Chapitre 5

Implantation de la procédure abductive

Dans ce chapitre, nous étudions brièvement l'implantation de la procédure de preuve. Ce qui suit est dicté par les enseignements tirés de l'implantation d'un prototype de la procédure. Nous décrivons tout d'abord le cadre général de l'implantation, en particulier l'introduction du typage, puis nous présentons un algorithme incrémental et efficace pour la gestion de contraintes temporelles métriques. Pour finir, nous étudions la faisabilité d'une machine abstraite pour l'exécution de la procédure abductive.

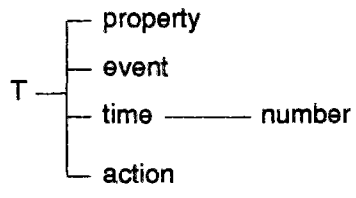
5.1 Cadre général de l'implantation

Il y a plusieurs possibilités pour l'implantation de la procédure abductive. Une première solution est d'en faire un méta-interprète au dessus de Prolog, comme c'est le cas pour le planificateur CHICA de Missiaen. Cette solution présente l'avantage d'utiliser un cadre où la résolution SLD est déja implantée, et où le parcours de l'espace de recherche est une notion naturelle. Par contre, Prolog souffre de sa trop grande simplicité, en particulier de son absence de système de typage natif et bien intégré. Nous avons pu voir, durant la description de la procédure, que beaucoup de notions devaient être adaptées à la sémantique des termes utilisés : par exemple, l'égalité ne s'interprète pas de la même manière entre deux instants, qu'entre deux événements, ou entre deux robots. Dans le premier cas, cette égalité signifie que les deux instants sont à la même position sur une droite temporelle, alors que dans le calcul d'événements, où chaque événement est représenté par un terme, l'égalité entre deux événements est équivalente à l'égalité (isomorphie) entre les deux termes.

L'introduction des types dans la programmation logique, et en particulier pour le langage Prolog, a reçu beaucoup d'attention et fait l'objet de nombreux travaux. On peut citer le système de (Mycroft & O'Keefe, 1984), dont (Lakshman & Reddy, 1991) présente une reconstruction formelle. L'inconvénient de beaucoup de ces sytèmes est l'absence de possibilités de sous-typage, car celle-ci détruit la propriété de « consistance des types ». Cette propriété signifie que l'application d'une règle d'inférence, la résolution SLD dans ce cas, à des expressions (clauses) bien formées produit une expression bien formée, ce qui dispense de la vérification des types lors de l'exécution (Deitrich & Hagl, 1988).

Il faut noter qu'une des motivations de H.J. Bürckert pour la mise au point du mécanisme général de la résolution avec contraintes était d'utiliser des systèmes comme KL-ONE (Brachman & Levesque, 1985) pour représenter les connaissances taxonomiques d'un domaine particulier, en parallèle avec des systèmes déductifs basés sur la résolution. Dans cette approche, les descriptions de typage, les relations entre les types des termes, sont des contraintes dont la satisfaisabilité est déterminée à l'aide d'un système à la KL-ONE.

Notre solution est plus proche de cette dernière que des extensions habituelles de la programmation logique par des types. De la même manière que Bürckert, la résolution SLD avec contraintes est utilisée aussi avec des contraintes de typage sur les termes des clauses. Chaque clause est donc associée à des contraintes temporelles, à des contraintes de typage et éventuellement à des contraintes de domaine. Ces contraintes sont combinées lors du calcul de résolvant et leur consistance est vérifiée en évaluant le type des termes et en le comparant à celui indiqué par la contrainte de typage. Il n'y a pas d'intéraction entre ce système de contraintes de typage et les contraintes temporelles, et la consistance globale est équivalente à la consistance de chaque type de contraintes, si l'on ne prend pas en compte les contraintes sur domaines finis.



Act : event × action
Clipped : time × property × time
Happens : event
Holds_at : property × time
Initiates : event × property
Succeeds : event
Terminates : event × property

date : event → time

FIG. 5.1 - *types et signatures pour le calcul d'événements.*

Nous définissons les informations taxonomiques à l'aide d'une hiérarchie de types, qui sont représentés par des symboles, et chaque symbole de fonction et de prédicat est associé à une signature. Les variables sont déclarées avec un type lors de l'écriture des clauses. Par exemple, pour le calcul d'événements, les types et les signatures sont définis sur la figure 5.1.

L'intérêt du typage est de pouvoir définir simplement la manière dont sont gérées les différentes contraintes simplement à partir du type des termes qui les composent. L'expression d'une contrainte de finitude du domaine peut être associée à un type et vient restreindre automatiquement toute variable déclarée de ce type, d'autres restrictions pouvant toujours être apportées au cours de la résolution. La résolution avec contraintes calcule les résolvants en réunissant les contraintes de chaque type provenant des clauses parentes dans la clause résolvante. Chaque forme de contrainte, temporelle, de typage, ou de domaine, est gérée à l'aide des algorithmes correspondants. Cependant, suivant les domaines d'interprétation mis en jeu, la consistance globale n'est pas forcément acquise dès lors que les consistance de chaque ensemble de contraintes est prouvée. En particulier, les contraintes sur domaine finis doivent être manipulées avec précaution (cf. chapitre 4, paragraphe 4.5).

5.2 Un algorithme efficace et incrémental pour la résolution de contraintes temporelles

Nous décrivons ici une étude réalisée dans le cadre d'un système de contraintes temporelles métriques entre instants en vue de son intégration dans un système déductif basé sur la procédure de preuve présentée au chapitre 4. L'objectif principal de ce travail est de fournir un ensemble complet de procédures permettant de :

1. décider de manière complète de la satisfiabilité d'un ensemble de contraintes temporelles ;
2. disposer d'un algorithme incrémental pour cette tâche ;

3. déterminer la relation minimale entre deux instants.

Le premier chapitre de ce mémoire a décrits plusieurs de ces systèmes, et nous en avons proposé des extensions dans le chapitre 2 pour les termes temporels avec constructeurs. Ici, notre motivation est principalement de prendre en compte l'information quantitative, en accordant un statut identique à tous les termes temporels utilisés, c'est à dire sans distinguer les variables, les constantes, les termes fonctionnels, etc.

En dehors des études sur CLP($\Re$), où les relations quantitatives peuvent être exprimées à l'aide d'équations et d'inéquations linéaires, et où les techniques de satisfaction sont basées sur l'élimination de Gauss et la méthode du Simplexe (Jaffar et al., 1992b), les seuls systèmes qui permettent l'expression de relations quantitatives sont basés sur des modèles analogues à celui des STP de (Dechter et al., 1991). Notre propos se place toujours dans ce cadre, mais nous proposons, sur la base d'une idée proposée dans (Dechter et al., 1991), un ensemble d'algorithmes réalisant les objectifs précités, pour un coût intéressant, puisque plus faibles que celui du plus connu des algorithmes applicables ici : PC-2.

5.2.1 Le cadre : contraintes métriques entre instants

Le cadre dans lequel nous nous plaçons est celui des contraintes métriques entre instants décrit par (Dechter et al., 1991) sous l'appellation de « Simple Temporal Problem » (STP). Ce cadre a déja fait l'objet d'une description dans le chapitre 1 de ce mémoire, aussi, nous ne rappellerons que l'essentiel.

Les contraintes manipulées dans ce formalisme lient chacune deux variables en bornant la distance entre ces deux variables. Entre deux variables x et y, la contrainte $[a, b]$ signifie $a \leq y - x \leq b$. Un ensemble de ces contraintes peut se représenter dans un graphe orienté dont les nœuds sont les variables et dont les arcs sont étiquetés par l'intervalle de la contrainte. Nous noterons un tel graphe par $G = (V, E)$ où V est l'ensemble des nœuds du graphe et E l'ensemble des arcs. La contrainte portée par l'arc (i, j) est notée T_{ij} et est donc un intervalle de la forme $[a, b]$: en pratique, il n'est pas nécessaire que l'arc inverse (j, i) soit présent dans E puisque l'on peut déterminer immédiatement T_{ji} comme étant l'intervalle $[-b, -a]$.

Un tel formalisme permet donc de représenter toutes les relations telles que :

- les symboles de relation sont $<$, $=$ ou $\leq$, respectivement traduits par les intervalles $]0, +\infty[$, $[0, 0]$ et $[0, +\infty[$;

- les instants, c'est à dire les variables, sont soit des termes « atomiques », soit de la forme $t+n$ ou $t-n$ où t est une variable et n est un nombre. Les contraintes entre éléments de cette dernière forme sont transformées à l'aide des règles

suivantes :

$$\begin{aligned}(t+n):[a,b]:t' &\rightarrow t:[a+n,b+n]:t'\\ (t-n):[a,b]:t' &\rightarrow t:[a-n,b-n]:t'\\ t:[a,b]:(t'+n) &\rightarrow t:[a-n,b-n]:t'\\ t:[a,b]:(t'-n) &\rightarrow t:[a+n,b+n]:t'\end{aligned}$$

Pour un graphe de contraintes $G = (V, E)$, on définit le *graphe des distances* comme un graphe orienté qui comporte le même ensemble V de nœuds, et tel qu'à chaque arc $[a, b]$ de x à y dans G correspond dans le graphe des distances un arc de x à y valué par b et un arc réciproque de y à x valué par $-a$ [1].

Ce domaine, au sens de la programmation logique avec contraintes (Jaffar & Maher, 1994), est un sous ensemble du domaine des équations et inéquations linéaires sur $\Re$. L'intérêt de cette remarque est que, dans un cas particulier, ce domaine vérifie la propriété dite « d'indépendance des contraintes négatives » (Lassez & McAloon, 1989), ce qui permet de traiter simplement les contraintes de la forme $t_1 \neq t_2$ comme cela a été expliqué au chapitre 2. Il est donc possible de transformer tout ensemble de contraintes positives ou négatives en un ensemble de contraintes où les seules contraintes négatives utilisent le symbole $=$. Pour cela, il suffit de réécrire les contraintes à l'aide des deux règles suivantes :

$$\begin{aligned}\neg(x < y) &\rightarrow y \leq x\\ \neg(x \leq y) &\rightarrow y < x\end{aligned}$$

Cette transformation seulement partielle des contraintes permet d'éviter la réécriture des contraintes $\neg(x = y)$ en $(x < y) \vee (y < x)$ qui donne potentiellement lieu à une explosion combinatoire, soit dans le nombre des clauses, soit lors du test de consistance des restrictions de celles-ci.

Une fois que l'on dispose d'un ensemble de contraintes où les seules contraintes négatives sont des négations d'égalités, l'inconsistance d'une contrainte $\neg(x = y)$ avec la contrainte positive (ou conjonction de contraintes positives) C se ramène à tester l'implication $C \Rightarrow (x = y)$. Comme cela a été établi précédemment, il suffit de déterminer la relation minimale entre x et y : si celle-ci est une égalité (intervalle $[0, 0]$ dans le formalisme qui nous intéresse) la contrainte est inconsistante avec C, sinon elle ne l'est pas.

5.2.2 Determination de la consistance

La première solution possible pour déterminer de manière complète la consistance est d'utiliser un algorithme de consistance de chemin par propagation de type PC-2 comme cela a été décrit dans le chapitre 1. Ce type d'algorithme a un coût en $O(n^3)$ où n est le nombre de nœuds du graphe. De plus, il est aussi possible, sur le modèle de (Loganantharaj et al., 1994) par exemple, d'en dériver une version incrémentale.

1. En général, il est inutile de construire le graphe des distances puisqu'il se déduit très simplement à partir du graphe de contraintes.

Un tel algorithme calcule aussi la relation minimale entre tous les nœuds et il est alors possible de la déterminer en temps constant par simple examen du réseau de contraintes produit.

Néanmoins, le coût en n^3 présente une croissance trop élevée pour des applications pratiques comme l'ont montré (Yampratoom & Allen, 1993). Il est donc intéressant de rechercher d'autres algorithmes permettant d'accomplir la même tache pour une complexité plus faible.

Il y a deux critères équivalents (Dechter et al., 1991) à la consistance globale d'un réseau de contraintes métriques :

1. le premier critère est que le graphe de contraintes ne doit avoir aucun circuit non valide, c'est à dire tel que la composition de tous les intervalles le long de ce circuit produit un intervalle ne contenant pas 0 ;
2. le deuxième critère est que le graphe des distances associé ne contienne aucun circuit de poids négatif. Ce deuxième critère est équivalent au premier.

Une première méthode plus efficace que PC-2 pour déterminer la consistance est d'examiner tous les cycles du graphe de contraintes ou de distances. La nécessité d'examiner *tous* les cycles, et pas seulement ceux dont on peut déterminer les éléments par recherche des composantes fortement connexes (algorithme linéaire en $O(|V|+|E|)$) fait perdre de son intérêt à cette solution. De plus, contrairement aux approches semblables dans l'algèbre d'instants (où les relations sont seulement symboliques) il n'est pas possible de détecter par ce moyen les égalités implicites entre instants.

Dechter (1991) donne un autre moyen de tester la consistance d'un ensemble de contraintes métriques. Celui-ci repose sur une condition de consistance de chemins plus faible que celle établie par l'algorithme PC-2 puisqu'elle repose sur un ordre entre les nœuds : cette condition, appelée *consistance de chemin directionnelle* est la suivante

Définition 5.1 (Consistance de chemins directionnelle) *Soit $G = (V, E)$ un graphe de contraintes métriques et $\prec$ un ordre sur les éléments de V. On note alors $V = \{1, \ldots, n\}$ suivant l'ordre $\prec$. Alors G vérifie la* consistance de chemins directionnelle, *que l'on notera $\prec$-dPC, si et seulement si pour tous nœuds i et j tels que $i \prec j$, et pour tout nœud k telque $i \prec k$ et $j \prec k$, on a :*

$$T_{ij} \subseteq T_{ik} \circ T_{kj}$$

où T_{ij} désigne l'intervalle de la contrainte de i à j et $\circ$ la composition de ces contraintes.

L'algorithme 5.1 permet d'établir cette forme de consistance. Dechter montre que cet algorithme est correct et complet pour déterminer la consistance globale

DPC(V, E) :
  pour $k = n$ a 1 **faire**
    pour tout (i, j) **tel que** $(i, k), (j, k) \in E$ et $i, j \prec k$ **faire**
      $T_{i,j} \leftarrow T_{i,j} \wedge (T_{i,k} \circ T_{k,j})$
      $E \leftarrow E \cup \{(i, j)\}$
      si $T_{ij} = \emptyset$ **alors retourner** (Echec)

Algorithme 5.1: *Algorithme DPC qui transforme un graphe (V, E) en un graphe qui satisfait la propriété de consistance de chemin directionnelle par rapport à l'ordre $\prec$. Pour faciliter la notation, les nœuds du graphe sont représentés par des entiers suivant l'ordre $\prec$.*

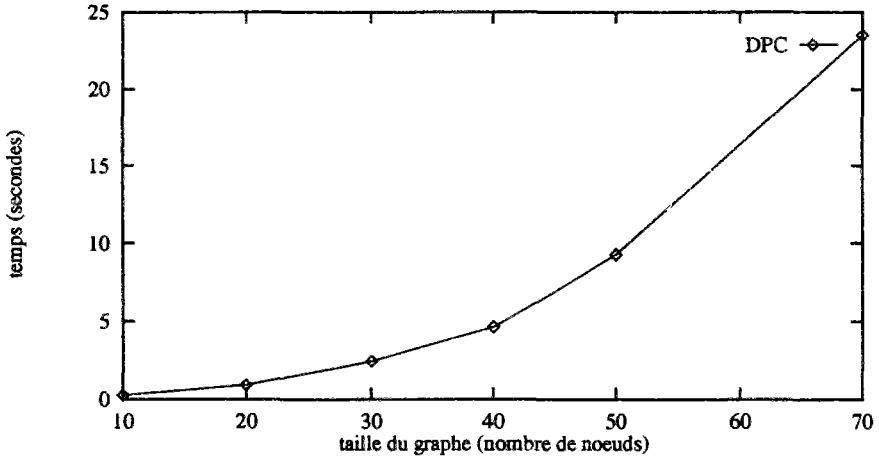


FIG. 5.2 - *Coût de la propagation totale par l'algorithme DPC en fonction du nombre de nœuds du graphe.*

d'un ensemble de contraintes. L'algorithme retourne « Echec » si un tel graphe est inconsistant sinon le graphe obtenu à la fin de l'algorithme est $\prec$-dPC.

Le premier intérêt de cet algorithme est que l'ordre $\prec$ est arbitraire. Le deuxième est sa complexité qui est linéaire en fonction de n : plus exactement, cette complexité est $O(nW^2)$ où W est le nombre maximal de parents qu'un nœud possède dans le graphe (en ne considérant que les arcs dans le sens de $\prec$). La figure 5.2 montre l'évolution du temps de calcul[2] en fonction de la taille du graphe[3].

L'algorithme DPC présente un inconvénient important : il n'est pas incrémental.

2. Les algorithmes ont été programmés avec Le_Lisp version 15.25, et les temps ont été mesurés sur le code interprété avec une station Sun IPC. Les chiffres présentés sont plus élevés que ce que donnerait une implémentation compilée puisque les tests ont montré que tous les algorithmes présentés ici étaient accélérés d'un facteur compris entre 20 et 25 par la compilation avec le compilateur modulaire Complice.

3. Tous les tests de cette partie ont été réalisés avec des graphes de contraintes consistants et générés aléatoirement.

C'est à dire que si l'on dispose de l'ensemble des contraintes élément par élément et que l'on est intéressé par la consistance après chaque ajout d'une contrainte, l'utilisation de DPC tel qu'il est donné par Dechter (1991) engendre un surcoût non négligeable car l'algorithme ne tient pas compte de la consistance directionnelle déjà établie lors des passages précédents.

Le problème est donc d'obtenir une version incrémentale de cet algorithme. Pour cela, nous allons d'abord détailler le fonctionnement de DPC. La boucle principale de cet algorithme examine une fois chaque nœud en descendant dans l'ordre $\prec$: chaque étape sur un nœud k établit la propriété $\prec$-dPC pour tous les triplets de nœuds (i, j, k) tels qu'il existe un arc de i à k et un arc de j à k dans le graphe. L'arc éventuellement créé ou modifié par l'opération de relaxation est (i, j) dont la valeur ne dépend alors que des contraintes des arcs (i, k) et (j, k). Ces observations permettent de concevoir un algorithme incrémental basé sur les principes suivants :

- l'algorithme doit fonctionner à partir d'un graphe vérifiant déja la propriété $\prec$-dPC ;
- lors de l'ajout ou de la modification d'une contrainte de l à m (que l'on peut sans perte de généralité supposer tels que $l \prec m$), tous les triplets (i, j, k) dont le nœud k est supérieur à m dans $\prec$ sont inchangés au regard de la propriété $\prec$-dPC. On peut donc envisager de ne commencer l'examen par la boucle principale qu'à partir du nœud m ;
- seuls les nœuds k qui sont à l'extrémité supérieure d'une contrainte précédemment modifiée (ou créée) par l'algorithme doivent être considérés dans la boucle principale ;
- de la même manière, seuls les parents de k tels que l'arc de ce parent à k a été modifié ou créé auront une influence sur le graphe.

Nous proposons l'algorithme 5.2 comme un algorithme incrémental pour rétablir la propriété $\prec$-dPC sur un graphe $G = (V, E)$ auquel on ajoute (ou restreint) une contrainte de i_1 à i_2. Cet algorithme utilise une file Q pour conserver les nœuds à partir desquels il faudra rétablir la propriété $\prec$-dPC. Le nœud à traiter k est choisi dans cette file comme le plus grand dans l'ordre $\prec$. Pour n'examiner que les parents de k concernés par les modifications précédentes, Π est une table qui retient pour chaque nœud j les nœuds i tels que la contrainte portée par l'arc (i, j) a été modifiée.

L'algorithme IDPC termine : en effet, lorsque l'on traite le nœud k, les seuls nœuds éventuellement rajoutés à la file Q sont strictement inférieurs à k dans l'ordre $\prec$ et de plus l'ensemble des nœuds est fini. La correction de IDPC découle de celle de l'algorithme DPC et des remarques faites précédemment. La complexité de IDPC peut être évaluée ainsi :

- la boucle principale est parcourue un nombre de fois inférieur ou égal au rang de k dans l'ordre $\prec$;

IDPC(i_1, i_2) :
  soit $Q \leftarrow \{i_2\}$; **soit** $\Pi[i_2] = \{i_1\}$
  $E \leftarrow E \cup \{(i_1, i_2)\}$
  tant que $Q \neq \emptyset$ **faire**
    soit $k \leftarrow \max_{\prec}(Q)$; $Q \leftarrow Q \setminus \{k\}$
    pour tout (i,j) **tel que** $\begin{cases} i \prec j \prec k \text{ et } (i,k),(j,k) \in E \\ \text{et (soit } i \in \Pi[k], \text{ soit } j \in \Pi[k]) \end{cases}$
      $T_{i,j} \leftarrow T_{i,j} \wedge (T_{i,k} \circ T_{k,j})$
      $E \leftarrow E \cup \{(i,j)\}$
      si $T_{ij} = \emptyset$ **alors retourner** (Echec)
      si T_{ij} est modifiee **alors**
        $Q \leftarrow Q \cup \{j\}$
        $\Pi[j] \leftarrow \Pi[j] \cup \{i\}$

Algorithme 5.2: *Algorithme IDPC qui ajoute la contrainte* (i_1, i_2) *(supposée telle que* $i_1 \prec i_2$*) au graphe* (V, E) *et qui propage celle-ci pour rétablir la consistance de chemin directionnelle par rapport à* $\prec$.

- le choix du nœud à traiter k et son extraction de la file Q peut se faire en un temps linéaire en fonction de la longueur de Q si celle-ci est implantée par une liste simple, ou mieux encore en un temps logarithmique si l'on utilise un *tas binaire* (« binary heap ») pour implanter Q ;
- si $W(k)$ est le nombre de parents de k, le nœud couramment traité par l'algorithme, l'opération de relaxation des arcs (i,j) est faite un nombre de fois inférieur où égal à $W(k)^2$;
- le test de la condition pour le choix des arcs (i,j) à relaxer peut être réalisé en temps constant si la structure Π est une table indexée par les nœuds (plus exactement par leur rang dans $\prec$) et que chaque élément de cette table est lui même un vecteur de booléens indiquant l'appartenance de chaque nœud à l'ensemble correspondant ;
- avec les choix de structure exposés plus haut, l'ajout d'un nœud à la file Q prend un temps constant si celle-ci est est représenté par une liste simple, ou un temps logarithmique dans le cas d'un tas binaire. L'ajout d'un nœud à une position de Π prend toujours un temps constant quelle que soit la structure choisie pour les éléments de Π.

Il ressort de ce qui précède que la complexité en pire cas de IDPC n'est pas meilleure que celle de DPC. En pratique cependant, le travail effectué par IDPC pour l'ajout d'une contrainte est largement inférieur à celui qu'effectue DPC pour le même travail. La figure 5.3 montre l'évolution du temps de calcul pour rétablir la propriété $\prec$-dPC après ajout d'une contrainte dans le graphe. Les contraintes ajoutées sont toutes

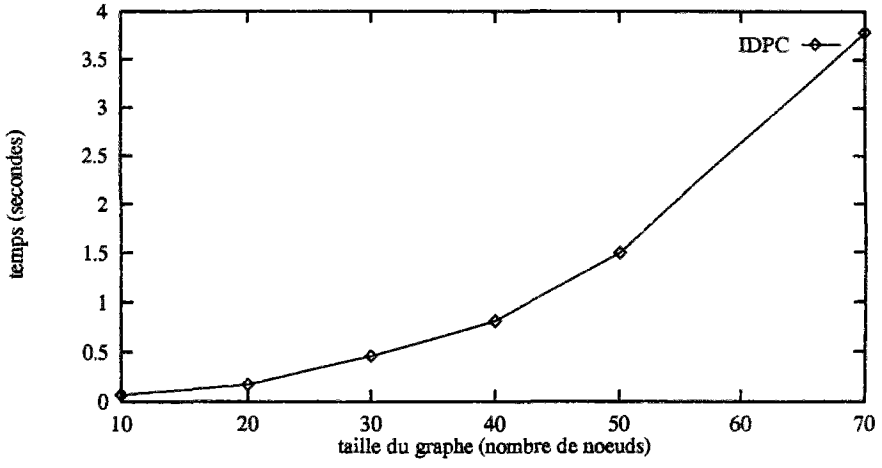


FIG. 5.3 - *Coût d'ajout et de propagation d'une contrainte par l'algorithme IDPC.*

tirées aléatoirement et sont connues pour être consistantes avec le graphe dans lequel elles sont insérées : ce cas est celui où l'algorithme de propagation effectue le plus de travail.

Pour le même travail, on peut estimer à partir de la courbe de la figure 5.2 le temps que prend l'algorithme DPC pour rétablir la propriété $\prec$-dPC : cette comparaison souligne bien le caractère incrémental de notre algorithme.

Dans le cadre d'un système basé sur la résolution avec contraintes, il est souvent nécessaire d'ajouter, non pas une, mais plusieurs contraintes à la fois. L'algorithme IDPC permet ceci : lorsque l'on souhaite ajouter à un graphe G un ensemble de contraintes dont les arcs sont $E' = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, il suffit d'initialiser la file Q de l'algorithme par $\{y_1, \ldots, y_m\}$ et d'initialiser $\Pi[y]$ par $\{x : (x, y) \in E'\}$ pour chaque $y \in Q$. Nous avons évalué la performance de IDPC pour ce type d'ajout. La figure 5.4 montre l'évolution du temps de propagation nécessaire pour rétablir la consistance de chemins directionnelle en fonction du nombre de contraintes ajoutées conjointement.

Ces derniers tests ont été réalisés sur des graphes consistants comportant vingt nœuds, connectés (comprenant donc au moins 19 arcs formant un arbre), et qui vérifiaient la propriété $\prec$-dPC. Les contraintes ajoutées étaient tirées aléatoirement et telles que les graphes finaux étaient toujours consistants. A titre de comparaison, l'agorithme DPC prend environ 4 secondes pour établir la consistance sur ces mêmes graphes. Ceci donne une indication sur la limite au delà de laquelle l'ajout incrémental et simultané de plusieurs contraintes devient équivalent à l'algorithme non incrémental : ici cette limite s'établit aux environs de 35 arcs pour 20 nœuds. Sur ces mêmes graphes, la propagation de la totalité des arcs initiaux par IDPC prend environ 5 secondes : ce qui correspond à un surcoût d'environ 25% par rapport à DPC pour la même tâche.

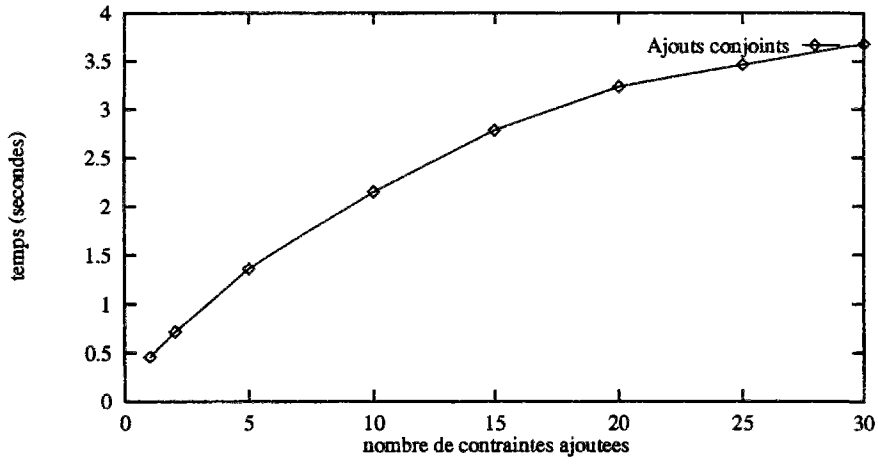


FIG. 5.4 - *Coût d'ajout et de propagation simultanée de plusieurs contraintes par l'algorithme IDPC.*

5.2.3 Détermination de la relation minimale

La relation minimale entre deux instants l et m est définie comme l'intervalle $[a, b]$ tel que a est l'opposé du poids $\delta(m, l)$ du chemin de poids minimal de m à l dans le graphe des distances du graphe de contraintes initial, et b est le poids $\delta(l, m)$ du chemin de poids minimal de l à m dans ce même graphe des distances.

Cette relation minimale peut donc être calculée par deux recherches de chemin minimal dans le graphe des distances initial. Un algorithme adéquat est par exemple celui de Bellman et Ford dont la complexité est en $O(n|E|)$. En pratique, l'on dispose d'un graphe qui vérifie la propriété de consistance de chemins directionnelle et pour lequel on peut montrer la résultat suivant sur les chemins de poids minimal :

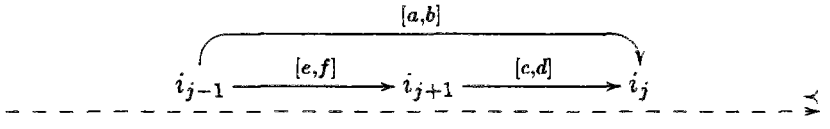
Théorème 5.1 *Soit $G = (V, E)$ un graphe de contraintes métriques consistant, et $G_d = (V, E_d)$ le graphe $\prec$-dPC obtenu par application de l'algorithme DPC sur G. Pour tous nœuds l et m, si $\delta(l, m)$ est le poids du chemin minimal de l à m dans le graphe des distances de G, alors il existe un chemin minimal de l à m dans le graphe des distances de G_d de même poids d, noté $(l, i_1, \ldots, i_k, m)$, et qui vérifie la propriété suivante :*

(i) aucun nœud i_j sur ce chemin n'est tel que $i_{j-1} \prec i_j$ et $i_{j+1} \prec i_j$.

où i_{j-1} est le prédécesseur direct de i_j sur le chemin minimal et i_{j+1} son successeur direct.

Preuve : Il suffit de montrer que, lorsqu'il existe un chemin minimal de l à m dans G[4], et qu'il ne satisfait pas la propriété (i), alors l'application de l'algorithme DPC sur G construit un chemin minimal de même poids dans le graphe des distances de G_d et tel que ce chemin vérifie alors la propriété citée.

Supposons que le chemin minimal dans G ne vérifie pas la propriété (i). Alors il existe au moins un nœud qui est supérieur (dans l'ordre $\prec$) à ses deux voisins directs dans le chemin. Soit i_j le plus grand de ces nœuds. Ce nœud i_j est tel que son prédécesseur direct i_{j-1} sur le chemin vérifie $i_{j-1} \prec i_j$ et son successeur direct i_{j+1} vérifie $i_{j+1} \prec i_j$:



Lorsque le nœud i_j est traité par l'algorithme DPC, la phase de relaxation va créer, s'il n'existe pas déja, un arc (i_{j-1}, i_{j+1}) étiqueté par

$$T_{i_{j-1},i_{j+1}} \wedge T_{i_{j-1},i_j} \circ T_{i_j,i_{j+1}} \quad = \quad [e,f] \cap [a-d, b-c]$$

et la contribution de cet arc au poids du chemin $(l, \ldots, i_{j-1}, i_{j+1}, \ldots, m)$ est :

$$\min(f, b-c)$$

Comme le chemin $(l, \ldots, i_{j-1}, i_j, i_{j+1}, \ldots, m)$ est minimal, on vérifie :

$$\min(f, b-c) = b-c$$

On a donc un nouveau chemin minimal de l à m de la forme $(l, \ldots, i_{j-1}, i_{j+1}, \ldots, m)$ et dont le poids est identique au chemin minimal initial. L'application de DPC va ainsi supprimer dans le chemin minimal tous les nœuds i_j qui sont supérieurs dans $\prec$ à leurs deux voisins directs dans le chemin minimal. L'ensemble des nœuds étant fini, et ceux-ci étant traités par ordre descendant de $\prec$, l'on est donc assuré que le chemin minimal dans le graphe final G_d vérifie la propriété (i). □

Un corollaire immédiat de ce résultat est le suivant :

Corollaire 5.1 *Soit G un graphe de contraintes métriques consistant, et G_d le graphe obtenu après application de l'algorithme DPC. Alors, pour tous nœuds l et m de ce graphe, le chemin minimal de l à m dans G_d vérifie que pour tout nœud i_j de ce chemin (autres que l et m), on a $i_j \prec \max_{\prec}(l, m)$.*

Ce résultat permet de calculer la relation minimale par deux recherches de chemin minimal qui, compte tenu du résultat précédent, peuvent se faire plus efficacement que par l'algorithme de Bellman et Ford, puisque la propagation des poids peut

4. Par abus de langage et pour alléger l'écriture, nous appellerons « chemin minimal dans G », le chemin de poids minimal dans le graphe des distances de G.

```
Min-path(s, t):
    pour tout u ∈ V faire d(u) ← ∞
    d(s) ← 0
    pour tout u = {s, ..., 1} faire
        pour tout (u, v) ∈ E tel que v ≺ u faire
            d(v) ← min(d(v), d(u) + W(u, v))
    pour tout u = {1, ..., t} faire
        pour tout (u, v) ∈ E tel que u ≺ v faire
            d(v) ← min(d(v), d(u) + W(u, v))
    retourner (d(t))
```

Algorithme 5.3: *Algorithme de calcul du poids $\delta(s,t)$ du chemin minimal du nœud s au noeud t dans le graphe des distances (V,E) d'un graphe de contraintes satisfaisant la propriété de consistance de chemin directionnelle. Les nœuds du graphe sont notés $\{1,\ldots,n\}$ dans l'ordre de $\prec$.*

se faire suivant certaines directions privilégiées. En effet, le résultat montre que le chemin minimal de s à t, notons le $(s, i_1, \ldots, i_k, t)$, ne passe que par des nœuds inférieurs dans $\prec$ à $\max_{\prec}(s,t)$ et que chacun de ces nœuds i_j (exceptés les nœuds extrémités s et t) vérifie une des trois conditions suivantes :

(1) $i_{j+1} \prec i_j \prec i_{j-1}$;

(2) $i_j \prec i_{j-1}$ et $i_j \prec i_{j+1}$, et un tel nœud, s'il existe, est unique ;

(3) $i_{j-1} \prec i_j \prec i_{j+1}$;

Nous proposons l'algorithme 5.3 qui est construit à partir d'une variante, par Yen, de l'algorithme de Bellman et Ford.

Après initialisation, la première boucle de l'algorithme propage les distances depuis s pour tous les nœuds inférieurs à s dans l'ordre $\prec$ et atteignables depuis s par un chemin qui ne comprend que des arcs dans le sens inverse de $\prec$. A la fin de cette première boucle, tous les nœuds $u \in \{1,\ldots,s\}$ tels que tous les nœuds d'un chemin minimal de s à u vérifient le cas (1) de la remarque précédente, sont étiquetés par $d(u) = \delta(s,u)$. La deuxième boucle prend en compte les cas (2) et (3) et propage ces distances à partir de ces nœuds vers les nœuds supérieurs dans l'ordre. Compte tenu de ceci, à la fin des deux boucles, tous les nœuds $u = 1,\ldots,t$ sont étiquetés par $\delta(s,u)$, et donc l'algorithme retourne $\delta(s,t)$.

La complexité de cet algorithme est linéaire en fonction de n et son coût est inférieur en pratique à celui de l'algorithme de Bellman et Ford puisque, pour un nœud donné, l'on ne considère à chaque fois que les arcs qui sont, soit dans le sens de $\prec$, soit dans le sens inverse. La figure 5.5 montre l'évolution du coût du calcul de

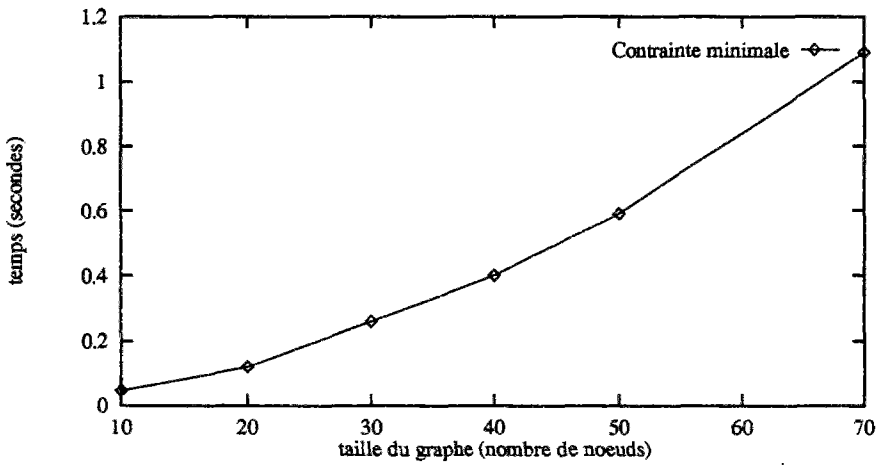


FIG. 5.5 - *Coût de la recherche de la contrainte minimale dans un graphe $\prec$-dPC.*

la contrainte minimale (c'est à dire deux recherches successives du chemin minimal) par cet algorithme en fonction de la taille du graphe.

5.3 Architecture d'une machine abstraite

Dans l'histoire de la programmation logique, la WAM (Warren Abstract Machine) (Warren, 1983) a joué un rôle primordial dans la diffusion et l'intérêt qui a été porté à la programmation logique, et plus particulièrement à Prolog. La WAM est une machine abstraite, dont la structure est particulièrement bien adaptée à l'exécution des programmes Prolog. Ces programmes sont compilés en une suite d'instructions qui réalisent l'enchaînement des buts, tout en construisant dans une zone de mémoire spéciale les termes qui seront donnés comme réponse pour les valeurs des variables du but initial. La simplicité de l'exécution de Prolog — la règle de sélection dans l'ordre, le parcours en profondeur d'abord et le retour arrière chronologique — permet donc cette compilation et les gains de temps en exécution sont particulièrement importants.

De plus, la WAM a ensuite été étendue pour prendre en compte différentes extensions de la programmation logique, en particulier la programmation logique avec contraintes avec CLP($\mathcal{R}$) (Jaffar, Michaylov, Stuckey, & Yap, 1992a), ou encore sur domaines finis avec CLP(fd) (Diaz & Codognet, 1993). On peut citer aussi la résolution SLG implantée sur une extension de la WAM (Swift & Warren, 1994).

Dans tous les cas le bénéfice de l'exécution sur une machine abstraite a compensé la rigidité introduite par la compilation et l'exécution déterministe des buts. Ces remarques motivent donc l'intérêt que nous portons à cette méthode d'implantation pour notre procédure abductive, malgré sa complexité plus grande que celle de la simple résolution SLDNF de Prolog.

Nous commençons par donner une description de la WAM originale, description qui est inspirée par celle de (Aït-Kaci, 1990), puis nous essayons d'identifier les raisons pour lesquelles la WAM permet un tel gain dans l'exécution des programmes logiques. Nous décrivons ensuite la manière dont cette architecture est étendue à la programmation logique avec contraintes. Pour finir, nous proposerons les grandes lignes d'une extension de la WAM pour la programmation logique abductive et l'implantation de notre procédure.

5.3.1 Qu'est ce que la WAM?

La WAM est une machine dont la structure générale apparaît sur la figure 5.6. Elle possède plusieurs registres spécialisés (P, CP, H, etc.), un ensemble extensible de registres dits « d'arguments », et plusieurs zones de données dont les rôles sont les suivants :

- La zone de code contient les instructions résultant de la compilation du programme logique. Chaque prédicat défini par une clause correspond à une adresse dans cette zone. Le registre P pointe sur la prochaine instruction à exécuter et CP retient l'adresse de l'instruction à exécuter après un appel réussi à un prédicat ;

- Le tas est une zone de données où sont construits dynamiquement des termes au cours de l'exécution du programme, c'est en particulier dans cette zone que sont construits les termes retournés comme réponses à une requête. Le registre H pointe sur la prochaine adresse à utiliser pour créer une structure dans cette zone. S et HB sont utilisés pendant l'exécution du programme.

- La pile est utilisée pour sauvegarder au cours de l'exécution les points de choix (pointés à l'aide du registre B) et les valeurs des registres arguments et CP avant qu'un appel à une procédure ne soit effectué.

- Le *trail* est une zone qui fonctionne comme une pile pour retenir les adresses des variables qui doivent être déliées lors des retours arrière.

- la zone PDL (*Push Down List*) est une pile utilisée par l'algorithme d'unification comme zone temporaire pour stocker des données.

L'idée principale de la WAM est qu'une clause qui définit un prédicat p/n peut être vue comme une procédure à n arguments qui lie les variables de ceux-ci à des termes construits dynamiquement dans la zone du tas. Ainsi, une clause de la forme

$$p_0(\ldots) \leftarrow p_1(\ldots), \ldots, p_n(\ldots).$$

est compilée en une suite d'instructions de la forme

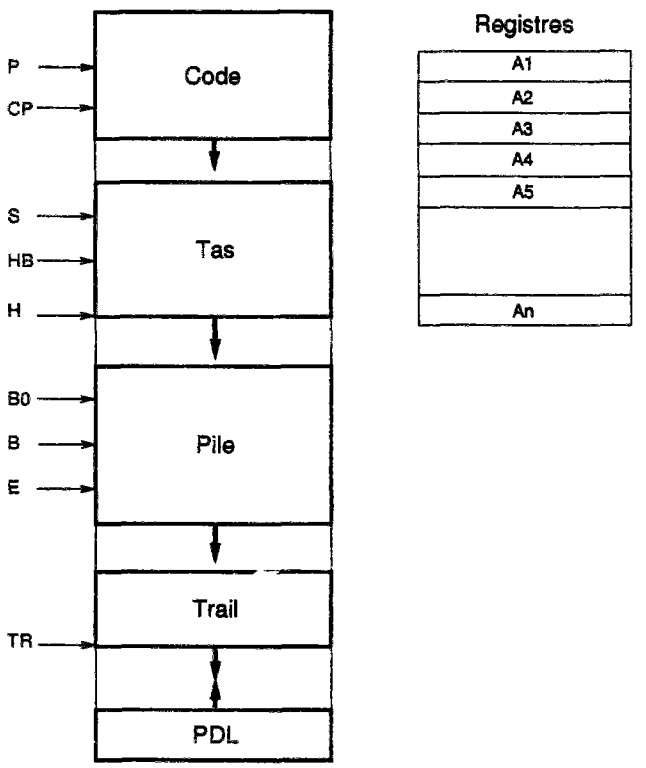


FIG. 5.6 - *Structure de la WAM.*

p_0 : `allocate`
obtenir les arguments de p_0
ranger les arguments de p_1
`call` p_1
⋮
ranger les arguments de p_n
`call` p_n
`deallocate`

Le principe du passage des arguments est d'utiliser les registres A_1, A_2, etc. pour cette tâche, en allouant ces registres dans l'ordre des arguments. L'expression « obtenir les arguments » signifie récupérer les contenus de ces registres arguments, soit par l'unification, soit sous la forme d'une liaison d'une variable si l'argument correspondant est une variable, soit en construisant une structure dans le tas matérialisant le terme argument si celui-ci est complexe (fonctionnel). L'expression « ranger les arguments » signifie placer dans les registres arguments la valeur des arguments du prédicat, qui est ensuite appelé par l'instruction `call`.

La représentation des termes dans le tas se fait comme un tableau de cellules de deux types différents : STR et REF, et complétées chacune d'une adresse dans ce tas. Les cellules STR représentent les termes complexes de la forme $f(t_1, \ldots, t_n)$: l'adresse placée dans la cellule est celle où sont stockés le nom et l'arité du symbole de fonction (ici f/n) et après cette adresse viennent les arguments du terme dans les n cellules suivantes. Les cellules REF représentent les variables, et elles ne sont pas instanciées si l'adresse associée à la cellule REF est celle de la cellule elle-même ; sinon la variable est liée (instanciée) par le terme pointé par l'adresse placée dans la cellule. Les variables identiques dans un terme sont regroupées en une seule cellule REF, c'est à dire que la forme générale de représentation des termes est celle de graphes acycliques et orientés (DAG).

Une requête, qui est un terme, est compilée en une suite d'instructions spécifiques qui construisent la structure du terme sur le tas, obtiennent les arguments par unification, placent ceux-ci dans les registres arguments, puis appellent le code de chaque prédicat de la requête. L'effet de l'exécution est de construire la liaison des variables de la requête à des termes construits dynamiquement sur le tas. Les différentes informations sauvegardées sur la pile retiennent les points de choix à l'intérieur d'un même prédicat (plusieurs clauses pour définir un prédicat), et les valeurs des registres.

5.3.2 Pourquoi la WAM est elle efficace?

Une première explication de l'efficacité de la WAM est que la résolution SLD est instanciée dans Prolog avec des règles de sélection et de choix qui sont très simples : on prend les littéraux dans l'ordre dans lequel ils apparaissent, on essaie les clauses dans l'ordre où elles sont données, etc. La structure de l'espace de recherche est celle

d'un arbre parcouru en profondeur d'abord, ce qui est très simple à faire sur une machine qui possède au moins une pile. La structure du code généré pour la WAM à partir d'un programme logique, et son exécution, suivent fidèlement cet ordre de parcours.

De plus, des instructions spécifiques sont prévues pour gérer des structures de données courantes dans les programmes Prolog comme les listes. L'unification est aussi optimisée : elle évite l'*occur-check* et utilise la structure des termes sous forme de DAG, ce qui lui donne une complexité très favorable, qui peut devenir linéaire (Baader & Siekmann, 1993).

5.3.3 Les extensions de la WAM pour la programmation logique avec contraintes

Les principes de fonctionnement sont identiques à ceux de la WAM originale. Une zone est ajoutée pour collecter les contraintes, couplée avec les algorithmes permettant de vérifier la consistance de celles-ci. Les contraintes apparaissant dans le corps des clauses sont compilées en des instructions spécifiques qui ajoutent des contraintes dans la zone concernée. Après chaque ajout des éléments d'une contrainte, la consistance est vérifiée, et un retour arrière est déclenché si celle-ci n'est pas obtenue (Jaffar et al., 1992a).

La méthodologie d'écriture des clauses impose que les contraintes soient placées en tête du corps de celles-ci, ce qui permet de les placer dans la zone concernée dès le début de l'exécution du corps de la clause. Les échecs par inconsistance des contraintes sont alors immédiatement détectés.

Par contre, les littéraux ordinaires des clauses sont compilés comme dans la WAM originale, et le passage d'arguments se réalise toujours par unification et à l'aide des registres arguments. L'opération d'unification détecte les types des variables et des termes en jeu, suivant qu'il s'agit de variables qui n'apparaissent que dans les contraintes, ou que dans les littéraux, ou encore dans les deux à la fois à l'intérieur du corps d'une clause.

Cette technique d'extension de la WAM est suffisament simple, dans son principe, à défaut de l'être dans sa réalisation, pour que son usage dans une machine abstraite adaptée à la procédure abductive soit possible. Nous allons maintenant esquisser les grandes lignes d'une telle machine.

5.3.4 Extension de la WAM pour la procédure abductive

Afin d'exécuter la procédure abductive, il faut noter les avantages et les inconvénients que les principes de la WAM imposent à l'implantation de notre procédure. Une première remarque est que la WAM est assez « rigide » : les règles de sélection sont simples et non adaptables, et le programme est une donnée statique qui n'est

pas modifiable à l'exécution. *A contrario* notre procédure suppose de pouvoir traiter différemment les littéraux suivants qu'ils sont supposables ou non, suivant que l'on retarde leur sélection, ou qu'on les sélectionne tous ensemble.

Malgré tout, toutes les dérivations se font à l'aide du même programme si l'on trouve le moyen de gérer les littéraux supposables et les hypothèses sans en faire un programme. Ceci est un avantage qui peut être utilisé.

Une solution serait de s'inspirer des techniques déja utilisées dans le compilateurs de Prolog sur la WAM, où des buts spécifiques, comme $op/2$ qui permet de définir de nouveaux opérateurs et leurs règles d'association, permettent d'influer sur la résolution. Ici, la déclaration des littéraux supposables pourrait se faire par un mécanisme analogue, tout comme la règle de sélection à utiliser, et diriger la compilation du programme en conséquence :

- les littéraux non supposables sont compilés de manière habituelle :
- suivant la règle de sélection, soit les littéraux sont réordonnées dans le corps des clauses (si l'on a une règle de sélection retardée), soit ils restent à leur place ;
- le code généré pour chaque littéral supposable tente de résoudre celui-ci avec les hypothèses existantes, et en cas d'échec de prouver leur consistance en tant qu'hypothèse.

Une nécessité est de représenter l'ensemble des hypothèses Δ de manière dynamique, ce qui nécessite une zone spécifique dont l'organisation serait analogue au tas de la WAM. Il faut alors implanter la résolution entre un littéral supposable et les hypothèses par une instruction spécifique de la WAM.

La phase de résolution avec les contraintes d'intégrité peut être préparée lors de la compilation puisque les littéraux supposables sont connus lors de cette phase. Pour une hypothèse de prédicat h/n, le code généré enchainerait les motifs suivants :

```
h:  allocate
    obtenir les arguments de h
    ranger les arguments de a1
    call a1
    ⋮
    ranger les arguments de an
    call an
    deallocate
```

pour chaque contrainte d'intégrité de la forme

$$\leftarrow a_1, \ldots, h, \ldots, a_n$$

Globalement, il faut aussi que la machine fonctionne suivant deux modes, l'un pour les dérivations abductives où la génération d'une clause vide (c'est à dire

l'exécution complète d'un appel sans échec) retourne les liaisons des variables, les contraintes et l'ensemble courant d'hypothèses, et un autre mode pour les dérivations de consistance, où l'exécution complète d'un appel déclenche l'équivalent du cas C_1 de la dérivation de consistance, et où l'occurrence d'un échec force la continuation de l'exécution sur la clause suivante parmi celles générées entre l'hypothèse et les contraintes d'intégrité.

5.4 Conclusions

Dans ce chapitre, nous avons étudié différents éléments de l'implantation de la procédure abductive décrite au chapitre 4 de ce mémoire. Nous avons choisi un cadre de programmation logique typée qui s'est révélé à l'usage suffisamment souple pour implanter et étendre notre procédure. L'introduction du typage s'est en particulier révélé une aide précieuse lors de l'ajout de différents systèmes de contraintes, en particulier les contraintes sur domaine finis qui ont été exposées dans le paragraphe 4.5 du chapitre 4.

Les idées présentées dans la dernière partie de ce chapitre sur l'architecture d'une machine abstraite ne sont que le résultat d'une étude prospective. Elles restent à valider de manière concrète. Nous pensons qu'une telle implantation, malgré la rigidité qu'elle introduit dans la procédure, peut être utile pour une classe assez large de problèmes. Les exemples en planification avec le calcul d'événements présentés dans le chapitre 4 montrent que l'on peut se contenter de règles de sélection simples et prédictibles et obtenir malgré tout des résultats significatifs.