apparaissent pas encore dans le DAG, alors les littéraux $L_{j1}, \ldots, L_{j\,t-1}, L_{j\,t+1}, \ldots, L_{j\,kj}$ seront inclus dans le DAG comme fils de L.

Le littéral par lequel τ' a été "appelé" peut être ou ne pas être L'. Dans le premier cas, le problème se résout en profitant des sous-DAGs: $L_{j1}, \ldots, L_{j\,t-1}, L_{j\,t+1}, \ldots, L_{j\,kj}$, c'est-à-dire que l'on peut factoriser. Dans le deuxième cas, il y a un sous-DAG qui n'apparaît pas (le sous-DAG qui correspond au précédent point d'entrée de τ': L"). Dans la section 4.3, on verra que le deuxième cas se produit lorsque notre graphe de connexions (qui doit être un DAG) a un ensemble de connexions que nous appellerons *cycle*. Le problème causé par les cycles est présenté dans la section 4.3, nous verrons dans la section 4.3.2 comment détecter l'apparition éventuelle des cycles dans notre graphe de connexions.

4.2.4. L'architecture proposée

Dans la section 4.2.2, nous avons décrit la structure qui permet de formaliser notre méthode. Cette structure est un Dag. Dans la section 4.2.3, nous avons présenté un algorithme pour la construction de nos Dags. L'objectif de cette section est d'aboutir à une architecture inspirée par nos Dags. Pour parvenir à cette architecture, il est important de souligner deux faits:

1. Chaque littéral apparaît tout au plus une fois dans un Dag ;

2. Les littéraux appartenant à une même clause ont toujours un père commun. Ce père est de type root ou intern.

Le premier fait nous permet d'estimer que les littéraux apparaissent une seule fois dans le sous-réseau qui représentera un Dag en particulier.

Le deuxième fait nous laisse penser qu'il est raisonnable de regrouper les littéraux en clauses.

Les liaisons de père aux fils dans le Dag, établissent une relation de dépendance qui permet de savoir:

1. Quels fils sont déjà actifs ;

2. Quels nœuds (littéraux) se satisfont chaque fois qu'une connexion est incluse dans le Dag.

Le premier fait est important car il nous permet de connaître les littéraux dont il faut tenir compte pour calculer le Spanning Set.

Le deuxième fait permet de déterminer quand on a trouvé une solution et quand est il impossible d'en avoir une.

L'architecture que l'on propose ne garde pas les relations père-fils, mais elle permet de savoir quels sont les littéraux à satisfaire, quand une solution est obtenue ou quand il est inutile de continuer la recherche.

L'architecture comprend un "master process" (Σ) qui peut communiquer avec ρ "supervisor processes" σ_i i= 1...ρ. Chaque "supervisor process" (σ_i) a α "worker processes" ω_{ij} j= 1...α ou α est le nombre de clauses de la formule. Le sous-réseau formé par un "supervisor process" σ_i et ses "worker processes": ω_{ij} sera nommé SUB-NETWORK η_i.

Pour la matrice de connexions de la figure 4.13, l'architecture est montrée dans la figure 4.14.

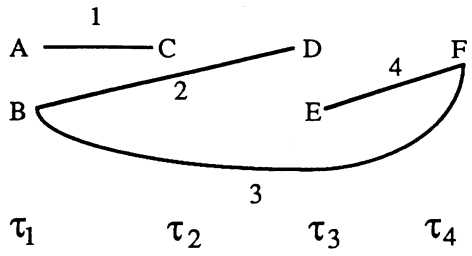


Figure 4.13

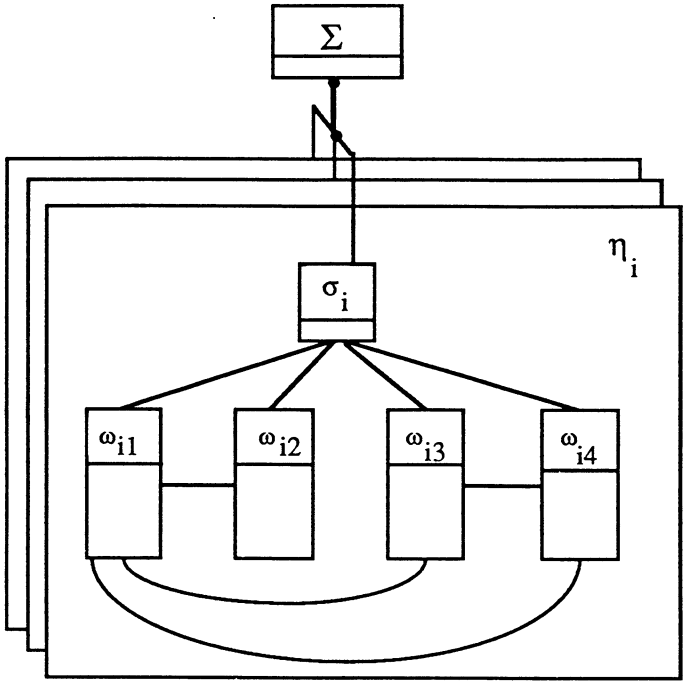


Figure 4.14

Le fonctionnement général du réseau est le suivant :

Supposons que la formule à prouver à "m" clauses différentes.

Le "master process" active "m" "supervisor processes" σ_i (i=1...m).

Le "supervisor process" σ_i active le "worker process" ω_{ii} (ω_{ii} représente la clause τ_i de la formule).

Un "worker process" stocke tous les littéraux de la clause qu'elle représente et les coordonnées des littéraux complémentaires à chacun de ces littéraux. L'objectif de tout "worker process" actif est de satisfaire tous ses littéraux. Pour satisfaire un littéral L, un "worker process" ω doit se synchroniser avec un autre "worker process" ω' où il y a un littéral complémentaire à L : $\neg$L, $\neg$L est aussi satisfait. Si ω' n'est pas actif alors il est activé. Si L a

"p" littéraux complémentaires : $L_1,\ldots,L_p$ alors pour chacun de ces littéraux un nouveau SUB-NETWORK η est activé où les processus de ce SUB-NETWORK η stockent la même information que les processus du SUB-NETWORK courant. Dans le SUB-NETWORK η_i (i=1...p) L sera satisfait avec $\neg L_i$.

Lorsque tous les "worker processes" ω_{ij} (j=1.m) actifs d'un SUB-NETWORK η_i ont satisfait tous ses littéraux, le "supervisor process" σ_i se synchronise avec les ω_{ij} pour recevoir l'information sur les connexions qui forment le Spanning Set calculé par ce SUB-NETWORK η_i. Si $\exists\ \omega_{ij}$ (j=1.m) actif tel que un de ses littéraux ne peux pas être satisfait, le SUB-NETWORK η_i ne donne aucun résultat.

Les fonctions plus détaillées des processus du réseau sont décrites ci-dessous.

Le "master process" (Σ) s'occupe de:

- activer un "supervisor process" par clause de la formule ;
- recevoir les Spanning Sets calculés par les "supervisor processes" ;
- copier un SUB-NETWORK η_i en plusieurs SUB-NETWORK η . La copie d'un réseau R en "n" réseaux $R_1,\ldots,R_n$ consiste à activer les processus du réseau R_i (i=1...n) en leur envoyant l'information stockée dans les processus correspondants du réseau R.

La fonction Compute-Spanning-Sets et le "master process" ont les mêmes buts.

Un "supervisor process" (σ_i) a les fonctions suivantes:

- Activer les "worker processes" ($\omega_{i1},\ldots,\omega_{i\alpha}$) qui représentent la clause indiquée par le "master process" ;
- Connaître les "worker processes" actifs ;
- Détecter la terminaison du travail de son sous-réseau SUB-NETWORK η_i :
  - Le sous-réseau SUB-NETWORK η_i termine avec succès lorsque tous les "worker processes" actifs finissent avec succès.
  - Le sous-réseau SUB-NETWORK η_i termine avec échec si il y a un processus actif que ne finisse pas avec succès.
- Se synchroniser avec le "master process" (Σ) pour réaliser des copies de son SUB-NETWORK η.

Pour notre exemple, un "supervisor process" avec ses "worker processes" est :

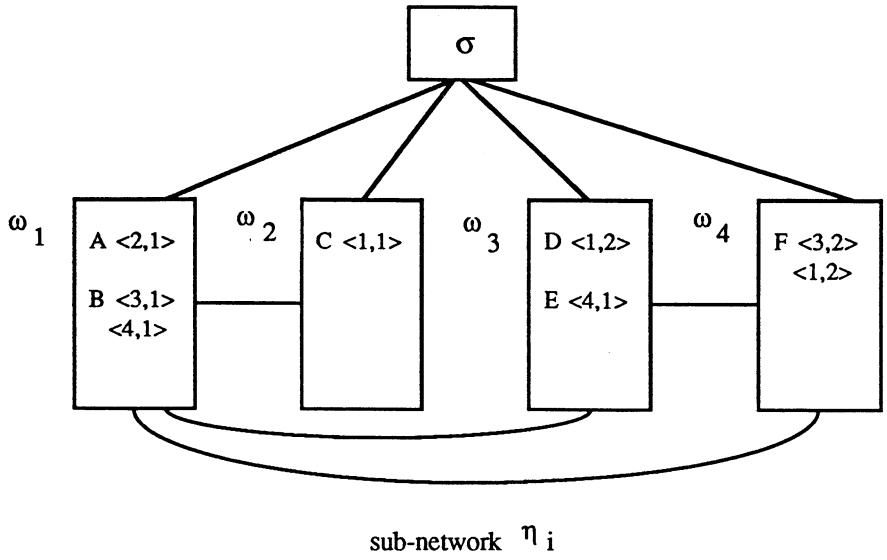


Figure 4.15

La fonction d'un "worker process" est de se synchroniser avec les autres "worker processes" de son SUB-NETWORK η pour obtenir un Spanning Set.

A "worker process" contient l'information de la clause qu'il représente. Les clauses sont composées par des littéraux. Un "worker process": ω_{ij} peut être actif soit parce que son "supervisor" l'active, soit parce qu'il y a un littéral L qui veut être satisfait et qui est complémentaire au littéral $\neg L \in \omega_{ij}$. Dans le dernier cas, $\neg L$ sera étiquete avec "entry-point".

A chaque littéral L dans un "worker process" est associée une liste de 3-uples avec l'information de tous les littéraux complémentaires à L. Les éléments de cette liste ont la forme $< b,n_1,n_2 >$.

- "n_1" et "n_2" sont les coordonnées d'un littéral complémentaire à L: $\neg L$.

  n_1 désigne la clause à laquelle appartient $\neg L$.

  n_2 désigne la position que $\neg L$ occupe dans sa clause.

- "b" est un valeur de vérité pour indiquer si la connexion $<L,\neg L>$ appartient au spanning set.

  Lorsqu'un des 3-uples a sa valeur b = true le littéral est satisfait.

Deux "worker processes" ω_{ij}, ω_{ik} (au moins un des deux est déjà actif) peuvent se synchroniser lorsque $\exists L \in \omega_{ij}$, $\neg L \in \omega_{ik}$ tels que L et $\neg L$ sont complémentaires.

Chaque "worker process" doit satisfaire tous ses littéraux. Si $L \in \omega_{ij}$ et $\delta_1,\ldots,\delta_n$ sont incidents à L alors ω_{ij} communique avec son "supervisor" σ_i et σ_i communique avec Σ pour faire "n" copies du sous-réseau η_i. Dans chacun de ces sous-réseaux L sera satisfait avec une connexion δ_p différente.

Lorsque un littéral L ($L \in \omega_{ij}$) va être satisfait avec une connexion $\delta = \langle L, \neg L \rangle$ ($\neg L \in \omega_{ik}$), le "worker process" ω_{ij} se synchronise avec ω_{ik} , comme conséquence de cette synchronisation :

- ω_{ik} est activé (s'il ne l'était pas déjà) ;
- L et $\neg L$ sont satisfaits ;
- L stocke les coordonnées de $\neg L$ et vice versa.

Si le processus ω_{ik} est déjà actif, L est satisfait si et seulement si $\neg L$ est le point d'entrée de ω_{ij}.

Un "worker process" termine avec succès si tous ses littéraux sont satisfaits, sinon il échoue.

4.3. Un problème: les cycles

Supposons que nous voulons prouver la formule :

$$\neg P(F(F(a))) \vee \exists Y\,(P(Y) \wedge \neg Q(Y)) \vee \exists X\,(Q(F(X)) \wedge \neg P(X)) \vee Q(a)$$

la matrice de connexions associée est la suivante :

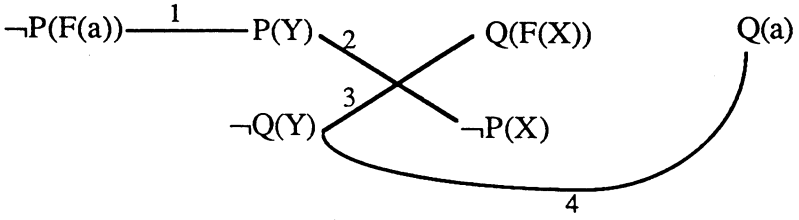


Figure 4.16

Cette matrice a deux candidats à Spanning Set : {1,4} et {1,2,3,4}. Cependant, seulement {1,2,3,4} est un Spanning Set.

Nous pouvons réécrire la matrice comme suit :

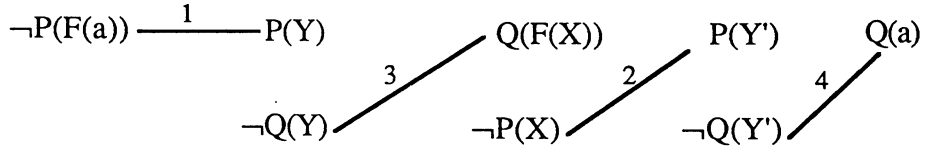


Figure 4.17

L'algorithme que l'on decrit dans la section 4.2.1 peut calculer {1,2,3,4} seulement lorsque la matrice est écrite comme dans la figure 4.17. Mais nous ne voulons pas faire de réécritures des matrices de connexions pour calculer les candidats à Spanning Sets. Notre propos est celui de savoir quand notre algorithme a besoin des réécritures et comment faire pour calculer les candidats à Spanning Sets sans avoir besoin de réécrire les matrices.

Lorsqu'une connexion $\delta = \langle L, \neg L \rangle$ ($L \in \tau$, $\neg L \in \tau'$) est ajoutée à un candidat à Spanning Set pour satisfaire le littéral L, cela ne signifie pas que tous les chemins qui passent par L ont été considérés. En effet, si $\tau' = L_1 \ldots L_{i-1}\ \neg L\ L_{i+1} \ldots L_n$, il manque encore pour considérer les chemins :

$\ldots L \ldots L_1 \ldots$

$\ldots$

$\ldots L \ldots L_{i-1} \ldots$

$\ldots L \ldots L_{i+1} \ldots$

$\ldots$

$\ldots L \ldots L_n \ldots$

les littéraux L1...Li-1 Li+1...Ln doivent être satisfaits. L'algorithme peut boucler indéfiniment en essayant de satisfaire les littéraux L1...Li-1 Li+1...Ln. Pour voir ce cas, analysons la matrice complémentaire suivante:

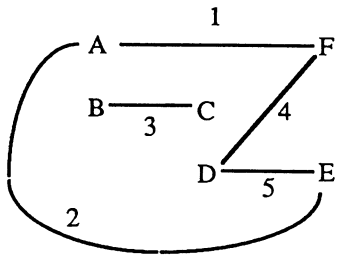


Figure 4.18

Voici les chemins de la matrice et les connexions qui les rendent complémentaires:

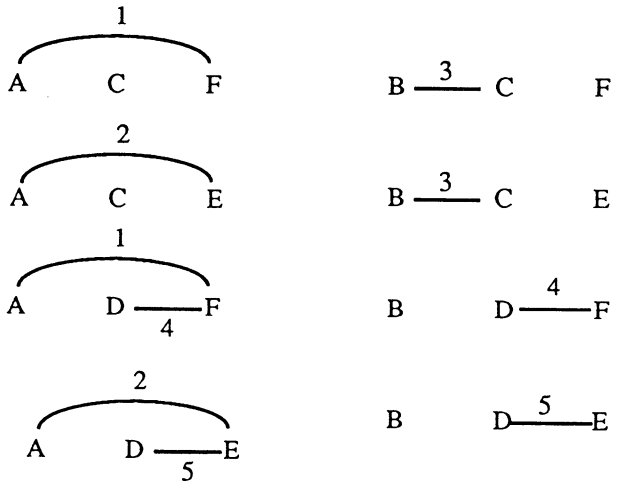


Figure 4.19

le seul spanning set de connexions est donc S = { 1,2,3,4,5 }.

Essayons de construire S en simulant l'activité de la machine décrite dans la section précédente.

Pour voir si tous les chemins de la matrice sont complémentaires, il faut s'assurer que:

1.- Les chemins qui passent par A sont complémentaires.

2.- Les chemins qui passent par B sont complémentaires.

Pour travailler avec les chemins qui passent par A, il y a deux possibilités:

1.1.- Prendre la connexion 1 et voir si les chemins qui passent par E sont ou ne sont pas complémentaires

ou

1.2.- Prendre la connexion 2 et analyser les chemins dans lesquels F apparaît.

En prenant l'alternative 1.1, nous avons considéré un sous-ensemble des chemins qui passent par A : ACF et ADF et nous devons analyser les chemins qui passent par E pour pouvoir considérer les chemins qui manquent : ACE et ADE. On a de nouveau deux options:

1.1.1.- En prenant la connexion 2, il reste à considérer tous les chemins où B apparaît.

1.1.2.- En prenant la connexion 5, il reste à considérer les chemins dans lesquels C apparaît.

Prenons la deuxième alternative. Pour travailler avec les chemins qui passent par C, nous prenons la connexion 3. Il ne nous reste donc plus qu'à considérer les chemins suivants:

Figure 4.20

La prochaine étape sera l'analyse des chemins dans lesquels A apparaît. On se trouve donc à nouveau dans le cas 1. Il existe pourtant une faible (mais non négligeable) probabilité pour que les mêmes actions soient répétées. On peut donc répéter indéfiniment les actions que l'on vient de décrire sans pourtant arriver jusqu'à la solution.

L'ensemble { 1,3,5 } est un cycle en M.

Supposons que l'on construit un Spanning Set S en utilisant notre méthode et que les connexions $\delta_1,\ldots,\delta_m$ sont déjà dans S.

Soit L l'ensemble des littéraux qui doivent être satisfaits pour que S soit un Spanning Set.

Soit L ε L, essayons de satisfaire L avec la connexion $\delta = < L,L' >$.

Si L' appartient à une clause $\tau' = < L'_1,\ldots,L'_n,L' >$ alors la satisfaction de L dépend de la satisfaction des L'_i.

Supposons qu'aucune connexion de S n'ait touché τ'. Comme les L'_i n'appartiennent pas à L, l'inclusion de δ en S ne provoque pas des problèmes d'itération indéfinie.

Supposons que τ' ait été déjà touché par au moins une connexion de S. Cela signifie qu'une des situations suivantes se produit:

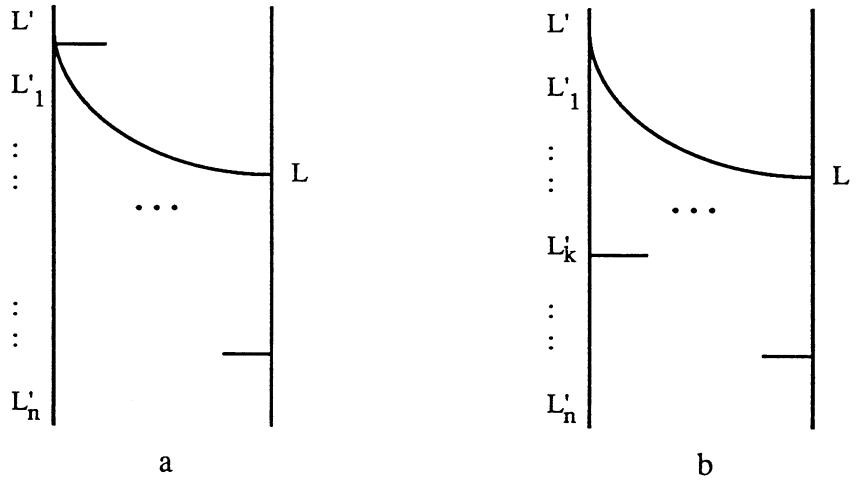


Figure 4.21

Lorsque la situation montrée dans la figure 4.21.a se produit, la satisfaction de L' dépend de L et la satisfaction de L dépend de $L'_1,\ldots,L'_n$. Il n'y a pas pourtant de problèmes d'itération indéfinie. Ces problèmes se présentent lorsque la situation montrée dans la figure 4.21.b se produit.

Ensuite, on définit la structure qui provoque les problèmes d'itération indéfinie (figure 4.21.b).

Définition 4.1

Soit M une matrice.

Soit A l'ensemble de connexions de M.

(M, A) est un graphe de connections.

Un **cycle** Δ dans un graphe de connections (M,A) est un sous-ensemble de A:

$\Delta_0 = \{ \delta_0,\ldots,\delta_{n-1} \}$

où $\forall i \in [0 \ldots n-1]$: $\delta_i = < L^i, L'^i >$, $L^i \in \tau_i$, $L'^i \in \tau_{i+1 \bmod n}$ et $\tau_i \neq \tau_{i+1 \bmod n}$.

Fin de la définition

On dira que L^i *et* L'^i **appartiennent** *à* Δ et on dira aussi que τ_i ($i \in [0 \ldots n-1]$) *appartient à* Δ ou que *les clauses* τ_i *sont* **touchées** *par* Δ.

Soit "m" le nombre de clauses touchées par Δ. Si m = n on dira que Δ est un **cycle simple** (figure 4.22), si m < n on dira que Δ est un **cycle complexe.**

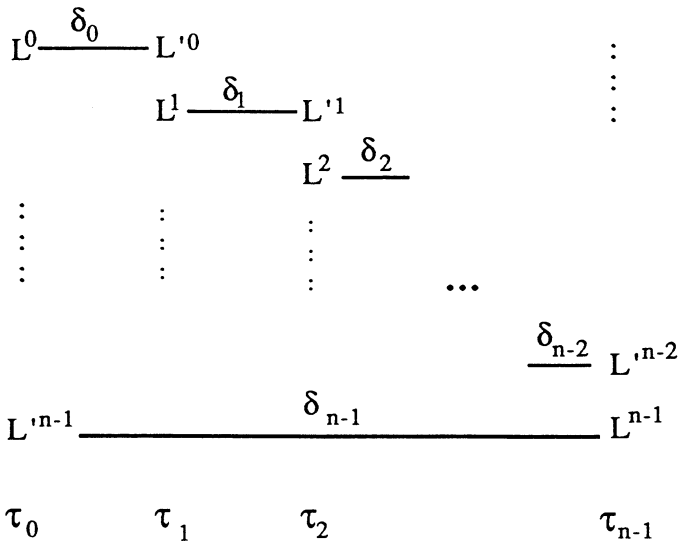


Figure 4.22

Exemple 4.2 :

Soit la matrice de connexions suivante:

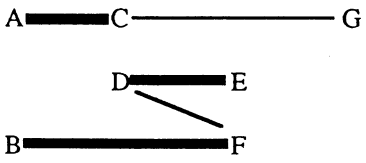


Figure 4.23

Dans la matrice, il y a un cycle A0 = { <A,C>, <D,E>,<B,F> }. Les connexions du cycle rendent complémentaires tous les chemins qui passent par les trois premières clauses (en utilisant uniquement les littéraux du cycle) sauf les S.C.s: ADF et BCE. Ces S.C.s sont appelés **S.C.s dangereux** (voir définition 4.2) et les connexions <D,F> et <C,G> permettent aux S.C.s dangereux d'être complémentaires. Elles sont appelées **connexions de fuite** (voir section 4.4). Fin de l'exemple

Définition 4.2

Soit Δ un cycle dont les connexions touchent les clauses: $\tau_0,\ldots,\tau_{n-1}$. Soient $L_0,\ldots,L_{n-1}$ des littéraux touchés par les connexions de Δ tels que $\forall i \in [0\ldots n-1]$: $L_i \in \tau_i$.

$L_0,\ldots,L_{n-1}$ est un **sous-chemin dangereux (S.C.D).** si et seulement si les connexions de Δ ne le rendent pas complémentaires.

Fin de la définition

LEMME 4.1

Soit M une matrice de connexions.

Soit $\Delta = \{ \delta_0,\ldots,\delta_{n-1} \}$ un cycle simple, $\forall i \in [0\ldots n-1]$: $\delta_i = \langle L^i, L'^i \rangle$ où $L^i \in \tau_i$ et $L'^i \in \tau_{i+1 \bmod n}$ (voir figure §§).

Soit P l'ensemble de S.C. de M de la forme: $X_0\ldots X_{n-1}$ où $\forall i \in [0\ldots n-1]$: $X_i \in \Delta$.

Si l'on considère uniquement les connexions de Δ, alors P n'a que deux S.C. qui ne sont pas complémentaires: $L^0 L^1 \ldots L^{n-1}$ et $L'^{n-1} L'^0 \ldots L'^{n-2}$.

Preuve

Formons deux ensembles C_1 et C_2 avec les littéraux du cycle Δ:

$C_1 = \{ L^0, L^1 \ldots, L^{n-1} \}$

$C_2 = \{ L'^0 \ldots L'^{n-1} \}$

Remarquons le fait que chaque connexion de Δ lie un littéral de C_1 avec un littéral de C_2.

Soit $X_0 X_1 \ldots X_{n-1}$ un S.C. π de M formé avec les littéraux du cycle Δ. $X_0 \in \{L^0, L'^{n-1}\}$ et $\forall i \in [1\ldots n-1]$: $X_i \in \{L^i, L'^{i-1}\}$.

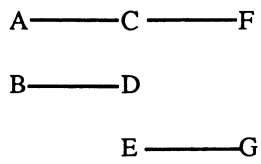
Si dans π ils existent $X_i, X_{i+1 \bmod n}$ tels que $X_i \in C_1$ et $X_i \in C_2$ cela veut dire que $X_i = L^i$ et $X_{i+1 \bmod n} = L'^i$, le S.C. est donc complémentaire.

Par conséquent, il y a seulement deux S.C. dans P qui ne sont pas complémentaires: $L^0 L^1 \ldots L^{n-1}$ et $L'^{n-1} L'^0 \ldots L'^{n-2}$.

Fin de la preuve

Exemple 4.3 :

Ensuite nous donnons une matrice de connexions où il y a un cycle complexe



A_1= { <A,C>, <B,D> }

A_2 = { <C,F>, <E,G> }

A_3 = { <A,C>, <E,G>, <F,C>, <D,B> }

Figure 4.24

A_1 et A_2 sont des cycles et A_3 est un cycle complexe. Le cycle A_3 a cinq S.C.D.: ADF, ADG, AEF, BCG et BEF.

Fin de l'exemple

Dans un cycle complexe, le nombre de S.C.D. n'est pas fixe. L'existence de cette sorte de cycles ne complique pas la situation grâce au résultat suivant:

LEMME 4.2

Soit $\Delta = < \delta_1,\ldots,\delta_n >$ un cycle complexe.

Si chaque cycle simple construit avec les littéraux de Δ est complémentaire, alors Δ est complémentaire.

Preuve

Soit $T = < \tau_1,\ldots,\tau_{n+1} >$ la trace de Δ, c'est-à-dire la séquence des clauses touchées par les connexions $\delta_1,\ldots,\delta_n$.

Examinons T de gauche à droite, soient $\tau_i,\tau_j \varepsilon T$ les deux premières clauses qui satisfont $\tau_i = \tau_j$.

A la séquence de clauses $T_{ij} = < \tau_1,\ldots,\tau_j >$ s'associe la séquence de connexions $\Delta_{i,j-1} = < \delta_1,\ldots,\delta_{j-1}>$. Chaque S.C. qui passe par les littéraux touchés par les connexions de $\Delta_{i,j-1}$ est complémentaire, où bien parce que les connexions de $\Delta_{i,j-1}$ forment un cycle simple ou bien parce que ces connexions constituent une structure de la forme:

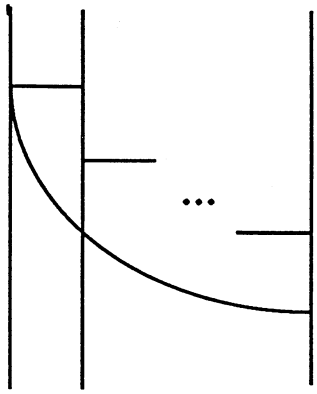


Figure 4.25

Dans des structures comme celle-ci tout S.C. qui peut se former est complémentaire.

Construisons l'ensemble C avec les clauses de Δ, déjà examinées. Initialement, $C = \{ \tau_k \mid i \leq k \leq j \}$. Construisons l'ensemble L avec les littéraux des clauses de C qui satisfont la propriété suivante: tout chemin qui passe par les littéraux de L est complémentaire. Initialement, L aura les littéraux touchés par les connexions $\Delta_{i,j-1}$.

Prenons les τ_q un par un, avec $q = < j+1,\ldots,n,i-1,\ldots,1 >$ (en suivant l'ordre de la séquence donnée). $C \leftarrow C \cup \{ \tau_k \}$.

Si $\exists\ \tau_p \in C$ tel que $\tau_p = \tau_q$, alors les connexions de $\Delta_{p,q-1}$ constituent ou bien un cycle simple, ou bien une structure comme celle de la figure §§. Dans les deux cas, tout S.C. formé avec les littéraux touchés par les connexions de $\Delta_{p,q}$ (on les appellera ψ) est complémentaire.

La prochaine étape consiste à inclure les littéraux de ψ dans L, mais on doit s'assurer pour cela que n'importe quel chemin construit en utilisant les littéraux de $\psi \cup L$ est complémentaire. Nous verrons ensuite les relations possibles entre les deux ensembles de littéraux: L et ψ.

Cas 1 : Les littéraux de L et ψ n'ont aucune clause en commun.

Tout chemin fait avec les littéraux de L et ψ est évidemment complémentaire donc $L \leftarrow L \cup \psi$.

Cas 2 Les littéraux de L et ψ ont des clauses en commun.

Cas 2a Une clause en commun, soit τ_m.

Tous les chemins pouvant se former en utilisant les littéraux de L ont la forme $\pi 1 = L_1,\ldots,L_p$ et $\pi 2 = L'_1,\ldots,L'_q$. Comme $L\pi \in \tau_m$ et $L'_1 \in \tau_m$, les chemins qui peuvent se former en utilisant les littéraux de L et ψ ont la forme: $\pi 3 = L_1,\ldots,L_{p-1},L'_1,\ldots,L'_q$ et $\pi 4$

$=L_1,\ldots,L_p,L'_2,\ldots,L'_q$. Comme $\pi 1$ et $\pi 2$ sont des chemins complémentaires, $\pi 3$ et $\pi 4$ sont alors des chemins complémentaires et $L \leftarrow L \cup \psi$.

Cas 2b Plus d'une clause en commun.

La manière dont on construit C et L rend ce cas impossible.

Fin de la preuve

W. Bibel dans son article : "On Matrices with Connexions" [Bib81] utilise la méthode de connexion pour :

- l'analyse de méthodes de preuve de théorèmes ;
- la simplification des formules à prouver pour le calcul propositionnel ;
- donner une vision unifiée de différents méthodes de preuve.

W. Bibel définit une structure appelée circuit simple qui resamble à notre définition de cycle. Les circuits lui permettent de réaliser des simplifications sur des formules du calcul propositionnel.

4.3.1. La représentation des cycles dans le DAG

Etant donné une matrice de connexions M, nous pouvons construire les DAGs qui représentent les spanning sets de connexions de M lorsqu'il n'y a pas de cycles dans le DAG (sections 4.2.2 et 4.2.3). Pour pouvoir aussi représenter les cycles dans les DAGs, nous utiliserons des nouveaux types de nœuds: BLOCK, CYCLE, CP et CL.

Nœud type **BLOCK**:

Un nœud type BLOCK réunit l'information nécessaire pour satisfaire les littéraux appartenant à un ensemble de clauses touchées par les connexions d'un même cycle. Les fils d'un nœud type BLOCK sont de deux sortes:

Nœuds type CYCLE: Un nœud de ce type pour chaque cycle formé dans les clauses du bloc.

Nœuds type CL: Un nœud de ce type pour chaque clause appartenant au bloc.

Les étiquettes qu'un nœud peut avoir sont:

"satisfied" Lorsque tous ses fils sont satisfaits.

$<L_1,\ldots,L_n>$ L_i est un point d'entrée au bloc.

Nœud type **CYCLE**:

Les littéraux touchés par les connexions des cycles du BLOC sont les descendants des nœuds de type CYCLE et sont groupés en S.C.D. (2 nœuds type CP).

Si un littéral appartient à plusieurs cycles, le sous-dag qui le représente apparaît une seule fois. Le sous-dag est partagé par tous les cycles où le littéral apparaît.

Un nœud type CYCLE peut avoir les étiquettes suivantes:

"satisfied" Un CYCLE est satisfait lorsque ses deux S.C.D. (CP_1 et CP_2) sont satisfaits.

$<\delta_1,\ldots,\delta_n>$ Les δ_i sont les connexions qui forment le cycle.

Nœud type **CP**:

Chaque nœud de type CP a comme fils les littéraux appartenant à un S.C.D. d'un cycle donné.

Lorsqu'un des fils de CP a été satisfait avec une connexion de fuite, le S.C.D. CP a l'étiquette "satisfied". Le lemme 4.11 (section 4.4) établit ce qu'est une connexion de fuite.

Nœud type **CL**:

Les fils d'un nœud type CL sont des littéraux d'une clause du bloc. Les fils d'un nœud type CL peuvent être de trois sortes:

- Des littéraux qui sont points des entrée au bloc (ils sont étiquetés avec "entry-point").

- Des littéraux qui appartient à un ou plusieurs cycles du bloc (ils sont étiquetés avec la séquence des cycles auquels ils appartiennent).

- Des littéraux qui n'appartient à aucun cycle et qui ne sont pas points des entrée au bloc.

Soient $L_1,\ldots,L_n$ les fils de CL. Si $\forall$ i = 1...n: L_i est un "entry-point" où père (L_i) est un nœud CP ou L_i est "satisfait" alors CL a l'étiquette "satisfait".

CL a associé la liste d'identificateurs de cycles qui touchent la clause.

4.3.2. La formation des cycles dans le DAG

Supposons que l'on construise un DAG D et que l'on veuille satisfaire le littéral L de la clause t. Supposons que L soit complémentaire au littéral L' de la clause τ'. La situation est illustrée dans la figure suivante:

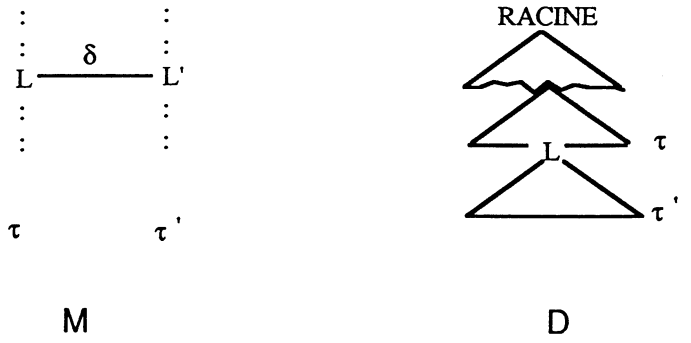


Figure 4.26

La clause τ' peut être ou ne pas être déjà en D. Si τ' n'est pas en D alors il n'y a pas de nouveau cycle, autrement il est possible qu'un nouveau cycle se forme. Dans le dernier cas, le DAG doit être transformé pour pouvoir représenter le nouveau cycle.

Avant d'établir dans quelles conditions se forment de nouveaux cycles, nous prouverons sur les blocs deux propriétés qui nous serons utiles dans cette section.

Définition 4.3

Soit M une matrice de connexions.

Soit $\{ \tau_1,\ldots,t_n \}$ un ensemble ordonné de clauses de M.

Une **route** entre les clauses τ_1 et t_n est une séquence ordonnée de connexions $<\delta_1,\ldots,d_n>$ telle que $\delta_i = < \tau_i,\ldots,t_{i+1} >$. Pour deux connexions δ_i, δ_{i+1} qui touchent la même clause τ_i, δ_i et δ_{i+1} ne touchent pas le même littéral de τ_i.

Fin de la définition

Les clauses d'un bloc sont touchées par un ou plusieurs cycles. Si une clause d'un cycle Δ est dans un bloc B, toutes les clauses touchées par Δ sont en B.

LEMME 4.3

Soient τ_i, τ_j deux clauses dans un bloc B. Il y a deux routes différentes entre τ_i et τ_j qui sont formées par des connexions des cycles de B.

Preuve

Procédons par cas.

CAS 1 : τi, τj appartiennent au même cycle Δ.

Soient $\delta_1,\ldots,\delta_k$ les connexions qui forment Δ.

Par les définitions de cycle et de route nous avons deux routes entre τ_i et τ_j:

$$\delta_p,\ldots,\delta_{p+q}$$

$$\text{et } \delta_1,\ldots,\delta_{p-1},\delta_{p+q+1},\ldots,\delta_k$$

CAS 2 : τi, τj appartiennent aux cycles Δ et Δ' respectivement.

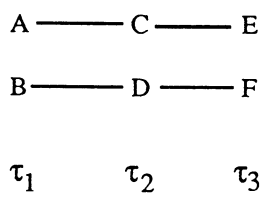
CAS 2a Δ et Δ' ont, au moins, une clause commune.

Soit τ une clause commune à Δ et Δ'. Comme τ_i et τ appartiennent au même cycle (Δ), et comme τ et τ_j appartiennent au même cycle (Δ') nous sommes dans les conditions du CAS 1.

Soient $\pi 1$ et $\pi 2$ les routes entre τ_i et τ.

Soient $\pi 3$ et $\pi 4$ les routes entre τ et τ_j.

La concaténation de $\pi 1$ (ou $\pi 2$) avec $\pi 3$ (ou $\pi 4$) ne produit pas toujours une route. Par exemple,



$\pi 1 = \langle A,C\rangle$, $\pi 2 = \langle B,D\rangle$, $\pi 3 = \langle C,E\rangle$, $\pi 4 = \langle D,F\rangle$

Figure 4.27

Supposons que nous voulions construire une route de $\tau 1$ à $\tau 3$, Append ($\pi 1$, $\pi 3$) n'est pas une route.

Le problème surgit parce que $\pi 1$ et $\pi 3$ ont un littéral commun: C. Etant donné que $\tau 1$ et $\tau 2$ appartiennent au même cycle (Cas 1), il y a une autre route pour aller de $\tau 1$ à $\tau 2$: $\pi 2$ (naturellement $\pi 2$ et $\pi 3$ n'ont pas de littéraux communs). Nous avons donc: append ($\pi 2$,$\pi 3$) est une route de $\tau 1$ à $\tau 3$.

Entre les routes suivantes: append ($\pi 1$,$\pi 3$), append ($\pi 1$,$\pi 4$), append ($\pi 2$,$\pi 3$), et append ($\pi 2$,$\pi 4$) il y a au moins deux routes pour aller de τ_i à τ_j.

CAS 2b : Δ et Δ' n'ont aucune clause commune.

Soient $\tau^\circ_1,\ldots,\tau^\circ_i,\ldots,\tau^\circ_{n0}$ les clauses de Δ ($\tau_i = \tau^\circ_i$).

Dans le bloc il y a d'autres clauses, donc il y a un autre cycle qui a quelque(s) clause(s) de Δ plus les clauses $\tau^1_1,\ldots,\tau^1_{n1}$.

Par le CAS 2a , il y deux routes de τ°_i à τ^1_k (k = 1...n1). Si nous répétons ces actions jusqu'à ce que l'on puisse travailler τ_j, nous pourrons construire deux routes de τ_i à τ_j.

Fin de la preuve

LEMME 4.4

Dans un bloc B, il y a au moins un cycle qui contient toutes les clauses de B.

Preuve

Par induction sur le nombre de cycles de B.

Lorsqu'il n'y a qu'un seule cycle en B, le lemme se satisfait évidemment.

CAS de base: Il y a deux cycles : Δ et Δ'.

Sans perte de généralité, supposons que Δ et Δ' aient au moins une clause commune: τ.

Soit $\tau_i \in \Delta$, $\tau_i \neq \tau$ et soit $\tau_j \in \Delta'$, $\tau_j \neq \tau$.

Par le lemme 4.3 (cas 1), il y a deux routes de τ_i à τ : $\pi 1$ et $\pi 2$ sont faites avec des connexions de Δ ; $\pi 1$ et $\pi 2$ touchent τ_i en deux littéraux différents (L_{i1},L_{i2}). De la même façon, $\pi 1$ et $\pi 2$ touchent τ en deux littéraux différents (L_1, L_2). Pareillement, il y a deux routes $\pi 3$ et $\pi 4$ qui vont de τ à τ_j et qui sont faites avec les connexions de Δ'. (Voir figure 4.28).

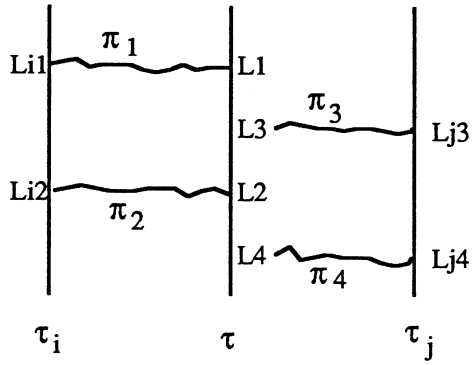


Figure 4.28

Par définition de route : L1 ≠ L3 ou L1 ≠ L4. Supposons que L1 ≠ L3, alors append (π1,π3) et append (π2,π4) sont deux routes de τi à τj et l'union des connexions de ces deux routes forme un cycle.

CAS inductif.

Soit Δ un cycle qui touche les clauses de "n" cycles: $\Delta_1,\ldots,\Delta_n$.

Prenons un nouveau cycle: Δ_{n+1}, supposons sans perte de généralité que Δ_{n+1} ait au moins une clause commune avec Δ et procédons comme dans le cas base pour former le nouveau cycle.

Fin de la preuve

On veut découvrir quand se forment de nouveaux cycles dans un Dag. C'est pour cela qu'on fait une étude des situations qui peuvent se présenter. En ayant une connexion δ = <L,L'> où L ε τ et L' ε τ', on a les cas suivantes :

- τ et τ' appartiennent au même bloc (section 4.3.2.1) ;
- Ni τ et τ' n'appartiennent à aucun bloc (section 4.3.2.2) ;
- Les clauses τ et τ' n'appartiennent pas au même bloc (section 4.3.2.3).

4.3.2.1. τ et τ' appartiennent au même bloc

Nous avons la connexion δ qui lie les littéraux L et L' des clauses τ et τ' respectivement.. Nous étudions la situation en considérant les littéraux comme appartenant ou n'appartenant pas au même S.C.D..

L et L' appartiennent au même S.C.D..

Un cycle peut se former dans certains cas. Nous n'étudierons pas dans quels cas apparaît un nouveau cycle mais nous donnerons deux exemples de situations pouvant se présenter.

La figure 4.29 montre deux matrices de connexions M1 (4.29.a) et M2 (4.29.b). Soient D1 et D2 deux DAGs pour M1 et M2 respectivement. Supposons que les connexions 1,2 et 3 soient déjà en D1 et D2. Lorsque la connexion 4 est incluse dans D1, il se forme un nouveau cycle, cependant lorsque la connexion 4 est incluse dans D2, il ne se forme pas de nouveau cycle.

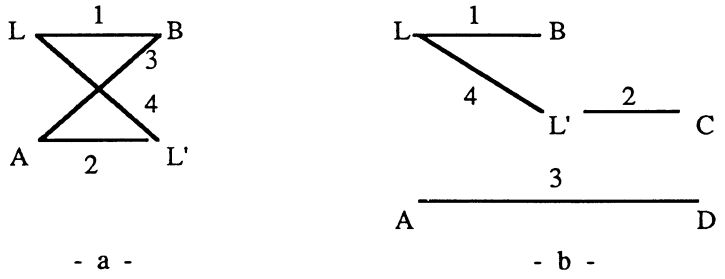


Figure 4.29

L et L' appartiennent à des S.C.D. différents.

Le lemme suivante montre que dans ce cas un nouveau cycle se forme.

LEMME 4.5

Soit $\delta = < L,L' >$ une connexion telle que $L \in \tau_i$ et $L' \in \tau_j$; τ_i, τ_j appartiennent à un bloc B.

L et L' appartiennent à des S.C.D. différents quelque soit le cycle choisi.

Lorsque δ est incluse dans le spanning set, un nouveau cycle se forme.

Preuve

Par le lemme 4.4, il y a un cycle Δ qui touche toutes les clauses de B. Δ touche donc les clauses τ_i et τ_j. La situation est montrée dans la figure suivante:

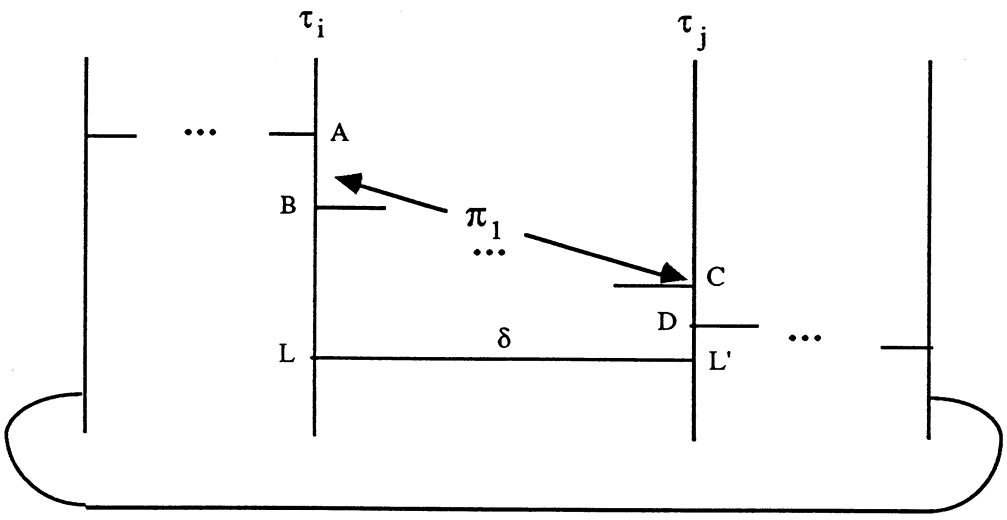


Figure 4.30

Avec les connexions de Δ, on peut construire une route π1 de τ_i à τ_j telle que deux de ses connexions soient incidentes à B et C respectivement (Lemme 4.3).

Procédons par cas:

CAS 1 : L ≠ A, L ≠ B, L' ≠ C, L' ≠ D.

Il y a deux nouveaux cycles:

1.- Connexions de π1 plus δ.

2.- Connexions de Δ moins connexions de π plus δ.

CAS 2 : L ≠ A, L = B, L' ≠ C, L' = D

Par hypothèse, L et L' n'appartiennent pas au même S.C.D., ce cas est donc impossible.

Les autres cas sont analogues au cas montrés ci-dessus.

Fin de la preuve

Au moins un des littéraux L ou L' n'appartient à aucun S.C.D..

Dans le lemme suivant, on prouve que lorsque les conditions citées ci-dessus se présentent, des nouveaux cycles se forment.

LEMME 4.6

Soient L et L', deux littéraux complémentaires liés par la connexion δ ou L ε τ_i et L' ε τ_j; τ_i et τ_j appartiennent au même bloc B.

Si au moins un des littéraux L ou L' n'appartient à aucun S.C.D., alors lorsque δ est incluse dans le spanning set, de nouveaux cycles se forment.

Preuve

Par le lemme 4.4, on peut construire un cycle Δ qui touche toutes les clauses de B.

En utilisant les connexions de Δ, on peut construire deux routes $\pi 1$ et $\pi 2$ qui aillent de τ_i à τ_j (lemme 4.3).

Nous allons considérer deux cas:

CAS 1 : L appartient à un S.C.D..

Cela veut dire que L' n'appartient à aucun S.C.D..

La situation est similaire à celle de la figure %%. Etant donné que L' n'appartient à aucun S.C.D., les connexions de Δ ne touchent pas L' et pourtant L' n'est pas touché par les connexions de $\pi 1$ ou $\pi 2$. Tout cela implique que L' $\neq$ C et L' $\neq$ D.

Maintenant il est facile de voir comment se forment de nouveaux cycles, en ayant L est égale ou n'est pas égale à A (B).

CAS 2 : Ni L ni L' appartiennent à S.C.D..

Cela veut dire que L $\neq$ A et L $\neq$ B et L' $\neq$ C et L' $\neq$ D, il y a donc deux nouveaux cycles:

1.- Les connexions de $\pi 1$ plus δ.

2.- Les connexions de Δ moins les connexions de $\pi 1$ plus δ.

Fin de la preuve

En conclusion, il faut chercher de nouveaux cycles chaque fois que l'on ajoute une connexion δ = <L,L'> (L,L' ε B) au DAG. Les nouveaux cycles se forment (éventuellement) dans B.

4.3.2.2. Ni τ ni τ' n'appartiennent à un bloc

Nous allons maintenant étudier la situation telle qu'elle se présente dans le DAG. Nous avons déjà les clauses τ et τ' dans le DAG et maintenant les littéraux Li et L'j de τ et τ' sont respectivement complémentaires et ils sont liés par la connexion δ. Nous avons deux cas à étudier:

- τ (τ') est ancêtre de τ'(τ) (Lemme 4.7) ;
- τ (τ') n'est pas ancêtre de τ'(τ) (Lemme 4.8)

LEMME 4.7

Soient τ et τ' deux clauses dans un DAG D. Ni τ ni τ' n'appartiennent à un bloc et τ (τ') n'est pas ancêtre de τ' (τ).

Soient $\delta_1 = < L_1, L'_1 >$ et $\delta_2 = < L_2, L'_2 >$ deux connexions appartenant à D. Soit L_i le littéral à satisfaire. Soit $\delta = < L_i, L'_j >$ la connexion qui sera incluse dans le DAG. L_1 et L_i appartiennent à τ, L'_2 et L_i appartiennent à τ'.

Si $L_i \neq L_1$ et $L'_j \neq L'_2$ alors un nouveau cycle se forme.

Preuve

τ et τ' doivent avoir un ancêtre commun en D, soit τ_a l'ancêtre de τ et τ'.

Il y a une route π1 de τ_a à τ, la dernière connexion de π1 est δ1. De manière similaire, il y a une route π2 de τ_a à τ' et sa dernière connexion est δ_2. Append (rev (π1), π2) est une route. La situation est démontrée dans la figure suivante:

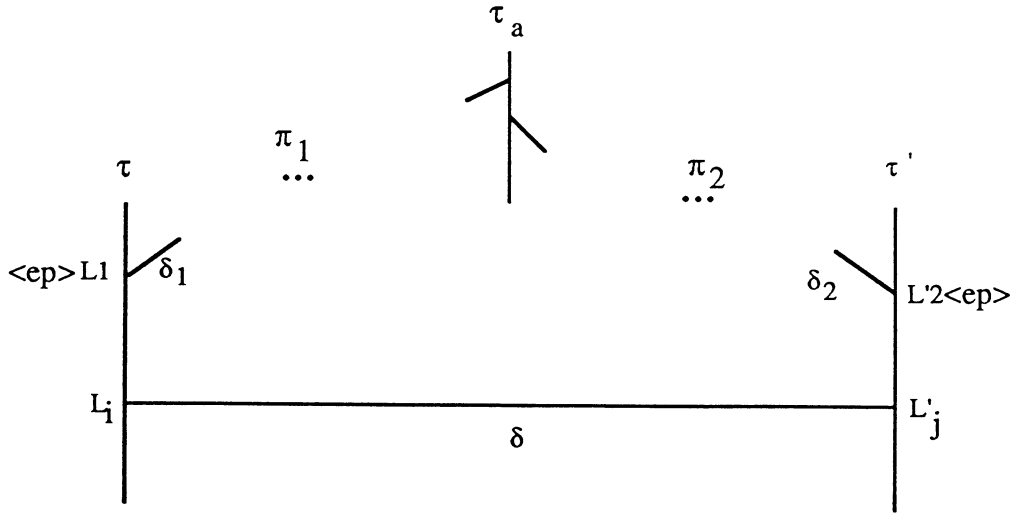


Figure 4.31

La connexion δ peut lier les clauses τ et τ' de quatre façons différentes:

$$L1 = L_i \wedge L'2 = L'_j.$$

$$L1 = L_i \wedge L'2 \neq L'_j.$$

$$L1 \neq L_i \wedge L'2 = L'_j.$$

$$L1 \neq L_i \wedge L'2 \neq L'_j.$$

L1 est point d'entrée à τ, et L_i est le littéral à satisfaire ; cependant $L1 \neq L_i$. Il est facile de voir qu'un nouveau cycle se forme seulement quand $L_i \neq L_1$ et $L'_j \neq L'_2$.

Fin de la preuve

LEMME 4.8

Soient τ et τ' deux clauses dans un DAG D. Ni τ ni τ' n'appartiennent à aucun bloc ; soit τ est ancêtre de τ', soit τ' est ancêtre de τ.

Soient $\delta_1 = < L_1,L'_1 >$ et $\delta_2 = < L_2,L'_2 >$ les deux connexions extrêmes de la route de τ à τ' qui peut être formée avec les connexions de D.

Soit $\delta = < L_i,L'_j >$ la connexion à inclure dans D, L_i et L'_j appartiennent à τ et τ' respectivement.

Sans perte de généralité, disons que τ' est ancêtre de τ (cela veut dire que L_i doit être satisfait).

Un nouveau cycle se forme si et seulement si $L_i \neq L_1$.

Preuve

La situation est montrée dans la figure suivante:

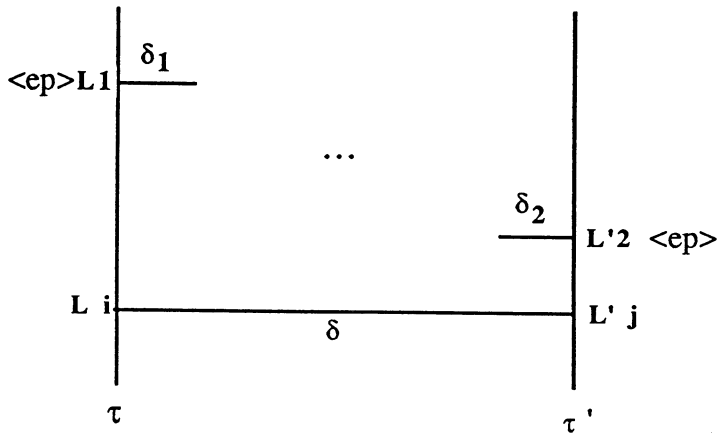


Figure 4.32

De la façon dont le DAG est construit, il y a une route $\pi = < \delta 1,\ldots,\delta 2 >$ de τ à τ'. Si $L_i \neq L_1$, alors les connexions de π plus δ forment un cycle ; si $L_i = L_1$, alors il n'y a pas de nouveaux cycles.

Fin de la preuve

Les lemmes 4.7 et 4.8 montrent qu'un nouveau cycle se forme lorsque le littéral complémentaire de L_i: L'j (L'j ε τ') n'est pas le point d'entré de la clause τ'. Les lemmes 4.7 et 4.8 montrent aussi quelles sont les connexions qui forment les cycles.

4.3.2.3. Les clauses τ et τ' n'appartiennent pas au même bloc.

Deux cas peuvent se présentes:

τ' n'appartient à aucun bloc (LEMME 4.9)

τ' appartient à un bloc B' (LEMME 4.10)

LEMME 4.9

Soit B un bloc formé dans un DAG D. Soient $L_1 \ldots L_n$ les points d'entré de B.

Soient τ = ...L... et τ' = ...L'... deux clauses telles que $\tau \varepsilon$ B et τ' n'appartient à aucun bloc.

Soit δ = <L,L'> une connexion qui lie les clauses de τ et τ'.

Soit δ_i une connexion d'entré à B telle que δ_i est incidente au littéral L_i de la clause τ_i, (L_i ε $L_1 \ldots L_n$).

Soit δ' une connexion d'entré à la clause τ' (δ' est incidente au littéral L'' de la clause τ').

Un nouveau cycle simple se forme si et seulement si

($\tau \neq \tau_i$ et $L'' \neq L'$ et ™$L_i \varepsilon \{L_1 \ldots L_n\} : L_i \neq L$ et il y a une route de τ à τ_i telle que la route ne touche ni L ni L_i)

ou

($\tau = \tau_i$ et $L_i \neq L$ et $L' \neq L''$)

Preuve

τ et τ' ont un ancêtre commun: τ_{ai} pour chaque point d'entrée L_i.

Soient π et π' deux routes de τ_{ai} à τ_i et de τ_{ai} à τ' respectivement. La figure suivante montre la situation.

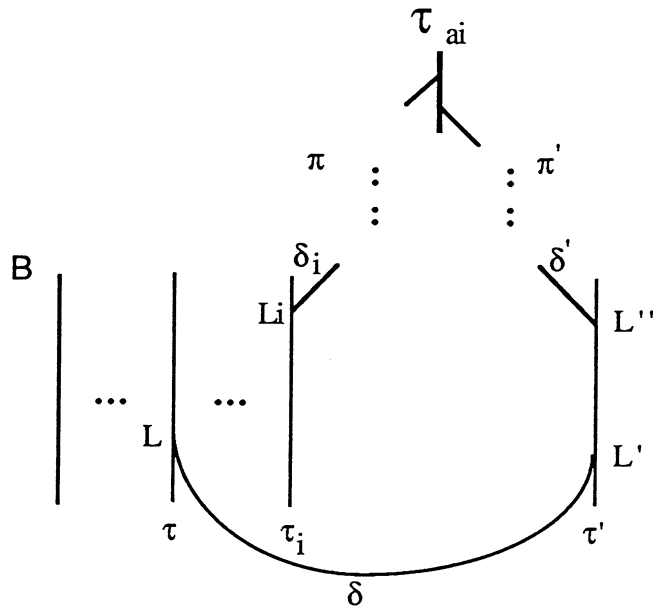


Figure 4.33

La preuve est par cas.

Fin de la preuve

LEMME 4.10

Soient B et B' deux blocs formés dans un DAG D. Soient $L_1 \ldots L_m$ les points d'entrée à B et $L'_1 \ldots L'_n$ les points d'entrée à B'.

Soient τ = ...L... et τ' = ...L'... deux clauses appartenant aux blocs B et B' respectivement et soit δ = <L,L'> une connexion qui lie τ et τ'.

Soit δ_i une connexion d'entrée à B telle que δ_i est incidente au littéral L_i de la clause τ_i.

Soit δ'_j une connexion d'entrée à B' telle que δ'_j est incidente au littéral L'_j de la clause τ'_j.

Un nouveau cycle simple se forme si et seulement si

[

($\tau = \tau_i$ et $L \neq L_i$)

ou

($\tau \neq \tau_i$ et il y a une route de τ à τ_i qui ne touche ni L ni L_i)

]

et

[

($\tau' = \tau'_j$ et $L' \neq L'_j$)

ou

($\tau' \neq \tau'_j$ et il y a une route de τ'_j à τ' qui ne touche ni L'_j ni L')

]

Preuve

τ et τ' ont un ancêtre commun: τ_{aij} pour chaque pair de points d'entrée L_i, L'_j.

Soient π_1 et π_2 deux routes de τ_{aij} à τ_i et de τ_{aij} à τ' respectivement. La figure suivante décrit la situation.

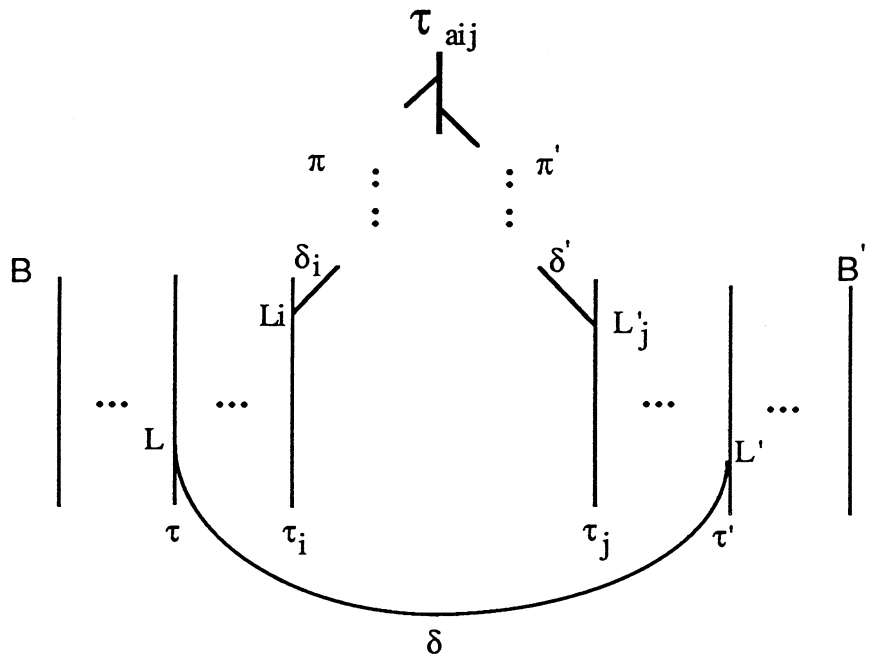


Figure 4.34

La preuve est par cas.

Fin de la preuve

La quantité de cycles qui se forment dépend de la quantité de points d'entrée différents aux blocs. Les clauses concernées dans les cycles sont celles de B (éventuellement celles de B') et celles qui sont dans les routes de τ_{ai} (τ_{aij}) à τ_i et de τ_{ai} (τ_{aij}) à τ_j.

4.4. Une solution: Les connexions de fuite.

Chaque fois qu'une connexion est ajoutée au spanning set, une famille de chemins a prouvé sa complémentarité. Les connexions d'un cycle simple garantissent la complémentarité de presque tous les chemins qui passent par les clauses du cycle en touchant uniquement les littéraux du cycle. Il y a seulement deux types de chemins qui ne sont pas complémentaires. Ce sont les chemins qui contiennent les S.C.D. du cycle.

Certaines connexions pourront rendre (ou moins contribueront à le faire) complémentaires les S.C.D.. On donne ensuite une classification de toutes les connexions possibles qui peuvent être incidentes aux littéraux d'un S.C.D., et on établit quelles sont les connexions de fuite. Il est évidemment nécessaire qu'au moins un littéral de chaque S.C.D. ait

incident une connexion de fuite. Dans cette section, nous identifions les connexions qui peuvent être considèrées comme des connexions de fuite.

LEMME 4.11

Soient $\tau = \ldots L \ldots$ et $\tau' = \ldots L' \ldots$ deux clauses d'une matrice de connexions M.

τ appartient au cycle Δ et L appartient à un S.C.D. de Δ.

L et L' sont littéraux complémentaires, soit $\delta = \langle L,L' \rangle$ la connexion qui les lie.

Les δ suivants sont connexions de fuite:

A.- δ ou L' appartient à Δ et au même S.C.D. que L.

B.- δ ou L' appartient à une clause τ' qui n'appartient à aucun cycle.

C.- δ ou L' appartient à une clause τ'. Cette clause appartient au cycle Δ' mais n'appartient pas au cycle Δ. L' appartient à Δ'.

D.- δ ou L' appartient à une clause τ'. Cette clause appartient au cycle Δ' mais n'appartient pas au cycle Δ. L' n'appartient pas à Δ'.

Aucune autre connexion n'est de fuite.

Preuve

Nous allons maintenant examiner toutes les possibilités pour L' et τ'.

CAS 1. L et L' appartiennent au même cycle Δ.

Les deux situations qui peuvent avoir lieu sont illustrées dans la figure suivante:

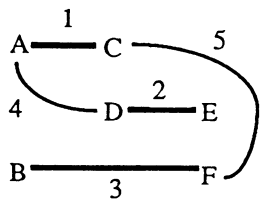


Figure 4.35

La matrice de connexions de la figure 4.35 a au moins le cycle $\Delta = \{ 1,2,3 \}$ avec les S.C.D.: ADF et BCE. La connexion 4 lie deux littéraux d'un même S.C.D., ce S.C. est

complémentaire grâce à la connexion 4. La connexion 5 lie deux littéraux appartenant à deux S.C.D. différents et ne contribue pas à garantir la complémentarité des nouveaux chemins.

CAS 1.1 : L et L' appartiennent au même S.C.D. π (cas A de l'hypothèse).

Soit π = ... L ... L' ..., le chemin dangereux. Comme L et L' sont complémentaires alors π est un chemin complémentaire.

CAS 1.2 : L et L' appartiennent à différents S.C.D. (π1 et π2).

Tous les chemins qui passent par L en utilisant les littéraux de Δ sont complémentaires sauf π1. L' $\notin$ π1 et L' ε Δ, par conséquent les chemins qui passent par L et L' sont complémentaires même si L et L' ne le sont pas. La connexion δ ne contribue à la complémentarité d'aucun nouveau chemin, c'est à dire que δ n'est pas une connexion de fuite.

CAS 2 : L' n'appartient pas à Δ.

CAS 2.1 : La clause τ' (L' ε τ') n'appartient pas à Δ.

CAS 2.1.1 : τ' n'appartient à aucun cycle (CAS B de l'hypothèse).

Exemple 4.4 :

Soit la matrice de connexions suivante:

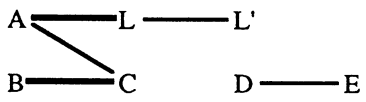


Figure 4.36

Les connexions: <A,L> et <B,C> constituent un cycle qui a deux S.C.D.: AC et BL. Le S.C. AC est complémentaire (cas A de l'hypothèse -déjà prouvée-). Les chemins qui passent par L sont de la forme: $X_1LL'X_4$ (ils sont complémentaires parce que L et L' sont complémentaires) ou de la forme X_1LDX_4. Il faut prouver que tous les chemins qui passent par D sont complémentaires (ils le sont en effet).

Fin de l'exemple

D'après notre méthode, si un littéral d'une clause τ est touché par une connexion du spanning set alors tous les autres littéraux de τ doivent être satisfaits. Nous enrichirons alors l'hypothèse afin de prouver une proposition plus générale.

Soit $\tau' = L' \wedge L'_1 \wedge \ldots \wedge L'_n$. Supposons que $\forall i \in [1...n]$: P_{Li} sont complémentaires, alors P_L sont complémentaires.

$\tau \neq \tau'$ donc les chemins P_{Li} ont la forme:

$$\ldots L \ldots L' \ldots$$

$$\text{or} \ldots L \ldots L'_i \ldots \forall i \in [1...n]$$

Par hypothèse, L et L' sont complémentaires , donc le premier type de chemin est complémentaire (δ est une connexion de fuite).

Etant donné que P_{Li} sont complémentaires, en particulier les chemins de la forme ... L .... L'_i ... sont complémentaires.

Par conséquent, les chemins P_L sont complémentaires.

CAS 2.1.2 : τ' appartient à un cycle Δ'.

CAS 2.1.2.1 : L' appartient à Δ'.

Dans le cas général, Δ et Δ' ont des clauses en commun.

Permutons les clauses de Δ et Δ' de manière à ce que les clauses de Δ, qui n'appartiennent pas à l'intersection soient placées le plus à gauche possible. Plaçons ensuite celles qui appartiennent à l'intersection et enfin les autres clauses de Δ'.

Soient $\pi 1_i$, $\pi 2_j$ ($i,j \in \{1,2\}$) les S.C.D. de Δ et Δ' respectivement.

Soient $\pi 1'_i \leq \pi 1_i$ les S.C.D. que n'ont pas de littéraux appartenant à l'intersection de clauses des cycles Δ et Δ'. De même, nous avons aussi $\pi 2'_j \leq \pi 2_j$.

Les S.C.D. que l'on peut former en utilisant les littéraux de Δ et Δ' sont:

$\{ \text{append} (\pi 1_i, \pi 2'_j) \mid i,j = 1,2 \} \cup \{ \text{append} (\pi 1'_i, \pi 2_j) \mid i,j = 1,2 \}$

$L' \in \pi 2'_j$ ($j = 1,2$) par hypothèse.

Posons: $L \in \pi 1_p$ et $L' \in \pi 2'_q$. On n'a pas encore prouvé la complémentarité des S.C.:

{ append ($\pi 1_i, \pi 2'_j$) | i,j = 1,2 } - { append ($\pi 1_p, \pi 2'_q$) } ∪ { append ($\pi 1'_i, \pi 2_j$) | i,j = 1,2 }

Pourtant la connexion δ garantit la complémentarité de nouveaux chemins, elle est donc une connexion de fuite.

CAS 2.1.2.2 : L' n'appartient pas à Δ' (Cas D de l'hypothèse).

Ce cas est analogue au cas 2.1.1

CAS 2.2 : L' ε τ', τ' ε Δ.

τ' a la forme: $L'_1 \ldots L'_i \ldots L' \ldots L'_m$ ou $L'_i \varepsilon \Delta$. Les chemins qui passent par L ont la forme:

$\pi 1$: ... L ... L'_1 ...

$\pi 2$: ... L ... L'_i ...

$\pi 3$: ... L ... L' ...

...

$\pi 4$: ... L ... L'_m ...

Les S.C.D. de Δ où L apparaît sont inclus dans les chemins de type $\pi 2$. La connexion δ garantit uniquement la complémentarité des chemins de type $\pi 3$ donc δ n'est pas une connexion de fuite.

Fin de la preuve.

.4.4.1. La construction du DAG à partir d'une matrice de connexions

Dans cette section, nous décrivons un algorithme qui construit les Spanning Sets d'une matrice de connexions. A la différence de l'algorithme présenté dans la section 4.2.3, cet algorithme peut travailler avec les cycles en utilisant les résultats obtenus dans les sections 4.3.2 et 4.4.

L'algorithme construit un Dag pour chaque clause de la matrice M - fonction Compute-Spanning-Sets -.

Les actions à réaliser sur chaque Dag D sont les suivantes:

• Choisir un littéral à satisfaire et la connexion δ avec laquelle L sera satisfait - fonction choose-cnx -.

Si tous les littéraux de D ont été déjà satisfaits, alors D représente un Spanning Set de M.

Si l'on ne peut pas choisir un littéral à satisfaire, mais il y a des littéraux qui ne sont pas satisfaits, alors D ne peut pas devenir un Spanning-Set de M.

• Déterminer si la connexion δ est nécessaire pour satisfaire le littéral L - fonction cnx-necessary -.

• Si L n'appartient à aucun S.C.D., alors δ est nécessaire.

• Si L appartient à un S.C.D. alors, δ est nécessaire si et seulement si δ est une connexion de fuite (Lemme 4.11).

Lorsque δ n'est pas nécessaire, il faut choisir un autre littéral à satisfaire.

• Ajouter la connexion δ dans le Dag - fonction add-cnx -.

Avant d'ajouter δ à D, il faut déterminer si de nouveaux cycles se forment en D. Dans la section 4.3.2, on présente les conditions sous lesquelles se forment de nouveaux cycles. Lorsque de nouveaux cycles se forment, il faut les représenter sur le Dag ; D doit être donc transformé.

• Une fois que δ est ajoutée à D, il faut déterminer quels nœuds du Dag ont été satisfaits - fonction satisfy-dag -. Une nouvelle connexion sera ensuite choisie.

La description détaillée des fonctions : Compute-Spanning-Sets, choose-cnx, cnx-necessary,add-cnx et satisfy-dag se trouve dans l'annexe2.

4.5. La méthode pour le calcul de spanning sets est correct

Nous avons représenté notre méthode (sections 4.1 et 4.5) par des DAGs (sections 4.2.2, 4.2.3, 4.3.1, 4.4.1). Nous utilisons ensuite la structure DAG pour prouver que notre méthode calcule des spannings sets.

LEMME 4.12

Soit B un bloc avec n+1 clauses: $\tau_0,\ldots,\tau_n$.

Soit $L_0,\ldots,L_n$ une séquence de littéraux telle que $L_i \in \tau_i$, chaque L_i appartient à un S.C.D. d'un cycle de B et $\forall$ i,j: L_i et L_j ne sont pas complémentaires.

Il y a un cycle Δ en B qui a tous les littéraux $L_0,\ldots,L_n$.

Preuve

Construisons une route de τ_i à τ_{i+1} qui touche le littéral L_i mais qui ne touche pas à $L_{i+1 \bmod n}$.

L_i appartient à un cycle. L_i a donc, au moins, une connexion incident à $\delta_i = < L_i,L_j >$. Prenons δ_i comme part de la route que nous voulons construire et maintenant construisons une route de τ_i à $\tau_{i+1 \bmod n}$. Par le lemme 4.3 il y a deux routes $\pi 1$ et $\pi 2$ de τ_i à $\tau_{i+1 \bmod n}$. Soit $\pi 1$ la route dont aucune de ses connexions est incidente à $L_{i+1 \bmod n}$. δ_i et les connexions de $\pi 1$ appartiennent à Δ.

Le cycle est formé par l'union de toutes les connexions des routes de L_i à $L_{i+1 \bmod n}$

(i = 0...n).

Fin de la preuve

THEOREME 4.1

Soit M une matrice de connexions.

Soit D un DAG construit à partir de M en suivant la méthode décrite dans la section 4.4.1.

Les étiquettes de D forment un spanning set de connexions de M.

Preuve

Nous procédons par induction structurel sur D.

CAS DE BASE: Nœuds de type FEUILLE

Soit L le père d'une feuille.

Si L est père d'une feuille cela veut dire qu'il existe un littéral L' appartenant à une clause unitaire et que L,L' sont complémentaires.

Etant donné que L' appartient à une clause unitaire, tout chemin de la matrice contient L' pourtant tout chemin qui passe par L, passe aussi par L'. Comme L et L' sont complémentaires, tout chemin qui passe par L est complémentaire.

CASES INDUCTIFS

Nœuds de type INTERNE.

Suposons que les nœuds $L_1,\ldots,L_n$ sont satisfaits.

Soit $<\delta> L < L_1,\ldots,L_n >$ le nœud interne qui nous intéresse.

Si $<\delta> L < L_1,\ldots,L_n >$ est dans le DAG, cela veut dire que M a les clauses: $\tau = \ldots \wedge L \wedge \ldots$ et $\tau' = L' \wedge L_1 \wedge \ldots \wedge L_n$.

Les chemins qui passent par L ont la forme:

$$\text{type } \pi 0 : \ldots L \ldots L' \ldots$$

$$\text{type } \pi i : \ldots L \ldots L_i \ldots \quad \forall i \in [1 \ldots n]$$

Les chemins de type $\pi 0$ sont complémentaires parce que L et L' sont complémentaires. Par hypothèse inductive chaque chemin qui passe par les L_i est complémentaire, en particulier les chemins de type πi sont complémentaires et pourtant tout chemin qui passe par L est complémentaire.

Nœuds de type BLOC

Supposons que nous avons un nœud de type BLOC qui a "n" cycles $\Delta_1,\ldots,\Delta_n$. Tous les S.C.D. de ces cycles sont satisfaits par des connexions de fuite appropriées. Chaque littéral du BLOC que n'appartient à aucun cycle est satisfait. La figure suivante montre la structure générale d'un BLOC.

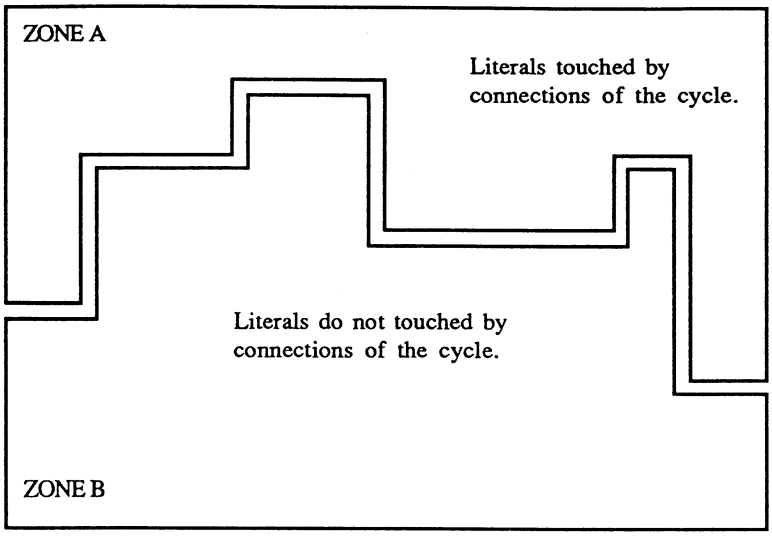


Figure 4.37

Il y a trois types différents de S.C. qui passent à travers d'un ensemble de clauses appartenant à un BLOC:

Type 1 : S.C. qui touchent seulement les littéraux de la zone A.

Type 2 : S.C. qui touchent seulement les littéraux de la zone B.

Type 3 : S.C. qui touchent les littéraux des deux zones.

Par hypothèse inductive, chaque littéral qui n'appartient à aucun cycle est satisfait, cela veut dire que tout chemin qui passe par ces littéraux est complémentaire. Les S.C. de type 2 et 3 sont donc complémentaires.

Par le lemme 4.12 il est toujours possible de construire un cycle (en général un cycle complexe) qui contient tous les littéraux de n'importe quel S.C. de la zone A. Par hypothèse inductive tout cycle simple en B est satisfait. Le cycle est donc satisfait (Lemme 4.2). Pourtant les S.C. de type 1 sont complémentaires.

Fin de la preuve

4.6. L'architecture pour le Calcul des Spanning Sets

Dans cette section on présente brièvement une architecture qui permet le calcul des Spanning Sets d'une formule en logique du premier ordre. Dans la section 4.2.4 on a déjà présenté une architecture pour des matrices sans cycles. L'architecture présentée se base sur un dag simple (décrit dans la section 4.2.3). Le dag simple a été enrichi avec nœuds de type: BLOCK,CYCLE,CL et CP (section 4.3.1) pour pouvoir traiter les cycles. Dans la section

précédent nous avons vu que les Dags construits selon l'algorithme décrit dans la section 4.4.1 représentent les Spanning Sets d'une formule en logique du premier ordre.

Les Dags enrichis groupent les clauses touchées par cycles en blocs. L'information importante est l'information sur les cycles. Pour un cycle en particulier, le Dag a deux groupes de littéraux qui forment les S.C.D. du cycle. Il y a plusieurs décisions dans notre méthode qui sont facilitées par cette organisation, à savoir:

- Dans chaque S.C.D. il faut satisfaire un littéral.

- Les règles qui déterminent quand un littéral est satisfait et celles qui déterminent quand se forment des nouveaux cycles, dépendent de cette organisation.

L'architecture que l'on propose a deux types de sous-réseaux:

- Un sous-réseau semblable au SUB-NETWORK η: N-FORMULA.

- Un sous-réseau pour gérer les cycles: N-CYCLES.

Le sous-réseau N-FORMULA$_i$ se charge de calculer les Spanning Sets. Pour cela, N-FORMULA$_i$ travaille de manière similaire au réseau SUB-NETWORK η_i, la différence consiste à vérifier si un nouveau cycle est formé lorsqu'une nouvelle connexion est ajoutée au Spanning Set. Dans le cas où un nouveau cycle est formé, les C.D. de ce cycle sont envoyés au sous-réseau N-CYCLES$_i$ qui determine s'il s'agit d'un cycle déjà connu et qui détermine à quel bloc appartient le cycle. Avec cette information, N-FORMULA se charge d'ajouter des connexions de fuite au Spanning Set.

Parlons d'abord du sous-réseau qui gère les cycles. Ce sous-réseau comprend un processus BLOCK: Θi et plusieurs processus CYCLE: $\theta_{i1},\ldots,\theta_{im}$. Θ est relié à tous les θ_{ij}. Les processus CYCLE forment une structure d'anneau et travaillent en pipeline. Le processus θ_{i1} est relié à N-FORMULA$_i$.

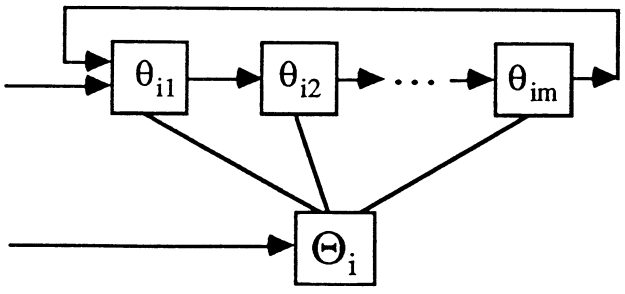


Figure 4.38

Les processus CYCLE

Ces processus se chargent de stocker l'information sur les cycles sans la répéter.

Un cycle est caractérisé par l'information suivante:

$< n_1,n_2 >$: $n_1, n_2 \in$ Nat, n_1 est l'identification du bloc[1] auquel appartient le cycle et n_2 est le nombre qui caractérisé le cycle dans le bloc n_1.

$\Pi = << L^0,\ldots,L^{n-1} >,$

$< L'^{n-1},L'^{0},\ldots,L'^{n-2} > >$: Les séquences de littéraux qui forment les S.C.D. du cycle.

"satisfaid": Si le cycle est satisfait alors il a cette étiquette.

Chaque θ_i a l'information de 0, 1 ou plusieurs cycles.

Si des nouveaux cycles sont découverts dans N-FORMULA, ses S.C.D.: π_1 et π_2 sont envoyés à θ_1.

Lorsque un θ_i reçoive les S.C.D. π_1 et π_2, θ_i vérifie si ces deux S.C. coïncident avec les S.C.D. d'un de ses cycles. Dans le cas positif π_1 et π_2 ne circulent plus. Dans le cas négatif π_1 et π_2 sont envoyés à θ_{i+1} ($i < m$).

Si π_1 et π_2 ont parcouru tous les θ_i et n'ont pas coïncidé avec les S.C.D. d'autres cycles ou si π_1 et π_2 arrivent à un θ_i vide, alors il faut établir la communication avec Θ:

- θ_i envoie à Θ l'identification des clauses du cycle (à partir des coordonnés des littéraux)
- Θ donne à ce cycle la paire de naturels qui lui identifie: $< n_1,n_2 >$
- le cycle sera stocké dans un des processus cycle.

Le processus BLOCK.

Ce processus se charge de:

- stocker l'information sur les blocs
- donner l'information sur les cycles à N-FORMULA.

---

[1] Si deux clauses τ_1 et τ_2 sont touchées par le même cycle alors τ_1 et τ_2 appartient au même block.

Pour chaque bloc, le processus BLOCK a l'information suivante:

n:	$n \varepsilon$ Nat, "n" est l'identification du bloc.
$< L_1,\ldots,L_m >$:	La liste de ses points d'entré.
$< t_1,\ldots,t_k >$:	La liste des identificateurs des clauses du bloc.
$<<p_1,q_1>\ldots>$:	$p_i \varepsilon$ Nat, p_i est l'identificateur d'un processus CYCLE.
	$q_i \varepsilon$ Nat, q_i est l'identificateur d'un cycle du bloc.
	Cette liste permet de retrouver l'information des cycles du bloc parmi les processus CYCLE.
"satisfied":	Si le bloc est satisfait alors il a cette étiquette.

Lorsque Θ reçoive la liste de clauses ($< \tau_1,\ldots,\tau_p >$) touchées par le cycle Δ qu'un θ_i l'envoi, Θ vérifie si cette liste est sous-ensemble d'un de ses cycles. Si le résultat de la vérification est positif alors Δ appartiendra à un bloc déjà existant. Si un bloc B touche l'ensemble de clauses $< \tau'_1,\ldots,\tau'_q >$ et $\{ \tau_1,\ldots,\tau_p \} \cup \{ \tau'_1,\ldots,\tau'_q \} \neq \{\}$ alors Δ est ajouté à B et maintenant B touche les clauses $\{ \tau'_1,\ldots,\tau'_q \} \cup \{ \tau_1,\ldots,\tau_p \}$. Sinon un nouveau bloc est crée pour stocker l'information de Δ.

Dans touts les cas θ_i reçoive l'identification donné à Δ.

Le processus BLOCK se communique avec N-FORMULA pour lui donner l'information sur le nouveau cycle.

Le sous-réseau N-FORMULA a la même structure que le sous réseau base (SUB-NETWORK η) décrit dans la section 4.2.4: un "supervisor process": σ_i relié à plusieurs "worker processes": $\omega_{i1},\ldots,\omega_{in}$.

Chaque N-FORMULA a des nouvelles connexions que lui permettent de se communiquer avec son N-CYCLES.

σ_i est relié à θ_{i1} pour pouvoir envoyer les cycles qui soient détectés.

Chaque ω_{ik} (k=1...n) est relié à Θ_i pour pouvoir recevoir l'information des cycles trouvés.

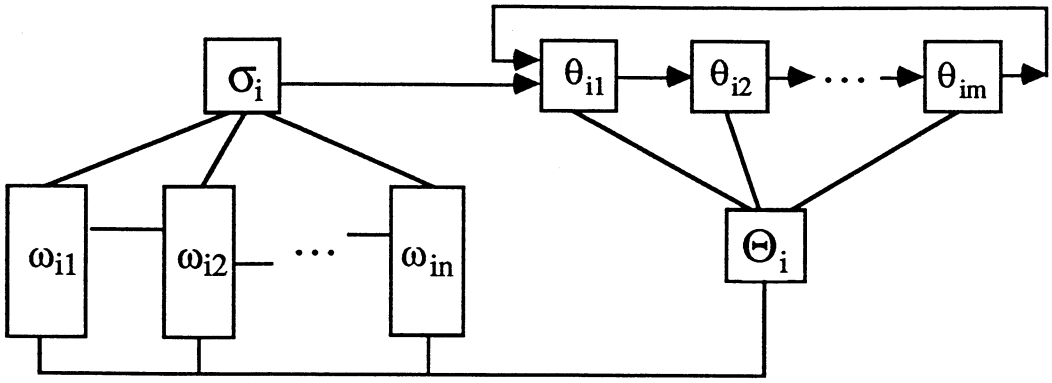


Figure 4.39

Avant de décrire les processus du réseau N-FORMULA on montre comment les cycles sont détectés.

Lorsque un "worker process": ω_{ij} ajoute une nouvelle connexion au Spanning Set et que cette connexion peut appartenir à des nouveaux cycles, ω_{ij} demande à σ_i de trouver les cycles du réseau.

Le "supervisor process" envoi une signal à tous ses "worker processes" pour commencer à chercher les cycles. Lorsque cette signal arrive à ω_{ij}, ω_{ij} prépare un nombre de messages égale au nombre de connexions du spanning set associées à ses littéraux.

Chaqu'un de ces messages initiaux est envoyé vers le littéral complémentaire du littéral qui l'origine.

Un message est constitué par l'identification des littéraux touchés dans son trajet. Le littéral qui originé le message (littéral-origine) et le littéral vers lequel le message est envoyé (littéral-destin) sont distingués de façon spécial.

Lorsque un ω_{ij} reçoive un de ces messages:

Si le littéral-origine ε ω_{ij} et le littéral-destin $\neq$ le littéral-origine alors

Le message contient l'information sur un cycle. Le cycle est envoyé à σ_i.

sinon

Si le littéral-origine ε ω_{ij} alors

Il faut détruire le message

sinon

$\forall$ L ε ω_{ij}, L $\neq$ littéral-destin,

Soient $L_1,\ldots,L_n$ les littéraux complémentaires à L qui ont été reliés à L dans ce Spanning Set .

Faire "n" copies du message et envoyer chaque copie à un "worker process" .

Lorsque il n'y a plus de messages dans le réseau σ_i arrête la recherche de cycles.

Une fois qu'un cycle Δ est inclue dans un bloc B, le processus BLOCK notifie le placement de Δ en B aux clauses et littéraux concernés.

Les "supervisor processes" et les "worker processes" ont les mêmes informations et les mêmes fonctions que ses processus homologues dans le SUB-NETWORK η plus les informations et les fonctions qui les permettent de bien gérer les cycles. L'information additionnelle nécessaire pour travailler avec les cycles est requise seulement par les "worker processes". Les "worker processes" ont besoin de stocker dans quel bloc se trouve la clause qu'ils représentent et les cycles qui la touchent, en plus ses littéraux doivent avoir l'information sur les cycles auxquels appartient.

Pour ce qui concerne les fonctions des processus elles ont été montrées dans les paragraphes ci-dessus.

Le réseau a la forme suivante :

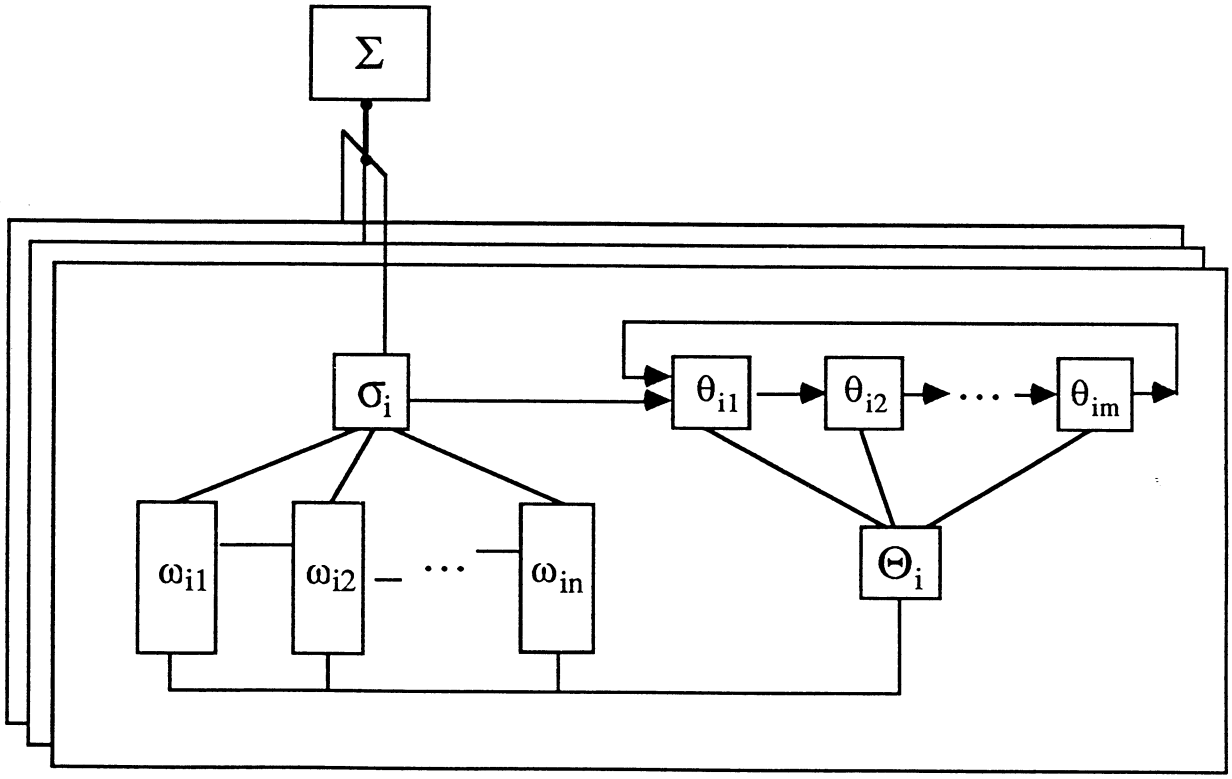


Figure 4.40

4.7. Conclusion

Pour finir le chapitre, nous voulons montrer à l'aide des exemples comment utiliser les candidats à Spanning Sets que nous calculons avec notre algorithme.

Pour une matrice sans cycles telle que :

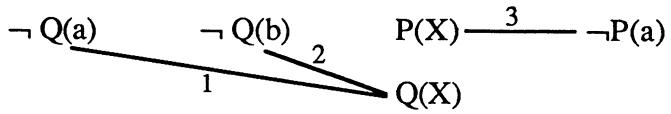


Figure 4.41

les candidats à Spanning Sets ne contiennent pas de cycles. Pour notre exemple il y a deux ensembles : {1,3} et {2,3}. Pendant l'exécution, l'ensemble {1,3} permettra de prouver la formule mais {2,3} n'y arrivera pas. {1,3} est un véritable Spanning Set et {2,3} ne l'est pas.

L'étape d'exécution peut être réalisée à la façon décrite dans [BiK87], c'est-à-dire en unifiant deux termes qui contiennent les littéraux touchés par les connexions du candidat à Spanning Set. Pour l'ensemble {1,3} ces deux termes sont les suivantes :

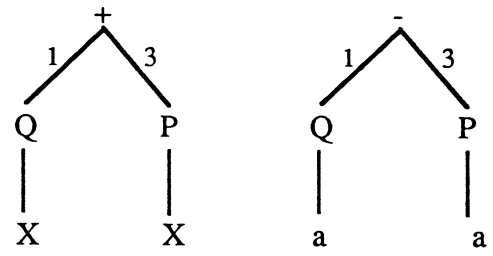


Figure 4.42

Le terme de gauche : tg contient les littéraux positifs touchés par les connexions de l'ensemble {1,3} et le terme de droite : td contient les littéraux négatifs. Si la connexion $\delta_i = <L_i, L'_i>$ alors

$$tg = + (t_1,\ldots,t_{i-1},L_i,t_{i+1},\ldots,t_n) \quad \text{et}$$

$$td = - (s_1,\ldots,s_{i-1},,L'_i,s_{i+1},\ldots,s_n)$$

Pour une matrice sans cycles, l'architecture proposée dans la section 4.2.4 est capable de trouver tous les candidats à Spanning Sets.

Analysons la matrice suivante :

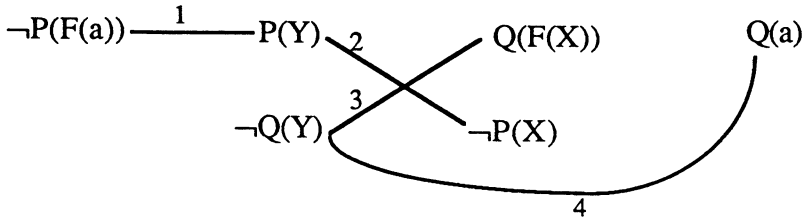


Figure 4.43

Cette matrice a deux candidats à Spanning Sets : {1,4} et {1,2,3,4} (seulement {1,2,3,4} est un Spanning Set). L'algorithme de la section 4.2.4 ne peut pas calculer le Spanning Set {1,2,3,4}. Cependant, si nous réécrivons la matrice dans sa forme équivalente :

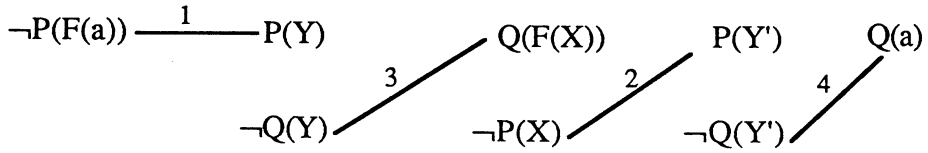


Figure 4.44

l'algorithme de la section 4.2.4 peut calculer le Spanning Set {1,2,3,4}. Cette situation se produit lorsqu'il y a des cycles (les connexions {2,3} forment un cycle) dans la matrice.

Les cycles permettent de réaliser les itérations des programmes logiques. Comme toute itération au niveau de programmation, elle a un point d'entrée et un point de sortie. Dans cette chapitre nous avons montré comment détecter les cycles et comment calculer les points d'entrée et de sortie avec les connexions de fuite.

Pour cette formule {1,2,3,4} est un véritable Spanning Set car les termes ci-dessous sont unifiables (unificateur plus générale = {Y->f(a), X->a, X -> Y', Y' -> a}.

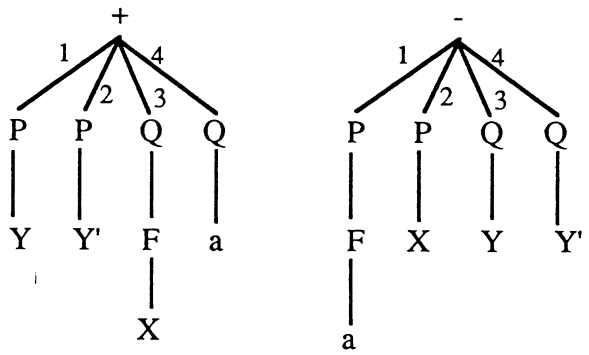


Figure 4.45

Examinons maintenant la matrice suivante :

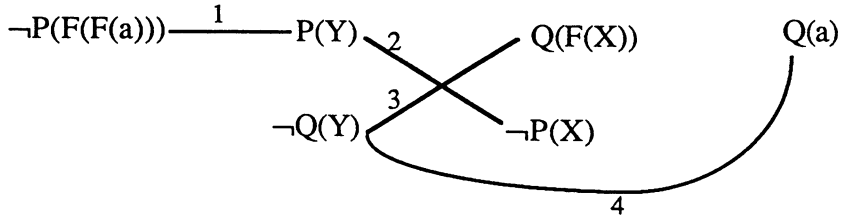


Figure 4.46

Les termes ci-dessous ne sont pas unifiables :

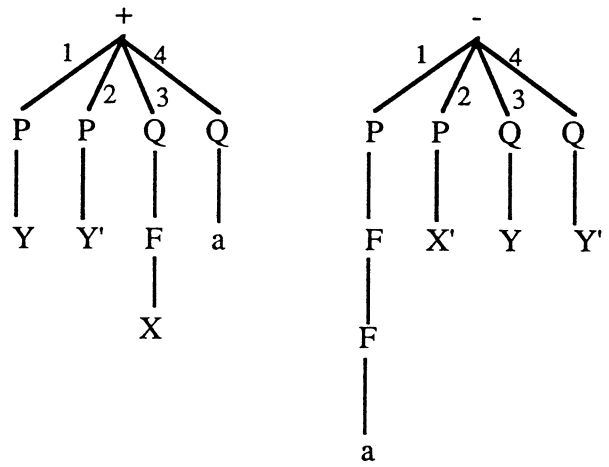


Figure 4.47

mais si nous utilisons dans la preuve deux fois les connexions 2 et 3 les termes qui résultent (voir la figure ci-dessous) sont unifiables.

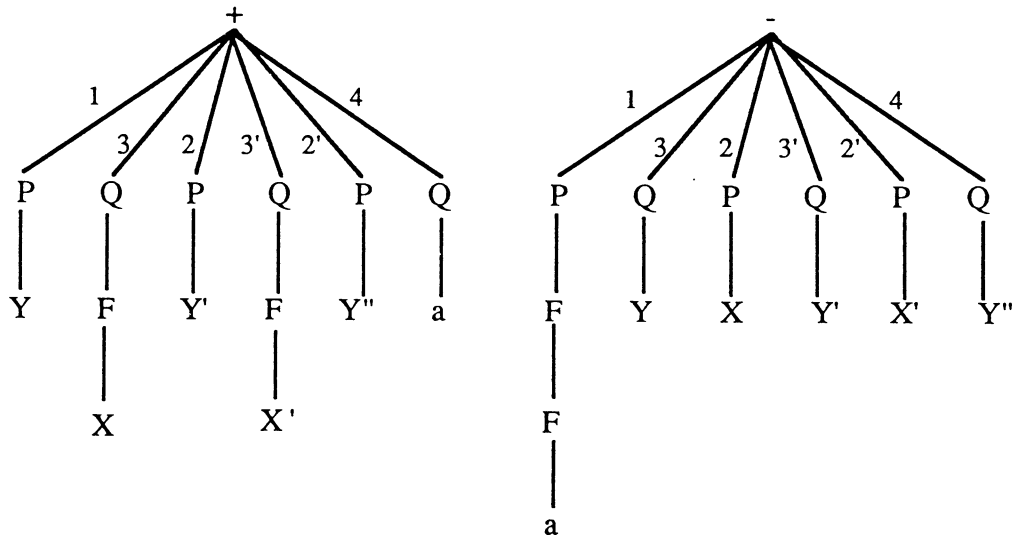


Figure 4.48

l'unificateur le plus général est : {Y->f(f(a)), X -> f(a), Y' -> f(a), X' -> a, Y" -> a }.

Le multi-ensemble de connexions {1,2,3,2,3,4} est un véritable Spanning Set. L'utilisation des connexions du cycle deux fois permet la réalisation de la preuve.

Nous n'avons pas considéré les **connexions internes** (connexions qui lient deux littéraux d'une même clause). Ces connexions apparaissent couramment dans des formules qui ont besoin de calcul pour être prouvées (par exemple le factoriel). les connexions internes sont la forme plus simple de cycle. Si dans la preuve d'une formule participe une clause qui a une connexion interne, cette connexion peut faire partie d'un véritable Spanning Set.

Examinons ensuite comment il est posible d'utiliser les techniques connues de parallélisme ET et OU pour les clauses de Horn dans la logique du premier ordre.

Si un littéral L est complémentaire aux littéraux : $L_1,\ldots,L_n$ cela veut dire qu'il y a "n" façons différentes de satisfaire L, il y a donc "n" candidats à Spanning Sets. Si la construction des Spanning Sets se fait au fur et à mesure que l'on fait l'exécution du programme, alors les techniques connues de parallélisme OU peuvent être appliquées.

L'exécution du programme signifie l'unification des sous-arbres que l'on a présenté dans cette section. L'unification des deux arbres peut être faite en parallèle de manière totale ou partielle sur les deux arbres. Ainsi l'on réalise le parallélisme ET.

CONCLUSION

Les modèles de machines à inférence parallèles connus travaillent sur les clauses de Horn et utilisent comme règle d'inférence la résolution.

Dans cette thèse, nous avons décrit d'abord une machine à inférence parallèle pour les Clauses de Horn qui exploite le parallélisme OU et qui utilise comme mécanisme d'inférence la résolution. L'adoption du langage FP2 pour la spécification de cette machine nous a permis d'utiliser le mécanisme de communication du langage pour réaliser l'opération de base dans l'inférence : l'unification. Nous avons présenté ensuite une méthode correcte basée sur la méthode de connexion, pour calculer les ensembles de paires de littéraux à résoudre (candidats à Spanning Sets) dans une formule du premier ordre. Cela représente le pas le plus difficile à franchir pour la spécification d'une machine à inférence parallèle pour la logique du premier ordre en utilisant la méthode de connexion.

Machine à Inférence Parallèle pour les Clauses de Horn basée sur la résolution.

Le modèle décrit pour les clauses de Horn construit un réseau de processus pour chaque programme écrit dans le langage des Clauses de Horn. Chacun de ces réseaux reflète la structure syntaxique du programme. En effet, le processus qui représente le but L du corps d'une clause est lié à un processus OR qui représente les clauses de tête L. Lorsqu'un but L du corps d'une clause doit être satisfait, pour activer les clauses qui peuvent résoudre L, il suffit d'envoyer un message du processus qui représente L vers le processus OR auquel il est lié. Nous n'avons donc pas besoin de chercher ces clauses dans des bases de données, comme cela serait nécessaire dans les modèles séquentiels connus.

La conception modulaire de la machine a pour avantage de permettre d'étendre la machine de manière à supporter le parallélisme OU et ET par de simples modifications.

Notre modèle appartient à la famille des modèles de granularité fine dont le désavantage principal reste dans les temps de communication. Cependant, nous avons bénéficié du mécanisme de communication de FP2 pour réaliser l'unification.

Machine à Inférence Parallèle pour la Logique du Premier Ordre basée sur la Méthode de Connexion.

L'espace de recherche de la preuve d'une formule des clauses de Horn contient uniquement les axiomes de la preuve plus la résolvante actuelle. Pour prouver une formule du premier ordre, cet espace est insuffisant. Prouver une formule consiste à choisir le littéral L à résoudre et à choisir le ou les littéraux qui peuvent le résoudre. Pour résoudre ce problème, nous travaillons avec la méthode de connexion.

La preuve d'une formule du premier ordre est effectuée en trois étapes en utilisant la Méthode de Connexion :

• Obtenir les connexions de la matrice.

• Calculer les "candidats Spanning Sets".

• Vérifier que les "candidats Spanning Sets" obtenus dans l'étape précédente sont effectivement des Spanning Sets, et qu'ils sont unifiables.

Le principal apport de cette thèse est de donner un algorithme pour le calcul des candidats Spanning Sets, c'est-à-dire l'ensemble de paires de littéraux dont la résolution peut donner une preuve.

Pour obtenir l'algorithme qui calcule les candidats Spanning Sets, nous avons d'abord décrit l'algorithme en utilisant un DAG. Puis nous avons prouvé des propriétés sur ce DAG pour aboutir à une preuve de correction de l'algorithme. Enfin nous avons présenté une version parallèle de l'algorithme.

Une fois que l'on sait choisir les littéraux à résoudre, on peut appliquer toutes les techniques connues pour exploiter les parallélismes OU et ET. La prochaine étape du travaile consiste donc à intégrer les techniques du parallélisme OU et ET dans l'algorithme d'obtention des Spanning Sets.

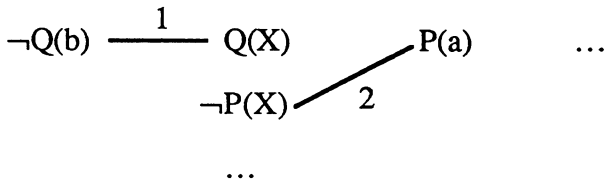
Selon notre schéma, une fois que l'on a les candidats Spanning Sets, on peut réaliser la preuve de la formule. Cependant, il est toujours possible de réaliser la preuve au fur et à mesure que l'on calcule les candidats Spanning Sets, cela offre trois avantages principaux :

1. Réduire le temps de la preuve, du fait que l'on fait la preuve de la formule au fur et à mesure que l'on ajoute ces connexions dans l'ensemble.

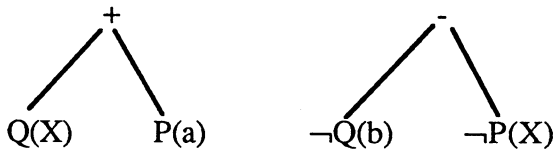
2. Arrêter la continuation du calcul de candidats Spanning Sets qui n'aboutiron pas.

Exemple de la situation 2:

Soit la matrice de connexions suivante :



Supposons que les connexions 1 et 2 appartiennent au candidat Spanning Set que l'on calcule et que le calcul de cet ensemble continue pendant que l'on réalise la partie de la preuve qui utilise ces deux connexions.



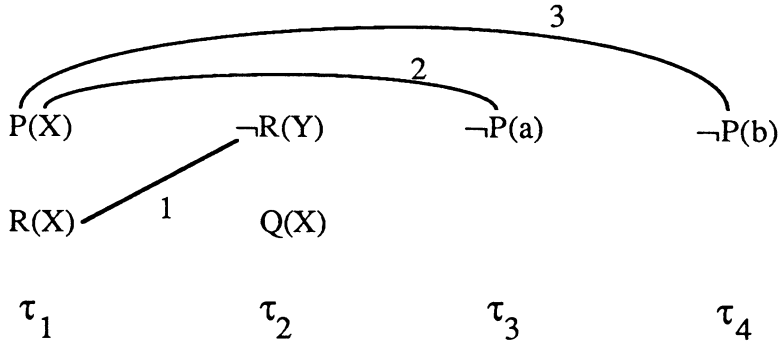
L'unification des deux arbres de la figure ci-dessus échoue et il n'y a pas de cycles dans les candidats Spanning Sets, c'est-à-dire qu'il n'y a pas d'autre façon de faire la preuve. A ce moment, il est possible de conclure que l'ensemble calculé n'est pas un candidat Spanning Set et d'empêcher que le calcul de cet ensemble ne continue.

Fin de l'exemple

3. Appliquer les techniques connues pour l'exploitation du parallélisme OU.

Exemple de la situation 3 :

Soit la matrice de connexions suivante :



Supposons que la première connexion qui est mise dans le candidat Spanning Set soit la connexion 1, les deux arbres de la figure ci-dessous peuvent donc être unifiés :

Comme conséquence de l'unification, les variables X et Y sont liées entre elles. Le littéral P(X) de la clause τ_1 peut être satisfait de deux façons différentes, donc deux candidats Spanning Sets sont possibles : {1,2,...} et {1,3,...}, l'unification de R(X) et ¬R(Y) est commune à ces deux candidats Spanning Sets et le parallélisme OU peut être exploité.

Fin de l'exemple

L'exécution de la preuve suivant l'approche d'arbres introduite par [BiK87], exige la résolution du problème de l'échec dans l'unification des arbres de manière efficace. Lorsqu'un échec se produit en unifiant les arbres, certaines liaisons des variables doivent être faites les problèmes à résoudre est alors de savoir lesquelles. Ensuite, certaines clauses devant être copiées, l'information sur les cycles de la formule permet de déterminer lesquelles (voir conclusion du chapitre 4). Si plusieurs cycles peuvent être choisis, il est possible d'exploiter le parallélisme OU. Lorsqu'un cycle est choisi, les littéraux aux extrémités des connexions du cycle peuvent être résolus en parallèle (parallélisme ET).

ANNEXE 1

Fonctions secondaires

assign: Type1 x Type2 → true

block: Dag x Node → Dag

La fonction étiquette tous les fréres du nOud avec l'étiquette "blocked".

blocked: Dag x Node → Bool

Si le nœud a l'étiquette "blocked" le résultat est "true" sinon le résultat est "false".

card: Seq-Connection → Nat

Elle donne le nombre de connexions qui a la séquence.

compl: Dag x Literal x Literal → Bool

Le résultat est "true" lorsque il v a la connexion δ = <L,L'> dans le Dag.

clause-of-literals: Connection-Matrix x Literal → Seq-Literal

Elle donne la séquence de littéraux qui sont dans la clause ou le littéral se trouve.

clause-p. Connection-Matrix x Nat → Seq-Literal

Elle donne la sequence des littéraux qui sont dans la clause indiqué par le naturel.

copy: Dag x Node x Connection → Dag x Node x Connection

Elle fait une copie du dag.

cycles: Dag x Node → Seq-Connexion

La fonction rend la Seq-Connexion qui forment les cycles qui touchent le Node. Le Node peut être du type BLOCK,CL ou INTERN.

destroy: Dag → "fail"

entries: Dag x Node → Seq-Literal

La fonction rend les littéraux qui sont points d'entre au Node du type block.

entry-point: Dag x Node → Bool

Lorsque node est le point d'entré à sa clause le résultat est "true" sinon le résultat est "false".

erase-incidents: Dag x Literal x Seq-Connexion → Seq-Connexion

La fonction efface de la Seq-Connexion celles où le littéral intervienne.

i-esime: Seq-Element x Nat → Element

Elle donne l'élement de la séquence qui est dans la possition indiqué par le naturel.

incident: Connection-Matrix x Node → Seq-Connection

Elle donne les connexions de la matrice que sont incidents à node.

in-dag: Dag x Seq-Literal → Bool

Lorsque la séquence de littéraux est dans le Dag le résultat est "false".

is: Dag x Node x Class → Bool

Lorsque node appartient à la class, le résultat est "true" autrement le résultat est "false".

links: Dag x Node → Seq-Connection

Elle donne la séquence de connexions liées à node.

link-sons: Dag x Node x Seq-Literal → Dag

Node aura comme fils les littéraux de Seq-Literal

number-of-clauses: Connection-Matrix → Nat

Elle donne le nombre de clauses qu'a la matrice de connexions.

put-label: Dag x Node x Label → Dag

Elle etiquette le node avec label.

route: Dag x Seq-Literal x Seq-Literal → Seq-Connexion

La fonction rend une route qui part d'un littéral de la premier séquence des littéraux et qui arrive à un littéral de la deuxième séquence.

satisfied: Dag x Node → Bool

Le résultat est "true" lorsque le nœud a l'étiquette "satisfied", et "false" autrement.

simple-route: DagxSeq-LiteralxSeq-Literal x Seq-Connexion→Seq-Connexion

La fonction rend une route qui part d'un littéral de la premier séquence des littéraux et qui arrive à un littéral de la deuxième séquence sans passer deux fois par les clauses et en utilisant uniquement les connexions de la Seq-Connexion.

solution: Dag → Seq-Connection

Le dag est parcuru en prennant les connexions qu'étiquettent ses nodes.

son: Dag x Node → Seq-Node

La fonction donne la séquence de nœuds fils d'un nœud.

trans: Dag x Node x Connexion → Dag

Cette fonction réalisse la transformation sur le Dag qui pernetre ajouter la Connexion au Dag et tenir en compte les nouveaux cycles qui se forment. Le Node indique la racine du sub-Dag à partir duquel il faut faire la transformation.

unit-clause: Seq-Literal → Bool

Lorsque la séquence a un littéral, le résultat est "true" sinon le résultat est "false".

ANNEXE 2

Compute-Spanning-Sets: Connection-Matrix → { Seq-Connection }+ | fail

A partir d'une matrice de connexions, la fonction donne les spanning sets de la matrice si cela est possible sinon le résultat est "fail".

Si la matrice de connexions a "m" clauses alors "m" Dags sont construits, un Dag pour chaque clause.

Compute-Spanning-Sets (M) **is**

number-of-clauses (M) = m ∧ ∀i = 1...m: clause-p (M,i) = <L_{i1},...,L_{iki}>

∀i = 1...m: | | choose-cnx (M, link-sons (ROOT,ROOT, <L_{i1},...,L_{iki}>),<>,<>)

choose-cnx: Connection-Matrix x Dag x Literal x Connexion → { Seq-Connection }$^+$ | fail

Cette fonction s'encherge de:

- Décider la terminaison (avec succès ou échec) de la construction du Dag.

- Continuer avec la construction du Dag:

en choisisant le littéral à satisfaire

ou

en invocant la fonction cnx-necessary

choose-cnx (M, D, L, δ) **is**

Terminaison avec succès

$$\frac{\exists\ \text{ROOT} \in D\text{: is (D,ROOT,root)} \wedge \text{satisfied (D,ROOT)}}{\text{solution (D)}}$$

Terminaison avec échec:

1.- La racine n'est pas satisfaite

et

2.- Tout littéral dans le Dag est bloqué ou a une connexion associé.

$$\frac{\begin{array}{c}\exists\ \text{ROOT} \in D\text{: is (D,ROOT,root)} \wedge \text{not satisfied (D, ROOT)} \wedge \\ \forall\ X \in D\text{: is (D,X, intern)} \wedge \\ (\ \text{links (D,X)} \neq \{\} \vee \text{entry-point (D,X)} \vee \text{blocked (D,X)}\ )\end{array}}{\text{destroy (D)}}$$

Il faut choisir un littéral du Dag et le satisfaire. Lorsque le littéral appartient à un sub-chemin dangéreux π, il faut bloquer les autres littéraux de π.

$$L = <> \wedge \exists CP \in D: is(D,CP,cp) \wedge \exists X \in D: is(D,X,intern) \wedge links(D,X) = \{\} \wedge not\ son(D,CP) = <...X...>$$

$$\forall i = 1...card(incident(M,X)): \mid\mid choose\text{-}cnx(M,D,X,i\text{-}esime(incident(M,X),i))$$

$$L = <> \wedge \exists CP \in D: is(D,CP,cp) \wedge \exists X \in D: is(D,X,intern) \wedge links(D,X) = \{\} \wedge son(D,CP) = <...X...> \wedge not\ blocked(D,X)$$

$$\forall i = 1...card(incident(M,X)): \mid\mid choose\text{-}cnx(M,block(D,X),X,i\text{-}esime(incident(M,X),i))$$

Le littéral L sera satisfait avec la connexion δ.

$$L \neq <>$$

$$cnx\text{-}necessary(M,D,L,\delta)$$

cnx-necessary: Connection-Matrix x Dag x Literal x Connection → { Seq-Connection }+ | fail

Cette fonction détermine si la connexion $\delta = <L,L'>$ est néccesaire pour satisfaire le littéral L.

cnx-necessary (M,D,L,<L,L'>) **is**

Si L n'appartient à aucun sub-chemin dangereux alors δ est nécessaire.

$$\frac{\text{not } (\exists\ CP \in D\text{: is } (D,CP,cp) \wedge \text{son } (D,CP) = <\ldots L \ldots>)}{\text{add-cnx } (M,D,L,<L,L'>)}$$

Dans le cas contraire (L appartient à au moins un sub-chemin dangereux), si les conditions du lemme §.2.5.1 sont vérifiées alors δ est une connexion nécessaire.

Cas A: $\delta = < L,L' >$, L et L' appartient au même sub-chemin dangéreux d'un cycle Δ.

$$\frac{\exists\ CP \in D\text{: is } (D,CP,cp) \wedge \text{son } (D,CP) = <\ldots L \ldots L' \ldots>}{\text{add-cnx } (M,D,L,<L,L'>)}$$

Cas B: $\delta = < L,L' >$, L' appartient a une clause τ' qui n'appartient à aucun cycle.

$$\frac{\begin{array}{c}\exists\ CP, X \in D\text{: is } (D,CP,cp) \wedge \\ \text{son } (D,CP)=<\ldots L \ldots> \wedge (\text{is } (D,X,\text{intern}) \vee \text{is } (D,X,\text{root})) \wedge \text{son } (D,X) = <\ldots L' \ldots>\end{array}}{\text{add-cnx } (M,D,L,<L,L'>)}$$

Cas C: $\delta = < L,L' >$, $L \in \Delta$, L' appartient a une clause τ'. Cette clause appartient au cycle Δ' mais n'appartient pas au cycle Δ. $L' \in \Delta'$.

$$\frac{\begin{array}{c}\exists\ CYCLE\_i, CYCLE\_j, CL \in D: \\ \text{is } (D, CYCLE\_i,\text{cycle}) \wedge \text{is } (D, CYCLE\_j,\text{cycle}) \wedge \text{is } (D, CL,cl) \wedge \\ \text{son } (D,CYCLE\_i) = < CP\_i1,CP\_i2 > \wedge \text{son } (D,CYCLE\_j) = < CP\_j1,CP\_j2 > \wedge \\ \exists\ k : \text{son } (D,CP\_ik) = <\ldots L \ldots> \wedge \exists\ k : \text{son } (D,CP\_jk) = <\ldots L' \ldots> \wedge \\ \text{son } (D,Cl,cl) = <\ldots L' \ldots> \wedge CYCLE\_j \in \text{cycles } (D,CL) \wedge CYCLE\_i \notin \text{cycles } (D,CL)\end{array}}{\text{add-cnx } (M,D,L,<L,L'>)}$$

Cas D: $\delta = < L,L' >$, $L \in \Delta$, L' appartient à une clause τ'. Cette clause appartient au cycle Δ' mais n'appartient pas au cycle Δ. $L' \notin \Delta'$.

$$\exists \text{ BLOCK,CYCLE\_i: is (D, CYCLE\_i,cycle) } \wedge \text{ is (D, BLOCK,block) } \wedge$$
$$\text{son (D,CYCLE\_i) = < CP\_1,CP\_2 > } \wedge \exists \text{ k: son (D,CP\_k) = < ... L ... > } \wedge$$
$$\text{son (D,BLOCK) = < ... CL ... > } \wedge \text{ is (D, CL,cl) } \wedge \text{ son (D,CL) = < ... L' ... > } \wedge$$
$$\text{CYCLE\_i} \notin \text{ cycles (D,CL) } \wedge \text{ cycles (D,L') = <>}$$

---

add-cnx (M,D,L,<L,L'>)

Si les conditions du lemme 4.2.5.1 ne se satisfasent pas alors il faut choisir une autre connexion.

not ($\exists$ CP ε D: is (D,CP,cp) $\wedge$ son (D,CP) = <...L...L'...>

---

add-cnx (M,D,L,<L,L'>)) $\wedge$
not ($\exists$ CP, X ε D: is (D,CP,cp) $\wedge$
son (D,CP)=<...L...> $\wedge$ (is (D,X,intern) $\vee$ is (D,X,root)) $\wedge$ son (D,X) = <...L'...>

---

add-cnx (M,D,L,<L,L'>)) $\wedge$
not ($\exists$ CYCLE_i, CYCLE_j, CL ε D:
is (D, CYCLE_i,cycle) $\wedge$ is (D, CYCLE_j,cycle) $\wedge$ is (D, CL,cl) $\wedge$
son (D,CYCLE_i) = < CP_i1,CP_i2 > $\wedge$ son (D,CYCLE_j) = < CP_j1,CP_j2 > $\wedge$
$\exists$ k : son (D,CP_ik) = < ... L ... > $\wedge$ $\exists$ k : son (D,CP_jk) = < ... L' ... > $\wedge$
son (D,Cl,cl) = <...L'...> $\wedge$ CYCLE_j ε cycles (D,CL) $\wedge$ CYCLE_i $\notin$ cycles (D,CL)

---

add-cnx (M,D,L,<L,L'>)
) $\wedge$
not ($\exists$ BLOCK,CYCLE_i: is (D, CYCLE_i,cycle) $\wedge$ is (D, BLOCK,block) $\wedge$
son (D,CYCLE_i) = < CP_1,CP_2 > $\wedge$ $\exists$ k: son (D,CP_k) = < ... L ... > $\wedge$
son (D,BLOCK) = < ... CL ... > $\wedge$ is (D, CL,cl) $\wedge$ son (D,CL) = < ... L' ... > $\wedge$
CYCLE_i $\notin$ cycles (D,CL) $\wedge$ cycles (D,L') = <>

---

choose-cnx (M, D, <>,<>)

add-cnx: Connection-Matrix x Dag x Literal x Connexion → { Seq-Connection }+ | fail

Cette fonction s'encherge d'ajouter la connexion $\delta = < L_i, L'_j >$ au Dag[1] . Si en ajoutant δ se forme un nouveau cycle[2] ,une transformation sur le Dag doit être fait.

add-cnx (M, D, L, <L_i,L'_j>) **is**

Selon la section 4.2.6.1: Lorsque $L_i \in \tau$ et $L'_j \in \tau'$, τ et τ' appartient au même bloc, il faut chercher des nouveaux cycles dedans le bloc.

$$\exists \text{ BLOCK} \in D: \text{is} (D,\text{BLOCK},\text{block}) \wedge$$
$$\text{son} (D,\text{BLOCK}) = < \text{CYCLE\_1},\ldots,\text{CYCLE\_m},\text{CL\_1},\ldots,\text{CL\_n} > \wedge$$
$$[\exists p: (\text{son} (D,\text{CYCLE\_p}) = < \text{CP\_p1},\text{CP\_p2} > \wedge \exists k: \text{son} (D,\text{CP\_pk}) = <\ldots L_i \ldots>)$$
$$\vee$$
$$\exists p: \text{son} (D,\text{CL\_p}) = <\ldots L_i \ldots>]$$
$$\wedge$$
$$[\exists p: (\text{son} (D,\text{CYCLE\_p}) = < \text{CP\_p1},\text{CP\_p2} > \wedge \exists k: \text{son} (D,\text{CP\_pk}) = <\ldots L'_j \ldots>)$$
$$\vee$$
$$\exists p: \text{son} (D,\text{CL\_p}) = <\ldots L'_j \ldots>]$$

---

satisfy-dag (M, trans(D,BLOCK,<Li,L'>))

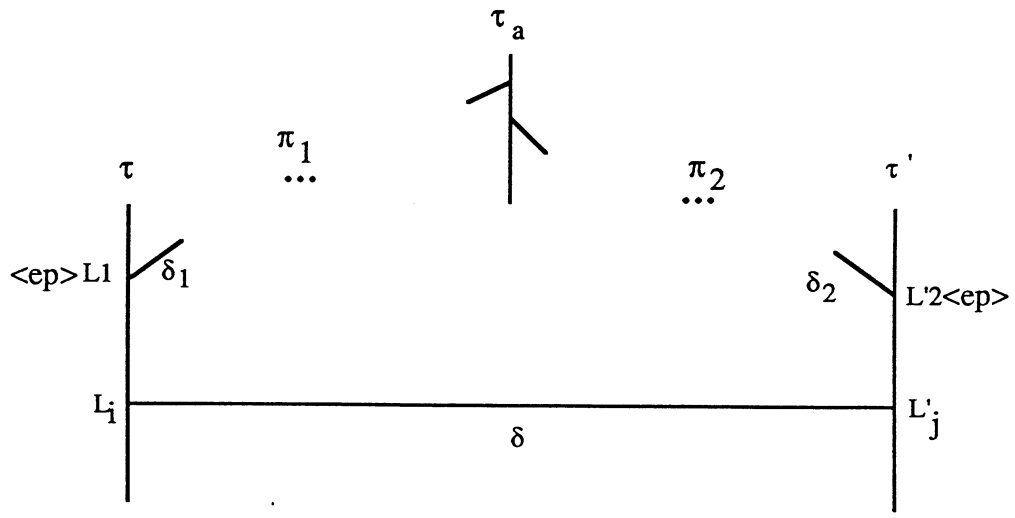
Selon la section 4.2.6.2: Lorsque $L_i \in \tau$ et $L'_j \in \tau'$, ni τ ni τ' n'appartient à aucun bloc. En dépendant si τ (τ') est ancêtre de τ' (τ) ou pas, nous avons des différentes possibilitées de formation des cycles.

Selon le lemme 4.2.6.5 si τ (τ') n'est pas l'ancêtre de τ' (τ), un nouveau cycle se forme lorsque Li ≠ L1 et L'j ≠ L'2, (L1 et L'2 sont points d'entré à τ et τ' respectivement).

---

[1] Le littéral L_i sera satisfait avec la connexion δ.

[2] selon les résultats de la section &.2.6.



$$\exists X,Y,Z \in D :$$
$$(is(D,X,intern) \vee is(D,X,root)) \wedge son(D,X) = <L_0,\ldots,L_m> \wedge$$
$$(is(D,Y,intern) \vee is(D,Y,root)) \wedge son(D,Y) = <L'_0,\ldots,L'_n> \wedge$$
$$route(D,son(D,X),son(D,Y)) = \{\} \wedge route(D,son(D,Y),son(D,X)) = \{\} \wedge$$
$$(is(D,Z,intern) \vee is(D,Z,root)) \wedge$$
$$route(D,son(D,Z),son(D,X)) \neq \{\} \wedge route(son(D,Z),son(D,Y)) \neq \{\} \wedge$$
$$not\ entry\text{-}point(D, L'_j)$$

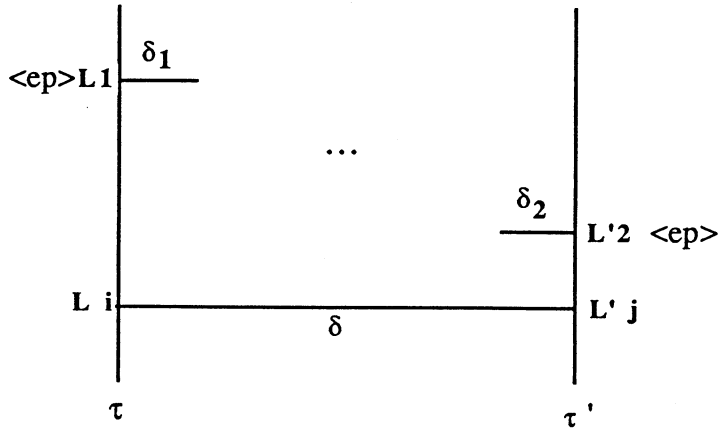
$$satisfy\text{-}dag(M, trans(D,Z,<L_i,L'_j>))$$

Si $L_i = L1$ ou $L'_j = L'2$ il n'y a pas des nouveaux cycles.

$$\exists X,Y,Z \in D :$$
$$(is(D,X,intern) \vee is(D,X,root)) \wedge son(D,X) = <L_0,\ldots,L_m> \wedge$$
$$(is(D,Y,intern) \vee is(D,Y,root)) \wedge son(D,Y) = <L'_0,\ldots,L'_n> \wedge$$
$$route(D,son(D,X),son(D,Y)) = \{\} \wedge route(D,son(D,Y),son(D,X)) = \{\} \wedge$$
$$(is(D,Z,intern) \vee is(D,Z,root)) \wedge$$
$$route(D,son(D,Z),son(D,X)) \neq \{\} \wedge route(D,son(D,Z),son(D,Y)) \neq \{\} \wedge$$
$$entry\text{-}point(D, L'_j)$$

$$satisfy\text{-}dag(M, put\text{-}label(D, L_i, L'_j))$$

Selon le lemme 4.2.6.6 si τ' est ancêtre de τ, un nouveau cycle se forme lorsque $L_i \neq L1$, (L1 et L2 sont points d'entré à τ et τ' respectivement).



$$\exists X,Y \in D :$$
$$(\text{is}(D,X,\text{intern}) \vee \text{is}(D,X,\text{root})) \wedge \text{son}(D,X) = \langle L_0,\ldots,L_m\rangle \wedge$$
$$(\text{is}(D,Y,\text{intern}) \vee \text{is}(D,Y,\text{root})) \wedge \text{son}(D,Y) = \langle L'_0,\ldots,L'_n\rangle \wedge$$
$$\text{route}(D,\text{son}(D,Y),\text{son}(D,X)) \neq \{\} \wedge$$
$$\text{not entry-point}(D, L'_j)$$

$$\text{satisfy-dag}(M, \text{trans}(D,X,\langle L_i,L'_j\rangle))$$

Si L_i = L1 ou L'_j = L2 il n'y a pas des nouveaux cycles.

$$\exists X,Y \in D :$$
$$(\text{is}(D,X,\text{intern}) \vee \text{is}(D,X,\text{root})) \wedge \text{son}(D,X) = \langle L_0,\ldots,L_m\rangle \wedge$$
$$(\text{is}(D,Y,\text{intern}) \vee \text{is}(D,Y,\text{root})) \wedge \text{son}(D,Y) = \langle L'_0,\ldots,L'_n\rangle \wedge$$
$$\text{route}(D,\text{son}(D,Y),\text{son}(D,X)) \neq \{\} \wedge$$
$$\text{entry-point}(D, L'_j)$$

$$\text{satisfy-dag}(M, \text{put-label}(D, L_i, L'_j))$$

Ensuite on suppose que la connexion à ajouter est $\delta = \langle L,L' \rangle$. τ appartient à un bloc B et τ' n'appartient à aucun bloc. Un nouveau cycle se forme lorsque les conditions du lemme 4.2.6.7 se satisfasent.

$$\exists \text{ BLOCK, X, Y, Z } \varepsilon \text{ D: is (D,BLOCK,block)} \wedge$$
$$\text{son (D,BLOCK)} = < \text{CYCLE\_1,...,CYCLE\_m,CL\_1,...,CL\_n} > \wedge$$
$$\forall i = 1...m\text{: is (D,CYCLE\_i,cycle)} \wedge \forall i = 1...n\text{: is (D,CL\_i,cl)} \wedge$$
$$\exists \text{ j: son (D,CL\_j)} = <...L...> \wedge$$
$$(\text{ is (D,X,intern)} \vee \text{is (D,X,root))} \wedge \text{son (D,X)} = <...L'...L''...> \wedge$$
$$(\text{ is (D,Y,intern)} \vee \text{is (D,Y,root))} \wedge \text{son (D,Y)} = <...L_i...> \wedge$$
$$L_i \varepsilon \text{ entries (D,BLOCK)} \wedge$$
$$\text{route (D,son(D,Z),son(D,Y))} \neq \{\} \wedge \text{route (D,son(D,Z),son(D,X))} \neq \{\} \wedge$$
$$[\ L \neq L_i \wedge \text{entry-point (L'')} \wedge \text{not entry-point (D,L')} \wedge$$
$$\text{simple-route (D,son (D,CL\_j), son (D,Y),}$$
$$\text{erase-incidents(D,L,erase-incidents (D,}L_i\text{, cycles (D,BLOCK))))} \neq \{\}$$
$$\vee$$
$$(\text{ son (D,CL\_j)} = < ...L_i,L... > \wedge L_i \neq L\ )\ ]$$

$$\text{satisfy-dag (M, trans(D,Z,<L,L'>))}$$

Si les conditions du lemme ne se satisfasent pas alors ne se forment pas des nouveaux cycles.

$$\exists \text{ BLOCK, X, Y, Z } \varepsilon \text{ D: is (D,BLOCK,block)} \wedge$$
$$\text{son (D,BLOCK)} = < \text{CYCLE\_1,...,CYCLE\_m,CL\_1,...,CL\_n} > \wedge$$
$$\forall i = 1...m\text{: is (D,CYCLE\_i,cycle)} \wedge \forall i = 1...n\text{: is (D,CL\_i,cl)} \wedge$$
$$\exists \text{ j: son (D,CL\_j)} = <...L...> \wedge$$
$$(\text{ is (D,X,intern)} \vee \text{is (D,X,root))} \wedge \text{son (D,X)} = <...L'...L''...> \wedge$$
$$(\text{ is (D,Y,intern)} \vee \text{is (D,Y,root))} \wedge \text{son (D,Y)} = <...L_i...> \wedge$$
$$L_i \varepsilon \text{ entries (D,BLOCK)} \wedge$$
$$\text{route (D,son(D,Z),son(D,Y))} \neq \{\} \wedge \text{route (D,son(D,Z),son(D,X))} \neq \{\} \wedge$$
$$\text{not [} L \neq L_i \wedge \text{entry-point (L'')} \wedge \text{not entry-point (D,L')} \wedge$$
$$\text{simple-route (D,son (D,CL\_j), son (D,Y),}$$
$$\text{erase-incidents(D,L,erase-incidents (D,}L_i\text{, cycles (D,BLOCK))))} \neq \{\}$$
$$\vee$$
$$(\text{ son (D,CL\_j)} = < ...L_i,L... > \wedge L_i \neq L\ )\ ]$$

$$\text{satisfy-dag (M, trans(D,Z,<L,L'>))}$$

τ appartient à un bloc B et τ' appartient à un bloc B'.

$$\exists \text{ BLOCK\_1,BLOCK\_2, X, Y, Z } \varepsilon \text{ D: is (D,BLOCK\_1,block) } \wedge \text{ is (D,BLOCK\_2,block) } \wedge$$
$$\text{son (D,BLOCK\_1) = < CYCLE\_11,...,CYCLE\_1m,CL\_11,...,CL\_1n> } \wedge$$
$$\forall \text{ i = 1...m: is (D,CYCLE\_1i, cycle) } \wedge \ \forall \text{ i = 1...n: is (D,CL\_1i, cl) } \wedge$$
$$\exists \text{ p: son (D,CL\_1p) = <...L...> } \wedge$$
$$\text{son (D,BLOCK\_2) = < CYCLE\_21,...,CYCLE\_2m,CL\_21,...,CL\_2n> } \wedge$$
$$\exists \text{ q: son (D,CL\_2q) = <...L'...> } \wedge$$
$$\text{(is (D, X,intern) } \vee \text{ is (D,X,root)) } \wedge \text{ son (D,X) = <...L}_i\text{...> } \wedge$$
$$\text{L}_i \ \varepsilon \text{ entries (D,BLOCK\_1) } \wedge$$
$$\text{(is (D, Y,intern) } \vee \text{ is (D,Y,root)) } \wedge \text{ son (D,Y) = <...L'j...> } \wedge$$
$$\text{L'j } \varepsilon \text{ entries (D,BLOCK\_2) } \wedge$$
$$\text{route (D,son (D,Z),son(D,X)) } \neq \{\} \wedge \text{ route (D,son (D,Z),son(D,Y)) } \neq \{\} \wedge$$
$$[\text{ L} \neq \text{Li } \vee$$
$$\text{simple-route (D, son(D,X),son(D,CL\_1p),}$$
$$\text{erase-incidents (D,L,erase-incidents (D,L}_i\text{,cycles(D,BLOCK\_1)))) } \neq \{\}]$$
$$\wedge$$
$$[\text{ L' } \neq \text{ L'j } \vee$$
$$\text{simple-route (D,son(D,Y),son(D,CL\_1q),}$$
$$\text{erase-incidents (D,L',erase-incidents (D,L'j,cycles(D,BLOCK\_2)))) } \neq \{\}]$$

$$\text{satisfy-dag (M, trans(D,Z,<L,L'>))}$$

Il n'y a pas de cycles...

$$\exists \text{ BLOCK\_1,BLOCK\_2, X, Y, Z } \varepsilon \text{ D: is (D,BLOCK\_1,block) } \wedge \text{ is (D,BLOCK\_2,block) } \wedge$$
$$\text{son (D,BLOCK\_1) = < CYCLE\_11,...,CYCLE\_1m,CL\_11,...,CL\_1n> } \wedge$$
$$\forall \text{ i = 1...m: is (D,CYCLE\_1i, cycle) } \wedge \ \forall \text{ i = 1...n: is (D,CL\_1i, cl) } \wedge$$
$$\exists \text{ p: son (D,CL\_1p) = <...L...> } \wedge$$
$$\text{son (D,BLOCK\_2) = < CYCLE\_21,...,CYCLE\_2m,CL\_21,...,CL\_2n> } \wedge$$
$$\exists \text{ q: son (D,CL\_2q) = <...L'...> } \wedge$$
$$\text{(is (D, X,intern) } \vee \text{ is (D,X,root)) } \wedge \text{ son (D,X) = <...L}_i\text{...> } \wedge$$
$$\text{L}_i \varepsilon \text{ entries (D,BLOCK\_1) } \wedge$$
$$\text{(is (D, Y,intern) } \vee \text{ is (D,Y,root)) } \wedge \text{ son (D,Y) = <...L'}_j\text{...> } \wedge$$
$$\text{L'}_j \varepsilon \text{ entries (D,BLOCK\_2) } \wedge$$
$$\text{route (D,son (D,Z),son(D,X)) } \neq \{\} \wedge \text{ route (D,son (D,Z),son(D,Y)) } \neq \{\} \wedge$$
$$\text{not [[L } \neq \text{ Li } \vee$$
$$\text{simple-route (D, son(D,X),son(D,CL\_1p),}$$
$$\text{erase-incidents (D,L,erase-incidents (D,L}_i\text{,cycles(D,BLOCK\_1)))) } \neq \{\} \text{]}$$
$$\wedge$$
$$\text{[L' } \neq \text{ L'j } \vee$$
$$\text{simple-route (D,son(D,Y),son(D,CL\_1q),}$$
$$\text{erase-incidents (D,L',erase-incidents (D, L'j,cycles(D,BLOCK\_2)))) } \neq \{\} \text{]]}$$

---

satisfy-dag (M, trans(D,Z,<L,L'>))

satisfy-dag : Connection-Matrix x Dag → { Seq-Connection }+ | fail

Cette fonction s'encherge d'étiquetter avec "satisfied" les nœuds qui vient d'être satisfaits.

satisfy-dag (M, D) **is**

La racine

$$\exists \text{ ROOT } \varepsilon \text{ D: is (D,ROOT,root)} \wedge$$
$$\text{son (D, ROOT)} = < X_1,\ldots,X_n > \wedge \forall i = 1\ldots n\text{: satisfied (D,}X_i \text{)}$$

---

choose-cnx (M, put-label (D, ROOT, "satisfied"), <>, <>)

Les nœuds interns

$$\forall X \varepsilon \text{ D: is (D,X,intern)} \wedge$$
$$\text{son (D,X)} = < X_1,\ldots,X_n > \wedge \forall i = 1\ldots n\text{: satisfied (D, }X_i \text{)}$$

---

choose-cnx (M, put-label (D, X, "satisfied"), <>, <>)

Les nœuds bloc

$$\forall \text{ BLOCK } \varepsilon \text{ D: is (D,BLOCK,block)} \wedge$$
$$\text{son (D,BLOCK)} = < \text{CYCLE\_1},\ldots,\text{CYCLE\_m,CL\_1},\ldots,\text{CL\_n} > \wedge$$
$$\forall i = 1\ldots m\text{: is (D,CYCLE\_i,cycle)} \wedge \forall i = 1\ldots n\text{: is (D, CL\_i,cl)} \wedge$$
$$\forall i = 1\ldots m\text{: satisfied (D, CYCLE\_i)} \wedge \forall i = 1\ldots n\text{: satisfied (D,CL\_i)}$$

---

choose-cnx (M, put-label (D,BLOCK, "satisfied"), <>, <>)

Les nœuds cl

$$\forall \text{ CL } \varepsilon \text{ D : is (D,CL,cl)} \wedge$$
$$\text{son (D,CL)} = < X_1,\ldots,X_n > \wedge$$
$$\forall i = 1\ldots n\text{: (entry-point (D,}X_i \text{)} \vee \text{cycles (D, }X_i \text{)} \vee \text{satisfied (D, }X_i\text{)}$$

---

choose-cnx (M, put-label (D,CL, "satisfied"), <>, <>)

Les nœuds cycle

$$\exists \text{ CYCLE } \varepsilon \text{ D: is (D,CYCLE,cycle)} \wedge \text{son (D,CYCLE)} = <\text{CP\_1,CP\_2}> \wedge \forall \text{ i = 1,2: satisfied (D,CP\_i)}$$

---

choose-cnx (M, put-label (D,CYCLE, "satisfied"), <>, <>)

Les nœudes cp

$$\forall \text{ CP } \varepsilon \text{ D: is (D,CP,cp)} \wedge \text{son (D,CP)} = < X_1,\ldots,X_n > \exists i\text{: satisfied (D, } X_i \text{)}$$

---

choose-cnx (M, put-label (D,CP, "satisfied"), <>, <>)

Les nœuds cp se satisfasent lorsque un des ses fils est satisfait avec une connexion de fugue, (lemme 4.2.5.1).

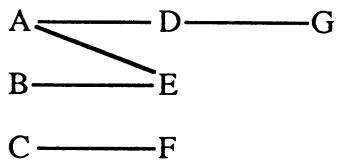
Cas A: $\delta = < L,L' >$, L et L' appartient au même sub-chemin dangéreux d'un cycle Δ.

$$\exists \text{ CP } \varepsilon \text{ D: is (D, CP,cp)} \wedge \text{son (D,CP)} < \ldots L \ldots L' \ldots > \wedge \text{compl (D,L,L')}$$

---

choose-cnx (M, put-label (D,CP, "satisfied"), <>, <>)

Nous ne pouvons pas affirmer que L et L' ont éte aussi satisfaits, exemple:



Les littéraux A et E sont conectés, son CP est donc satisfait. Les chemins CEG et AFG passent par E ou A et ne sont pas complémentaires. Ne pouvons donc pas dire que A ou E sont satisfaits.

Cas B: $\delta = < L,L' >$, L' appartient a une clause τ' qui n'appartient à aucun cycle.

$$\exists \text{ CP } \varepsilon \text{ D: is (D, CP,cp)} \wedge \text{son (D,CP)} < \ldots L \ldots > \wedge \exists \text{ X } \varepsilon \text{ D: (is (D,X,intern) } \vee \text{ is (D,X,root))} \wedge \text{son (D,X)} = < L'_1,\ldots,L'_n,L' > \wedge \text{compl (D,L,L')} \wedge \forall i = 1 \ldots n\text{: satisfied (D,L'}_i\text{)}$$

---

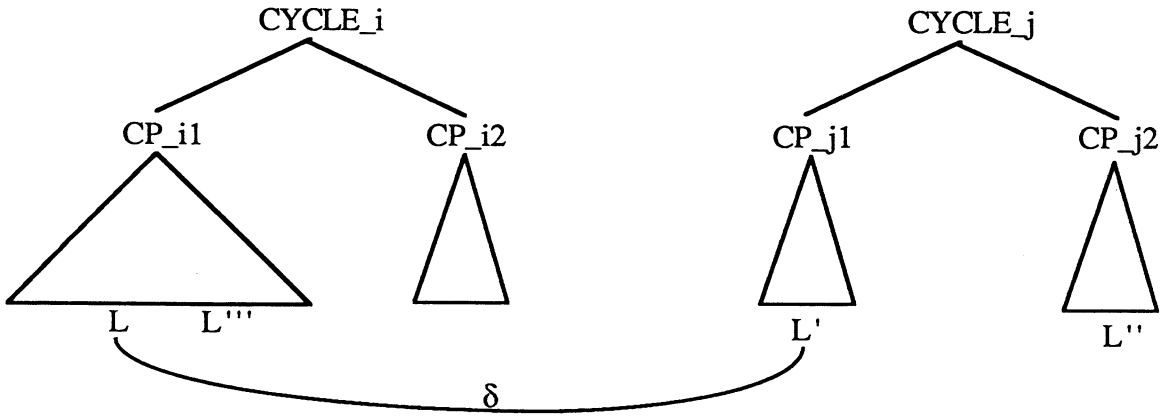
choose-cnx (M, put-label (D,L, "satisfied"), <>, <>)

L' se satisfait lorsque le BLOCK ou se trouve L est satisfait

$$\exists \text{ BLOCK} \in D : is(D,BLOCK,block) \wedge$$
$$son(D,BLOCK) = < CYCLE\_1,\ldots,CYCLE\_m,CL\_1,\ldots,CL\_n > \wedge$$
$$\exists j: son(D,CYCLE\_j) = < CP\_j1,CP\_j2 > \wedge$$
$$\forall i = 1 \ldots m: is(D,CYCLE\_i,cycle) \wedge$$
$$son(D,CP\_j1) = < \ldots L \ldots > \wedge$$
$$\exists L' \in D: is(D,L',intern) \wedge compl(D,L,L') \wedge satisfied(D,BLOCK)$$

$$choose\text{-}cnx(M, put\text{-}label(D,L', \text{"satisfied"}), <>, <>)$$

Cas C: $\delta = < L,L' >$, $L \in \Delta$, L' appartient a une clause τ'. Cette clause appartient au cycle Δ' mais n'appartient pas au cycle Δ. $L' \in \Delta'$.



Le CP_i1 est satisfait lorsque:

CP_j2 est déjà satisfait

ou

Il y a un littéral appartenant à CP_i1 qui est complémentaire à L".

$$\exists CYCLE\_i, CYCLE\_j \in D: is(D, CYCLE\_i,cycle) \wedge is(D, CYCLE\_j,cycle) \wedge$$
$$son(D,CYCLE\_i) = < CP\_i1,CP\_i2 > \wedge son(D,CYCLE\_j) = < CP\_j1,CP\_j2 > \wedge$$
$$\exists k_1: son(D,CP\_ik_1) = < \ldots L \ldots L''' \ldots > \wedge \exists k_2: son(D,CP\_jk_2) = < \ldots L' \ldots > \wedge$$
$$son(D,CP\_jk_2+1 \bmod 2) = < \ldots L'' \ldots > \wedge$$
$$compl(D, L,L') \wedge son(D,Cl,cl) = < \ldots L' \ldots > \wedge$$
$$CYCLE\_j \in cycles(D,CL) \wedge CYCLE\_i \notin cycles(D,CL) \wedge$$
$$(compl(D,L, L'') \vee compl(D,L''', L'') \vee satisfied(D,CP\_jk_2+1 \bmod 2))$$

$$choose\text{-}cnx(M, put\text{-}label(D,CP\_ik_1, \text{"satisfied"}), <>, <>)$$

Cas D: $\delta = < L,L' >$, $L \varepsilon \Delta$, L' appartient à une clause τ'. Cette clause appartient au cycle Δ' mais n'appartient pas au cycle Δ. $L' \notin \Delta'$.

$$\exists \text{ BLOCK,CYCLE } \varepsilon \text{ D: is (D,CYCLE,cycle)} \wedge \text{ is (D,BLOCK,block)} \wedge$$
$$\text{son (D,CYCLE)} = < \text{CP\_1,CP\_2} > \wedge$$
$$\exists \text{ k: son (D,CP\_k)} = < \ldots L \ldots > \wedge$$
$$\text{son (D,BLOCK)} = < \ldots CL \ldots > \wedge \text{ is (D,CL,cl)} \wedge$$
$$\text{son (D,CL)} = < L1,\ldots,Ln,L' > \wedge \text{ compl (D, L,L')} \wedge \text{ satisfied (D,BLOCK)} \wedge$$
$$\text{CYCLE\_j } \varepsilon \text{ cycles (D,CL)} \wedge \text{CYCLE\_i} \notin \text{ cycles (D,CL)}$$

$$\text{choose-cnx (M, put-label (D,CP\_k, "satisfied"), <>, <>)}$$

L appartient au BLOCK_1. L' est satisfait lorsque BLOCK_1 est satisfait.

$$\exists \text{ BLOCK\_1, BLOCK\_2 } \varepsilon \text{ D: is(D,BLOCK\_1,block)} \wedge \text{ is(D,BLOCK\_2,block)} \wedge$$
$$\text{son (D,BLOCK\_1)} = < \ldots \text{CYCLE\_1i} \ldots > \wedge \text{ is (D,CYCLE\_1i,cycle)} \wedge$$
$$\text{son (D,CYCLE\_1i)} = < \text{CP\_1i1,CP\_1i2} > \wedge$$
$$\exists k \text{ :son (D,CP\_1ik)} = < \ldots L \ldots > \wedge \text{ satisfied (D,BLOCK\_1)} \wedge$$
$$\text{son (D,BLOCK\_2)} = < \ldots \text{ CL\_2j } \ldots > \wedge \text{ is (D, CL\_2j, cl)} \wedge$$
$$\text{son (D,CL\_2j)} = < \ldots L' \ldots > \wedge \text{ compl (D,L,L')} \wedge$$
$$\text{CYCLE\_1i} \notin \text{ cycles (D,CL\_2j)} \wedge \text{ cycles (D,L')} = <>$$

$$\text{choose-cnx (M, put-label (D,L', "satisfied"), <>, <>)}$$

REFERENCES

[AsB85] Aspetsberger, K., Bayerl, S. Two parallel versions of the connection method for propositional logic on the L-Machine. In Proc. of the German Workshop on Artificial Intelligence, (Dassel/Solling, 1985), Springer-Verlag, 1985.

[Bib81] Bibel, W. On Matrices with Connections. Journal of the ACM, Vol. 28 N° 4 (Oct. 81), p. 633-645.

[Bib83] Bibel, W. Matings in matrices. Commun. ACM 26, p. 844-852, 1983.

[BiA85] Bibel, W., Aspetsberger K. A bibliography on parallel inference machines. Journal of Symbolic Computation, Vol 1 N°. 1, p. 115-118, 1985.

[BiJ86] Bibel, W., Jorrand P. (Ed.) Fundamentals of artificial intelligence. An advanced course. ACAI 85, LNCS 232, Springer-Verlag, 1986.

[Bib87] Bibel, W. Automated Theorem Proving. Second, revised edition. Vieweg, Braunschweig, 1987.

[BiK87] Bibel, W., Kurfess, F., et al. Parallel Inference Machines. Future Parallel Computers LNCS 272, Springer-Verlag, 1987, pp.185-226.

[Bor84] Borgwardt, P. Parallel Prolog using stack segments on shared memory multiprocessors. In Proceedings of the 1984 International Symposium on Logic Programming, (Atlantic City, NJ, Feb 6-9), 1984, pp. 2-11.

[Bru81] Bruynooghe, M. Solving combinatorial search problems by intelligent backtracking. Information Processing Letters 12, 1 (Feb. 1981), pp. 36-39.

[ClG81] Clark, K.L. and Gregory, S. A relational language for parallel programming. In Proceedings of the Conference on Functional Programming Languages and Computer Architecture, (Wentworth-by-the-Sea, NH, Oct. 18-22), ACM, 1981, pp. 171-178.

[ClM79] Clark, K.L. and McCabe, F. The control facilities of IC-Prolog. In Michie, D., editor, Expert Systems in the Microelectronic Age, Edinburgh University Press, 1979.

[Cha73] Chang, C.L., Lee, R. Symbolic Logic and Mathematical Theorem Proving. Academic Press, 1973.

[ChD85] Chang, J.H., Despain, A.M. Semi-intelligent backtracking of Prolog based on static dependency analysis. In Proceedings of the 1985 International Symposium on Logic Programming, (Boston, MA, July 15-18), 1985, pp. 10-21.

[CiH83] Ciepielewski, A. et Haridi, S. A formal model for OR-parallel execution of logic programss. In Information Processing 83, IFIP, 1983, pp. 299-305.

[CiH84] Ciepielewski, A. et Haridi, Control activities in the OR-Parallel token machine. In Proceedings of the 1984 International Symposium on Logic Programming, (Atlantic City, NJ, Feb 6-9), 1984, pp. 49-57.

[CiH86] Ciepielewski, A. Hausman, B. Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs. In Proceedings 1986 Symposium on Logic Programming. (Utah, Sept), 1986, pp. 246-257.

[ClA88] Clocksin, W.F., Alshawi, H. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. New Generation Computing 5, (1988), pp. 361-376.

[Con87] Conery, J.S. Parallel Execution of Logic Programs. Kluwer Academic Publishers, 1987.

[CoK81] Conery, J.S. Kibler, D.F. Parallel interpretation of logic programs. In Proceedings of the Conference on Functional Programming Languages and Computer Architecture, (Wentworth-by-the-Sea, NH, Oct. 18-22), ACM, 1981, pp. 163-170.

[CoK85] Conery, J.S. Kibler, D.F. AND Parallelism and Nondeterminism in Logic Programs. New Generation Computing 3, (1985), pp. 43-70.

[Cra85] Crammond, J.A. A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages. IEEE Transactions on Computers, Vol. c-34, N° 10, October 1985.

[DeG84] DeGroot, D. Restricted AND-parallelism. In Proceedings of the International Conference on Fifth Generation Computer Systems, (Tokyo, Japan), 1984, pp. 471-478.

[DeG88] DeGroot, D. A Technique for Compiling Execution Graph Expressions for Restricted And-Parallelism in Logic Programs. Journal of Parallel and Distributed Computing, Vol. 5, N° 5, October 1988.

[FNM82] Furukawa, K., Nitta, K., Matsumoto, Y. Prolog interpreter based on concurrent programming. In Proceedings of the First International Logic Programming Conference, (Faculté des Sciences de Luminy, Marseille, France, Sept.), 1982, pp. 38-44.

[HeN86] Hermenegildo, M. Nash, R. Efficient Management of Backtracking in AND-Parallelism. In Proceedings of the 3rd. International Conference on Logic Programming, London, 1986.

[Her86] Hermenegildo, M. In Proceedings of the 3rd. International Conference on Logic Programming, London, 1986. In Proceedings of the 3rd. International Conference on Logic Programming, London, 1986.

[HeT87] Hermenegildo, M., Tick, E. Performance Evaluation of the RAP-WAM Restricted AND-Parallel Architecture on Shared Memory Multiprocessors. Technical Report PP-085-87. MCC, (Austin, USA), March 1987.

[Her87] Hermenegildo, M. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In 4th. International Conference on Logic Programming, (Melbourne, Australia), 1987.

[HwB87] Hwang, K., Briggs, F. Computer Architecture and Parallel Processing. McGraw-Hill International Editions. Computer Science Series. Singapore. 1987.

[Iba87] Ibáñez, M.B. Parallel Implementation of the Connection Method on an Abstract FP2 Machine. ESPRIT 415, Deliverable D10. LIFIA, April, 1987.

[ISK85] Ito, N., Shimizu, H., Kishi, M., Kuno, E., and Rokusawa, K. Data-flow based execution mechanisms of parallel and Concurrent Prolog. New Generation Computing 3, 1 (1985), 15-41.

[Jor84] Jorrand P. FP2 : Functional Parallel Programming based on term substitution. In Proc. of AIMSA 84, Varna, Bulgaria, North-Holland, P. 95-112, 1984.

[Jor86] Jorrand P. Term rewriting as a basis for the design of a functional and parallel programming language. A case study : the language FP2. In ACAI 85, LNCS 232, Springer-Verlag, p. 221-276, 1986.

[Jor87] Jorrand, P. Design and implementation of a parallel inference machine for first order logic : an overview. In PARLE : Parallel Architectures and Languages Europe. Vol I : Parallel Architectures, (June 1987), 1987, pp. 434-445.

[Kow74] Kowalski, R.A. Predicate logic as programming language. In Information Processing 74, IFIPS, 1974, pp. 569-574.

[Kow79] Kowalski, R.A. Algorithm = logic + control. CACM 22, 7 (Jul. 79), pp.424-436.

[Kow88] Kowalski, R.A. The early years of logic programming. CACM 31, 1 (Jan. 1988), pp. 38-43.

[KuM86] Kumon, K. Masuzawa, H. Itashiki, A. Satoh, K. Sohma, Y. KABU-WAKE: A new parallel inference method and its evaluation. In COMPCON Spring 86, IEEE, 1986.

[LiP84] Lindstrom, G. Panangaden, P. Stream based execution of logic programs. In Proceedings of the 1984 International Symposium on Logic Programming, (Atlantic City, NJ, Feb 6-9), 1984, pp. 168-176.

[LiH85] Lipovski, G.H, Hermenegildo, M.V. B-Log: A branch and bound methodology for the parallel execution of logic programs. In Proceedings of the 1985 International Conference on Parallel Processing, (August 20-23), 1985, pp.560-567.

[LiM86] Li, P., Martin, A.J. The Sync Model for Parallel Execution of Logic Programming. In Proceedings 1986 Symposium on Logic Programming, (Sept. 1986), 1986, pp. 223-234.

[LiK88] Lin, Y.J, Kumar, V. An Execution Model for Exploiting AND-Parallelism in Logic Programs.New Generation Computing 5, (1988), pp. 393-425.

[LiW85] Li, G., Wah, B. MANIP-2: A multicomputer architecture for evaluating logic programs. In Proceedings of the 1985 International Conference on Parallel Processing, (August 20-23), 1985, pp. 123-130.

[LJA88] Lavington, S.H., Jiang, Y.J., Azmoodeh, M. Towards an implementation of Bibel's Connection Method. CSM-110, Dept. of Computer Science, University of Essex, Jan. 1988.

[Man74] Manna, Z. Mathematical Theory of Computation. Academic Press, 1974.

[Mar86] Marty, A. Placement d'un réseau de processus communicants décrit en FP2 sur une structure de grille en vue d'une implantation parallèle de ce langage. RR 606 IMAG 49 LIFIA I. Mai 1986.

[Nak84] Nakamura, K. Associative concurrent evaluation of logic programs. In Proceedings of the Second International Logic Programming Conference, (Uppsala, Sweden, July 2-6), 1984, pp. 321-331.

[Nil84] Nilsson, J.F. Formal Vienna-Definition-Method Models of PROLOG. In Implementations of PROLOG. Ellis Horwood Series Artificial Intelligence. 1984, pp. 281-308.

[PeP80] Pereira, L.M., Porto, A. An Interpreter of Logic Programs Using Selective Backtracking. Report 3/80, Dept. de Informatica, Univ. Nova de Lisboa, July 1980.

[Qui87] Quinn, M. Designing Efficient Algorithms for Parallel Computers. McGraw-Hill International Editions. Computer Science Series. Singapore 1987.

[Rob65] Robinson, J.A. A machine oriented logic based on the resolution principle. J. ACM 12, 1 (Jan. 1965), pp. 23-41.

[Rob83] Robinson, J.A. Logic Programming -Past, Present and Future-. New Generation Computing 1 (1983), pp. 107-124.

[Rog86] Rogé, S. Comparaison des comportements de processus communicants. Application au langage FP2. Doctoral Thesis, LIFIA, INPG, Grenoble 1986.

[Sch85] Schnoebelen, P. The semantics of concurrency in FP2. In Working Group on Semantics, ESPRIT Priject 415, 1985.

[Sch85a] Schnoebelen, P. μFP2 : a prototype interpreter for FP2. RR LIFIA 41, Grenoble, 1986.

[Sch85b] Schnoebelen, P. About the implementation of FP2. RR LIFIA 42, Grenoble, 1986.

[Sha83] Shapiro, E.Y. A Subset of Concurrent Prolog and its Interpreter. Tech. Rep. TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, Jan. 1983.

[Sti86] Stickel, M. An Introduction to Automated Deduction. In ACAI 85, LNCS 232, Springer-Verlag, Berlin, West Germany, pp.75-132, 1986.

[ScB86] Schwinn, B. Barth, G. An AND-parallel execution model of logic programs. In Lecture Notes in Computer Science N° 213. ESOP86, 1986.

[SoS85] Sohma, Y. Satoh, K. Kumon, K. Masuzawa, H. Itashiki, A. In IFIP TC-10 Working Conference on fifth generation computer architecture., (UMIST, Manchester, July 15-18), 1985.

[Ued85] Ueda, K. Guarded Horn Clauses. Technical Report TR-103, ICOT, Tokyo, 1985.

[UmT83] Umeyama, S. Tamura, K. A parallel execution model of logic programs. In Proceedings of the 10th International Symposium on Computer Architecture, (Stockolm, Sweden, June 13-17), 1983, pp. 349-355.

[War83] Warren, D. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, AI Center, Computer Science and Technology Division, 1983.

[Wil86] Wilson, G.V. Implementation of a connection method theorem prover for S5 modal logic. MsC thesis, Univ. of Edinburgh, Sept. 1986.

[YaN84] Yasuhara, H. Nitadori, K. ORBIT: A parallel computing model of Prolog. New Generation Computing 2, (1984), 277-288.

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de

- Monsieur Wolfgang BIBEL
- Monsieur Gérard COMYN

Mademoiselle IBANEZ-ESPIGA Maria-Blanca

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité

"Informatique"

Fait à Grenoble, le 31 Janvier 1990

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
P. VENNEREAU

Thèse de Doctorat de l'INPG.

Auteur : María-Blanca Ibáñez-Espiga

Etablissement : Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle

Titre : Inférence Parallèle et Processus Communicants pour les Clauses de Horn. Extension au premier ordre par la Méthode de Connexion.

Résumé :

Dans cette thèse, nous avons décrit une machine à inférence parallèle pour les Clauses de Horn qui exploite le parallélisme OU et qui utilise comme mécanisme d'inférence la résolution. Le modèle décrit pour les clauses de Horn part d'un réseau de processus qui représente la structure syntaxique du programme logique. Le fait d'avoir FP2 comme langage pour la specification des machines nous a permis d'utiliser le mécanisme de communication du langage pour réaliser l'opération de base dans l'inférence : l'unification.

L'espace de recherche de la preuve d'une formule des clauses de Horn contient uniquement les axiomes de la preuve plus la résolvante courrante. Pour prouver une formule du premier ordre, cet espace est insufisant. Nous avons presenté également une méthode correcte, fondée sur la Méthode de Connexion pour calculer les ensembles de paires de littéraux à résoudre dans formule du premier ordre. Cela représente le pas le plus difficile à franchir pour la spécification d'une machine à inférence parallèle pour la logique du premier ordre.

Mots clés : Parallélisme, Clauses de Horn, Inférence Parallèle, la Méthode de Connexion.