NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Apprentissage Incrémental de Contraintes Procédurales par une Méthode Mixte

par

Johanne Morin

Thèse déposée à
l'Ecole des études supérieures et de la recherche
en vue de l'obtention de la maîtrise ès sciences appliquées en informatique.

Université d'Ottawa

ISBN 0-315-68048-2

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Je déclare être le seul auteur de cette thèse.

J'autorise l'Université d'Ottawa à prêter cette thèse à d'autres institutions ou individus, dans le but de recherches académiques.

De plus, j'autorise l'Université d'Ottawa de reproduire cette thèse par photocopies ou autrement, en sa totalité ou en parties, à la demande d'autres institutions ou individus, dans le but de recherches académiques.

Johanne Morin

Table des matières

Liste des figures

Résumé

La méthode mixte d'apprentissage présentée ici et appelée ACP, permet l'acquisition de connaissances à partir d'exemples d'un concept. Ces exemples sont représentés par une séquence d'actions et de descripteurs. L'apprentissage consiste à générer les relations qui existent entre les actions et entre les descripteurs. Ces relations sont appelées contraintes procédurales. Les contraintes, qui existent implicitement dans les exemples, sont rendues explicites par la méthode. Notre approche constitue une extension essentielle à la méthode basée sur l'explication (`Explanation-Based Learning`: EBL). Elle permet d'apprendre à partir de plusieurs exemples positifs et négatifs et intègre l'apprentissage inductif avec EBL. C'est pourquoi on l'appelle méthode mixte. De plus, c'est une méthode incrémentale qui ne conserve pas en mémoire tous les exemples appris. En fait la méthode EBL est utilisée pour apprendre le plus possible à partir d'un seul exemple. Tant qu'à la méthode SBL, elle est utilisée pour améliorer les connaissances provenant des exemples précédemment appris pour accepter le nouvel exemple. La méthode ACP est basée sur les résultats obtenus par la méthode EBL. Les connaissances apprises sont représentées par une définition du concept. Cette définition doit couvrir les exemples positifs et rejeter les exemples négatifs. Des exemples du prototype pour *prendre le métro* sont illustrés tout au long de la discussion.

1. Introduction

Les méthodes principales d'apprentissage automatique [Michalski 1983; Mitchell et al. 1986] s'appliquent surtout au problème d'acquisition d'une définition du concept. Une telle définition est habituellement représentée par une formule logique; elle permet de vérifier si une instance appartient au concept ou non. La vérification se fait par substitution des constantes, qui se trouvent dans la description de l'instance, aux variables de la définition du concept. De cette substitution résulte une expression logique qui est à son tour évaluée: la valeur logique trouvée détermine l'appartenance de l'instance au concept.

De nombreuses applications de l'apprentissage nécessitent cependant l'acquisition de relations qui existent entre différentes parties de la définition d'un concept. Considérons la situation où le concept appris décrit un processus, tel qu'un voyage, et les parties du processus sont représentées par des actions, telles que le départ, le trajet, et l'arrivée. Nous pouvons bien connaître ces actions, ainsi que le fait qu'elles servent à réaliser le processus, et en même temps ignorer des relations entre les différentes actions (par exemple que le départ nécessite le trajet). Pour clarifier la discussion, nous appelons ces relations les *contraintes procédurales*. A notre connaissance, le problème d'apprentissage de contraintes procédurales n'a pas jusqu'ici été étudié. C'est précisément le problème auquel nous consacrons cette thèse. Le résultat de notre travail est une méthode d'apprentissage *mixte* qui permet d'apprendre les différentes relations implicites dans les exemples d'entraînement. Bien évidemment, notre approche est très différente des méthodes statistiques [Diday et al. 1982] qui servent à l'étude de données. Au contraire, l'approche ACP que nous présentons ici est basée sur des connaissances, et non - comme en statistique - sur une notion de distance entre les différents exemples du concept.

Situons maintenant la méthode d'apprentissage proposée ici par rapport à des méthodes existantes. On peut regrouper les approches menant à l'acquisition de définitions du concept en trois catégories, selon qu'elles soient *inductives*, *déductives* ou *mixtes*. Les méthodes inductives permettent d'obtenir la définition du concept à partir de plusieurs exemples sans avoir de connaissances approfondies du domaine étudié. Actuellement, l'apprentissage basé sur les similarités (`Similarity-Based Learning`: SBL) constitue la méthode inductive la plus utilisée [Michalski et al. 1986]. Quant aux méthodes déductives, l'apprentissage d'un concept donné s'effectue à partir d'un seul exemple, en présence de connaissances du domaine étudié. Ces connaissances nous servent à expliquer comment l'exemple en question est une instance du concept, donc la méthode s'appelle

apprentissage par recherche d'explication (`Explanation-Based Learning`: EBL) [DeJong et al. 1986]. Il existe quatre perspectives différentes à partir de cette méthode: la généralisation (EBG) [Mitchell et al. 1986], la mémorisation par bloc, l'opérationnalisation et l'analogie justifiée [Ellman 1989]. On entend par *généralisation*: une méthode de généralisation à partir d'exemples s'effectuant en deux étapes, soient l'explication et la généralisation de l'exemple; par *mémorisation par bloc*: une conversion d'une séquence d'opérateurs en un seul "macroopérateur" ayant le même effet que la séquence entière d'opérateurs; par l'*opérationnalisation*: une reformulation de l'expression originale en termes de données et d'actions accessibles au système; par *analogie justifiée*: une mise en correspondance d'une séquence d'explications pour une situation déjà vue, analogue à la nouvelle situation. Finalement les méthodes mixtes regroupent à la fois les approches inductives et déductives. Elles combinent les caractéristiques de la méthode EBL et de l'apprentissage inductif, pour apprendre à partir d'exemples positifs et négatifs.

Les méthodes actuelles, tout en présentant certains avantages s'avèrent inadéquates pour de nouvelles avenues, tel l'apprentissage incrémental. Celui-ci fait référence à un apprentissage à partir de plusieurs exemples, mais à raison d'un à la fois. L'apprentissage y est continu et s'ajoute à la généralisation antérieure. Jusqu'à ce jour, la seule méthode reposant sur l'instruction du système à partir de plusieurs exemples sans aucune connaissances approfondies du domaine étudié est SBL. Toutefois elle s'avère dépendante du domaine d'application. Les caractéristiques utilisées pour décrire le concept ne sont pas toujours pertinentes à la définition de celui-ci et la généralisation n'est pas démontrable. De plus pour tout nouvel exemple, souvent ces méthodes (VS, AQ15 incrémental, ARCH) reprennent le processus d'apprentissage à partir du début et améliore la définition. Les inconvénients de la méthode SBL ont été corrigés par le développement de la méthode EBG. Par contre il semble irréaliste de croire qu'un seul exemple soit suffisant pour décrire à la fois les règles générales et les exceptions d'un concept. Les méthodes mixtes ont montré jusqu'à maintenant qu'il est possible de profiter des avantages et bénéfices de chacune de ces méthodes pour apprendre de façon continue.

Jusqu'à ce jour la méthode EBG est la seule méthode de généralisation à partir d'un seul exemple. Par cette méthode, il est possible d'identifier les caractéristiques pertinentes ainsi qu'une généralisation démontrable pour l'exemple étudié. La méthode inductive développée dans cette thèse utilise de tels résultats pour modifier ce qui est déjà appris du concept en fonction de ce qui est nouvellement appris par l'exemple. L'idée de base de cette

méthode, constituant la contribution de cette thèse, provient de la méthodologie employée en Vérification de Logiciels destinée à définir des cas d'essais [Myers 1979].

Pour appliquer la méthode d'apprentissage, il faut un domaine où il existe des règles préétablies sur la façon de procéder. Notre méthode utilise un domaine d'application emprunté à Alterman: *prendre le métro* [Alterman 1988]. C'est un domaine riche en connaissances, où il existe des règles préétablies sur la façon de procéder.

Notre méthode ACP d'apprentissage de contraintes procédurales est une méthode mixte: elle commence par l'explication d'un exemple d'entraînement, et ensuite combine plusieurs explications à l'aide d'un processus inductif. La méthode est incrémentale et ne requiert pas la mémoire complète des exemples précédants. Cette méthode a pour but d'apprendre une définition du concept qui couvre tous les exemples positifs et rejette tous les exemples négatifs du concept. Les chapitres 3, 4 et 5 de cette thèse décrivent les trois grandes étapes suivies pour développer une telle méthode:

1) apprendre à partir des exemples positifs (chapitre 3);

2) ajouter une théorie négative pour expliquer et apprendre à partir d'exemples négatifs (chapitre 4);

3) ajouter des connaissances générales pour raffiner l'apprentissage, lorsqu'un exemple négatif est explicable par la théorie positive (chapitre 5).

L'apprentissage incrémental de contraintes procédurales à partir d'exemples positifs a été implanté en Prolog. Cet prototype nous a servi à réaliser une application intéressante: la génération de données pour tester les spécifications du protocole de communication: ABP (Alternating Bit Protocol) [Geldrez et al. 1989].

Le chapitre 2 présente une introduction à la méthode EBG. Le chapitre 3 présente les phases déductive et inductive de l'apprentissage incrémental pour les exemples positifs[2]. Le chapitre 4 présente l'apprentissage incrémental à partir d'exemples négatifs rendu possible avec l'ajout d'une théorie négative du domaine au système. Le chapitre 5

[2] Un résumé de ce chapitre a paru dans [Matwin et al. 1989]

montre comment il est possible d'apprendre d'avantage à partir des exemples négatifs, avec l'ajout de connaissances générales. Le chapitre 6 présente la revue de littérature. Le chapitre 7 présente les résultats et la conclusion. L'appendice A présente tous les exemples d'entraînement utilisés pour cette thèse et l'appendice B présente le programme Prolog et quelques exemples.

2. Généralisation basée sur l'explication (EBG)

Nous allons présenter EBL sous sa forme de réalisation la plus répandue, celle de EBG (`Explanation-Based Generalization`). A notre connaissance, la méthode EBG a donné les meilleurs résultats concernant l'apprentissage à partir d'un seul exemple. La méthode EBG est constituée de deux étapes. La première étape génère une explication en fonction de l'exemple donné. Le but de celle-ci est de saisir le principe général d'opération de l'exemple. L'explication est générée par le système à l'aide de connaissances du domaine qui lui sont fournies. La seconde étape analyse l'explication et l'exemple de manière à générer la définition du concept la plus générale possible. La généralisation englobe des exemples qui décrivent un principe d'opération identique. Ceux-ci pourront être compris en déduisant la même explication. La méthode EBG utilise les connaissances du domaine pour déterminer quelles caractéristiques d'un exemple peuvent être généralisées. La généralisation est déductivement justifiable, car elle peut être expliquée en terme de connaissances du domaine.

La définition du problème de la figure 2.1, résume la méthode EBG.

```
Donné:
• Concept étudié
• Exemple d'entraînement
• Théorie du domaine
• Critère d'opérationnalité

A déterminer:
• Une définition du concept qui à la fois
   - constitue une généralisation de l'exemple
     d'entraînement
   - satisfait le critère d'opérationnalité
```

Fig. 2.1 *Définition d'un problème pour la méthode EBG.*

Prendre le métro à Paris (RER) de la figure 2.2 donne un exemple d'un problème couvert par la méthode EBG[3].

[3] Les exemples utilisés pour cette thèse sont écrits en anglais, car ils proviennent d'articles anglais qui décrivent la méthode proposée ici.

Donné:

1. Concept étudié:

```
take subway ⇔
          pay for Access ∧
          enter station with Access ∧
          ride ∧
          exit station with Access
```

2. Exemple d'entraînement:

```
is not in tunnel
has no mailvan
has no sleeping car
buy ticket from cashier
has 432 passengers
receive ticket
put ticket into turnstile
receive ticket
enter through turnstile at pt-a
has no washroom
driver jimmy
ride
put ticket into turnstile
exit through turnstile at pt-b
distance between pt-a and pt-b is 4
```

3. Théorie du domaine:

```
pay for Acc ⇔
          buy Acc from Disp ∧
          receive Acc

enter_station ⇔
          get Acc ∨
          enter through Ent_Mach at Ent_pt
          receive Acc

get Acc ⇔
          put Acc into Acc_Mach1 ∧
          receive Acc

exit_station ⇔
          put Acc into Exit_Mach ∧
          exit through Exit_Mach at Exit_pt
```

4. Critère d'opérationnalité

Les seules caractéristiques opérationnelles sont celles utilisées dans les exemples d'entraînement

A déterminer:

Une définition du concept qui à la fois:
- couvre l'exemple
- satisfait le critère d'opérationnalité

Fig. 2.2 *Exemple d'un problème de généralisation EBG. Les mots commençant par une lettre majuscule dénotent des variables logiques. Les caractéristiques en caractères gras représentent la séquence d'actions de l'exemple.*

Comme l'indique la figure 2.2, la définition d'un problème EBG nécessite quatre sources d'information:

1. Concept étudié:

Le concept étudié équivaut au concept à apprendre. Reconnaître une situation de *prendre le métro* représente le concept étudié de notre exemple. Il est défini en termes tels que `pay for Acc, enter station, ride` et `exit station`; tandis que l'exemple est défini en termes, initialement de plus bas niveau, tels que `buy Acc from Disp, enter through Ent_Mach at Ent_pt, receive Acc, ride`, etc, ...

2. Exemple d'entraînement:

Un exemple d'entraînement est un exemple du concept étudié. *Prendre le métro* à Paris (RER) constitue l'exemple du concept *prendre un métro*.

3. Théorie du domaine:

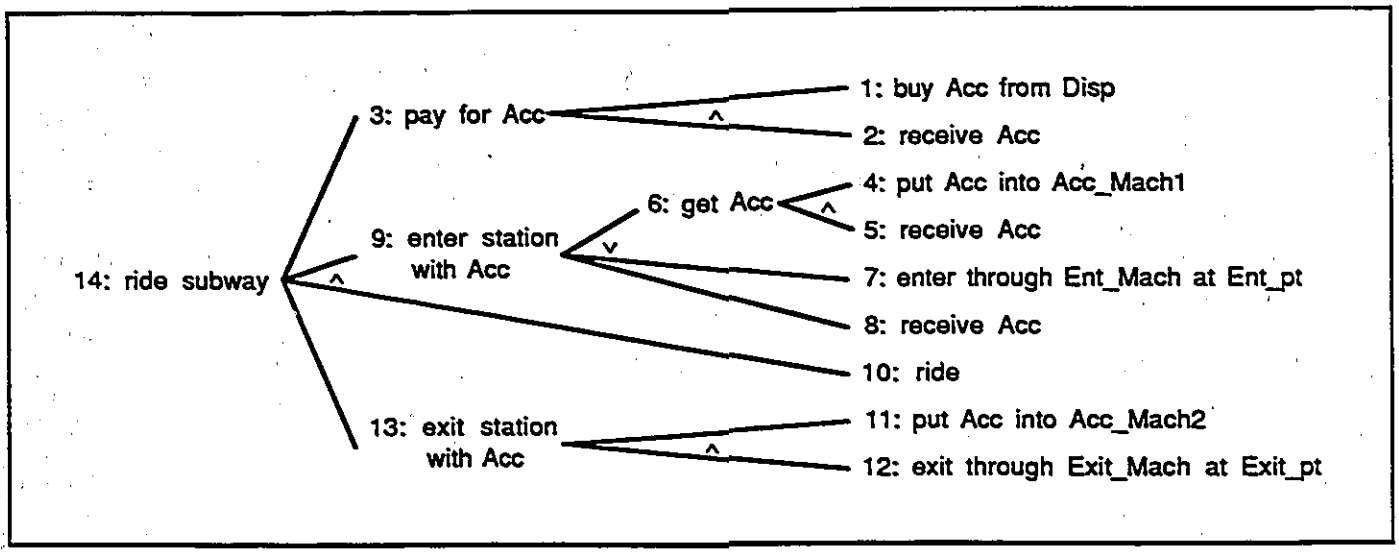


Fig. 2.3 *La structure en arbre de la théorie du domaine. Les mots commençant par une lettre majuscule représentent des variables logiques de chacun des arguments.*

La théorie du domaine est représentée par un ensemble de règles et de faits utilisé pour expliquer comment un exemple fait partie du concept étudié. La théorie pour *prendre le métro* définit les règles pour inférer le concept, notamment

`pay for Acc`, `enter station` et `exit station`. Cette théorie est illustrée par la structure en arbre ET (∧)/OU (∨) de la figure 2.3.

4. Critère d'opérationnalité:

Le critère d'opérationnalité définit les expressions valides pour exprimer la définition du concept étudié. Dans le cas de prendre le métro, les expressions valides correspondent aux caractéristiques de l'exemple du métro de Paris (RER).

Etant donné les quatre sources précédantes d'information, la méthode consiste à déterminer une définition du concept qui couvre l'exemple et satisfait le critère d'opérationnalité. Dans notre exemple, cela correspond à déterminer une règle générale pour *prendre le métro* à partir de l'exemple du métro de Paris (RER).

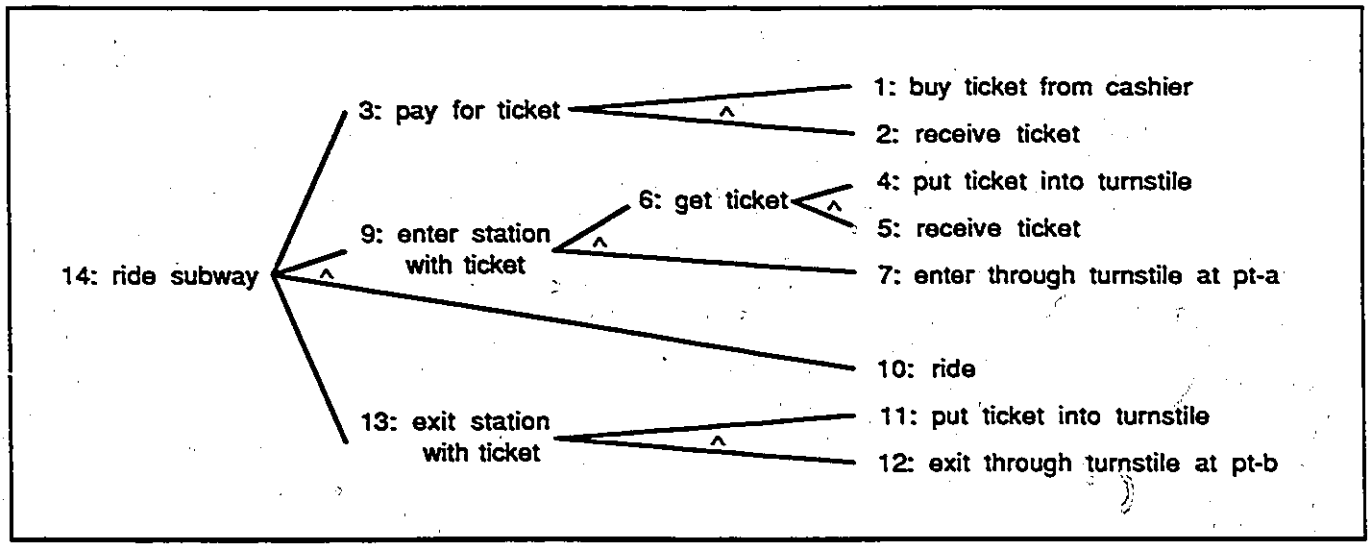


Fig. 2.4 *La structure explicative pour* prendre le métro *RER. On peut remarquer que certains faits appartenant à l'exemple d'entraînement ne participent pas à l'explication.*

La méthode EBG se réalise en deux étapes:

1) **Expliquer** comment l'exemple fait partie du concept à l'aide de la théorie du domaine. Une structure explicative en arbre démontre l'explication obtenue et chacune de ses branches se termine par une caractéristique opérationnelle (figure 2.4). Prendre le métro à Paris (RER) fait partie du concept étudié *prendre un métro*, car les caractéristiques `pay for ticket`, `enter station with`

`ticket at pt-a`, `ride` et `exit station with ticket at pt-b` sont inférées par `ride subway`. A leur tour, ces caractéristiques sont déduites par d'autres caractéristiques. Ce processus est répété jusqu'à ce que les caractéristiques déduites soient opérationnelles. De cette façon la structure explicative isole les caractéristiques pertinentes de l'exemple, tels `buy ticket from cashier`, `receive ticket`, `put ticket into turnstile`, etc, ...

```
ride subway ⇔ buy Acc from Disp ∧
              receive Acc ∧
              put Acc into Acc_Mach1 ∧
              receive Acc ∧
              enter through Ent_Mach at Ent_pt ∧
              ride subway ∧
              put Acc into Acc_Mach2 ∧
              exit through Exit_Mach at Exit_pt
```

Fig. 2.5 *La définition du concept pour reconnaître une situation de* prendre un métro.

2) **Généraliser** l'explication en régressant le concept au travers de la structure explicative en arbre. Chaque occurrence d'une règle instanciée de la structure explicative est remplacée par la règle générale associée à la théorie du domaine. La conjonction des caractéristiques opérationnelles généralisées forme la définition du concept (figure 2.5). Elle définit les caractéristiques requises pour qu'un exemple fasse partie du concept *prendre un métro*.

3. L'Apprentissage Incrémental

Notre méthode ACP propose un apprentissage incrémental. Son but est:

1º d'apprendre une définition du concept à partir de plusieurs exemples au lieu d'un seul. Cette définition du concept satisfait le critère d'opérationnalité et couvre tous les exemples positifs. L'ajout de contraintes procédurales aux opérateurs, rend explicites les liaisons implicitement représentées par la séquence d'actions de chacun des exemples.

2º d'apprendre des contraintes procédurales de façon continue;

3º de spécifier les opérateurs inconnus de la théorie du domaine, avec des opérateurs et leurs contraintes procédurales appris, pour obtenir une théorie complètement spécifiée du domaine (FSDT).

L'utilisation de plusieurs exemples sert à décrire les situations possibles d'un concept plutôt que de restreindre ou à diriger la recherche de la définition du concept, comme c'est le cas pour les méthodes inductives. L'ordre d'entrée de ceux-ci n'a aucune influence sur la définition du concept. Ils doivent être judicieusement choisis de façon à ce que les explications exercent le plus possible la théorie du domaine. Les caractéristiques de l'exemple peuvent exprimer une action ou un descripteur. Toutes les actions et l'ordre dans lequel elles apparaissent sont pertinents. Tant qu'aux descripteurs, ils peuvent être pertinents ou non pour un exemple. Ces derniers sont traités plus particulièrement au chapitre 5. La définition du concept est aussi influencée par une telle représentation des exemples. Disons simplement pour l'instant, qu'elle est composée d'une conjonction d'actions (appelées `Act`) et de descripteurs (appelés `Desc`) possibles pour un exemple.

A partir d'un exemple concret d'apprentissage incrémental voici un scénario regroupant toutes les situations possibles pour démontrer la portée de notre méthode d'apprentissage incrémental. La première fois que nous prenons le métro, nous déduisons les règles de base, telles que 1º payer l'accès, 2º avant de sortir remettre le ticket reçu au moment de payer, etc... Nous généralisons ensuite la façon de *prendre un métro* en supposant que tous les métros se prennent de la même façon, jusqu'à preuve du contraire. Imaginons qu'en sortant du métro, nous devons *prendre un taxi* pour se rendre à la gare. Nous nous apercevons que la séquence d'actions pour prendre le taxi diffère de celle reliée

au métro. Une fois à la gare, nous pouvons *prendre le train* en utilisant le même procédé que pour prendre le métro. La représentation mentale, que nous avons de *prendre le métro* est alors trop générale puisqu'elle englobe aussi de *prendre le train*. A partir de cette expérience, nous restreignons l'idée que nous avons à *prendre le métro* à l'aide des caractéristiques discriminant les situations *prendre le métro* et *prendre le train*. Une fois devant un nouveau métro la situation peut correspondre, à peu de chose près (i.e. au lieu de payer ticket, nous payons un jeton) à la situation vue précédemment. Donc, nous ne nous posons pas vraiment de question (i.e. pas de déduction nécessaire ici, nous utilisons la règle générale) et nous prenons le métro. Par contre, on peut faire face à une situation nouvelle, telle que: nous ne recevons pas de ticket après avoir payé ou, nous ne remettons aucun ticket avant de sortir. Cette situation est différente de l'idée générale que nous nous étions fixée. Nous devons donc déduire en partie ou en totalité les étapes à suivre. Comme pour la première fois, nous généralisons à partir de cette nouvelle situation, puis nous améliorons la représentation mentale pour qu'elle englobe ce que nous connaissions déjà de la situation *prendre le métro*, avec ce que nous venons d'apprendre.

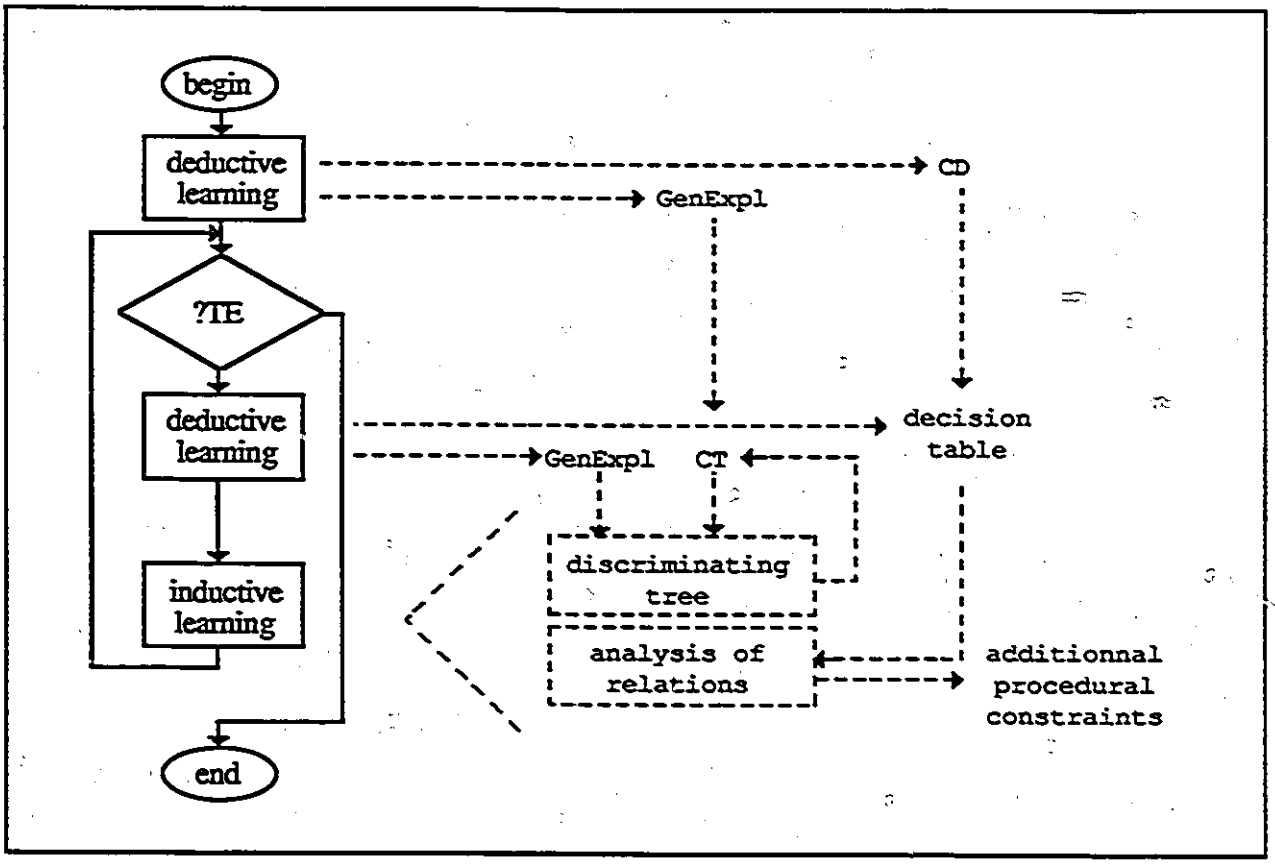


Fig. 3.1 *L'apprentissage incrémental, une méthode mixte.*

Tout comme le processus *prendre le métro*, l'apprentissage incrémental combine les méthodes d'apprentissage déductif et inductif (figure 3.1). La méthode classique EBG sert de méthode d'apprentissage déductif. La méthode EBG appliquée à un exemple particulier fournit des résultats, notamment: une structure explicative généralisée et une définition du concept. La première structure explicative (`GenExpl`) est gardée dans une structure temporaire appelée théorie courante (`CT`). Cette structure et la prochaine structure explicative généralisée obtenue pour un exemple sont comparées lors de la discrimination des arbres (`discriminating trees`) et le résultat est à nouveau gardé dans la structure de la théorie courante. Toutes les définitions du concept sont emmagasinées dans une structure appelée table de décision (`decision table`) pour générer des contraintes procédurales additionnelles (`additionnal procedural constraints`) par l'analyse des relations (`analysis of relations`). La théorie complètement spécifiée du domaine est obtenue en combinant la théorie courante et les contraintes procédurales additionnelles. La définition du concept est ensuite générée à partir de cette théorie.

Ce chapitre présente: la définition et l'algorithme général de l'apprentissage incrémental, l'utilisation de la méthode EBG, l'apprentissage inductif et un exemple complet d'apprentissage incrémental à partir d'exemples positifs.

3.1 La définition du problème pour l'apprentissage incrémental

La définition du problème de la figure 3.2, résume la méthode de l'apprentissage incrémental à partir d'exemples positifs.

Donné:
- Concept étudié
- Exemples d'entraînement
- Théorie sous-spécifiée du domaine
- Critère d'opérationnalité

A déterminer:
- Une théorie du domaine complètement spécifiée
- Une définition du concept qui à la fois
  - satisfait le critère d'opérationnalité
  - couvre tous les exemples d'entraînement

Fig. 3.2 *Définition du problème de l'apprentissage incrémental.*

Le concept étudié

Le concept étudié de notre exemple est de reconnaître une situation de *prendre le métro*. Il est exprimé par des actions non opérationnelles, tels que pay for Acc, enter station, ride et exit station, tandis que les exemples d'entraînement sont décrits par des actions opérationnelles, notamment buy Acc from Disp, receive Acc.

Les exemples d'entraînement (TEs)

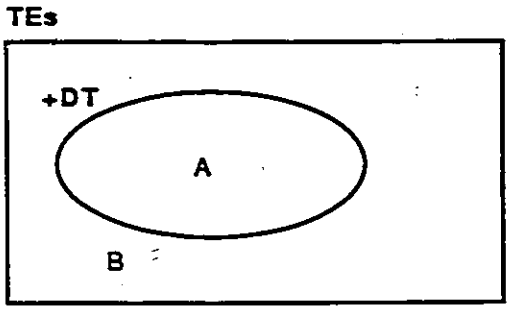


Fig. 3.3 *Les types A et B d'exemples positifs d'entraînement.*

Chaque exemple d'entraînement fourni au système est accompagné de son signe: positif ou négatif. Seuls les exemples positifs sont considérés pour l'instant. Tous les exemples positifs possibles pour *prendre le métro* peuvent être regroupés selon qu'ils sont explicables ou non par la théorie du domaine (figure 3.3). Le type A englobe les exemples explicables par la théorie du domaine. Le type B représente tous les exemples qui n'appartiennent pas au type A. Ces exemples sont non explicables par la théorie du domaine. Ce dernier type d'exemples est une conséquence de la théorie incomplète et ils ne sont d'aucune utilité pour l'apprentissage incrémental présenté ici, sinon de montrer que la théorie du domaine est incomplète.

Tous les exemples d'entraînement classés selon leur type sont présentés en appendice A. Chaque caractéristique d'un exemple est opérationnelle et peut représenter une action ou un descripteur. Les actions sont représentées en caractères gras.

La théorie positive du domaine sous-spécifiée (DT)

La théorie positive du domaine inclut (figure 3.4) un ensemble de règles et de faits permettant d'expliquer comment les exemples d'entraînement font partie du concept étudié. La théorie du domaine est *incomplète*, car elle ne peut pas expliquer tous les exemples positifs du concept étudié. Il existe toujours des exceptions ou de nouveaux exemples inexplicables par la théorie du domaine. Entre autres, il est possible que l'expert ne pense pas à tous les exemples représentatifs du domaine ou bien le domaine peut évoluer; c'est-à-dire que de nouveaux exemples peuvent être acceptés dans le domaine. La théorie du domaine est *inconsistante*, car elle peut expliquer des exemples négatifs du concept étudié.

Une théorie du domaine définissant de façon trop générale la situation de *prendre le métro*, risque d'accepter aussi des exemples négatifs, telles les situations de *prendre le train*.

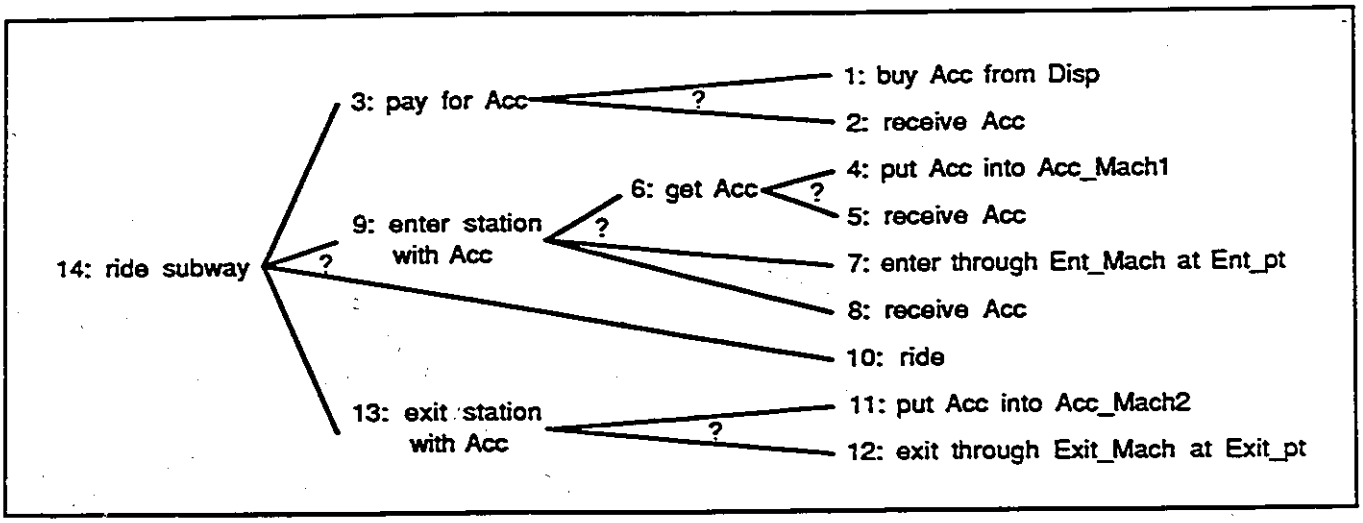


Fig. 3.4 *La théorie positive du domaine sous-spécifiée. Les opérateurs dénotés par "?" sont inconnus. Lors de l'apprentissage, ces opérateurs sont remplacés soit par ET ou, par OU et une des contraintes procédurales. Les mots commençant par une lettre majuscule représentent des variables logiques de chacun des arguments. Seul le chiffre accompagnant l'argument sera utilisé pour simplifier des exemples et des figures. Le nom complet d'un argument sera utilisé à des fins d'interprétation seulement.*

La théorie du domaine est *sous-spécifiée*, puisque les connecteurs logiques de la théorie dénotée par le "?" sont inconnus. Par conséquent la théorie sous-spécifiée du domaine n'élimine pas les combinaisons impossibles d'actions pour un exemple. Elle représente l'organisation de la séquence d'actions possibles du concept étudié sans liaison entre elles. Cette assomption représente le fait que dans la réalité, nous pouvons connaître une définition non-opérationnelle d'un processus quelconque, telle que si deux actions surviennent, alors une des deux doit précéder l'autre[3]. Nous pouvons être au courant par exemple que pour prendre le métro une personne a besoin d'un ticket. Au même moment nous pouvons ne pas savoir que mettre le ticket dans la barrière ouvrante, après avoir pris le métro, force la personne à sortir (il n'est pas possible de reprendre le métro ou de transférer de métro).

[3] Les actions simultanées ne font pas l'objet de cette recherche.

La théorie du domaine (figure 3.4) est représentée par une structure en arbre ET (∧)/ OU (∨) avec des opérateurs INCONNUS (?), où l'ordre d'évaluation de chacun des arguments est connu. Tant qu'un opérateur ne peut être généré à partir des observations faites sur les exemples d'entraînement, il reste INCONNU (?). Les opérateurs INCONNUs sont représentés par les points d'interrogation (?). Par exemple, l'opérateur INCONNU (?) de l'arbre `14` (`ride subway`), s'interprète comme suit: on ne connaît pas encore la liaison existante entre les sous-arbres `3, 9, 10` et `13` (`pay for Acc , enter station, ride` et `exit station`). Cet opérateur deviendra un opérateur ET (∧) ou OU (∨) avec des contraintes procédurales s'il y a lieu, lors de la séance d'apprentissage incrémental. La nature exacte de la liaison entre deux actions sert à restreindre l'opérateur de niveau supérieur. Les opérateurs sont générés à partir des combinaisons d'actions fournies par les exemples d'entraînement.

Le critère d'opérationnalité

Le critère d'opérationnalité correspond aux actions utilisées dans les exemples d'entraînement.

La théorie complètement spécifiée du domaine (FSDT)

La théorie complètement spécifiée du domaine est représentée par une structure en arbre ET (∧), OU (∨). La structure comprend toutes les actions utilisées par les exemples d'entraînement. Par exemple, si les actions opérationnelles de la figure 3.4 (actions du dernier niveau) apparaissent dans au moins un des exemples d'entraînement, alors la théorie complètement spécifiée du domaine est représentée par la même structure en arbre que la théorie du domaine. De plus, suite aux liaisons apprises sur l'organisation de la séquence d'actions des exemples, tous les opérateurs de la théorie complètement spécifiée du domaine sont générés et certaines de ces actions peuvent être liées par des contraintes procédurales. La théorie du domaine une fois spécifiée n'explique pas les exemples que la théorie sous-spécifiée du domaine ne peut expliquer. Elle est incomplète de la même manière que la théorie du domaine l'est. Par exemple, la théorie du domaine peut expliquer l'utilisation des actions `11` et/ou `12`. Si ces actions sont absentes pour tous les exemples d'entraînement, alors elles sont aussi absentes dans la théorie complètement spécifiée du domaine, ainsi que les actions de niveau supérieur ne référant qu'à ces deux actions comme c'est le cas pour l'action `13` ici. La théorie est spécifiée dans le sens que les opérateurs ont été appris et restreints par des contraintes procédurales pour éliminer des combinaisons

impossibles d'actions. Ces combinaisons d'actions sont non observables par les exemples d'entraînement et elles sont interprétées provisoirement avec les assomptions du monde fermé (`closed world assumption`). En d'autres termes les exemples avec les combinaisons d'actions non représentées sont considérés négatifs. Etant donné que la méthode est incrémentale, si un exemple présente une nouvelle combinaison d'actions, alors la théorie complètement spécifiée du domaine est révisée. Ce qui était considéré comme un exemple négatif (basé sur les assomptions du monde fermé) peut maintenant être accepté comme étant un exemple positif.

Les exemples inexplicables par la théorie sous-spécifiée du domaine ne le seront pas davantage après une séance d'apprentissage incrémental, par la théorie complètement spécifiée du domaine. Cette dernière n'explique pas plus d'exemples que la théorie sous-spécifiée du domaine. Elle peut expliquer au plus les mêmes exemples. La théorie complètement spécifiée du domaine possède tout simplement plus d'information que la théorie sous-spécifiée du domaine sur les liaisons existantes entre les actions, qui ont été apprises à partir des exemples fournis lors de l'apprentissage, pour éliminer des combinaisons impossibles d'actions.

La définition du concept (CD)

La définition du concept, c'est une règle générale couvrant tous les exemples positifs. Pour l'instant disons simplement qu'elle est composée d'une séquence d'actions accompagnées de contraintes procédurales liant certaines actions. Une séquence d'actions (appelée `Act`) est une conjonction/disjonction d'actions. La définition du concept est utilisée pour vérifier si un nouvel exemple diffère de ce qui a déjà été appris à partir des exemples précédents. Si l'exemple satisfait les combinaisons d'actions exprimées par cette règle, alors il est déjà appris par celle-ci. Elle est exprimée par des actions satisfaisant le critère d'opérationnalité et elle est modifiée à chaque fois qu'un nouvel exemple est fourni.

La définition du concept est définie à partir de la théorie complètement spécifiée du domaine. Les opérateurs et les contraintes procédurales de la structure de la théorie complètement spécifiée du domaine, propagés au dernier niveau définissent les liaisons entre les actions opérationnelles. De telles actions accompagnées des liaisons forment la définition du concept. Cette définition satisfait ainsi le critère d'opérationnalité et couvre tous les exemples d'entraînement.

3.2 La phase déductive

Lors d'une séance d'apprentissage incrémental, les exemples positifs sont entrés un à un dans le système. Chaque exemple peut être soit inexplicable par la théorie du domaine, déjà appris, le premier à être expliqué par la théorie du domaine ou bien, expliqué sans être le premier. Dans les deux premiers cas, aucune structure explicative généralisée n'est obtenue par la méthode EBG. Etant donné que l'apprentissage du système est basé sur les structures généralisées, l'algorithme ne peut rien apprendre de cet exemple et passe à l'exemple suivant. Par contre pour les deux derniers cas, la méthode EBG permet au système d'apprendre en généralisant l'explication donnée pour un exemple. L'algorithme de l'apprentissage incrémental de la figure 3.5 illustre de telles possibilités.

Les délimiteurs possibles sont le début (begin[4]) et la fin (end) de l'algorithme. Les conditions vérifiées sont:

1º la définition du concept existe ou n'existe pas (condition vraie ou fausse) (∃CD?);

2º l'exemple satisfait ou non (condition vraie ou fausse) une des combinaisons d'actions définies par la définition du concept (∈Ac?)

3º l'exemple appartient ou non (condition vraie ou fausse) à la théorie sous-spécifiée du domaine (∈DT?). Les processus exécutables possibles sont: la première généralisation donnée pour un exemple positif (1st generalization) et l'apprentissage inductif (inductive learning) pour les autres généralisations. Aucun processus est n'exécuté et l'algorithme passe à l'exemple positif suivant lorsque l'exemple a déjà été appris (already learned) ou que la théorie du domaine est incomplète pour expliquer l'exemple (incomplete DT).

[4] Les termes soulignés du texte réfèrent aux termes utilisés dans les figures.

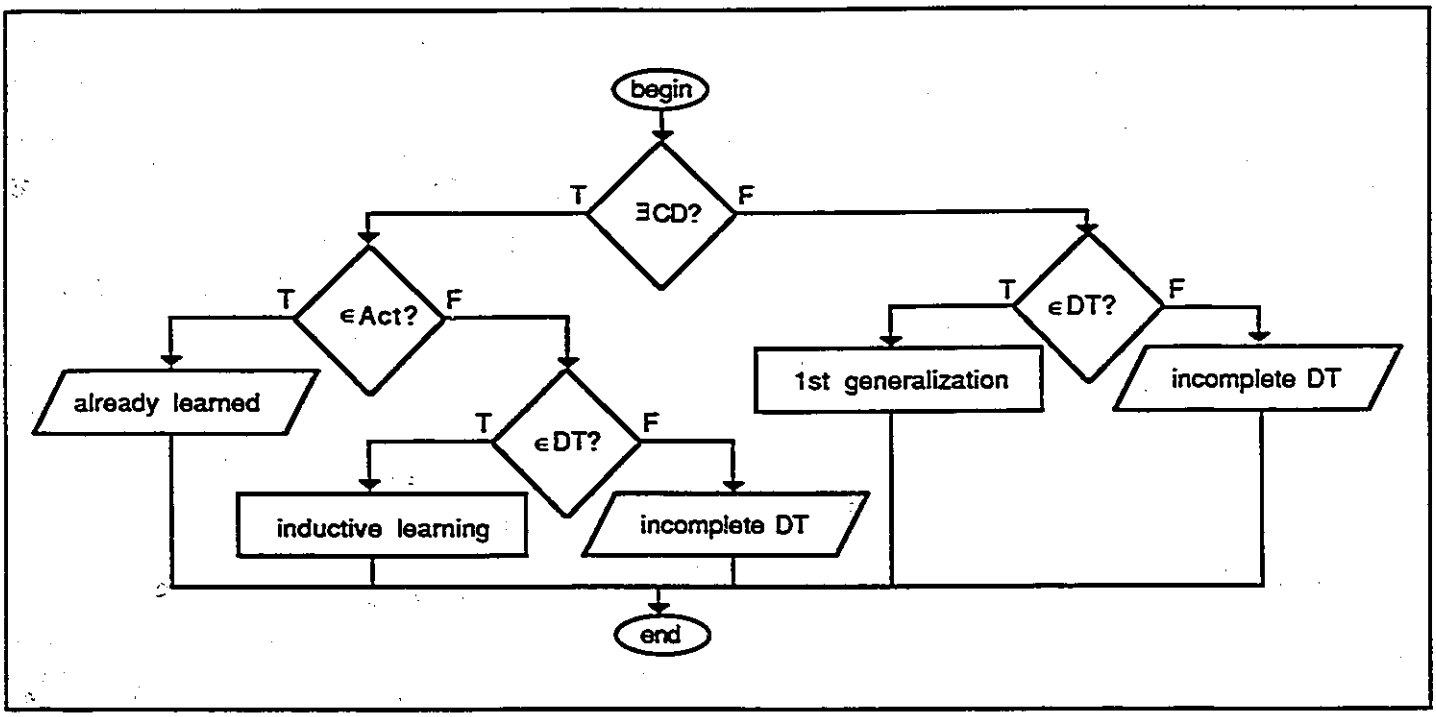


Fig. 3.5 *L'algorithme général de l'apprentissage incrémental. Les ellipses illustrent les délimiteurs de l'algorithme, les losanges: les conditions vérifiées, les rectangles: les processus exécutés et les parallélogrammes: les messages possibles lorsqu'aucun processus ne peut être exécuté.*

De façon plus détaillée, on vérifie d'abord s'il existe une définition du concept (∃CD?). Si c'est le cas et si la séquence d'actions de l'exemple satisfait une des combinaisons d'actions possibles de la définition du concept (∈Act?), alors l'algorithme passe au prochain exemple, car il a déjà été appris (`already learned`). S'il existe une définition du concept (∃CD?) que l'exemple positif ne satisfait pas (∉Act?), alors la méthode EBG tente de généraliser l'exemple à l'aide de la théorie sous-spécifiée du domaine. Si on obtient une structure explicative généralisée, l'exemple appartient à la théorie sous-spécifiée du domaine (∈DT?), on passe donc au processus de l'apprentissage inductif (`inductive learning`). Ce processus modifie la théorie complètement spécifiée du domaine ainsi que la définition du concept en fonction de la nouvelle structure explicative généralisée obtenue. Si aucune généralisation n'est obtenue, la théorie sous-spécifiée du domaine est incomplète (`incomplete DT`), c'est-à-dire l'exemple est valide, mais la théorie n'est pas assez élaborée pour expliquer que l'exemple fait partie du concept étudié. De façon analogue, si aucune définition du concept n'existe (∄CD?) alors on applique la méthode EBG. Si la théorie sous-spécifiée du domaine peut expliquer l'exemple

($\in_{DT2}$), on obtient la structure explicative généralisée de l'exemple. Cette structure explicative initialise la théorie complètement spécifiée du domaine, ainsi que la définition du concept. Et encore une fois aucune structure explicative généralisée n'est obtenue, car la théorie du domaine est incomplète (incomplete_DT).

3.2.1 Quelques modifications à la méthode EBG

Lors de l'apprentissage incrémental, les résultats obtenus par l'apprentissage déductif servent à l'apprentissage inductif. Pour cette raison, on a apporté quelques modifications aux résultats obtenus par la méthode EBG, tels:

1º on conserve la structure explicative généralisée au lieu d'une simple structure explicative. Ainsi, on peut couvrir tous les exemples qui ne s'expliquent pas de la même façon que l'exemple étudié. Cette structure est utilisée lors de la discrimination des arbres.

2º la définition du concept est basée sur les assomptions du monde fermé. Nous assumons que la valeur logique d'une action absente pour un exemple est "0". Cette interprétation permet de générer des contraintes procédurales additionnelles sur les actions opérationnelles par l'analyse des relations.

La pertinence de l'ordre des actions et les résultats escomptés entraînent aussi une légère modification de l'algorithme. La théorie du domaine est complètement parcourue, en largeur d'abord et une seule fois. Si la théorie couvre l'exemple alors une structure explicative généralisée et une définition du concept sont obtenues pour l'exemple de la façon suivante: pour chaque action de l'exemple; si l'action est expliquée par le chemin courant de la théorie du domaine alors ce chemin fait partie de la structure explicative de l'exemple et la valeur logique de cette action pour la définition du concept est "1"; sinon la valeur logique de cette action est "0" et l'algorithme passe au chemin suivant. Et ce, jusqu'à ce que toutes les actions de l'exemple soient expliquées et jusqu'à ce qu'il n'y ait plus de chemin à parcourir dans la théorie sous-spécifiée du domaine.

Toutes les actions de l'exemple d'entraînement doivent être expliquées pour qu'une explication soit valide, sinon la théorie est incomplète. Contrairement à la méthode EBG, certaines caractéristiques d'un exemple sont nécessairement pertinentes, c'est-à-dire qu'elles doivent apparaître dans l'explication. Ces caractéristiques représentent les actions

de l'exemple. Cette restriction est due au fait que l'occurrence et l'ordre dans lequel une action apparaît sont importants pour rendre explicites les relations qui existent entre elles. De la même façon qu'auparavant, la généralisation de l'explication est obtenue en régressant le concept au travers de la structure explicative, et la définition du concept est maintenant exprimée par les valeurs logiques ("0": absence; "1": présence) associées aux actions.

3.3 La phase inductive

L'algorithme général de l'apprentissage incrémental passe de l'apprentissage déductif à l'apprentissage inductif, aussitôt qu'au moins deux exemples de type A sont expliqués par la théorie du domaine. La méthode déductive est appliquée pour chacun des exemples. Pour un exemple de type A, ce processus permet entre autres d'obtenir une structure explicative généralisée de l'exemple. Les deux étapes de l'apprentissage inductif sont présentées dans les sous-sections suivantes. Elles explorent les combinaisons possibles des actions pour tous les exemples fournis. La première étape appelée discrimination des arbres, explore les combinaisons possibles des actions à partir des structures en arbre. Pour cette étape, l'opérateur ainsi que les contraintes générées pour une action non-opérationnelle lient des actions opérationnelles et non-opérationnelles. La seconde étape appelée l'analyse des relations basée sur les assomptions du monde fermé, ajoute des contraintes procédurales aux opérateurs qui existent déjà, pour lier seulement des actions opérationnelles. Les contraintes procédurales lors de la phase inductive seulement sont d'abord présentées, puis suivies des étapes de l'apprentissage inductif.

3.3.1 Les contraintes procédurales

L'idée de base des contraintes procédurales est de rendre explicites les liaisons entre les actions, représentées implicitement par les exemples. Une contrainte procédurale accompagne un opérateur OU ($\vee$) et restreint la présence ou l'absence ainsi que l'ordre d'une action en particulier. Les contraintes procédurales apparaissent dans la théorie complètement spécifiée du domaine et dans la définition du concept, pour éviter qu'elles acceptent des séquences impossibles d'actions.

On se limite à six contraintes procédurales (figure 3.6). Trois d'entre elles correspondent aux contraintes employées pour définir les cas d'essais en Vérification de logiciels [Myers 1979]. Ce sont les contraintes **E** (ou exclusif), **I** (ou inclusif) et **O** (un et

seulement un). Les contraintes **N** (nécessaire) et **R** (requiert) sont dérivées à partir d'une seule des contraintes employées par Myers (1979). Celui-ci emploie la contrainte **R** bidirectionnellement, il ne fait pas la distinction entre les actions passées et futures. Pour l'apprentissage incrémental, la contrainte **R** exprime une seule direction (en relation avec les actions passées) et la contrainte procédurale **N** exprime l'autre direction (en relation avec les actions futures). La contrainte procédurale **C** (obligatoire) stipule que l'action est obligatoire pour un exemple. Cette contrainte a été définie par nécessité. En effet, il semble pertinent de connaître les actions apparaissant dans chacun des exemples d'entraînement et ainsi définir les actions obligatoires de la théorie complètement spécifiée du domaine ou de la définition du concept.

Les contraintes **N** et **R** peuvent être générées lors de la première étape de l'apprentissage inductif (i.e. discrimination des arbres). La seconde étape du même processus appelée analyse des relations, permet de générer toutes les contraintes, y compris **N** et **R**.

Les contraintes procédurales sont présentées en ordre alphabétique à la figure 3.6.

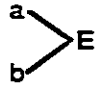
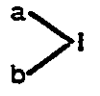
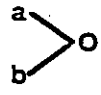
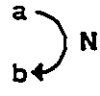
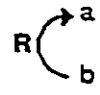
#	Symbol	Constraint	Condition(s)	Combinations	Graph
1	C a	Compulsory	a must be 1	1 0 1 1	
2	a E b	Exclusive or	a and b cannot be 1 simultaneously	0 0 0 1 1 0	
3	a I b	Inclusive or	a and b cannot be 0 simultaneously	0 1 1 0 1 1	
4	a O b	One and only one	a and b cannot be 1 simultaneously & a and b cannot be 0 simultaneously	0 1 1 0	
5	a N b	Necessitates	for a equal to 1, b must be 1	0 0 0 1 1 1	
6	b R a	Requires	for b equal 1, a must be 1	0 0 1 0 1 1	

Fig. 3.6 *Les contraintes procédurales. La représentation symbolique de la contrainte apparaît en deuxième colonne, son nom en troisième colonne, les conditions à satisfaire pour être générées en quatrième colonne, suivie en cinquième colonne des combinaisons binaires (0: absence; 1: présence) correspondantes et enfin la représentation graphique de la contrainte en sixième colonne.*

Une contrainte C a , dénotée par C, stipule que l'action a est présente pour chaque exemple fourni. La seconde contrainte procédurale a E b, dénotée par E signifie que les actions a et b ne sont pas présentes simultanément pour aucun des exemples; tandis que la contrainte suivante a I b , dénotée I, spécifie que les actions a et b ne sont pas absentes simultanément pour aucun des exemples. La contrainte procédurale a O b , dénotée O, est une combinaison des deux contraintes précédentes. Les actions a et b ne peuvent être présentes ou absentes simultanément pour un même exemple. La cinquième contrainte a N b, dénoté N, représente la présence de l'action a nécessairement suivie de la présence de l'action b. La dernière contrainte b R a, dénotée R, représente la présence de l'action b dans une séquence d'actions qui requiert la présence de l'action précédente a. Il n'est pas nécessaire que deux actions se suivent ou se précèdent immédiatement pour être reliées. Il

peut exister d'autre(s) action(s) entre elles. Etant donné que les contraintes procédurales N et R sont directionnelles, on ne peut pas générer que *a* N *b* à partir de *b* R *a*.[5]

Considérons toutes les combinaisons binaires possibles pour deux actions *a* et *b* : (1, 1), (1, 0), (0, 1), (0, 0). Le résultat des ces combinaisons pour un opérateur de type OU (v) serait: 1, 1, 1 et 0. En ajoutant une contrainte, on restreint le nombre de combinaisons possibles. Par exemple: la contrainte *a* N *b* stipule que si l'action *a* est présente, alors elle est suivie de la présence de l'action *b* pour toutes les séquences d'actions. La combinaison binaire (1, 0) est donc éliminée des possibilités pour les deux actions *a* et *b* liées par la contrainte N, puisque l'action *a* est présente, mais pas suivie de l'action *b* (voir la cinquième colonne de la contrainte dénotée N).

3.3.2 Discrimination des arbres

La discrimination des arbres explore les combinaisons des exemples à partir des structures en arbre. Les structures de la théorie complètement spécifiée du domaine et la structure explicative généralisée (GenExpl de la figure 3.1) du dernier exemple sont nécessaires pour exécuter le processus de discrimination des arbres. La théorie complètement spécifiée du domaine garde toutes les combinaisons possibles pour les exemples passés. Initialement la théorie complètement spécifiée du domaine correspond à la structure explicative généralisée du premier exemple (i.e. la seule combinaison possible des actions pour l'exemple positif passé). Le résultat temporaire du processus est gardé sous le nom de nouvelle théorie (NT). Avant de passer à l'exemple suivant le processus inductif affecte la théorie courante (CT) avec la nouvelle théorie.

---

[5] Considérons *a* N *c*, et *b* N *c*. Puisque *c* n'a pas une seule cause, il n'est pas possible de dire que *c* R une autre cause.

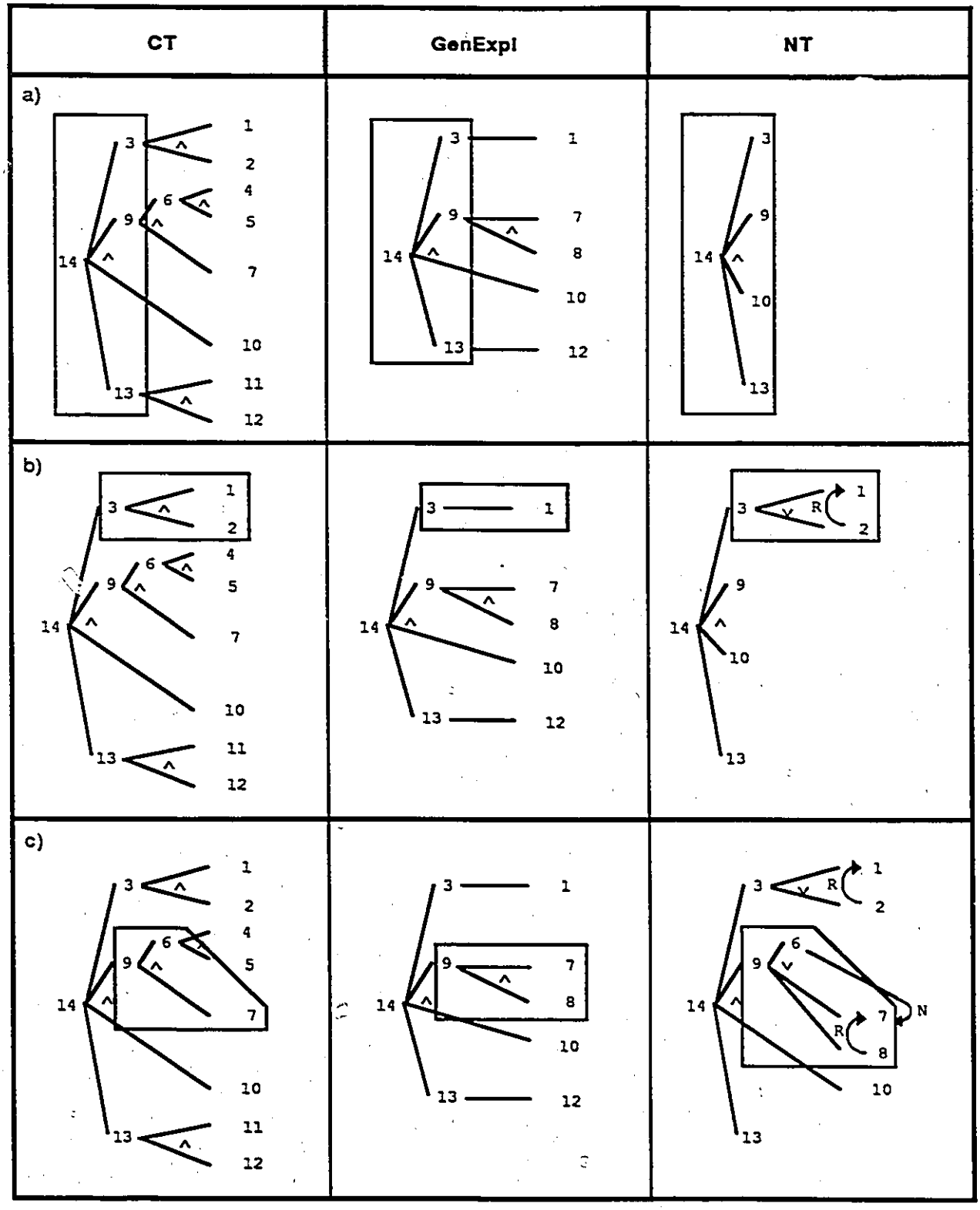
L'algorithme traverse les arbres en largeur d'abord. Pour chaque arbre rencontré:

1) si l'arbre existe dans une seule des structures alors l'arbre est reproduit dans la nouvelle théorie;

2) la même chose se produit si les arbres sont égaux (i.e. la racine et les sous-arbres identiques);

3) si les arbres sont *discriminants* (i.e. la racine ou les sous-arbres différents) alors le *plus large* des arbres (i.e.la racine avec l'ensemble correspondant à l'union de tous les sous-arbres possibles des deux arbres) est produit dans la nouvelle théorie avec l'opérateur OU (v) et s'il y a lieu, des contraintes procédurales sont générées pour ce nouvel arbre de la nouvelle théorie;

4) si un arbre n'a aucun sous-arbre (dernier niveau), alors il apparaît déjà dans la nouvelle théorie.


CT
GenExpl
NT
a)
b)
c)
1
2
3
4
5
6
7
9
10
11
12
13
14
R
N

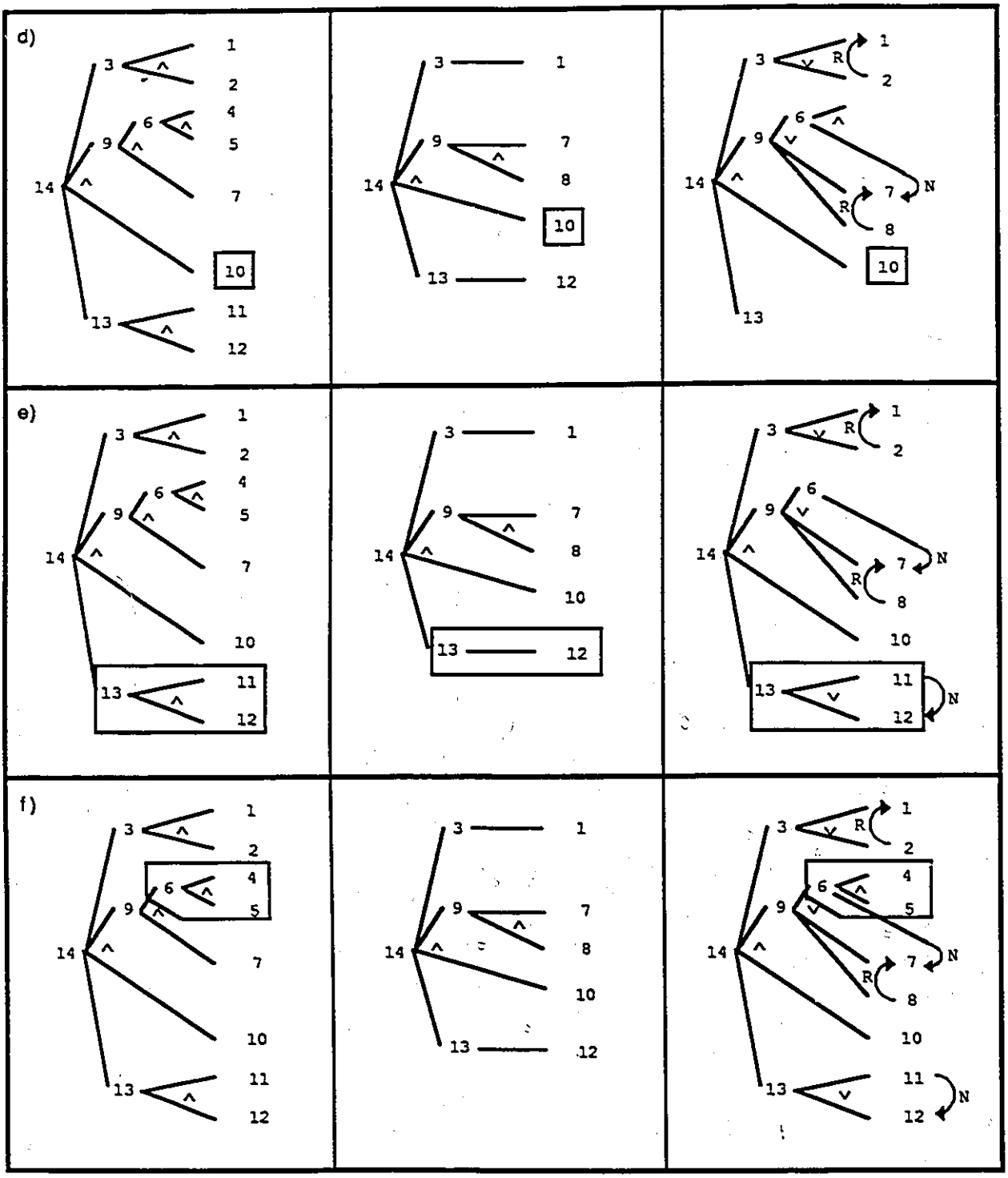


Fig. 3.7 *L'apprentissage de contraintes procédurales par la discrimination des arbres. La première colonne représente la théorie courante (CT). La deuxième colonne représente la structure explicative généralisée (GenExpl). La nouvelle théorie apparaît en troisième colonne. L'arbre encadré de la troisième colonne illustre le résultat de la discrimination des arbres correspondants des deux premières colonnes. Les feuilles entres autres sont dénotées: 1:* `buy Acc from Disp`, *2:* `receive Acc`, *4:* `put Acc into Acc_Mach1`, *5:* `receive Acc`,

7: `enter through Ent_Mach at Ent_point`, *8:* `receive Acc`, *10:* `ride subway`, *11:* `put Acc into Acc_Mach2` *and 12:* `exit through Exit_Mach at Exit_point`. *Les contraintes procédurales requiert et nécessite sont représentées par les flèches identifiées par R et N respectivement.*

Puisque l'ordre des sous-arbres qui correspond à l'ordre d'évaluation des arguments est fixe, alors le processus discriminant implique $O((n-1)^2)$ étapes, où n est le nombre de prédicats dans la théorie du domaine.

Prenons un exemple concret. L'algorithme est indifférent à l'ordre d'entrée des exemples, mais pour montrer certaines de ses caractéristiques, les exemples de prendre les métros 6 (RER) et 7 (Toronto) sont choisis. La figure 3.7 illustre six des étapes effectuées lors de la discrimination des arbres pour les deux exemples choisis. Afin d'éliminer la répétition, un seul exemple de discrimination des arbres du dernier niveau est illustré. L'apprentissage déductif produit une structure explicative généralisée pour l'exemple 6. La théorie courante est initialisée avec celle-ci, puisque c'est la première généralisation trouvée. Etant donné qu'un seul exemple est généralisé, l'algorithme général continue d'appliquer la méthode déductive sur les exemples fournis. Une structure explicative généralisée est aussi produite pour l'exemple 7. Maintenant deux exemples positifs sont expliqués par la théorie du domaine, alors l'algorithme général passe à la méthode inductive.

L'algorithme parcourt les structures en arbre de la théorie du domaine et de la structure explicative généralisée, en largeur d'abord. Si les arbres sont égaux, comme ceux encadrés en première et deuxième colonnes à la figure 3.7a (14), alors le même arbre est produit en troisième colonne, dans la nouvelle théorie avec le même opérateur. Si les arbres sont différents, comme c'est le cas pour ceux encadrés de la figure 3.7b (3), alors l'arbre le plus large est produit dans la nouvelle théorie; l'opérateur devient OU (v). L'algorithme génère ensuite la contrainte procédurale **R** (requiert), pour les actions opérationnelles 1 et 2 (formulée 2 **R** 1 ou 2 requiert 1), puisque chaque occurrence de l'action 2 apparaît avec l'occurrence précédente de l'action 1, dans les deux arbres. Comme on a vu à la figure 3.6 la contrainte procédurale **R** élimine le cas ou l'occurrence de l'action 2 apparaît sans l'occurrence précédente de l'action 1. Les arbres suivants (9 de la figure 3.7c) sont différents, alors l'arbre le plus large est produit dans la nouvelle théorie; l'opérateur devient OU (v). L'algorithme génère les contraintes procédurales 6 **N** 7 (ou 6 nécessite 7) et 8 **R** 7. Etant donné que l'occurrence de l'action 6 apparaît dans les deux arbres, et qu'elle est suivie de l'occurrence de l'action 7, alors la contrainte procédurale 6 **N** 7 est générée. Si l'occurrence de l'action 8 est présente alors l'occurrence précédente de

l'action 7 l'est également, ce qui permet de générer la contrainte procédurale 8 R 7. On ne peut rien générer à partir de l'action 7, puisqu'elle apparaît dans la théorie complètement spécifiée du domaine, précédée de la présence de l'action 6 et dans l'autre structure (structure explicative généralisée) elle est présente suivie de l'occurrence de l'action 8. L'action 7 n'apparaît pas toujours accompagnée (précédée ou suivie) de la même action, alors aucune contrainte procédurale n'est générée. Si un arbre n'a aucun sous-arbre, alors il apparaît déjà dans la nouvelle théorie, comme c'est le cas pour la racine 6. Comme pour les racines des arbres 3, les racines des arbres 13 sont discriminantes; l'arbre le plus large est reproduit dans la nouvelle théorie et la contrainte procédurale 11 N 12 est générée, car chaque occurrence de l'action 11 apparaît avec l'occurrence suivante de l'action 12. Le dernier arbre parcouru (6) apparaît dans une seule des deux structures, donc il est reproduit dans la nouvelle théorie avec son opérateur et aucune contrainte n'est générée.

La théorie complètement spécifiée du domaine qui correspond à la théorie courante finale est ensuite passée à la prochaine étape de l'apprentissage inductif, pour y ajouter des contraintes procédurales. Afin de mieux montrer le processus de l'analyse des relations, d'autres exemples ont été choisis.

3.3.3 Analyse des relations

L'analyse des relations garde les définitions du concept basées sur les assomptions du monde fermé pour chacun des exemples. Si une action est présente dans l'exemple positif, alors sa valeur logique est "1", sinon la valeur logique "0" est générée en conséquence aux assomptions du monde fermé. L'analyse des relations permet d'explorer les combinaisons possibles des actions et d'établir des contraintes procédurales, telles qu'une action est obligatoire (C); que l'opérateur OU (v) est exclusif (E), inclusif (I) ou sans contrainte; que la présence d'une action est requise (R) ou nécessaire (N) pour la présence d'une autre action. Les contraintes procédurales générées par cette étape ne lient que des actions opérationnelles du domaine. Ces contraintes ne seront pas conservées avec la théorie courante lors de la discrimination des arbres pour un prochain exemple appris.

L'analyse de toutes les combinaisons possibles d'actions pour tous les exemples appris s'effectue chaque fois qu'un nouvel exemple d'entraînement est fourni. L'analyse s'effectue d'abord sur les actions individuellement, pour générer les contraintes procédurales C (actions obligatoires). Le processus combine ensuite, deux par deux, tous les résultats (0: absence; 1: présence) des actions opérationnelles obtenus pour les exemples

et génère, s'il y a lieu, des contraintes procédurales correspondant à ces combinaisons binaires (figure 3.6).

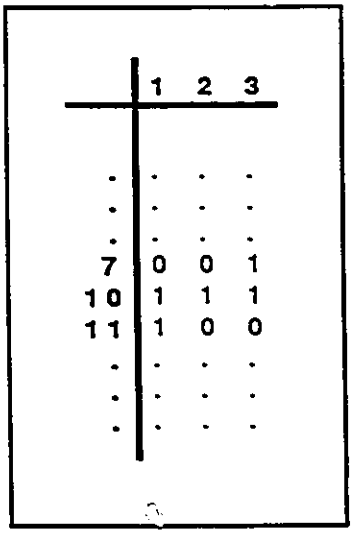


Fig. 3.8 Les définitions du concept basées sur les assomptions du monde fermé pour trois exemples positifs: 1) Bart, 2) Montréal et 3) Paris et pour les actions opérationnelles: 7, 10 et 11.

La figure 3.8 représente l'analyse des relations pour les actions 7, 10 et 11 pour les trois exemples de prendre le métro, soient: `1: Bart, 2: Montréal` et `3: Paris`. Si une action est présente pour l'exemple, alors sa valeur logique est "1", sinon la valeur logique "0" est générée par le monde fermé des assomptions.

L'algorithme vérifie d'abord les actions obligatoires. Rappelons que la contrainte C *a*, dénotée par C, stipule que l'action *a* est obligatoire. Une action est obligatoire si elle est présente (i.e. sa valeur logique est "1") pour tous les exemples. Aucune contrainte procédurale n'est générée pour l'action `7`, puisque celle-ci est absente dans les exemples `1` et `2`. Cette action n'est pas obligatoire. La contrainte procédurale C `10` est générée, car la valeur logique pour les trois exemples positifs est "1". Etant donné que l'action `11` est absente pour les exemples positifs `2` et `3`, alors aucune contrainte procédurale n'est générée pour cette action. L'algorithme combine ensuite les actions, deux par deux, pour générer d'autres contraintes, telles que **E, I, N** et **R** à partir des valeurs logiques pour chacune d'elles. Pour donner un exemple de telles contraintes, supposons les combinaisons possibles pour les actions `7` et `11` (figure 3.9). La combinaison binaire manquante (1, 1)

permet de générer la contrainte 7 **E** 11, c'est-à-dire que les actions 7 et 11 ne sont pas présentes simultanément dans aucun des exemples.

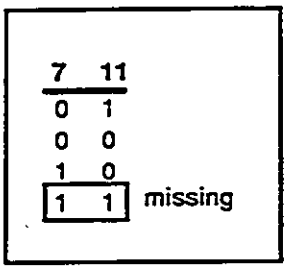


Fig. 3.9 *Les combinaisons binaires possibles pour les actions* 7 *et* 11.

L'analyse des relations génère des contraintes au niveau des feuilles de la structure de la théorie du domaine (opérationnel). Il y a au plus $n - 1$ feuilles dans cet arbre. Puisque les feuilles de l'arbre sont ordonnées et que l'analyse des relations implique la comparaison de paires ordonnées de feuilles, le maximum d'étapes requis est $O((n - 1)^2)$.

3.4 Exemple d'une séance d'apprentissage incrémental

Un exemple d'une séance d'apprentissage incrémental démontre la portée de l'algorithme général de l'apprentissage incrémental (figure 3.5) à partir d'exemples positifs. L'ordre des exemples n'a pas d'influence sur le résultat final. Chaque exemple est choisi pour couvrir un scénario en particulier. Un scénario correspond à un embranchement de l'algorithme. Il existe cinq embranchements dans cet algorithme, alors cinq différents scénarios sont possibles pour un exemple. Les résultats obtenus pour un exemple sont gardés et composent les conditions environnantes pour l'exemple suivant. Les conditions environnantes d'un exemple positif déterminent le choix du scénario à suivre.

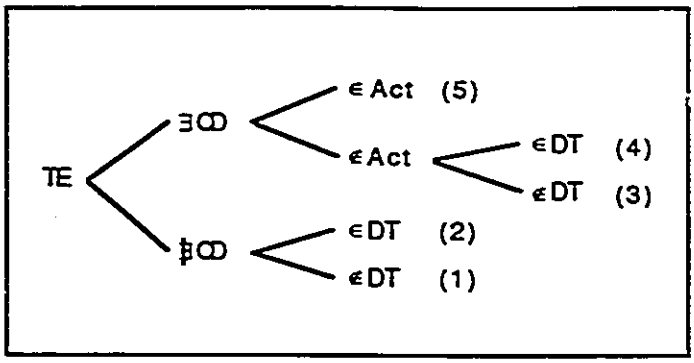


Fig. 3.10 *Les scénarios possibles pour exemple positif. Chaque feuille représente un scénario et chaque chemin correspond aux conditions environnantes à satisfaire pour un scénario.*

Les scénarios possibles pour un exemple positif sont illustrés par la structure en arbre de la figure 3.10. La racine de la structure en arbre représente un exemple positif. Chaque sous-arbre de la structure représente une condition environnante possible pour l'exemple. Si la condition environnante est satisfaite, alors on se déplace en profondeur dans la structure en arbre (i.e. on passe à la condition suivante). Si la condition n'est pas satisfaite, alors on se déplace en largeur dans la structure (i.e. on regarde une alternative à ce niveau).

#	TE's type	TE's identification	∃CD	∈Act	∈DT	process
1	B	Moscow	0	-	0	incomplete DT
2	A	NYC	0	-	1	1st generalization
3	B	Boston	1	0	0	incomplete DT
4	A	Toronto	1	0	1	inductive learning
5	A	Mexico	1	1	-	already learned

Fig. 3.11 *Les exemples positifs utilisés pour la séance d'apprentissage incrémental. La valeur "1" représente une condition vraie, la valeur "0" représente une condition fausse et le symbole "-" signifie que la condition n'est pas vérifiée. La dernière colonne décrit le processus exécuté pour l'exemple.*

Avec cinq exemples d'entraînement il est donc possible de montrer la portée de l'algorithme. La figure 3.11 se veut un résumé pour les exemples utilisés pour la séance d'apprentissage incrémental qui suit. Le chiffre de la première colonne identifie le scénario couvert et correspond au chiffre entre parenthèses de la figure 3.10 et en deuxième colonne

il y a le type de l'exemple. Rappelons ici que les exemples de type A et B sont positifs. Les exemples de type A sont explicables par la théorie du domaine, contrairement à ceux de type B. En troisième colonne il y a l'identification de l'exemple. Le choix de ce dernier n'a pas vraiment d'importance, pourvu qu'il appartienne au type mentionné en deuxième colonne. Les quatrième, cinquième et sixième colonnes représentent les conditions environnantes possibles:

1º l'existence ou non (1 ou 0) d'une définition du concept ($\exists_{CD}$);

2º l'appartenance ou non (1 ou 0) de l'exemple à la définition du concept ($\in_{Act}$), si celle-ci existe d'abord;

3º l'appartenance ou non (1 ou 0) de l'exemple à la théorie du domaine ($\in_{DT}$), si la définition du concept n'existe pas ou si l'exemple n'appartient pas à la définition du concept.

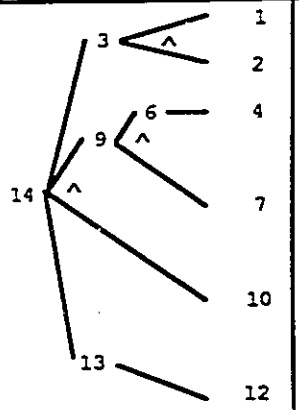
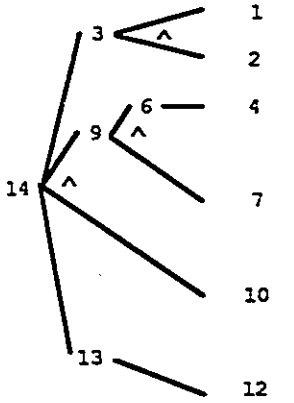
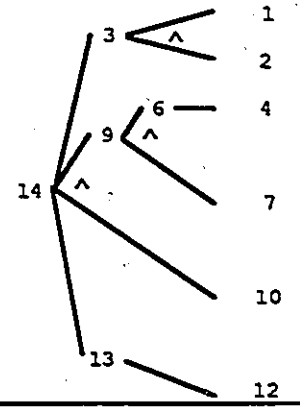
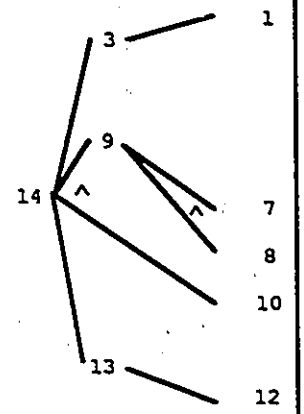
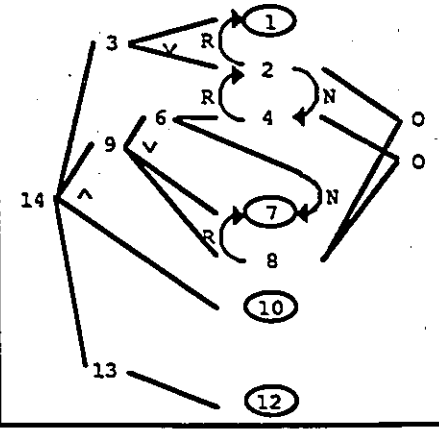
L'appartenance de l'exemple à la séquence d'actions (`Act`) de la définition du concept signifie que l'exemple satisfait une des combinaisons d'actions exprimées par cette définition; tandis que l'appartenance de l'exemple à la théorie du domaine signifie que la théorie sous-spécifiée du domaine peut expliquer comment l'exemple fait partie du concept étudié.

Illustrons à l'aide de deux exemples l'interprétation du tableau de la figure 3.11.

1º Prenons le premier exemple positif décrivant la situation de *prendre le métro*: `Moscow`, de type B. C'est le premier exemple, il est donc normal qu'il n'existe pas de définition du concept ("0" dans la quatrième colonne, "-" en cinquième colonne, pour signifier que la vérification de l'appartenance de l'exemple à la définition du concept ne s'applique pas). L'exemple ne peut être expliqué par la théorie du domaine ("0" en sixième colonne), parce que la théorie du domaine est incomplète (`incomplete DT` en dernière colonne).

2º Prenons le quatrième exemple identifié `Toronto`. Il existe une définition du concept, provenant des exemples précédents ("1" dans la quatrième colonne), mais la séquence d'actions de l'exemple ne satisfait pas la combinaison d'actions exprimée par celle-ci ("0" en cinquième colonne). La théorie du

domaine peut expliquer comment l'exemple fait partie du concept ("1" en sixième colonne). Il est donc possible d'apprendre à partir de cet exemple (`inductive_learning` en dernière colonne).

TE *Process*	GenExpl	FSDT	CD
#1 Moscow (+) *incomplete DT*			
#2 NYC (+) *1st generalization*			1∧2∧4∧7∧10∧12
#3 Prague (+) *incomplete DT*			1∧2∧4∧7∧10∧12
#4 Toronto (+) *inductive learning*			(1 v 2) ∧ (4 v 7 v 8) ∧ 10 ∧ 12 ∧ C1 ∧ C7 ∧ C10 ∧ C12 ∧ 2R1 ∧ 2N4 ∧ 4R2 ∧ 4N7 ∧ 8R7 ∧ 2O8 ∧ 4O8

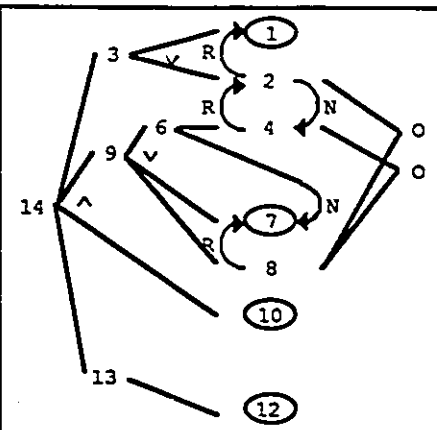
#5 Mexico (+) *already learned*			(1 ∨ 2) ∧ (4 ∨ 7 ∨ 8) ∧ 10 ∧ 12 ∧ C1 ∧ C7 ∧ C10 ∧ C12 ∧ 2R1 ∧ 2N4 ∧ 4R2 ∧ 4N7 ∧ 8R7 ∧ 2O8 ∧ 4O8

Fig. 3.12 *La séance d'apprentissage incrémental. La première colonne identifie l'exemple et le processus exécuté (en italique). La deuxième représente la structure explicative généralisée (GenExpl). La troisième colonne représente la théorie complètement spécifiée du domaine (FSDT). La quatrième colonne représente la définition du concept (CD) générée à partir de la FSDT. Les ellipses représentent la contrainte procédurale C (action obligatoire).*

La séance d'apprentissage incrémental à partir d'exemples positifs est illustrée à la figure 3.12. Initialement la définition du concept et la théorie complètement spécifiée du domaine n'existent pas. `Moscow` est le premier exemple positif entré. Il est inexplicable par la théorie du domaine, car elle est incomplète (`incomplete_DT`) (figure 3.5). Il n'est donc pas possible d'apprendre quoi que ce soit à partir de cet exemple. Le second exemple positif est celui de `NYC`. Il ne satisfait pas la définition du concept, puisqu'elle n'existe pas, mais il appartient à la théorie du domaine. C'est donc la première structure explicative généralisée (`1st_generalization`) donnée par la méthode déductive pour un exemple. La théorie complètement spécifiée du domaine est initialisée avec cette généralisation et la définition du concept est générée à partir de cette théorie. Comme pour le premier exemple, la théorie complètement spécifiée du domaine et la définition du concept sont inchangées pour l'exemple de `Prague`, car la théorie du domaine est incomplète (`incomplete_DT`). L'exemple de *prendre le métro* à `Toronto` ne satisfait pas non plus la définition du concept, mais il est explicable par la théorie sous-spécifiée du domaine. Ainsi le processus de l'apprentissage inductif (`inductive_learning`) est exécuté pour réviser la théorie complètement spécifiée du domaine et la définition du concept, pour qu'elles acceptent la généralisation du nouvel exemple. Le cinquième et dernier exemple de la figure: `Mexico` a déjà été appris (`already_learned`) puisque sa séquence d'actions satisfait une des combinaisons possibles d'actions exprimées par la définition du concept. La théorie complètement spécifiée du domaine et la définition du concept restent inchangées encore une fois et représentent le résultat final de l'apprentissage incrémental à partir des exemples présentés.

La définition du concept est formée de la combinaison d'actions et de contraintes procédurales qui relient certaines actions pour restreindre les combinaisons possibles. Elle couvre tous les exemples possibles explicables par la théorie sous-spécifiée du domaine et appris jusqu'à maintenant. Plus concrètement, la définition du concept est générée à partir de la théorie complètement spécifiée du domaine en propageant les opérateurs et les contraintes procédurales au niveau opérationnel (i.e. de la racine aux feuilles dans la structure en arbre). Entre autres c'est pourquoi la contrainte `6 N 7` dans la structure en arbre est représentée par la contrainte `4 N 7` dans la définition du concept.

La première ligne de la définition du concept représente les combinaisons possibles d'actions pour un exemple et s'interprète comme suit:

1) au moins une des actions `1 (buy Acc from Disp)` ou `2 (receive Acc)` est présente ((1 ∨ 2)) et

2) au moins une des actions `4 (put Acc into Acc_Mach1)`, `7 (enter through Ent_Mach at Ent_point)` ou `8 (receive Acc)` est présente ((4 ∨ 7 ∨ 8)) et

3) les actions `10 (ride subway)` et `12 (exit through Exit_Mach at Exit_point)` sont présentes (10 ∧ 12) pour un exemple.

Les autres lignes de la définition du concept représentent les contraintes procédurales qui restreignent les combinaisons d'actions. La deuxième ligne représente les actions obligatoires `1 (buy Acc from Disp)`, `7 (enter through Ent_Mach at Ent_point)`, `10 (ride subway)` et `12 (exit through Exit_Mach at Exit_point)` (C1 ∧ C7 ∧ C10 ∧ C12) présentes pour un exemple.

Les troisième et quatrième lignes restreignent aussi les combinaisons d'actions de sorte que:

1) la présence de l'action `2 (receive Acc)` requiert la présence de l'action `1 (buy Acc from Disp)` (2 **R** 1);

2) la présence de l'action `2 (receive Acc)` nécessite la présence de l'action `4 (put Acc into Acc_Mach1)` (2 **N** 4);

3) la présence de l'action 4 (`put Acc into Acc_Mach1`) requiert la présence de l'action 2 (`receive Acc`) (4 **R** 2);

4) la présence de l'action 4 (`put Acc into Acc_Mach1`) nécessite la présence de l'action 7 (`enter through Ent_Mach at Ent_point`) (4 **N** 7);

5) la présence de l'action 8 (`receive Acc`) requiert la présence de l'action 7 (`enter through Ent_Mach at Ent_point`) (8 **R** 7);

6) les actions 2 (`receive Acc`) et 8 (`receive Acc`) ne sont pas absentes ou présentes simultanément, pour un exemple (2 **O** 8);

7) les actions 4 (`put Acc into Acc_Mach1`) et 8 (`receive Acc`) ne sont pas absentes ou présentes simultanément, pour un exemple (4 **O** 8).

3.5 Implantation

L'apprentissage incrémental à partir d'exemples positifs décrit dans ce chapitre a été implanté en Quintus Prolog sur une station SUN-3/60 et testé avec les exemples positifs du prototype pour *prendre le métro* (appendice A). Le langage Prolog s'avère adéquat pour implanter un tel système, puisque les inférences effectuées par des méthodes comme EBL fonctionnent particulièrement par chaînage arrière, tout comme le moteur d'inférence de Prolog. Une étape de résolution Prolog consiste à choisir un littéral L dans le but B (B étant une conjonction de littéraux) en:

1) choisissant une clause, dont la tête s'unifie avec L

2) propageant l'unification au travers de B, en remplaçant L par le corps de la clause; si L est un fait, il n'existe pas de corps à la clause et L est tout simplement rejeté.

Le programme Prolog pour apprendre incrémentalement les contraintes procédurales à partir de plusieurs exemples positifs apparaît à l'appendice B. Celui-ci comprend 856 lignes de codes. Des exemples d'apprentissage pour *prendre le métro*

accompagnent le programme. Tous les exemples positifs de l'appendice A sont appris en 9 secondes, ce qui est relativement rapide comparativement aux méthodes inductives.

Pour utiliser le système, l'utilisateur doit fournir une théorie du domaine, plusieurs exemples positifs et l'ordre dans lequel les exemples doivent être appris. Le concept étudié correspond à la règle du plus haut niveau de la théorie du domaine. Tous les faits de la théorie (qui correspondent aux faits utilisés pour la représentation des exemples) sont opérationnels. A la sortie, il est possible d'observer les résultats de l'apprentissage pour chacun des exemples.

Ce prototype nous a aussi servi à générer des cas d'essais pour tester les spécification d'un protocole de communication tel que ABP (`Alternating Bit Protocol`) [Geldrez et al. 1989].

4. L'apprentissage incrémental à partir d'exemples positifs et négatifs

L'apprentissage incrémental à partir d'exemples positifs permet d'obtenir une théorie positive complètement spécifiée du domaine, ainsi qu'une définition positive du concept satisfaisant le critère d'opérationnalité et couvrant tous les exemples positifs appris. Une définition du concept est souvent trop générale si elle n'est définie qu'à partir d'exemples positifs, elle peut aussi couvrir des exemples négatifs du concept. Il arrive parfois que ce soit plus facile de définir un concept par ce qu'il n'est pas, plutôt que de tenter de le définir entièrement par ce qu'il est. C'est pourquoi on veut apprendre à partir d'exemples négatifs.

On applique la méthode d'apprentissage incrémental aux exemples négatifs de la même façon qu'aux exemples positifs du chapitre précédent. Les exemples négatifs sont maintenant acceptés par le système. Ils sont regroupés selon trois nouveaux types, soient les types C, D et E. Le concept étudié reste le même: reconnaître une situation de *prendre le métro*.

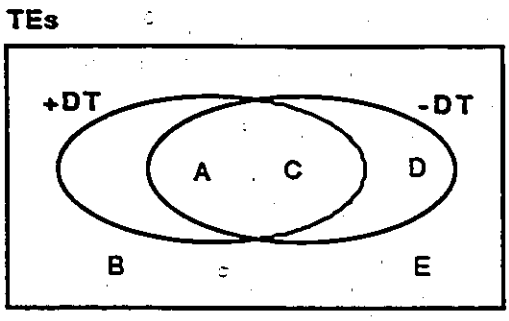


Fig. 4.1 *Les cinq types d'exemples d'entraînement*

La figure 4.1 présente les types d'exemples d'entraînement possibles pour l'apprentissage incrémental à partir d'exemples positifs et négatifs. Tous les exemples d'entraînement utilisés dans cette thèse sont listés dans l'appendice A. Les exemples positifs de type A et B ont été introduits au chapitre précédent. En réalité, les différents type d'exemples correspondent à un ensemble d'exemples. Le type A décrit des situations de *prendre le métro*, possibles pour le système (i.e. explicables par la théorie positive). Le type B décrit des situations de *prendre le métro*, impossibles pour le système (i.e. inexplicables par la théorie positive). Le type C correspond aux exemples pour prendre le

train, qui peuvent s'expliquer par la même séquence d'actions que pour prendre le métro (ils sont explicables par les deux théories). Le type D représente n'importe quelles situations autres que celle de *prendre le métro* et possibles pour le système (i.e. explicables par la théorie négative sous-spécifiée du domaine). Le type E correspond aux exemples qui ne décrivent pas une situation de prendre le métro et qui ne sont pas possibles pour le système (i.e inexplicables par la théorie négative sous-spécifiée du domaine).

Les exemples de type A, C et D sont pertinents pour l'apprentissage incrémental à partir d'exemples positifs et négatifs. Les exemples de type A permettent de spécifier la théorie positive du domaine et de générer la définition positive du concept. Les exemples de type C et D permettent de spécifier la théorie négative du domaine et la définition négative du concept. On verra au prochain chapitre que les exemples négatifs de type C permettent aussi de spécialiser la définition positive du concept. Les exemples de type B et E ne sont d'aucune utilité pour l'apprentissage incrémental, puisqu'aucune généralisation n'est obtenue par la méthode déductive avec leur théorie respective.

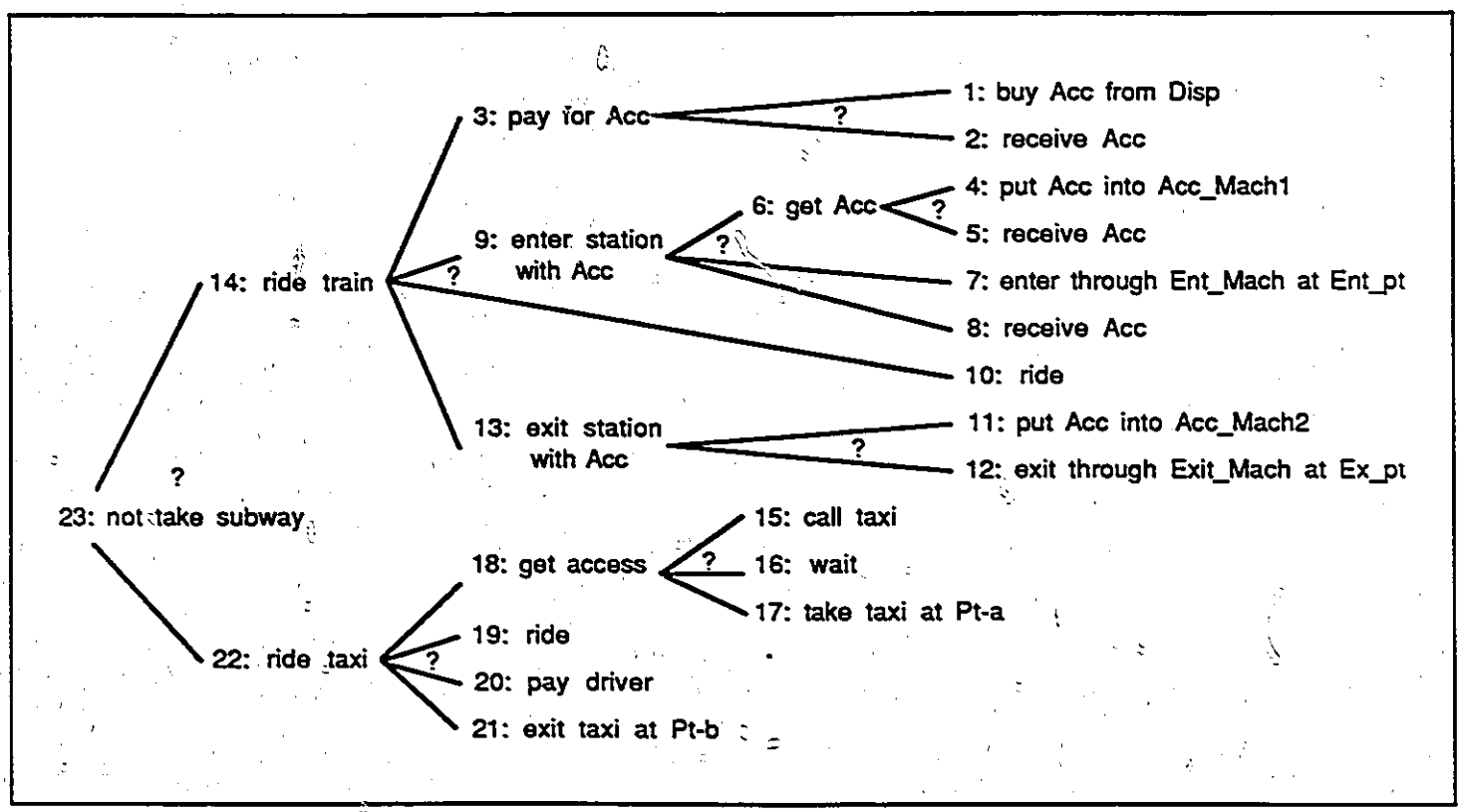


Fig. 4.2 *La théorie sous-spécifiée du domaine négative (-DT)*

La théorie négative sous-spécifiée du domaine (figure 4.2) inclut un ensemble de règles permettant d'expliquer comment les exemples négatifs ne font pas partie du concept étudié. La théorie négative sous-spécifiée du domaine est *incomplète*, *inconsistante* et *sous-spécifiée* pour des raisons similaires à celles exposées pour la théorie positive sous-spécifiée du domaine: incomplète, car elle ne peut pas expliquer tous les exemples négatifs du concept étudié; inconsistante, car elle peut expliquer des exemples positifs du concept étudié et sous-spécifiée, puisqu'elle n'a pas les informations nécessaires pour éliminer les combinaisons impossibles d'actions pour un exemple négatif. La théorie négative sous-spécifiée du domaine représente l'organisation de la séquence d'actions possibles pour un exemple négatif.

La théorie négative inclut toutes les règles et faits de la théorie positive entre autres. Cela correspond en réalité au fait qu'il est possible que deux concepts soient décrits par la même séquence d'actions. Par exemple, la séquence d'actions pour *prendre le métro* est similaire et peut même être identique, dans certains cas, à la séquence d'actions pour *prendre le train*. On verra au prochain chapitre comment il est possible de différencier les deux concepts. Idéalement, la théorie négative couvrirait tous les concepts autres que celui de *prendre le métro*, pour arriver à expliquer tous les exemples négatifs, chose difficile à réaliser encore de nos jours. La théorie négative comprend aussi des règles pour expliquer des exemples négatifs dont la séquence d'actions diffère de celle de la théorie positive. Ici, la théorie négative couvre aussi le concept de *prendre un taxi*. Ces règles serviront à expliquer les exemples négatifs dont la séquence d'actions correspond à une de ces situations.

La théorie négative du domaine complètement spécifiée se décrit de la même façon que la théorie positive du domaine complètement spécifiée. La différence réside dans le fait que la théorie négative du domaine est complètement spécifiée pour éliminer des combinaisons d'actions impossibles pour les exemples négatifs, au lieu d'éliminer les combinaisons d'actions impossibles pour les exemples positifs.

Comme auparavant la définition du concept est la règle générale couvrant tous les exemples appris. Pour être plus spécifique maintenant, la définition du concept[6] est divisée

[6] L'expression "définition du concept" sera dorénavant utilisée, pour signifier la conjonction de la définition positive et la négation de la définition négative du concept.

en deux généralisations. La première est appelée la définition positive du concept et la seconde la définition négative du concept. La première définition couvre tous les exemples positifs et correspond à la définition décrite au chapitre 3. La seconde définition couvre tous les exemples négatifs, sauf ceux déjà couverts par la définition positive du concept. Cette dernière est aussi composée pour l'instant d'une séquence d'actions accompagnée de contraintes procédurales. Pour obtenir une définition du concept plus consistante, la définition positive du concept est suivie de la négation de la définition négative du concept.

L'ordre d'entrée des exemples positifs et négatifs peut influencer les résultats finaux, telles la définition du concept et les théories spécifiées. Celles-ci peuvent être influencées par l'ordre d'entrée des exemples positifs de type A et des exemples négatifs de type C. Prenons un exemple positif de type A et un exemple négatif de type C avec une séquence identique d'actions pour les deux. Appelons les EE+A et EE-C. Deux cas sont possibles pour la représentation finale de `Act+` et `Act-`:

1) **Act+ ∩ Act- = {}**: `Act+` et `Act-` n'ont aucune combinaison identique d'actions.

- Aucun exemple négatif de type C a été fourni au système lors de l'apprentissage.

- EE+A a été appris avant EE-C[7] , `Act+` a été modifiée par l'apprentissage de l'EE+A. La séquence d'actions de EE-C satisfait `Act+`, alors la négation des descripteurs pertinents de EE-C s'ajoute à `Desc+`.

2) **Act+ ∩ Act- ≠ {}**: `Act+` et `Act-` ont au moins une combinaison identique d'actions.

- EE-C a été appris avant EE+A. EE-C ne satisfait pas `Act+`, alors `Act-` est généralisée pour accepter EE-C. EE+A ne satisfait pas la définition positive du concept, alors `Act+` est généralisée pour accepter le nouvel EE+A. Puisqu'il

[7] EE-C appris avant EE+A nous amène au deuxième cas.

n'est pas possible[8] de retirer une seule généralisation de la définition négative du concept.

4.1 L'algorithme principal

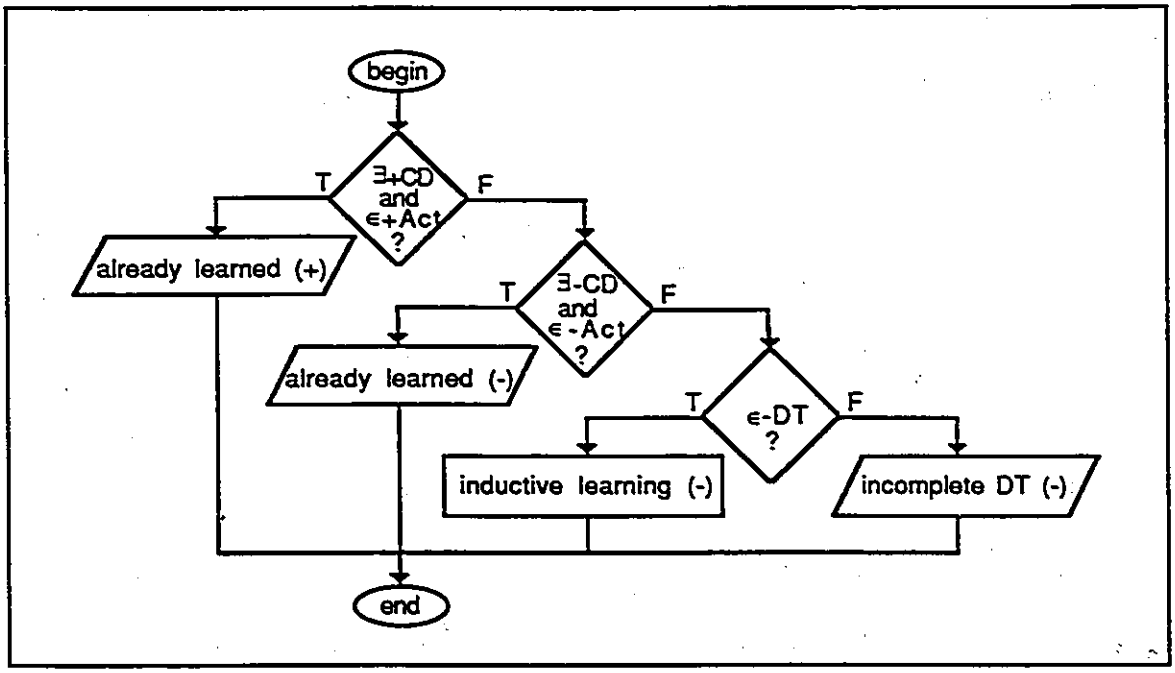


Fig. 4.3 *L'algorithme principal de l'apprentissage incrémental à partir d'exemples négatifs.*

L'algorithme (figure 4.3) fonctionne tant qu'il existe des exemples d'entraînement. Si l'exemple est positif, alors on applique l'algorithme général de l'apprentissage incrémental (figure 3.5) pour un exemple positif. Si l'exemple n'est pas positif, alors on vérifie s'il existe une définition positive du concept et si les actions de l'exemple satisfont une des combinaisons d'actions exprimées par cette définition (`∃+CD and ∈+Act`). Si oui, alors l'exemple a déjà été appris par la définition positive du concept (`already learned (+)`) (cela est dû au fait que la définition positive du concept est trop générale et couvre aussi des exemples négatifs). Si non, alors on vérifie s'il existe une définition négative du concept et si les actions de l'exemple satisfont une des combinaisons d'actions exprimées par cette définition (`∃-CD and ∈-Act?`). Si c'est le cas, alors l'exemple a déjà

[8] Une fois qu'une définition du concept est généralisée pour accepter un nouvel exemple, il est difficile voire même impossible (si ce n'est la première généralisation) de retrouver les actions, les opérateurs et les contraintes procédurales qui correspondent à un exemple négatif particulier à partir de Act-.

été appris par la définition négative du concept (already_learned_(-)). Si non, alors on vérifie si l'exemple peut être généralisé par la théorie négative du domaine (∈-DT?). Si oui, alors l'algorithme passe à l'apprentissage inductif pour un exemple négatif (inductive_learning_(-)) et si aucune généralisation n'est obtenue, c'est que la théorie négative du domaine est incomplète (incomplete_DT_(-)).

4.2 Exemple de l'apprentissage incrémental à partir d'exemples positifs et négatifs.

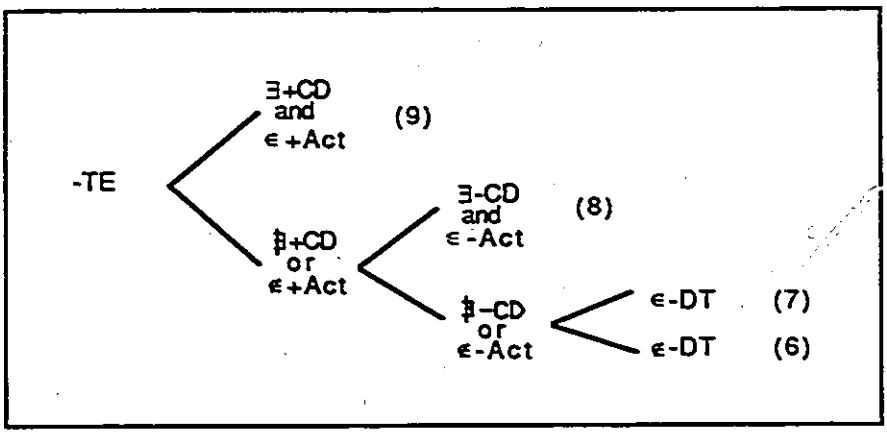


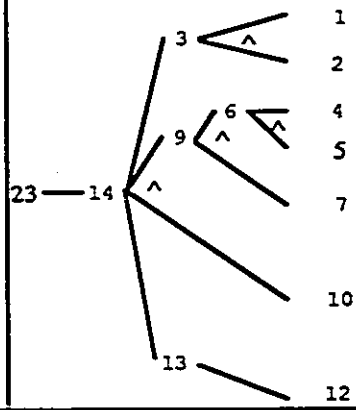
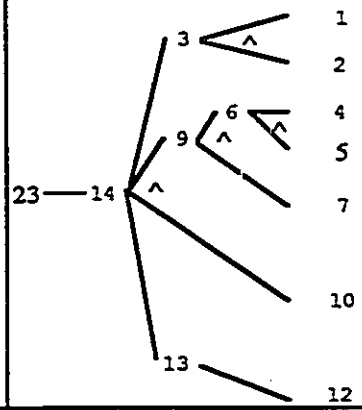
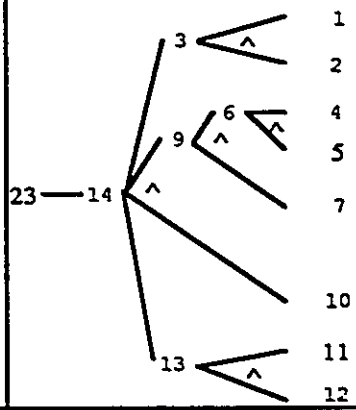
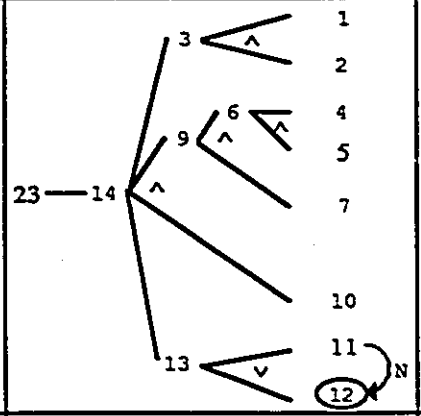
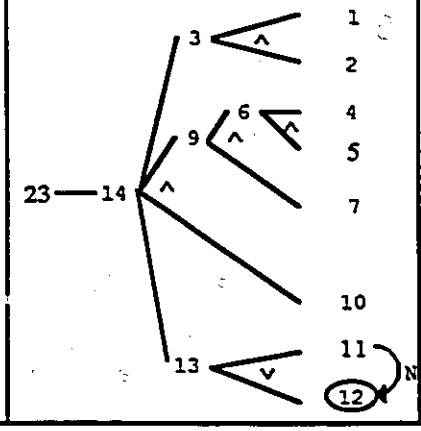
Fig. 4.4 *Les scénarios possibles pour les exemples négatifs.*

Encore une fois, pour montrer la portée de l'algorithme, un exemple couvrant tous les scénarios possibles à partir d'exemples négatifs est donné à la figure 4.4. Pour satisfaire les conditions environnantes des exemples présentés ici, on poursuit la séance d'apprentissage incrémental du chapitre précédent (figure 3.12). Les scénarios possibles pour les exemples positifs restent inchangés.

#	TE's type	TE's identification	∃+CD and ∈+Act	∃-CD and ∈-Act	∈-DT	Process
6	E	Cinema	0	0	0	incomplete DT (-)
7	C	Train1	0	0	1	inductive learning (-)
7	C	Train4	0	0	1	inductive learning (-)
8	C	Train2	0	1	-	already learned (-)
9	C	Train3	1	-	-	already learned (+)

Fig. 4.5 *Les exemples utilisés pour la séance d'apprentissage incrémental. La valeur "1" signifie que la condition est vraie, la valeur "0" signifie que la condition est fausse et "-" signifie que la condition ne s'applique pas.*

De la même façon qu'au chapitre 3, le tableau de la figure 4.5 se veut un résumé de la séance d'apprentissage incrémental qui suit. Si la séquence d'actions de l'exemple satisfait une des combinaisons représentées par la définition positive du concept (∃+CD and ∈+Act), alors l'exemple négatif est déjà appris par cette définition (already learned (+)). Si la séquence d'actions de l'exemple satisfait une des combinaisons représentées par la définition négative du concept (∃-CD and ∈-Act), alors l'exemple négatif est déjà appris par cette définition (already learned (-)). Si l'exemple n'a pas déjà été appris, parce qu'il n'existait pas de définition du concept et qu'il appartient à la théorie négative du domaine (∈-DT), alors c'est la première généralisation obtenue (inductive learning (-)). De la même façon qu'un exemple positif peut ne pas être explicable par la théorie positive sous-spécifiée du domaine, un exemple négatif peut ne pas être explicable par la théorie négative sous-spécifiée du domaine (∉-DT), alors l'algorithme passe à l'exemple suivant (incomplete DT (-)). Dans le cas où l'exemple n'est pas déjà appris, qu'il existe une définition du concept et que l'exemple est explicable par la théorie négative sous-spécifiée du domaine, alors la définition négative du concept est spécifiée davantage par l'apprentissage inductif (inductive learning (-)) pour accepter cet exemple.

TE *Process*	GenExpl (-)	FSDT (-)	CD (-)
#6 Cinema (-) *incomplete DT (-)*			
#7 Train1 (-) *inductive learning (-)*			1∧2∧4∧5∧7∧10∧12
#7 Train4 (-) *inductive learning (-)*			1∧2∧4∧5∧7∧10∧(11∨12)∧ C11∧11N12
#8 Train2 (-) *already learned (-)*			1∧2∧4∧5∧7∧10∧(11∨12)∧ C11∧11N12

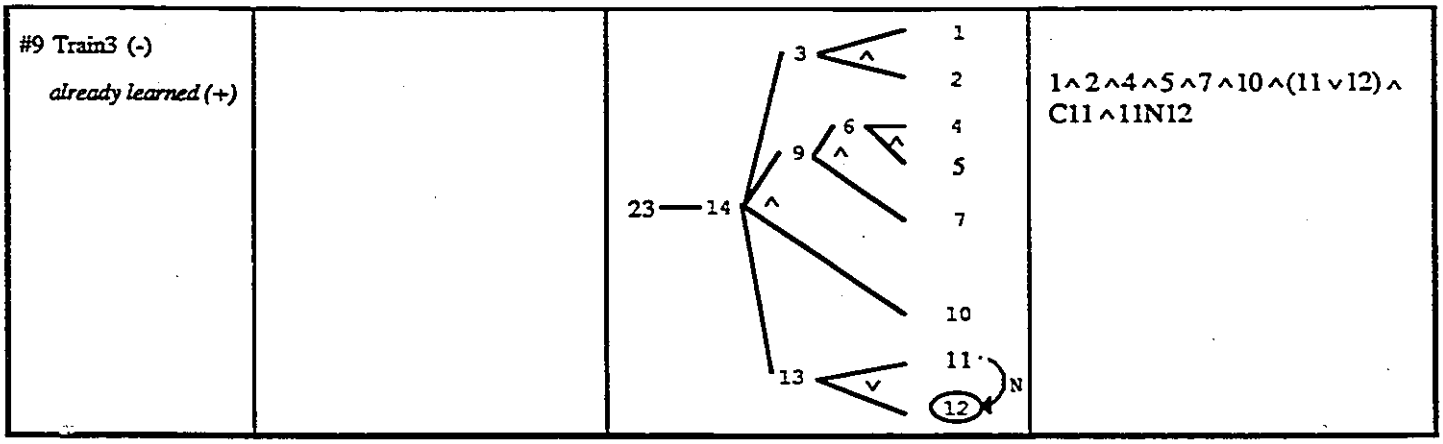


Fig. 4.6 *Exemple de l'apprentissage incrémental à partir d'exemples négatifs.*

De façon similaire aux scénarios présentés pour l'apprentissage incrémental à partir d'exemples positifs, les scénarios pour les exemples négatifs (figure 4.6) s'interprètent comme suit. Le premier exemple négatif (`Cinema`) n'influence pas les résultats de l'apprentissage, puisque la théorie négative du domaine est incomplète (`incomplete DT(-)`).

Le second exemple (`Train1`) appartient à aucune des définitions, positive ou négative, de la définition du concept existante. Il est le premier exemple négatif à être généralisé par la méthode déductive. La définition négative du concept est générée à partir de la théorie négative complètement spécifiée qui correspond à la généralisation obtenue.

La séquence d'actions du troisième exemple (`Train4`) ne satisfait pas les conditions exprimées par les définitions du concept positive et négative, mais il est quand même possible d'apprendre à partir de celui-ci, car il appartient à la théorie négative sous-spécifiée du domaine. L'algorithme passe à l'apprentissage inductif (`inductive learning (-)`) avec la généralisation obtenue, pour raffiner davantage la théorie spécifiée négative et régénérer une définition négative du concept.

L'avant-dernier exemple négatif de la séance d'apprentissage (`Train2`) a déjà été appris (`already learned(-)`), car la séquence d'actions de l'exemple satisfait une des combinaisons possibles exprimées par la définition négative du concept. Il y a rien de nouveau à apprendre à partir de cet exemple et les résultats de l'apprentissage incrémental restent inchangés.

Enfin la séquence d'actions du dernier exemple (Train3) est identique à une séquence d'actions d'un exemple positif déjà appris, puisqu'il satisfait une des combinaisons possibles exprimée par la définition positive du concept. Il a déjà été appris par le système. C'est donc dire que la définition positive du concept est trop générale et accepte aussi des exemples négatifs. On verra au prochain chapitre comment la définition positive du concept est rendue plus consistante.

Etant donné la pertinence des exemples négatifs fournis au système, la définition négative du concept correspond aux séquences d'actions apprises, qui ne représentent pas une situation de *prendre le métro*. En fait, il existe bien d'autres situations qui ne correspondent pas à *prendre le métro*, mais celles apprises dépendent de la théorie sous-spécifiée du domaine et des exemples négatifs fournis au système. Ici, les exemples appris ont permis de générer la définition négative du concept de façon à ce qu'elle reconnaisse les séquences d'actions pour *prendre le train* de la même façon que les exemples Train1, Train2 et Train4.

```
CD ⇔
     ((1 ∧ 2) ∧ (4 ∧ 7 ∧ 8) ∧ 10 ∧ 12 ∧
     C1 ∧ C7 ∧ C10 ∧ C12 ∧2R1 ∧ 2N4 ∧ 4R2 ∧ 4N7 ∧ 8R7 ∧ 2O8 ∧ 4O8) ∧
     not (1 ∧ 2 ∧ 4 ∧ 5 ∧ 7 ∧ 10 ∧ (11 ∨12) ∧ C12 12R11)
```

Fig. 4.7 *La définition du concept pour les exemples positifs et négatifs. Les deux premières lignes correspondent à la définition positive du concept (figure 3.12) et la dernière ligne correspond à la négation de la définition du concept négative.*

La définition finale du concept (figure 4.7) correspond à la conjonction de la définition positive du concept (figure 3.12) et à la négation de la définition négative du concept (figure 4.6). La définition du concept couvre les exemples positifs appris par la définition positive et exclut les exemples négatifs appris par la définition négative. La définition positive du concept couvre tous les exemples qui s'expliquent de la même façon que les exemples de *prendre le métro* de NYC et de Toronto (se référer à la section 3.4 pour un exemple de son interprétation). La définition négative du concept couvre les exemples négatifs qui s'expliquent de la même façon que les exemples de *prendre le train* Train1 et Train4. Par contre, on a pu observer que la définition positive du concept est trop générale et couvre aussi certains exemples négatifs, comme c'est le cas pour l'exemple de *prendre le train* Train3.

Sans changer la méthode, on veut une définition positive du concept plus consistante, pour que les exemples négatifs, avec une séquence identique d'actions à celle d'un exemple positif déjà appris, soient à l'avenir exclus par la définition positive du concept.

5. Apprendre davantage à partir d'exemples négatifs

Pour représenter l'idée de base de ce chapitre, reprenons encore une fois l'exemple du début. La représentation mentale qu'on a de *prendre le métro* est parfois trop générale, puisqu'elle englobe aussi *prendre le train*. Il est possible de prendre le train en utilisant le même procédé que pour *prendre le métro*. A partir de cette expérience, on restreint l'idée qu'on a à *prendre le métro* à l'aide des descripteurs qui distinguent la situation de *prendre le train* de celle de *prendre le métro*.

En fait, l'apprentissage présenté dans les deux chapitres précédents peut apprendre une définition du concept qui se contredit. Cela est possible si les séquences d'actions sont identiques pour les exemples positifs et négatifs. De cette façon toutes les combinaisons d'actions acceptées par la définition positive du concept sont aussi exclues par la négation des actions acceptées par la définition négative du concept. C'est donc insuffisant d'apprendre une définition du concept seulement basée sur les combinaisons des actions des exemples. On a besoin d'apprendre davantage à partir des exemples, quelque chose qui nous permette de distinguer les exemples positifs et négatifs pour que chacun d'eux soit couvert par la définition positive du concept et exclu par la définition négative du concept.

On peut remarquer qu'une partie de la représentation de l'exemple n'a pas servi jusqu'à maintenant aux buts de l'apprentissage. En fait chaque exemple est composé de deux niveaux connus de connaissances du système. Le premier niveau participe activement à l'apprentissage de l'exemple, tandis que le second y participe passivement. En réalité, le premier niveau de connaissances correspond aux actions inférées par la théorie sous-spécifiée du domaine et le deuxième niveau, aux descripteurs attachés à ces actions et inférés par les connaissances générales.

Auparavant seulement les actions pour un exemple pouvaient être pertinentes, maintenant toutes les caractéristiques d'un exemple peuvent être pertinentes, que ce soit une action ou un descripteur. Les exemples négatifs dont la séquence d'actions a déjà été apprise par la définition positive du concept, doivent être éliminés par celle-ci pour obtenir une définition plus consistante. Si la séquence d'actions n'est pas encore apprise par la définition positive du concept, alors cet exemple sera couvert par la définition négative du concept.

On veut éviter que la définition positive du concept soit trop générale au point de couvrir des exemples négatifs. On veut la rendre plus consistante en éliminant les exemples négatifs qui sont déjà couverts par celle-ci à l'aide des descripteurs pertinents qui accompagnent la séquence d'actions de ces exemples. Si les descripteurs sont pertinents pour un exemple négatif, alors ils sont impertinents pour un exemple positif et vice versa.

Les situations intéressantes à observer dans ce chapitre, sont celles où la séquence d'actions pour *prendre le train* sont identiques à une situation pour *prendre le métro*. Elles correspondent aux exemples négatifs de type C (i.e. explicables par les deux théories).

```
Donné:
• Concept étudié
• Exemples d'entraînement positifs et négatifs
• Théories du domaine positive et négative
• Connaissances générales positives et négatives
• Critère d'opérationnalité

A déterminer:
• Des théories positive et négative du domaine avec des
  opérateurs connus (FSDT+/-)
• Une définition du concept qui:
- satisfait le critère d'opérationnalité
- couvre tous les exemples positifs et exclut tous les exemples
  négatifs
- rend explicites les contraintes procédurales implicitement
  représentées dans les exemples d'entraînement
```

Fig. 5.1 *Définition du problème de l'apprentissage incrémental à partir d'exemples positifs et négatifs.*

La figure 5.1 définit à nouveau le problème de l'apprentissage incrémental à partir d'exemples positifs et négatifs. De la même façon qu'au chapitre précédent, on veut déterminer une définition du concept qui couvre les exemples positifs, exclut les exemples négatifs et satisfait le critère d'opérationnalité. La définition du concept est généralisée par la définition positive du concept, puis spécialisée par la définition négative du concept. Par contre la définition positive du concept peut être trop générale et couvre parfois des exemples négatifs de type C. L'utilisation de descripteurs permet de définir une définition du concept plus consistante. Par l'ajout de descripteurs au système, les exemples négatifs couverts par la définition positive du concept sont maintenant exclus. Grâce aux connaissances générales données au système, il est possible de générer une définition du concept et des théories spécifiées du domaine positive et négative plus consistantes encore.

5.1 Les connaissances générales positives et négatives

Les connaissances générales positives et négatives permettent de rendre explicites d'autres liens implicitement représentés par les exemples d'entraînement. Ces connaissances sont représentées indépendamment de la théorie du domaine, sous forme utilisable par le système. Elles permettent d'identifier les descripteurs pertinents pour un exemple, puisqu'il n'est pas possible d'apprendre s'il est impossible de distinguer ce qui est important de ce qui est accidentel [Winston 1984]. Ce sont des données qui ont déjà été fournies au système, mais n'ont pas été utilisées pour obtenir l'explication de l'exemple d'entraînement. Les connaissances générales positives, représentent les descripteurs pertinents possibles pour un exemple positif. A l'opposé les connaissances générales négatives représentent les descripteurs pertinents possibles pour qu'un exemple négatif ne fasse pas partie du concept. Une fois appris, de tels descripteurs permettent de distinguer les exemples positifs et les exemples négatifs ayant une séquence identique d'actions. Par exemple, si *prendre le métro* et *prendre le train* sont décrits par la même séquence d'actions, alors un descripteur tel que le train ne se déplace pas dans un tunnel (`is not in a tunnel`) permet de distinguer entre les deux exemples. La négation de cette caractéristique est ajoutée à la définition positive du concept (`Desc+`) par l'apprentissage inductif. Les exemples positifs qui vont suivre et qui ont une séquence identique d'actions à celle de l'exemple positif déjà appris, devront satisfaire cette nouvelle condition de la définition positive du concept. En revanche tous les exemples négatifs suivants représentés par une séquence déjà apprise d'actions par un exemple positif et ayant cette caractéristique sont dorénavant exclus par la définition positive du concept.

A l'opposé des presque-exemples (`near miss`) utilisés par Winston [Winston 1984], les exemples négatifs peuvent avoir plus d'une différence avec un exemple positif du concept, pour qu'il soit possible de restreindre la définition du concept.

```
BK1+: distance between Entrance and Exit ≤50 km
BK2+: has no sleeping car
BK3+: has no washroom
BK4+: has ≤5 employees

BK1-: has commodity to travel ⇔
         has washroom v
         has sleeping car v
         has goodsvan v
         has mailvan v
         has restaurant car
BK2-: is not in tunnel
BK3-: distance between Entrance and Exit > 50 km
BK4-: number of employees >5
```

Fig. 5.2 *Les connaissances générales positives (BK+) et négatives (BK-) qui rendent la définition du concept et la théorie du domaine plus consistantes.*

Voyons dans la figure 5.2, à titre d'exemple, quelles connaissances générales, positives ainsi que négatives, permettent de raffiner la théorie du domaine *prendre le métro* de sorte que chaque exemple correspondant à *prendre le train* soit dorénavant éliminé. Nous obtenons une définition du concept et une théorie du domaine plus consistantes qu'auparavant.

5.2 EBG et les connaissances générales

A l'aide des connaissances générales positives et négatives, il est possible d'apprendre davantage à partir de chacun des exemples positifs et négatifs. Pour chaque exemple, on connaît les expressions qui représentent les actions et celles qui représentent les descripteurs de l'exemple. Les actions sont expliquées par la théorie du domaine. Pour être pertinent, un descripteur doit faire partie de la structure généralisée de l'exemple. Pour faire partie de cette structure, un descripteur doit apparaître dans l'exemple et dans les connaissances générales appropriées. Un descripteur qui n'a pas vraiment d'importance pour la description de la situation est un descripteur impertinent. Par exemple la couleur du métro ou le nom du chauffeur importe peu dans la description de *prendre le métro*. Soit que ces descripteurs n'apparaissent pas dans tous les exemples, ou ils n'apparaissent pas dans les connaissances générales.

Pour chaque expression de l'exemple, si c'est un descripteur, l'algorithme vérifie s'il appartient aux connaissances générales. Si oui, alors ce descripteur fait partie de l'explication de l'exemple: il est pertinent. Sinon l'algorithme passe à l'expression suivante

de l'exemple. Si l'expression ne représente pas un descripteur, alors c'est une action et l'algorithme de la phase déductive (section 3.2) est exécuté. Le processus est répété jusqu'à ce qu'il n'y ait plus d'expressions à expliquer pour l'exemple.

5.3 Les théories du domaine complètement spécifiées

Les théories du domaine spécifiées sont plus spécifiques encore grâce à l'ajout de connaissances générales au système d'apprentissage incrémental. Maintenant en plus de représenter toutes les combinaisons d'actions possibles, les théories du domaine spécifiées représentent aussi toutes les combinaisons possibles de descripteurs pertinents pour les exemples appris.

La théorie positive spécifiée du domaine est modifiée par les exemples positifs de type A et dans certains cas par les exemples négatifs de type C. Ces derniers peuvent modifier les descripteurs positifs (`Desc+`) que si leur séquence d'actions correspond à une des combinaisons d'actions positives (`Act+`). La théorie négative spécifiée du domaine est modifiée par les exemples négatifs de type C dans certains cas et par les exemples négatifs de type D. La structure des actions (`Act`) des théories spécifiées du domaine est initialisée avec la première structure généralisée pour un exemple. Chaque nouvel exemple qui n'a pas déjà été appris par le système, entraîne la modification de la structure des actions de la théorie spécifiée du domaine. Un nouvel exemple positif de type A modifie la séquence positive d'actions (`Act+`) et un nouvel exemple négatif de type C ou D modifie la séquence négative d'actions (`Act-`). Toute modification d'une théorie spécifiée du domaine entraîne la modification de la définition du concept. Ainsi la théorie positive spécifiée du domaine couvre tous les exemples positifs et exclut certains exemples négatifs de type C. La théorie négative spécifiée du domaine couvre les exemples négatifs, sauf ceux déjà couverts par la théorie positive spécifiée du domaine.

5.4 Une définition du concept plus consistante

La définition du concept est plus consistante qu'aux chapitres 3 et 4: elle couvre tous les exemples positifs et exclut tous les exemples négatifs appris. La définition du concept est une conjonction de la définition positive du concept et la négation de la définition négative du concept (figure 5.3). La définition positive du concept couvre tous les exemples positifs et exclut certains exemples négatifs, tandis que la définition négative du concept couvre les exemples négatifs (sauf ceux déjà couverts par la définition positive

du concept). La définition du concept exclut les exemples négatifs en utilisant la négation de la définition négative du concept. Les définitions du concept sont composées d'une séquence d'actions (`Act+/-`) et d'une liste de descripteurs (`Desc+/-`). Les séquences d'actions positives et négatives représentent les combinaisons possibles d'actions pour les exemples positifs et négatifs. Les descripteurs positifs et négatifs représentent les descripteurs pertinents pour les exemples positifs et négatifs. Chaque action ou descripteur de la définition du concept est opérationnel.

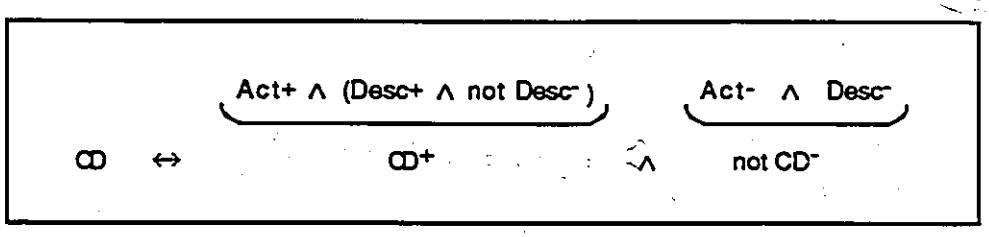


Fig. 5.3 *La définition du concept (CD) pour le concept étudié.*

`Act` est une conjonction/disjonction d'actions accompagnées de contraintes procédurales. Toutes les actions sont opérationnelles et correspondent aux expressions du dernier niveau de la théorie spécifiée du domaine. Les conjonctions/disjonctions sont définies selon les opérateurs de la théorie spécifiée du domaine. Les opérateurs et les contraintes procédurales de la théorie spécifiée du domaine sont propagés au dernier niveau de la structure et complètent ainsi les liaisons entre les actions de `Act`. Cette dernière représente toutes les combinaisons possibles d'actions pour un exemple.

`Desc+/-` est basée sur les assomptions du monde fermé des descripteurs pertinents pour les exemples positifs et négatifs de la même façon que `Act+` est basée sur les assomptions du monde fermé pour les actions (chapitre 3), c'est-à-dire la valeur logique pour un descripteur est "0" s'il est absent et "1" s'il est présent. `Desc+` est composée de deux listes de descripteurs: une liste pour les descripteurs appris par les exemples positifs et une liste de descripteurs appris par les exemples négatifs, lorsque la séquence d'actions de ce dernier satisfait `Act+`. Pour apparaître dans `Desc+` un descripteur doit être présent dans au moins un des exemples positifs, ou appartenir à un exemple négatif dont la séquence d'actions a déjà été apprise par un exemple positif et il doit être pertinent. `Desc+` est donc composée de descripteurs pertinents pour les exemples positifs et de la négation de descripteurs pertinents pour les exemples négatifs.

5.5 L'influence des exemples négatifs

Les exemples négatifs de type D (Fig. 4.1) influencent la théorie spécifiée du domaine et la définition du concept négatives de la même façon que les exemples positifs de type A influencent la théorie spécifiée du domaine et la définition du concept positives. Les nouveaux exemples négatifs de type D permettent de généraliser la théorie spécifiée du domaine et la définition du concept négatives pour qu'elles couvrent de nouvelles situations n'appartenant pas au concept étudié, de la même façon que la théorie spécifiée du domaine et la définition du concept positives sont généralisées pour couvrir de nouvelles situations appartenant au concept étudié par de nouveaux exemples positifs de type A. Rappelons que les exemples négatifs de type D sont explicables par la théorie négative du domaine seulement. Ces exemples ne satisfont dans aucun cas la définition positive du concept. Ils n'appartiennent pas non plus à la théorie positive du domaine. Ils peuvent ou non être couverts par la définition négative du concept. S'ils appartiennent à cette dernière définition, c'est qu'ils ont déjà été appris. Ils n'apportent rien de nouveau à apprendre pour le système à ce moment. S'il n'ont pas déjà été appris, alors ils permettent ainsi au système d'apprendre une nouvelle situation. La théorie spécifiée du domaine et la définition du concept négatives (`Act-` et `Desc-`) sont révisées en conséquence.

Les exemples négatifs de type E sont équivalents aux exemples positifs de type B. Aucune structure explicative généralisée ne peut être obtenue par EBG pour de tels exemples et le système d'apprentissage les ignore tout simplement et passe au suivant.

Les exemples négatifs de type C sont uniques par leur façon d'influencer le système d'apprentissage. Rappelons que ces exemples appartiennent aux deux théories du domaine et qu'ils sont négatifs. Ils n'ont aucun équivalent chez les exemples positifs. Chaque nouvel exemple négatif de type C peut influencer la définition positive du concept ou bien la théorie spécifiée du domaine et la définition du concept négatives, selon les conditions environnantes de l'exemple. Un exemple négatif de type C peut être déjà appris par la définition positive du concept ou par la définition négative du concept. Le premier cas se produit si le système a déjà appris un exemple positif dont la séquence d'actions est identique à l'exemple négatif. A ce moment-là, le système pour éviter de couvrir à nouveau un tel exemple à partir de la définition positive du concept, retient la négation des descripteurs pertinents de l'exemple négatif (dans la liste positive de descripteurs (`Desc+`) de la définition positive du concept) pour spécialiser la définition positive du concept. Le second cas (déjà appris par la définition négative du concept) se produit si aucun exemple

positif avec une séquence identique d'actions a été appris ou qu'un autre exemple négatif de C avec une séquence d'actions et des descripteurs pertinents identiques a été appris avant. Aucune modification n'est alors apportée à la définition négative du concept et le système passe à l'exemple suivant. Par contre si l'exemple négatif de type C appris avant, n'a que la séquence identique d'actions, alors le nouvel exemple négatif entraîne des modifications seulement pour Desc- de la définition négative du concept et définition du concept couvre ainsi l'exemple négatif. Si un exemple négatif de type C est le premier du genre à être fourni au système, c'est-à-dire qu'aucun autre exemple négatif de type C avec une séquence identique d'actions n'a été appris avant, alors la séquence d'actions négative (Act-) et la liste de descripteurs négatifs (Desc-) (de la définition négative du concept) sont généralisées pour couvrir cet exemple. La définition du concept est ainsi spécialisée.

5.6 Les scénarios possibles pour un exemple positif avec les connaissances générales positives

Les modifications apportées au système sont observables à partir des scénarios possibles pour un exemple positif. Aucune modification a été apporté à l'algorithme général de l'apprentissage incrémental (figure 3.5) pour les exemples positifs. C'est pour cela qu'il n'est pas présenté ici. Les seules modifications apparaissent plutôt au niveau des processus de l'algorithme. Ces modifications proviennent du fait que la définition positive du concept est maintenant représentée par une séquence d'actions suivie d'une liste de descripteurs pertinents pour un exemple positif, au lieu d'une simple séquence d'actions, comme c'est le cas pour les chapitres 3 et 4.

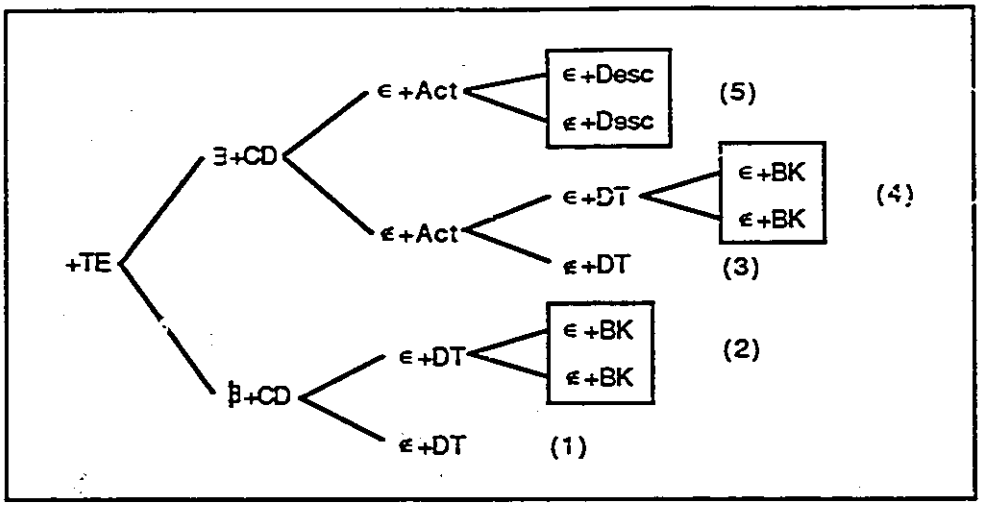


Fig. 5.4 *Les scénarios possibles pour un exemple positif.*

La figure 5.4 correspond à peu de chose près à la figure 3.10. Les scénarios possibles sont identifiés par les chiffres entre parenthèses à droite de la figure. Ils correspondent à ceux de la figure 3.10. Rappelons que chaque branche d'un sous-arbre correspond aux conditions environnantes à vérifier. Les sous-arbres encadrés de la structure représentent les différences entre les deux figures. Seules ces différences sont discutées ici.

Un exemple positif peut avoir déjà été appris, si au moins la séquence d'actions (y compris les opérateurs et les contraintes procédurales) satisfait une des combinaisons d'actions exprimées par Act+. Les descripteurs peuvent avoir été appris ou non (∈ ou ∉+Desc). Cela correspond au sous-arbre supérieur encadré. Les deux autres sous-arbres encadrés couvrent les conditions environnantes: appartenir aux connaissances générales positives ou non (∈ ou ∉+BK) accompagnées de différentes conditions environnantes. L'apprentissage inductif s'effectue de la même façon qu'à la section 3.3, si aucun descripteur pertinent n'est donné par l'exemple (∉+BK). Par contre s'il existe des descripteurs pertinents (∈+BK), le processus applique en plus, la table de décision aux descripteurs pertinents appris pour un exemple positif. Il génère ainsi des contraintes procédurales pour lier les descripteurs de chacune des listes de descripteurs de la définition positive du concept. Le dernier sous-arbre encadré représente la première généralisation obtenue pour un exemple positif par la phase déductive. La première généralisation pour un exemple est au moins composée d'une séquence d'actions et s'il existe des descripteurs pertinents pour l'exemple (∈+BK), alors elle comprend aussi une liste de descripteurs.

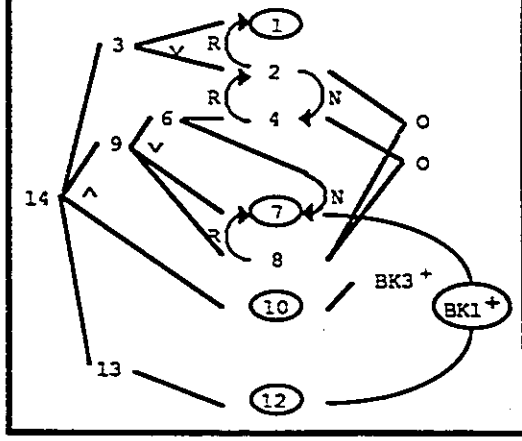
FSDT (+)	CD (+)
	(1 ∨ 2) ∧ (4 ∨ 7 ∨ 8) ∧ 10 ∧ 12 ∧ C1 ∧ C7 ∧ C10 ∧ C12 ∧ 2R1 ∧ 2N4 ∧ 4R2 ∧ 4N7 ∧ 8R7 ∧ 2O8 ∧ 4O8 ∧ (BK3+ ∨ BK1+) ∧ CBK1+

Fig. 5.5 *Les connaissances générales incorporées à l'apprentissage des exemples du chapitre 3.* BK1+ *et* BK3+ *représentent de telles connaissances.*

La figure 5.5 montre comment les connaissances générales positives sont incorporées au résultat de l'apprentissage des exemples du chapitre 3; on observe entre autres que:

1) la distance entre la station d'entrée et la station de sortie est moindre que 50 km (BK1+ lie la station d'entrée et la station de sortie de l'exemple);

2) il n'y a pas de salle de bain dans un métro (BK3+).

De plus l'analyse des relations a permis d'observer que le descripteur BK1+ est obligatoire pour les exemples positifs appris.

5.7 Les scénarios possibles pour un exemple négatif

Etant donné que le principal intérêt de ce chapitre s'adresse particulièrement aux exemples négatifs, dont la séquence d'actions est déjà apprise par la définition positive du concept, seulement la partie de l'algorithme traitant de tels exemples est illustrée ici (figure 5.6). L'algorithme pour les autres exemples négatifs est semblable à celui présenté pour les exemples positifs.

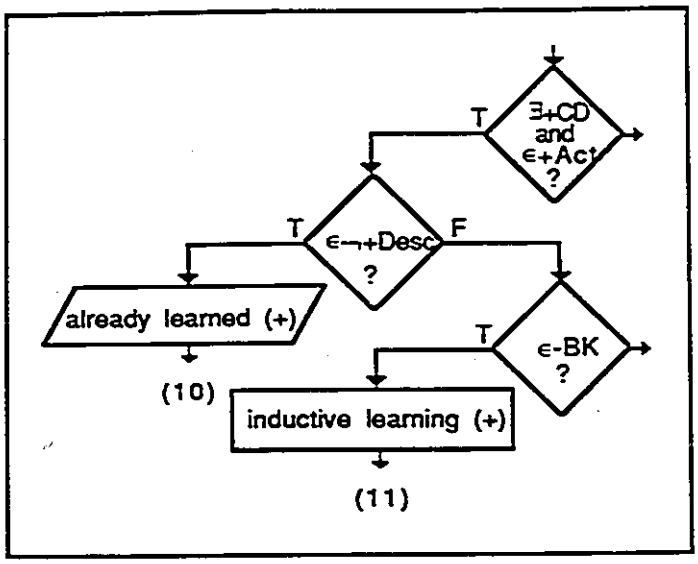


Fig. 5.6 *L'algorithme et les scénarios possibles pour les exemples négatifs dont la séquence d'actions est déjà apprise par la définition positive du concept.*

Seulement deux scénarios sont possibles pour les exemples négatifs en question. Ils correspondent aux deux branches de l'algorithme et sont identifiés par le chiffre entre parenthèses[9].

La séquence d'actions d'un exemple négatif est déjà apprise, s'il existe une définition positive du concept et si les actions de l'exemple satisfont une des combinaisons représentées par cette définition (∃+CD and ∈+Act).

Si au moins un des descripteurs de l'exemple négatif satisfait la négation d'un des descripteurs de la définition positive du concept (∈¬+Desc), alors l'exemple est déjà appris (already learned (+)) ou plutôt: déjà exclu. Dans le cas contraire et si l'exemple comprend des descripteurs pertinents (∈-BK), alors la négation de chacun d'eux est ajoutée à Desc+ de la définition du concept (inductive learning (-)).

5.8 Exemple

[9] La structure illustrant les différents scénarios ainsi que le tableau qui résume la séance d'apprentissage, ne sont pas présentés ici étant donné qu'il est facile d'identifier les conditions environnantes pour chacun des cas.

Les résultats des exemples positifs et négatifs appris jusqu'à maintenant (figures 5.5 et 5.7) sont utilisés pour poursuivre la séance d'apprentissage à partir d'exemples négatifs, dont la séquence d'actions est déjà apprise par la définition positive du concept.

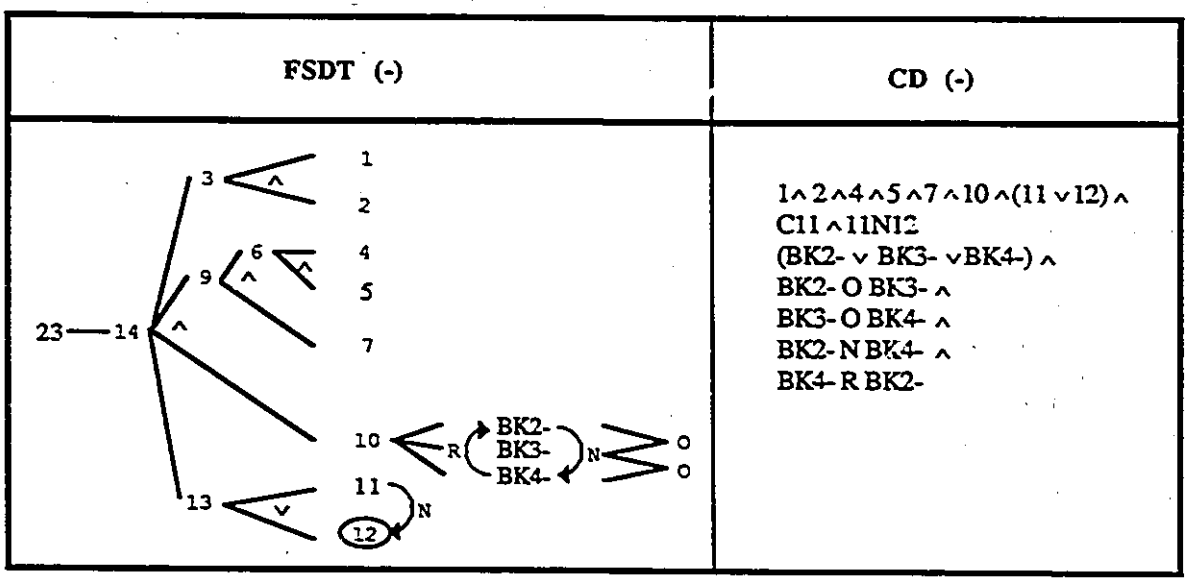


Fig. 5.7 *Les connaissance générales incorporées dans l'apprentissage des exemples négatifs du chapitre 4.* BK2−, BK3− *et* BK4− *représentent les connaissances générales pertinentes pour les exemples négatifs.*

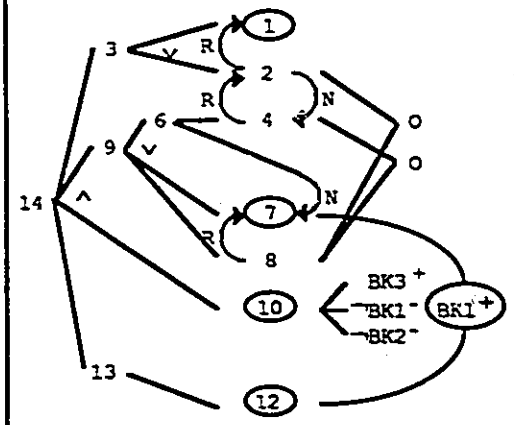
TE *Process*	FSDT (+)	CD (+)
#10 Train3 (-) *already learned (+)*		
#11 Train5 (-) *inductive learning (+)*		(1 ∨ 2) ∧ (4 ∨ 7 ∨ 8) ∧ 10 ∧ 12 ∧ C1 ∧ C7 ∧ C10 ∧ C12 ∧ 2R1 ∧ 2N4 ∧ 4R2 ∧ 4N7 ∧ 8R7 ∧ 2O8 ∧ 4O8 ∧ (BK3+ ∨ BK1+) ∧ CBK1+ ∧ ¬(BK1- ∧ BK2-)

Fig. 5.8 *La séance d'apprentissage incrémental à partir d'exemples négatifs dont la séquence d'actions est déjà apprise par la définition positive du concept. Le symbole "¬" signifie "not".*

L'exemple Train3 a déjà été appris, puisque sa séquence d'actions satisfait une des combinaisons d'actions représentées par Act+ et qu'un de ces descripteurs satisfait la négation d'un des descripteurs de Desc+. En effet, la négation de BK1+ (distance entre la station d'entrée et la station de sortie ≤ 50 km) correspond à BK3- (distance entre la station d'entrée et la station de sortie > 50 km). La définition positive du concept possède donc les connaissances suffisantes pour exclure un tel exemple, rien n'est changé aux résultats.

La séquence d'actions du second exemple (Train5) est déjà appris par la définition positive du concept, mais aucun descripteur (Desc+) ne permet pas de rejeter cet exemple, même s'il est négatif. On veut apprendre à partir de cet exemple, pour éviter qu'il soit accepté à l'avenir par cette définition. Pour ce faire, la négation de ces deux descripteurs pertinents (BK1- et BK2-) est ajouté à Desc+. Ainsi la prochaine fois qu'un tel exemple sera présenté, la définition positive du concept sera suffisamment consistante pour l'exclure (l'exemple sera déjà appris). La définition finale du concept à partir des théories positive et négative correspond à la conjonction de la définition positive et de la négation de la définition négative du concept.

6. Revue de littérature

Cinq méthodes mixtes d'apprentissage sont présentées ici. Chacune des méthodes mixtes utilise différemment les avantages et les bénéfices de l'apprentissage déductif et inductif. Entre autres, la méthode EBL est utilisée pour apprendre les combinaisons possibles de caractéristiques pour un exemple et la méthode SBL est utilisée pour apprendre les caractéristiques communes à plusieurs exemples. Toutes ces méthodes mixtes utilisent plusieurs exemples positifs et négatifs. Seulement les deux méthodes incrémentales ne conservent pas tous les exemples lors de l'apprentissage.

Un résumé des particularités de ces méthodes apparaît à la figure 6.1.

#	Method	Increm.	Positive Theory	Negative Theory	Positive Theory Learned	Negative Theory Learned	Concept Definition Learned	Knowl. Level
1	IOE	Y	imperfect	N	perfect consistent	N/A	N	N
2	Hirsh	Y	imperfect optional	imperfect optional	N	N	consistent	Y
3	OCCAM	Y/N	incorrect inconsistent	N	consistent correct	N/A	N	N
4	Bergadano	N	incomplete inconsistent	N	N	N/A	N	N
5	IOU	N	imperfect	N	N	N/A	has expl. and unexpl. parts	N
6	ACP	Y	incomplete inconsistent underspecified	incomplete inconsistent underspecified	consistent specified	consistent specified	consistent has expl. and unexpl. parts	Y

Fig. 6.1 *Résumé des méthodes mixtes.*

La plupart des méthodes courantes qui intègrent la méthode SBL et la méthode EBL, utilisent une de ces deux méthodes pour orienter ou fournir les informations nécessaires à l'autre. Normalement la méthode EBL sert de biais au système et le concept final est construit complètement par la méthode SBL. C'est le cas pour les quatre premières méthodes présentées ici. La cinquième méthode consiste plutôt à utiliser les méthodes pour apprendre différents aspects d'un concept, soit les aspects explicables en termes de fonctionnalité ou intentionnalité, ainsi que les aspects inexplicables par la théorie courante, qui sont plutôt conventionnels. Notre méthode ACP (la sixième méthode), qui combine les deux approches est ensuite comparée aux cinq méthodes mixtes.

1. IOE

Le but de la méthode IOE (`Induction Over Explanations`) [Dietterich et al. 1988], est d'améliorer la théorie du domaine lorsqu'elle est imparfaite (i.e. elle fournit plusieurs explications mutuellement exclusive pour un seul exemple) de façon à ce qu'elle produise une seule explication correcte pour un exemple positif.

Le système d'apprentissage requiert plusieurs exemples positifs et négatifs. Pour chacun d'eux la théorie du domaine est appliquée par la méthode déductive (EBL) pour construire toutes les explications possibles. Chaque explication est alors traitée comme un exemple de plus haut niveau. Les méthodes inductives sont appliquées à ces derniers exemples pour trouver une seule explication générale pour tous les exemples. Cette explication générale devient la nouvelle théorie parfaite du domaine.

La méthode est constituée de trois étapes:

1) **Expliquer:** Construire la structure explicative ET/OU en arbre, pour chaque exemple d'entraînement. Pour la méthode EBG, seulement une structure ET en arbre est construite pour représenter une explication, étant donné qu'à chaque fois qu'une disjonction apparaît dans la théorie du domaine, une seule branche apparaît dans l'explication. Dans la méthode IOE, pour tenir compte de toutes les explications pour un exemple, toutes les disjonctions de la théorie du domaine apparaissent dans l'explication. Une structure ET/OU en arbre est ainsi obtenue.

2) **Spécialiser:** Spécialiser la structure explicative en arbre le plus possible, pour qu'elle prévoit au moins une solution pour chaque exemple expliqué (i.e. garder une structure en arbre avec chaque branche des sous-arbres ET et au moins une branche des sous-arbres OU de la structure explicative).

3) **Extraire la nouvelle théorie:** Les noeuds de cette nouvelle structure explicative sont séparés en règles d'inférence individuelles pour constituer la théorie améliorée du domaine. Cette dernière corrige les parties trop générales de la théorie initiale et la remplace complètement.

IOE tente d'expliquer un exemple négatif de la même façon qu'un exemple positif: par la théorie du domaine. Si aucune explication n'est fournie, alors tout est correct, puisque c'est un exemple négatif. Dans le cas contraire, une structure ET/OU en arbre pour toutes les explications possibles est construite. Cette structure constitue un exemple négatif

pour le processus de spécialisation. La structure en arbre construite à l'étape 2 ne doit pas inclure de solutions possibles pour la structure négative en arbre.

Les exemples positifs généralisent donc l'explication recherchée (certaines caractéristiques doivent être retirées pour accepter un nouvel exemple positif) et les exemples négatifs la spécialisent (certaines caractéristiques sont ajoutées pour rejeter un nouvel exemple négatif).

2. Hirsh

Hirsh décrit dans [Hirsh 1989a] une approche qui combine l'apprentissage inductif et déductif en utilisant la méthode incrémentale de la fusion de l'espace [Hirsh 1989b] pour apprendre à partir de plusieurs exemples positifs et négatifs. La méthode EBL est utilisée comme méthode déductive pour générer toutes les explications possibles pour un exemple. Une généralisation de la méthode de l'espace de versions (`version space`) développée par Mitchell [Mitchell 1978] sert de méthode inductive. Cette méthode de base consiste à former un espace de versions avec les définitions consistantes du concept à partir de la généralisation obtenue par la méthode EBG pour chacun des exemples (au lieu de former l'espace de versions avec des définitions consistantes du concept à partir des valeurs des attributs pour chacun des exemples).

L'idée de base présentée par Hirsh est d'utiliser l'apprentissage inductif sur les résultats de l'apprentissage déductif, quand les connaissances du domaine sont disponibles (les connaissances du domaine sont requises pour appliquer la méthode EBL). Dans le cas contraire, l'apprentissage inductif utilise les exemples comme tels, pour former toutes les définitions du concept qui sont potentiellement pertinentes.

L'espace de versions s'applique à n'importe quel ensemble de définitions d'un concept, à condition que le langage de description soit représentable par les ensembles limites S et G. L'ensemble S contient les définitions les plus spécifiques du concept, tandis que l'ensemble G contient les définitions les plus générales. L'espace de versions contient les définitions du concept qui sont au moins plus générales que les définitions dans S et au moins plus spécifiques que les définitions dans G. L'espace de versions pour un exemple reflète la généralisation basée sur l'explication de l'exemple et représente l'ensemble des définitions consistantes du concept avec les exemples explicables de la même manière que l'exemple donné.

L'algorithme de fusion a pour effet de mettre à jour l'espace de versions avec plusieurs exemples qui partagent la même explication au lieu d'un seul exemple. De cette manière les caractéristiques impertinentes des exemples sont retirées et l'apprentissage converge vers une définition finale du concept en utilisant moins d'exemples.

De plus la méthode inductive permet un apprentissage au niveau des connaissances (`knowledge level`) [Dietterich 1986]. En effet, le langage de description utilisé par l'apprentissage empirique utilise des hiérarchies généralisées, incluant des connaissances générales. Ces connaissances ne font pas partie de la théorie du domaine et sont vraies en général. Le but de l'apprentissage empirique est de déterminer quelles sont les généralisations potentielles pertinentes pour les exemples et de les ajouter aux définitions consistantes du concept.

L'algorithme se présente comme suit:

1 a) • appliquer si possible la méthode EBG pour obtenir généralisation de l'exemple courant
• faire toutes les explications possibles
• si aucune explication n'est obtenue alors garder l'exemple tel quel

b) • former l'espace de versions de toutes les définitions du concept consistantes avec l'exemple (probablement généralisé dans 1a)
• s'il y a plusieurs explications, inclure les définitions consistantes du concept pour chacune des explications

2) faire l'intersection de ce nouvel espace de versions avec l'espace de versions des exemples précédents

3) • si aucune définition appartient aux deux ensembles S et G alors
• retourner à la première étape pour l'exemple suivant
• sinon la disjonction des définitions qui appartiennent à l'intersection de S et G forme la définition finale du concept

Encore une fois les exemples positifs sont appris pour généraliser la définition du concept, tandis que les exemple négatifs sont appris pour la spécialiser. La méthode est incrémentale et requiert la mémoire de tous les exemples seulement dans le pire des cas,

c'est-à-dire si aucune connaissance n'est donnée au système (dans ce cas, l'induction est faite à partir des exemples, puisqu'il n'y a aucune généralisation obtenue par EBG). A ce moment là, la méthode est équivalente à une méthode purement inductive.

3. OCCAM

Comprendre la cause d'un événement (ou exemple) permet d'expliquer, de prédire[11] et parfois même de contrôler l'événement. Pazzani dans [Pazzani 1987; Pazzani 1988] présente le système OCCAM basé sur une théorie d'apprentissage pour prédire et expliquer le résultat des événements. Deux sources d'information sont utilisées par OCCAM:

1) celle qui révèle les caractéristiques communes entre les exemples

2) celle qui explique les caractéristiques d'un exemple

La première source d'information est fournie par l'apprentissage inductif (SBL), tandis que la seconde est fournie par l'apprentissage déductif (EBL).

L'idée de base du système est d'expliquer et de généraliser l'exemple (ou l'événement) par la méthode EBL d'abord. Si ce n'est pas possible, alors la méthode SBL est appliquée. Les règles d'inférence (ou prédictions) apprises sont ensuite gardées et pourront être réutilisées pour les exemples suivants. Le fait d'ajouter ainsi des règles, peut amener les connaissances du système à devenir inconsistantes et/ou incorrectes (incorrectes dans le sens de [Pazzani 1988] i.e. une théorie du domaine avec des hypothèses fausses). Sans utiliser de méthode de préservation de vérité (`truth maintenance`), le système OCCAM a prévu à cet effet un mécanisme d'évaluation de ces connaissances. Brièvement, le système OCCAM procède comme suit:

- pour chaque règle d'inférence (ou prédiction) qui appartient au système:
- si elle prédit correctement un exemple, son niveau de confiance est augmenté (peut importe si elle provient de la méthode EBL ou SBL)

- sinon (c'est une mauvaise prédiction)

[11] Un événement est prédit, si une règle d'inférence du système satisfait cet événement sans l'avoir vu précédemment.

- si elle provient de la méthode SBL, alors son niveau de confiance est diminué (jusqu'à l'élimination possible de la prédiction)

- sinon (la prédiction provient de la méthode EBL)
  - vérifier toutes les règles d'inférence qui ont servi à l'inférer
  - si une règle n'apparaît plus dans les connaissances du système, alors retirer la prédiction

- appliquer la méthode EBL si possible
  (la définition du concept obtenue forme la prédiction que les exemples de la même classe[12] auront le même résultat)

- appliquer la méthode SBL (pour former la prédiction que les caractéristiques appartenant à tous les exemples précédents, apparaîtront aussi dans les exemples suivants)

Ainsi les connaissances, pour produire une explication, peuvent être acquises et révisées par le système OCCAM. OCCAM est un système à boucle-fermée[13] (`closed-loop`) [Michalski 1987]. En fait, la capacité de réviser l'explication obtenue par la méthode EBL (quand il est évident que les connaissances du système sont inconsistantes et/ou incorrectes) et l'habileté des méthodes inductives à acquérir de nouvelles règles pour compléter l'explication constituent les principaux avantages de la méthode. Par contre la méthode n'est pas totalement incrémentale, puisqu'elle procède seulement après avoir accumulé plusieurs exemples qui partagent les mêmes caractéristiques pour former une catégorie et créer une généralisation des exemples. De plus le fait d'augmenter ou de diminuer le niveau de confiance d'une prédiction peut orienter le système à apprendre davantage à partir de la fréquence des caractéristiques dans les exemples, plutôt que d'apprendre d'après les caractéristiques comme telles, peut importe la fréquence.

4. Bergadano

Bergadano et Giordana [Bergadano et al. 1988] présentent une méthode d'acquisition de concept en intégrant les méthodes traditionnelles de l'apprentissage

[12] Ceux qui s'expliquent de la même façon.

[13] Les deux méthodes d'apprentissage coopèrent en utilisant la même mémoire, pour garder (par la méthode SBL) et retirer (par la méthode EBL) les résultats.

déductif et inductif soient: EBL et SBL. Le but de cette méthode est d'apprendre une description discriminante et opérationnelle avec une théorie incomplète et/ou inconsistante du domaine à partir d'exemples positifs et négatifs.

A partir de la description non opérationnelle du concept initialement donnée, le processus d'apprentissage recherche de haut en bas dans un espace de formules logiques. Cette recherche est guidée par plusieurs sources d'information, incluant des informations statistiques, des heuristiques générales d'apprentissage et une théorie du domaine.

Le processus déductif utilise la théorie du domaine, qui peut être incomplète et/ou inconsistante par rapport aux exemples. Le processus inductif n'utilise pas les exemples, mais les résultats de la déduction. Ce processus permet de laisser tomber une partie de la théorie du domaine inconsistante par rapport aux exemples.

En fait l'algorithme de la méthode est simple:

1) spécialiser la description du concept par la déduction (en ajoutant des parties à la définition du concept pour ne pas couvrir d'exemples négatifs),

2) produire des statistiques[10] sur l'ensemble des exemples, selon la nouvelle description du concept

3) si certains exemples positifs ne sont pas couverts (d'après les statistiques) par la nouvelle description du concept, alors généraliser la description par l'induction (en retirant une partie de la description du concept qui empêche de couvrir plus d'exemples positifs),

4) répéter les trois étapes précédentes jusqu'à ce tous les exemples positifs soient couverts.

La description finale du concept est formée de la disjonction de toutes les descriptions partielles du concept qui couvrent un ensemble d'exemples positifs.

[10] Les statistiques comprennent les informations sur les parties concernées des exemples et le nombre d'exemples positifs et négatifs couverts

L'article [Bergadano and Giordana 1988] apporte une autre contribution à la méthode EBL et au critère d'opérationnalité. En fait, dans ce système l'explication est générée pour plusieurs exemples en même temps. Ceci permet de définir le critère d'opérationnalité sur une base statistique et ainsi, choisir une explication parmi plusieurs et guider le processus d'opérationnalisation.

Cette méthode permet entre autres:

1) d'appliquer à des problèmes du monde réel où la théorie du domaine peut être inconsistante et/ou incomplète,

2) d'obtenir une explication pour plusieurs exemples au lieu d'un seul; ce qui permet au système d'évaluer la consistance et la complétude des assertions générées pendant l'explication, par la méthode SBL,

3) d'exploiter les résultats même partiels, qui sont déductibles à partir de théorie imparfaite,

4) d'utiliser de plusieurs exemples au lieu d'un seul pour opérationnaliser le concept de façon statistique.

5. IOU

La méthode IOU (`Induction Over the Unexplained`) [Mooney et al. 1989] a été développée pour apprendre efficacement les aspects explicables et inexplicables d'un concept. Les caractéristiques, qui sont explicables pour un seul exemple, sont apprises par la méthode standard EBL. Ces caractéristiques sont retirées de l'exemple initial pour ne garder qu'une description réduite de l'exemple. Cette description et toutes les suivantes sont passées à un système d'apprentissage inductif pour y trouver des caractéristiques communes et ainsi ajouter la partie conventionnelle à la définition du concept. La définition finale du concept correspond à la conjonction de la "description expliquée" (disjonction de toutes les explications obtenues pour un exemple) apprise par EBL et de la description "non expliquée" apprise par SBL.

Le système utilise plusieurs exemples positifs et négatifs et assume que la théorie est correcte pour généraliser le concept étudié. La méthode n'est pas incrémentale. L'algorithme se présente comme suit:

1) construire et généraliser une explication pour chacun des exemples positifs

2) combiner disjonctivement les définitions résultantes, pour former la composante explicable (C_e) du concept

3) éliminer tous les exemples négatifs qui ne satisfont pas la composante explicable

4) retirer, de chacun des exemples positifs et des exemples négatifs restants, toutes les caractéristiques mentionnées dans la composante explicable

5) passer l'ensemble des exemples réduits à une méthode d'apprentissage inductif pour former la composante inexplicable du concept (C_n)

6) obtenir la définition finale du concept correspondant à la conjonction des composantes explicable et inexplicable des exemples ($C_e \wedge C_n$)

Le principal avantage de la méthode IOU, c'est qu'elle peut utiliser n'importe quel système SBL (pourvu qu'il supporte la description du langage), puisque l'algorithme du système IOU est indépendant des détails d'une méthode SBL. Son principal désavantage c'est qu'elle n'est pas incrémentale.

6. ACP

Comme le résume le tableau de la figure 6.1, notre méthode (ACP) présente de nouvelles caractéristiques pour l'apprentissage à partir d'une méthode mixte.

Cette méthode utilise plusieurs exemples positifs et négatifs, comme toutes les autres méthodes mixtes présentées ici et qu'elle est incrémentale comme certaines d'entre elles.

La méthode ACP utilise entre autres une théorie positive et une théorie négative du domaine qui sont sous-spécifiées (i.e. elles comprennent des opérateurs inconnus). Dans d'autres travaux sur les méthodes mixtes [Bergadano and Giordana 1988; Dietterich and Flann 1988; Hirsh 1989a; Mooney and Ourston 1989; Pazzani 1987; Pazzani 1988] les opérateurs logiques des théories positive et/ou négative fournies au système correspondent à ET ou OU. Les opérateurs logiques de la méthode ACP des théories (positive et négative) sous-spécifiées du domaine sont inconnus et sont générés lors de l'apprentissage.

Ces opérateurs inconnus représentent le fait qu'en réalité on peut connaître qu'une définition non-opérationnelle d'un processus quelconque.

De plus, la méthode ACP ne requiert pas plus de sources d'information que les autres méthodes, même que c'est la seule méthode où les résultats touchent les théorie et la définition du concept. En effet cette méthode fournit les sources d'information suivantes:

1 a) une théorie positive et une théorie négative spécifiées et consistantes du domaine (opérateurs ET/OU)

b) des contraintes procédurales pour révéler les relations entre les caractéristiques des exemples

2) une définition consistante du concept

L'apprentissage inductif pour les méthodes ACP et IOE utilise différemment les structure explicatives en arbre. A partir d'une théorie parfaite (dans le sens de [Dietterich and Flann 1988], i.e. une seule explication possible pour un exemple), les structures explicatives en arbre pour la méthode ACP permettent de tenir compte de toutes les combinaisons de caractéristiques possibles pour un exemple. L'explication des combinaisons possibles pour les exemple positifs est représentée par la théorie positive spécifiée du domaine et celle pour les exemples négatifs est représentée par la théorie négative spécifiée. Les opérateurs logiques de ces théories sont connus (ET/OU) et peuvent être accompagnés de contraintes procédurales.

Un peu comme la méthode de Hirsh, la méthode ACP apprend au niveau des connaissances (par exemple: avoir un wagon-lit et/ou une toilette peut-être remplacé par avoir les commodités pour voyager). Mais la méthode ACP va plus loin encore en apprenant des connaissances qui sont initialement inconnues du système et qui appartiennent à aucun des exemples. Ces connaissances correspondent aux contraintes procédurales et servent à rendre explicites les relations implicitement représentées par les exemples.

De plus la définition du concept pour la méthode ACP est composée de deux parties explicables et de deux parties inexplicables. Les connaissances qui appartiennent aux

parties inexplicables de la définition du concept n'appartiennent pas aux exemples, contrairement à la méthode IOU. Ce sont les contraintes procédurales.

7. Les résultats et la conclusion

La méthode crée une représentation du concept appris, dont chacune des parties est augmentée avec des contraintes procédurales. Ces contraintes représentent les relations entre les actions qui sont implicites dans les exemples d'entraînement, mais qui seraient absentes si le concept avait été appris par une méthode inductive classique (c.f. INDUCE [Hoff et al. 1982]).

Pour la méthode de l'apprentissage incrémental, la représentation mentale dont on a parlé au chapitre 3, correspond à la définition du concept. Dans l'exemple de ce même chapitre la définition du concept est trop générale et accepte aussi des situations pour *prendre le train*. Le système doit rendre la définition positive du concept plus consistante en rejetant les exemples négatifs couverts. Cela est rendu possible en ajoutant la négation d'un ou plusieurs descripteur(s) pertinent(s) pour ces exemples, à la définition positive du concept. Cette situation correspond particulièrement aux exemples de *prendre le train*, dont la séquence d'actions est déjà apprise par une séquence d'actions de *prendre le métro*.

Les connaissances générales positives représentent les descripteurs pertinents possibles pour qu'un exemple positif soit membre du concept. D'un autre côté, les connaissances générales négatives représentent les descripteurs pertinents possibles pour qu'un exemple négatif ne fasse pas partie du concept. Une fois appris, ces descripteurs permettent de couvrir les exemples positifs et de rejeter les exemples négatifs, lorsque leur séquence d'actions sont identiques. Par exemple: c'est possible de connaître comment *prendre le train* parce que la séquence d'actions est identique à celle déjà vue de *prendre le métro*. Ainsi un descripteur tel que: ne se déplace pas dans un tunnel (`is not in a tunnel`) permet de distinguer entre ces deux exemples. La négation de ce descripteur est ajoutée à la définition positive du concept par l'apprentissage incrémental. Les exemples positifs suivants devront satisfaire la séquence d'actions accompagnée du descripteur appris. Tous les exemples négatifs suivants, avec une séquence d'action déjà apprise par la définition positive du concept et satisfaisant le descripteur appris seront dorénavant exclus.

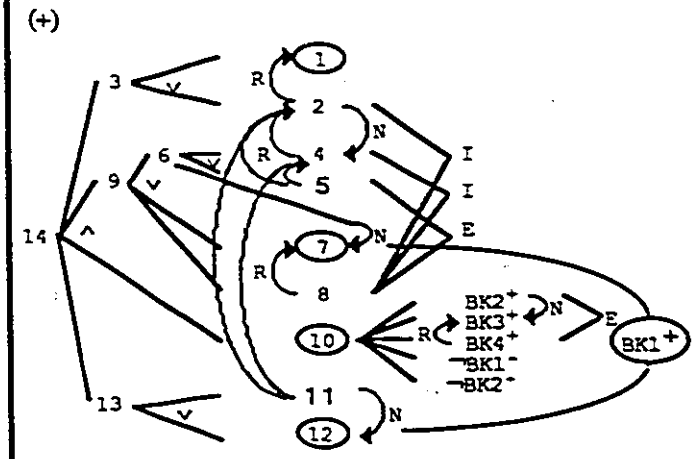
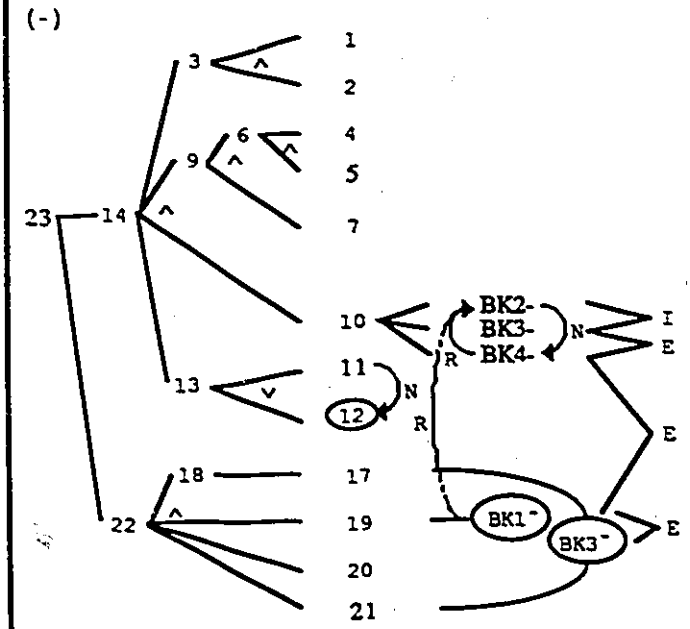
FSDT	CD
(+) 	(1 ∨ 2) ∧ ((4 ∨ 5) ∨ 7 ∨ 8) ∧ 10 ∧ (11 ∨ 12) ∧ C1 ∧ C7 ∧ C10 ∧ C12 ∧ 2R1 ∧ 2N4 ∧ 4R2 ∧ 5R2 ∧ 5R4 ∧ 4N7 ∧ 5N7 ∧ 8R7 ∧ 11R2 ∧ 11R3 ∧ 11N12 ∧ 2I8 ∧ 4I8 ∧ 5E8 ∧ (BK1+ ∨ BK2+ ∨ BK3+ ∨ BK4+) ∧ CBK1+ ∧ BK2+ N BK3+ ∧ BK2+ E BK4+ ∧ BK4+ R BK3+ ¬(BK1- ∨ BK2-)
(-) 	1 ∧ 2 ∧ 4 ∧ 5 ∧ 7 ∧ 10 ∧ (11 ∨ 12) ∧ C11 ∧ 11N12 ∧ (BK2- ∨ BK3- ∨ BK4-) ∧ BK1- R BK2- ∧ BK2- N BK4- ∧ BK2- I BK3- ∧ BK3- E BK4- ∧ BK4- R BK2- ∧ ((17 ∧ 19 ∧ 20 ∧ 21) ∧ (BK1- ∧ BK3-))

Fig. 7.1 *Les théories complètement spécifiées positive et négative et les définitions du concept générées à partir de celles-ci pour tous les exemples d'entraînement de l'appendice A.*

Les résultats finaux pour un apprentissage incrémental à partir de tous les exemples positifs et négatifs présentés en annexe, sont illustrés à la figure 7.1. L'algorithme a appris inductivement les opérateurs inconnus. Certains expriment une conjonction et d'autres expriment une disjonction accompagnés de contraintes procédurales. Par exemple:

1) avec les exemples appris jusqu'à maintenant, une situation pour prendre le métro est au moins exprimée par les actions 1, 7, 10 et 12 (actions obligatoires);

2) l'occurrence de l'actions 11 nécessite l'occurrence de l'action 12 (11N12), qui s'interprète comme suit: remettre l'accès dans la seconde machine (put Acc into Acc_Mach2) oblige la personne à sortir (exit through Exit_Mach at Exit_pt) et la même contrainte permet aussi une situation où la personne sort (exit through Exit_Mach at Exit_pt) sans avoir remis l'accès (put Acc into Acc_Mach2) avant.

Les théories complètement spécifiées du domaine montrent comment les connaissances générales sont incorporées dans les théories pendant le processus de l'apprentissage incrémental de contraintes procédurales. Quelques observations peuvent être faites dans le cas du métro, par exemple:

1) la distance parcourue par le métro est moindre que 50 km (descripteur obligatoire BK1+ qui lie la station d'entrée et la station de sortie du métro);

2) un métro se déplace dans un tunnel (¬BK2-);

3) un métro n'a pas de wagon-lit (BK2+).

De façon analogue la théorie négative du domaine complètement spécifiée couvre les exemples négatifs.

La définition positive du concept est générée à partir de la théorie positive du domaine complètement spécifiée. Elle représente les combinaisons d'actions et de descripteurs appris pour les exemples positifs. Ainsi cette définition couvre tous les exemples positifs et exclut certains exemples négatifs. La définition positive du concept satisfait le critère d'opérationnalité, puisqu'elle est exprimée par une conjonction/disjonction d'actions et de descripteurs opérationnels. Les actions peuvent être liées par des contraintes procédurales. Les opérateurs et les contraintes procédurales de la définition du concept sont obtenus en propageant ceux-ci dans la structure en arbre de la théorie positive du domaine complètement spécifiée au niveau des feuilles (i.e. actions opérationnelles).

La négation de la définition négative du concept est intégrée dans la définition du concept finale pour exclure d'autres exemples négatifs, qui sont explicables, mais acceptés par la définition positive du concept. La définition négative du concept est générée à partir de la théorie négative du domaine complètement spécifiée, de la même façon que la définition positive du concept.

Notre approche est une extension essentielle de la méthode EBL: elle apprend à partir de plusieurs exemples, elle se sert aussi bien d'exemples négatifs que d'exemples positifs, et elle intègre l'apprentissage inductif avec la méthode EBL. Le coût de l'apprentissage est comparable aux autres méthodes inductives qui produisent les descriptions disjonctives des concepts, puisque pour un seul exemple, notre méthode est limitée par $O(n^2)$ et pour m exemples d'entraînement, c'est $O((m - 1) n^2)$. Il nous faut cependant remarquer que notre approche se sert *d'explications* des exemples, qui sont plus riches en connaissances que les exemples mêmes. Par conséquent, il faut s'attendre que l'apprentissage à partir des explications requiert moins de données que l'apprentissage à partir des exemples.

Nous avons décrit une méthode d'apprentissage incrémental de contraintes procédurales qui permet l'acquisition de contraintes procédurales. Ces contraintes permettent d'apprendre différentes relations, implicites dans les exemples d'entraînement. La méthode proposée est décrite en trois étapes soit:

1) apprentissage à partir des exemples positifs;

2) l'ajout d'une théorie négative, pour expliquer et apprendre à partir d'exemples négatifs;

3) l'ajout de connaissances générales pour raffiner l'apprentissage, lorsqu'un exemple négatif est explicable par la théorie positive.

L'apprentissage incrémental de contraintes procédurales à partir d'exemples positifs a été implanté en Prolog.

Glossaire

connaissances générales positives et négatives: ce sont les informations complémentaires fournies au système représentées indépendamment de la théorie du domaine, sous forme utilisable par le système. Elles permettent d'identifier les descripteurs pertinents pour un exemple. Ce sont des données qui ont déjà été fournies au système, mais n'ont pas été utilisées pour obtenir l'explication de l'exemple d'entraînement. Les connaissances générales positives, représentent les descripteurs pertinents possibles pour un exemple positif. A l'opposé les connaissances générales négatives représentent les descripteurs pertinents possibles pour qu'un exemple négatif ne fassent pas partie du concept. Une fois appris, de tels descripteurs permettent de distinguer les exemples positifs et les exemples négatifs ayant une séquence identique d'actions.

contraintes procédurales: ce sont des relations qui existent entre deux actions ou entre deux descripteurs. Ces relations sont implicites dans la représentation des exemples et rendues explicites par la méthode. Une contrainte procédurale accompagne un opérateur OU (v) et restreint la présence ou l'absence ainsi que l'ordre d'une action en particulier. Les contraintes procédurales apparaissent dans la théorie complètement spécifiée du domaine et dans la définition du concept, pour éviter qu'elles acceptent des séquences d'actions et des combinaisons de descripteurs, impossibles pour un exemple.

définition finale du concept: c'est une description générale du concept qui couvre tous les exemples positifs et exclut tous les exemples négatifs appris. Cette définition est une conjonction de la définition positive du concept et la négation de la définition négative du concept

définition négative du concept: description générale et opérationnelle du concept qui couvre les exemples négatifs qui ne sont pas rejetés par la définition positive du concept. Elle est composée d'une séquence d'actions et d'une liste de descripteurs pertinents, reliés par des contraintes procédurales.

définition positive du concept: description générale et opérationnelle du concept qui couvre tous les exemples positifs et rejette certains des exemples négatifs qui

appartiennent aux deux théories. Elle est composée d'une séquence d'actions et d'une liste de descripteurs pertinents, reliés par des contraintes procédurales.

opérateur inconnu: c'est un connecteur logique de la théorie sous-spécifiée du domaine et représenté par "?". Lors de l'apprentissage, ces opérateurs sont remplacés soit par ET ou, par OU et une des contraintes procédurales.

théorie complètement spécifiée du domaine (négative et positive): chacune de ces théories est représentée par une structure en arbre ET ($\wedge$), OU ($\vee$). La structure comprend toutes les actions utilisées par les exemples d'entraînement. De plus, suite aux liaisons apprises sur l'organisation de la séquence d'actions des exemples, tous les opérateurs de la théorie complètement spécifiée du domaine sont générés et certaines de ces actions peuvent être liées par des contraintes procédurales. La théorie du domaine une fois spécifiée n'explique pas les exemples que la théorie sous-spécifiée du domaine ne peut expliquer. Cette théorie peut expliquer au plus les mêmes exemples que la théorie sous-spécifiée du domaine. La théorie est spécifiée dans le sens que les opérateurs inconnus ont été appris et restreints par des contraintes procédurales. Le but des contraintes procédurales est d'éliminer des combinaisons d'actions et de descripteurs pertinents impossibles, pour les exemples négatifs dans le cas de la théorie négative et positifs dans le cas de la théorie positive.

théorie sous-spécifiée du domaine (positive et négative): c'est un ensemble de règles d'inférence et de faits utilisés pour expliquer comment chacun des exemples positifs fait partie du concept et comment chacun des exemples négatifs ne fait pas partie du concept. Elle constitue les théories initiales du domaine et contiennent des opérateurs inconnus.

Appendice A: Les exemples d'entraînement

Type A

1.BART
has 110 passengers
buy ticket from machine
receive ticket
put ticket into machine2
enter through gate at pt-a
distance between pt-a and pt-b is 6
receive ticket
driver Jim
ride subway
is in tunnel
put ticket into machine3
exit through gate at pt-b

2.MONTREAL
has no washroom
buy ticket from cashier
is in tunnel
receive ticket
put ticket into turnstile
enter through turnstile at pt-a
ride subway
color blue
has no mailvan
exit through turnstile at pt-b
distance between pt-a and pt-b is 8
has 233 passengers

3.MEXICO
driver suzan
buy ticket from cashier
receive ticket
put ticket into turnstile
has 156 passengers
enter through turnstile at pt-a
ride subway
exit through turnstile at pt-b
distance between pt-a and pt-b is 11

4.NYC
buy token from teller
receive token
distance between pt-a and pt-b is 7
put token into turnstile
has no mailvan
enter through turnstile at pt-a
color red
ride subway
driver joe
exit through turnstile at pt-b
is in tunnel

5.PARIS
color green
buy ticket from cashier
receive ticket
has 70 passengers
put ticket into turnstile
receive ticket
has no washroom
enter through turnstile at pt-a
ride subway
has no sleeping car
exit through turnstile at pt-b
distance between pt-a and pt-b is 3

6.PARIS (RER)
is in tunnel
has no mailvan
has no sleeping car
buy ticket from cashier
distance between pt-a and pt-b is 4
has 432 passengers
receive ticket
put ticket into turnstile
receive ticket
enter through turnstile at pt-a
has no washroom
driver jimmy
ride subway
put ticket into turnstile
exit through turnstile at pt-b
has 3 employees

7.TORONTO
buy ticket from teller
has 32 passengers
has no goodsvan
enter through turnstile at pt-a
receive ticket
is in tunnel
ride subway
has no washroom
exit through turnstile at pt-b
distance between pt-a and pt-b is 14

Type B

1.BOSTON
- **have token**
- **put token into turnstile**
- has 328 passengers
- has no goodsvan
- **enter through turnstile at pt-a**
- **ride subway**
- has 3 employees
- **exit through gate at pt-b**

2.PRAGUE
- is in tunnel
- **put money into change machine**
- **receive coin**
- **put coin into turnstile**
- **enter through turnstile at pt-a**
- has 3 employees
- **ride subway**
- **exit through spec-gate at pt-b**

3.MOSCOW
- **have coin**
- **put coin into turnstile**
- is in tunnel
- **enter through turnstile at pt-a**
- has 3 employees
- **ride subway**
- is in tunnel
- **exit through turnstile at pt-b**

Type C

1.TRAIN1
color yellow
buy ticket from cashier
receive ticket
has 270 passengers
put ticket into turnstile
receive ticket
enter through turnstile at pt-a
ride subway
is not in tunnel
exit through turnstile at pt-b

2.TRAIN2
buy ticket from cashier
receive ticket
put ticket into turnstile
receive ticket
enter through turnstile at pt-a
driver john
ride subway
exit through turnstile at pt-b
distance between pt-a and pt-b is 88

3.TRAIN3
buy ticket from cashier
is not in tunnel
receive ticket
put ticket into turnstile
enter through turnstile at pt-a
ride subway
color blue
exit through turnstile at pt-b
distance between pt-a and pt-b is 125

4.TRAIN4
buy ticket from cashier
driver peter
receive ticket
put ticket into turnstile
enter through turnstile at pt-a
ride subway
color blue
exit through turnstile at pt-b

5.TRAIN5
driver suzan
buy ticket from cashier
receive ticket
put ticket into turnstile
has washroom
has 156 passengers
enter through turnstile at pt-a
ride subway
has sleeping car
is not in tunnel
exit through turnstile at pt-b

Type D

1.TAXI
- call taxi
- wait
- is not in tunnel
- driver robert
- take taxi at pt-a
- ride
- pay driver
- color blue
- exit atxi at pt-b
- distance between pt-a and pt-b is 31

Type E

1.GROSSERY
- enter grossery
- take food
- pay food
- exit

2.GAS
- enter station
- fill it up
- pay for gas
- exit

3.CLOTHES
- enter store
- take clothes
- pay clothes
- exit

Appendice B: Le programme Prolog et les résultats

```
%       learn_proc_const                                        6 avril 1990
%-------------------------------------------------------------------------
%
%
%       Apprentissage Incrementale de Contraintes Procedurales
%
%                           These de Maitrise
%
%
%
%                             Johanne Morin
%
%
%
%
%                           Universite d'Ottawa
%                               Avril 1990
%
%
%
%-------------------------------------------------------------------------

%       Quintus Prolog Release 2.4.2 (Sun-3, SunOS 3.4)

%       AnalRel:        analysis of relations
%       CD:             concept definition
%       CT:             theorie courante
%       CW:             closed-world assumption
%       DiscDSubt:      disriminating subtrees
%       DT:             domain theory
%       ebg:            Explanation-Based Generalization
%       ProcConst:      procedural constraints
%       TE:             training exemple

%       Leaves: actions operationnelle
%       Nodes:  toutes les actions possibles dans la DT (operationnelles ou non)

:- compile(library(not)).
:- consult(library(basics)),
        consult(subway),                                % fichier d'entree
        consult(common),
        consult(ebg),
        consult(already_learned),
        consult(discrim_subt),
        consult(analysis_of_relations),
        consult(concept_definition),
        consult(write_procedure).

        get_TE(TE_Id, TE):-
                retract(order(TE_Id)),                  % identifie le prochain T
                train_ex(TE_Id, TE).                    % prend la sequence d'act.

        get_goal_concept(GenGC):-
                goal_concept(GC),                       % le concept a apprendre
                GenGC =.. [GC, _],                      % forme une regle general
                !.

        explain_TE(TE_Id, TE, CW_Just, [DT_Leaves, DT_Nodes]):-
                get_goal_concept(GenGC),                % le concept a apprendre
                ebg(_GC, GenGC, TE, _,          % applique EBG pour le TE;
                _CD, and(_Just), and(Reg),              % CD, Just & Reg pourraie
                and(CW_Just), CW_CD,                    % ici, ils ne sont pas re
                DT_Leaves, unkn(DT_Nodes)),!,
                write_procedure(TE_Id, Reg),            % affichage de certain re
                assert(anal_of_rel(TE_Id, CW_CD)),      % CD based on close world
                !.
```

```
already_learned([CD, LConst], TE_Id, TE, New, New, [CD, LConst]):-      %
        satisfy_CD(CD, TE),                          % si satisfait la CD &
        satisfy_proc_const(LConst, TE),              % si satisfait les ProcCor
        write(TE_Id), write(' is ALREADY LEARNED !'),  nl, nl,
        !.

increm_learning(TE_Id, TE, [DT_Leaves, DT_Nodes], OldLConst, NewLConst_DSu
        explain_TE(TE_Id, TE, CW_Just, [DT_Leaves, DT_Nodes]),          %
        assert(cw_just(TE_Id, CW_Just)),                                %
        discrim_subt(DT_Nodes, OldLConst, NewLConst_DSubt),
        analysis_of_relations(DT_Leaves, LConst_ARel),
        concept_definition(DT_Nodes, NewLConst_DSubt, LConst_ARel, Concept
        !.

initialization(DT_Names):-
        nl, write('List of training examples to learn from: '), nl, nl,
        bagof(Id, order(Id), L),
        write_list(L),
        get_TE(TE_Id, TE),
        explain_TE(TE_Id, TE, CW_Just, DT_Names),       % le 1er TE doit e
        write(TE_Id),
        write('   First justification becomes the current theory ...'), nl
        assert(current_theo(CW_Just)),                  % initialisation d
        !.

%       other_TE(DT_Names, OldProcConst, NewProcConst, OldCD):
%               - pour chaque TE
%               - verifier s'il est deja appris
%               - sinon l'apprendre incrementalement
%
%       DT_Leaves:      liste des feuilles possibles dans la DT
%       DT_Nodes:       liste des noeuds possibles dans la DT
%       OldProcConst:   anciennes liste de ProcConst provenant seulement de DiscSu
%       NewProcConst:   nouvelle liste de ProcConst
%       OldCD:          ancienne CD: [CD, ProcConst]

        other_TE(DT_Names, Old, New1, OldCD):-
                get_TE(TE_Id, TE),
                (already_learned(OldCD, TE_Id, TE, Old, New, NewCD);
                increm_learning(TE_Id, TE, DT_Names, Old, New, NewCD)),
                other_TE(DT_Names, New, New1, NewCD).
        other_TE(_, Old, Old, _):-
                write('    No more training example !!! ..'), nl, nl, nl,
                !.

learn_proc_const:-
        initialization(DT_Names),                       % initialisation de la CT
        other_TE(DT_Names, [], _Proc_Const, []).        % apprentissage incrementa
learn_proc_const:- !.

%       - execute automatiquement le programme en tenant compte du temps d'executi
%       :- statistics(runtime, _),
%               learn_proc_const,
%               statistics(runtime, [_,T]),
%               format('Incremental Learning of Procedural Constraints took ~3d se
```

```
%         common                                              8 septembre 1989

%         procedures communes a plus d'un fichiers

n_th_elem(Stop, Stop, [H|_T], H):- !.
n_th_elem(Ctr, Stop, [_H|T], N_Elem):-  % trouver le nieme element de la liste
        Ctr1 is Ctr + 1,
        n_th_elem(Ctr1, Stop, T, N_Elem).

fact(Pred, 1, [Pred, _]):- !.                   % fait a 2 arg.
fact(Pred, 2, [Pred, _, _]):- !.                % fait a un arg.

retract_fact(Pred, N):-                         % retirer tous les faits qui correspondent
        fact(Pred, N, LFact),
        Fact =.. LFact,
        retract(Fact),
        fail.
retract_fact(_, _):- !.

nonop(A):- predicate_property(A, (dynamic)).            % predicat non operationn

compl_struct(_A, [], and([])):- !.                      % rien a ajouter
compl_struct(A, Just, and([t(A, and(Just))])):- !.      % ajouter un sous-arbre

compl_struct2(_A, [], unkn([])):- !.                    % rien a ajouter
compl_struct2(A, DT, unkn([t(A, unkn(DT))])):- !.       % ajouter un sous-arbre
```

```
%       already learned                                  6 avril 1990

equal_heads(Pred1, Pred2):-                     % meme predicats?, pour eviter qu
        Pred1 =.. [H|_P1],
        Pred2 =.. [H|_P2],
        !.

force_list([], 1):- !.                          % retourne 0 si au moins un elem.
force_list([0|_], 0):- !.                       % retourne 1 dans le cas contrair
force_list([1|T], V):-
        force_list(T, V).

mask_list([], 0):- !.                           % retourne 1 si au moins un elem.
mask_list([1|_], 1):- !.                        % retourne 0 dans le cas contrair
mask_list([0|T], V):-
        mask_list(T, V).

next_elem([], TE, TE, LSatisfy, LSatisfy):- !.          % plus de cond. a satisfa
next_elem(Pred1, [Pred2|T], T, LSatisfy, NLSatisfy):-   % retourne 1 au niveau su
        equal_heads(Pred1, Pred2),
        append(LSatisfy, [1], NLSatisfy), !.
next_elem(_, TE, TE, LSatisfy, NLSatisfy):-             % retourne 0 au niveau su
        append(LSatisfy, [0], NLSatisfy), !.

%       - passer au travers de la "struct." de la CD, pour verifier si le TE sati
satisfy_cond([], TE, TE, L, L):- !.
satisfy_cond([H|T], TE, Rest1_TE, L, NL):-              % satisfaire les branches
        satisfy_cond(H, TE, Rest_TE, L, L1),
        satisfy_cond(T, Rest_TE, Rest1_TE, L1, NL), !.
satisfy_cond(and(LArg), TE, Rest_TE, L, NL):-           % verifie un niveau ET
        satisfy_cond(LArg, TE, Rest_TE, [], L1), !,
        force_list(L1, ForceVal),
        append(L, [ForceVal], NL), !.
satisfy_cond(or(LArg), TE, Rest_TE, L, NL):-            % verifie un niveau OU
        satisfy_cond(LArg, TE, Rest_TE, [], L1), !,
        mask_list(L1, MaskVal),
        append(L, [MaskVal], NL), !.
satisfy_cond(Elem, TE, Rest_TE, L, NL):-                % feuille a satisfaire
        next_elem(Elem, TE, Rest_TE, L, NL), !.

satisfy_CD([and(LArg)], TE):-                           % arbre ET a satisfaire
        satisfy_cond(LArg, TE, [], [], L),
        force_list(L, 1), !.
satisfy_CD([or(LArg)], TE):-                            % arbre OU a satisfaire
        satisfy_cond(LArg, TE, [], [], L),
        mask_list(L, 1), !.

must_be_there(Pred, [HTE|_]):-                  % le predicat apparait dans TE
        equal_heads(Pred, HTE), !.
must_be_there(Pred, [_|TTE]):-
        must_be_there(Pred, TTE).

both(Pred1, Pred2, TE):-                                % les deux predicats doiv
        must_be_there(Pred1, TE),
        must_be_there(Pred2, TE), !.

get_pred(Param1, Param2, Pred1, Pred2):-                % retourne seulement les
        Param1 =.. [Pred1|_P1],
        Param2 =.. [Pred2|_P2], !.

split_proc_const(compul(Param), TE):-                   % obligatoire
        Param =.. [Pred|_], !,
        must_be_there(Pred, TE), !.
split_proc_const(eor(Param1, Param2), TE):-             % ou exclusif
        get_pred(Param1, Param2, Pred1, Pred2),
        not both(Pred1, Pred2, TE), !.
```

```
split_proc_const(ior(Param1, Param2), TE):-                    % ou inclusif
        get_pred(Param1, Param2, Pred1, Pred2),
        (must_be_there(Pred1, TE);
        must_be_there(Pred2, TE)), !.
split_proc_const(onl_o(Param1, Param2), TE):-                  % ou exclusif & ou inclus
        get_pred(Param1, Param2, Pred1, Pred2),
        not both(Pred1, Pred2, TE),
        (must_be_there(Pred1, TE);
        must_be_there(Pred2, TE)), !.
split_proc_const(req(Param1, Param2), TE):-                    % requiert
        get_pred(Param1, Param2, Pred1, Pred2),
        (both(Pred1, Pred2, TE);
        not must_be_there(Pred1, TE)), !.
split_proc_const(nec(Param1, Param2), TE):-                    % necessite
        get_pred(Param1, Param2, Pred1, Pred2),
        (both(Pred1, Pred2, TE);
        not must_be_there(Pred1, TE)), !.

pass_through_proc_const([], _TE):- !.                          % no more ProcConst
pass_through_proc_const([H|T], TE):-                           % pour chaque ProcConst
        split_proc_const(H, TE),                               % verifier si les actions
        pass_through_proc_const(T, TE).

satisfy_proc_const(LConst, TE):-                               % satisfaire toutes les P
        pass_through_proc_const(LConst, TE), !.
```

```
%       ebg     (Explanation Based Generalization)              20 septembre 1989

%       cw_just(L_Nodes, Val): compose la structure explicative basee sur le CW
%
%       L_Nodes:        liste des noeuds basee sur le CW
%       Val:            valeur que doit prendre le niveau superieur dans la struct
%

        cw_justif([], 0):- !.                                   % feuille absente
        cw_justif([t(1, nil)|_T], 1):- !.                       % feuille presente
        cw_justif([t(1, and(_SubL))|_T], 1):- !.                % sous-arbre present
        cw_justif([t(0, nil)|T], RetVal):-                      % sous-arbre absent
                cw_justif(T, RetVal).
        cw_justif([t(0, and(_SubL))|T], RetVal):-               % sous-arbre absent
                cw_justif(T, RetVal).

%       compl_cw(SubtTree, Tree): passer au travers de la structure explicative ba
        compl_cw(X, and([t(Y, and(X))])):-
                cw_justif(X, Y),
                !.

%-------------------------------------------------------------------------------
%       EBG modifie
%       ------------
%
%       ebg(Rule, GenRule, Input_TE, Output_TE, CD, and(Just), and(Reg), and(CW_Ju
%
%       Rule:           regle inferee
%       GenRule:        regle generale
%       Input_TE:       liste representant la sequence d'actions de l'exemple (orc
%       Output_TE:      liste des actions qui restent a verifier
%       CD:             definition du concept
%       Just:           structure explicative de l'exemple
%       Reg:            structure generalisee en arbre de l'exemple
%       CW_Just:        structure explicative de l'exemple basee sur les assomptic
%       CW_CD:          definition du concept basee sur les assomptions du monde f
%       DT_Leaves:      liste des regles operationnelles (generales: avec variable
%       DT_Nodes:       structure en arbre de toutes les regles d'inferences de la
%
%       Toutes les structures en arbre sont definies selon les structures d'entree
%       pour le programme d'affichage de structures en arbre du sous-repertoire
%       "ML/draw_tree" (programme de John Kowalski).
%
%       NB: Seulement les clauses de Horn sont acceptees

ebg((A, B), (GenA, GenB), TEI, TEOB, CD, and(Just), and(Reg), and(CW_Just),
        CW_CD, DT_Leaves, unkn(DT_Nodes)):-     % - plus d'une regle d'inference a
        !,                                      % les resultats ebg pour chacune c
        ebg(A, GenA, TEI, TEOA, CDA, and(JustA), and(RegA), and(CW_JustA), CW_CDA,
        ebg(B, GenB, TEOA, TEOB, CDB, and(JustB), and(RegB), and(CW_JustB),CW_CDB,
        append(CDA, CDB, CD),
        append(CW_CDA, CW_CDB, CW_CD),
        append(JustA, JustB, Just),
        append(RegA, RegB, Reg),
        append(CW_JustA, CW_JustB, CW_Just),
        append(DT_LeavesA, DT_LeavesB, DT_Leaves),
        append(DT_NodesA, DT_NodesB, DT_Nodes).
ebg(A, GenA, TEI, TEO, CD, RetJust, RetReg, RetCW_Just, CW_CD, DT_Leaves, RetDT):-
        nonop(A),                                               % A:-B is active c
        clause(GenA, GenB),                                     % true is A:-B is
        copy_term((GenA:-GenB),(A:-B)),                         % copying a term w
        ebg(B, GenB, TEI, TEO, CD, and(Just), and(Reg), and(CW_Just), CW_CD, DT_Le
        compl_struct(A, Just, RetJust),                         % dans le fichier
        compl_struct(GenA, Reg, RetReg),
        compl_cw(CW_Just, RetCW_Just),
        compl_struct2(GenA, DT, RetDT).                         % dans le fichier
```

```
%       - feuille de la DT correspond a la premiere action de TE
ebg(A, GenA, [A|Tail], Tail, [GenA], and([t(A, nil)]), and([t(GenA, nil)]), and([t
%       - feuille de la DT ne correspond pas a la premiere action de TE
%       - on est qaund meme interesse a garder sont nom dans la liste des feuille:
ebg(_A, GenA, TE, TE, [], and([]), and([]), and([t(0, nil)]), [0], [GenA], unkn([t
```

```
%       discrim_subt                                    12 mars 1990

:- dynamic
        current_theo/1,
        constraint/1,
        cw_just/1,
        tmp_list/1,
        col1/1,
        col2/1,
        tmp/2.

        mask(0, 0, 0):- !.                          % retourne 0 si deux 0
        mask(_, _, 1):- !.                          % retourne 1 si au moins un 1

        empty_subt([]):- !.                         % fin d'un niveau
        empty_subt([0|T]):- empty_subt(T).          % sous-arbre absent

        get_row1(N, Elem):- !,                      % chercher le Nieme elem. de la 1
                tmp_list(L),
                n_th_elem(1, N, L, Elem).

        revers_lists(Stp, Stp):-
                retract_fact(tmp_list, 1), !.
        revers_lists(Ctr, Stp):-                    % renverser le tableau tmp (range
                bagof(L_Elem, get_row1(Ctr, L_Elem), L),
                assert(col1(L)),
                assert(col2(L)),
                Ctr1 is Ctr + 1,
                revers_lists(Ctr1, Stp).

        constraint(req).                                                    %

        assert_proc_const(Const, Node1, Node2, OldLConst, NewLConst):-      %
                C =.. [Const, Node1, Node2],
                assert(proc_const(C)),
                append(OldLConst, [C], NewLConst), !.

        establ_const(_, 0, 0, _, _, _, _, OldLConst, OldLConst):- !.        %
        establ_const(N1, _, _, N2, 1, 1, Const, OldLConst, NewLConst):-     %
                assert_proc_const(Const, N1, N2, OldLConst, NewLConst).
        establ_const(N1, 1, 0, N2, 1, 0, Const, OldLConst, NewLConst):-     %
                assert_proc_const(Const, N1, N2, OldLConst, NewLConst).
        establ_const(N1, 0, 1, N2, 0, 1, Const, OldLConst, NewLConst):-     %
                assert_proc_const(Const, N1, N2, OldLConst, NewLConst).
        establ_const(_, _, _, _, _, _, _, OldLConst, OldLConst):-  !.       %

        make_const(_, _, _, [], OldLConst, OldLConst):- !.                  %
        make_const(LeavNam, X1, Y1, [LeavNam|T], OldLConst, NewLConst):-    %
                retract(col2([X2, Y2])),
                assertz(col2([X2, Y2])),
                retract(constraint(req)),
                assert(constraint(nec)),
                make_const(LeavNam, X1, Y1, T, OldLConst, NewLConst).
        make_const(LeavNam1, X1, Y1, [LeavNam2|T], OldLConst, New1LConst):-  %
                retract(col2([X2, Y2])),
                constraint(C),
                establ_const(LeavNam1, X1, Y1, LeavNam2, X2, Y2, C, OldLConst, Ne
                assertz(col2([X2, Y2])),
                make_const(LeavNam1, X1, Y1, T, NewLConst, New1LConst).

        rev_const:-                                 % reinitialise la contrainte a re
                retract(constraint(_)),
                assert(constraint(req)), !.

        procedural_constraint([], _, OldLConst, OldLConst):-    % passe les branc
                retract_fact(col2, 1), !.
```

```
procedural_constraint([LeavNam|T], LeaveNames, OldLConst, NewlLConst):-
        retract(coll([X, Y])),
        rev_const,
        make_const(LeavNam, X, Y, LeaveNames, OldLConst, NewLConst),
        procedural_constraint(T, LeaveNames, NewLConst, NewlLConst).

eq_op(L, L, _, and, []):- !.                          % operateurs et feuilles
eq_op(L1, _, _, and, []):-                            % sous-arbre ET apparait
        empty_subt(L1), !.
eq_op(_, L2, _, and, []):-                            % sous-arbre ET apparait
        empty_subt(L2), !.
eq_op(L1, L2, L3, or, NewLConst):-                    % sous-arbre OU, regarder
        assert(tmp_list(L1)),                         % ajout de 1 et de 0 dans
        assert(tmp_list(L2)),
        length(L3, Lgth),                             % retourne le nombre de f
        N is Lgth + 1,
        revers_lists(1, N),                           % renverser le tableau tm
        procedural_constraint(L3, L3, [], NewLConst), !.

combin([], []):- !.                                   % fin de la liste
combin([H1|T1], [H2|T2]):-                            % inserer la valeur (CW)
        assert(tmp(H1, H2)),
        combin(T1, T2).

check_const(C, [C]):-                                 % verifier si la ProcConst
        C =.. [_Const, Node1, Node2],
        tmp(1, Node1),
        tmp(0, Node2),
        retract(proc_const(C)).
check_const(_, []):- !.

look_constraints([], []):- !.
look_constraints([H|T], NewLConst):-                  % verifie toutes les Proc
        check_const(H, LConst),
        look_constraints(T, L1Const),
        append(L1Const, LConst, NewLConst).

diff_op(L, L, _, _):- !.                              % operateurs diff., mais
diff_op(_, L2, L3, LConst):-                          % regarde pour ajouter de
        bagof(Op, proc_const(Op), L),
        combin(L2, L3),
        look_constraints(L, LConst),
        retract_fact(tmp, 2),!.

make_struct(_Str, _Ts, and, [and(_Str)|_Ts]):- !.     % compose la NT,
make_struct(_Str, _Ts, or, [or(_Str)|_Ts]):- !.       % compose la NT,

trav_trees([], [], [], [], [], [], [], [], []):- !.   % fin de niveau
trav_trees([t(X1, nil)|T1], [t(X2, nil)|T2], [t(M_Val, nil)|M_Ts], [X1|Y
    [X3|Y3], LConst_to_add, LConst_to_retract):-
        mask(X1, X2, M_Val),
        trav_trees(T1, T2, M_Ts, Y1, Y2, T3, Y3, LConst_to_add, LConst_to_
trav_trees([t(X1, and(SubStr1))|T1], [t(X2, and(SubStr2))|T2], [t(M_Val, N
    [X1|Subt1], [X2|Subt2], [t(X3, unkn(SubStr3))|T3], [X3|Subt3], NewLCor
        mask(X1, X2, M_Val),
        trav_trees(SubStr1, SubStr2, M_SubStr, Z1, Z2, SubStr3, Z3, LConst
        eq_op(Z1, Z2, Z3, Op, L1Const_to_add),
        trav_trees(T1, T2, M_Ts, Subt1, Subt2, T3, Subt3, L2Const_to_add,
        append(L1Const_to_add, L2Const_to_add, L3Const_to_add),
        append(LConst_to_add, L3Const_to_add, NewLConst_to_add),
        append(LConst_to_retract, L1Const_to_retract, NewLConst_to_retract
        make_struct(M_SubStr, M_Ts, Op, [M_SubStrH|M_SubStrT]).
trav_trees([t(X1, or(SubStr1))|T1], [t(X2, and(SubStr2))|T2], [t(M_Val, or
    [t(X3, unkn(SubStr3))|T3], [X3|Subt3], NewLConst_to_add, NewLConst_to_
        mask(X1, X2, M_Val),
        trav_trees(SubStr1, SubStr2, M_SubStr, Z1, Z2, SubStr3, Z3, LConst
```

```
                diff_op(Z1, Z2, Z3, L1Const_to_retract),
                trav_trees(T1, T2, M_Ts, Subt1, Subt2, T3, Subt3, L1Const_to_add,
                append(LConst_to_add, L1Const_to_add, NewLConst_to_add),
                append(L1Const_to_retract, L2Const_to_retract, L3Const_to_retract)
                append(LConst_to_retract, L3Const_to_retract, NewLConst_to_retract

        del_elem(X, [X|T], T).                                  % retirer le ler e
        del_elem(X, [Y|T], [Y|T1]):-                            % passe a l'elemer
                del_elem(X, T, T1).

        retract_proc_const([], OldLConst, OldLConst):- !.       % plus de ProcCons
        retract_proc_const([H|T], OldLConst, NewlLConst):-      % pour chaque Proc
                del_elem(H, OldLConst, NewLConst),
                retract_proc_const(T, NewLConst, NewlLConst).

discrim_subt(DT_Nodes, OldLConst, NewlLConst):-                 % comparaison des
        retract(cw_just(_Expl_Id, Expl)),                       % explic. du nouve
        retract(current_theo(CT)),                              % CT
        trav_trees(CT, Expl, NT, _, _, DT_Nodes, _, LConst_to_add, LConst_to_retra
        append(LConst_to_add, OldLConst, NewLConst),            % ajoute les nouve
        retract_proc_const(LConst_to_retract, NewLConst, NewlLConst),   % retire l
        assert(current_theo(NT)),                               % nouvelle DT
        !.                                                      % NB: que les Proc
```

```
% id_te:                  - identification of the TE used
% cw_leaf_to_leaf_poss:   - rows become col. & col. become rows...
%
%                                        cw_leaves
%                                leaf_poss | 1 2 3 4 5 . . .                cw_l
%                                       --|---------------
%                                       1 | 1 1 1 0 1
%                                       2 | 1 1 1 0 0
%                                       3 | 1 1 1 1 0          =====>
%                                       . |
%                                       . |
%                                       . |
%
%
%
%
%                        - find # of Elem in the 1st list "cw_leaves",
%                          which corresponds to the # of leaves in the domain theo
% rows_as_col:           - for 1 to # of Elem
% get_row:                       - for 1 to # of "cw_leaves"
% n_th_elem:                             - take the N_th elem of the list "cw_leav
%                                - assert a list of diff. possibilities found for
%                                  with N and the N_th_Elem of each "cw_leaves"
% compulsory:            - a compulsory leaf is a leaf that was present in all the
%                          i. e. each elem of a particular list "leaf_poss" (which
%                                has to be = to '1'
% compul_list:                   - make a compulsory list for next comparaisons "C
%                                  i.e. a list of # of TE of '1s'
% compul_comp:                   - for 1 to # of "leaf_poss" (# of leaves)
%                                        - if "Compul_List" = "Leaf_Poss" then
%                                                assert a compulsory operator with
% oth_proc_const:        - look for others than compulsory constraints (i.e. req,
%                          retract the 1st row "leaf_poss"
%                        - for 1 to # of "leaf_poss" (retract & fail)
%                                - take the following "leaf_poss" list
% comp_lists:                    - compare the 2 lists ("leaf_poss")
% poss_match:                    - returns 1(s) in the corr. position of possibili
%                                  list found by the comparaison
% operators:             - look for
% uninst_var_as_0:       - instantiated to 0 uninstantiated variabes in the possib
% oper:                  - look for a possible (corresponding) operator(i.e. proce

        get_row(N, Elem):- !,                          % chercher le Nieme elem.
                anal_of_rel(_Id, L),
                n_th_elem(1, N, L, Elem).


        rows_as_col:-
                anal_of_rel(_TE_Id, List_Leaves),
                length(List_Leaves, Nb_Leaves),
                N is Nb_Leaves + 1,
                rows_as_col(1, N).
        rows_as_col(Stop, Stop):- !.                   % cf. revers_lists => dis
        rows_as_col(Ctr, Stop):-
                bagof(X, get_row(Ctr, X), L),
                assert(leaf_poss(Ctr, L)),             % "leaf_poss..."
                Ctr1 is Ctr + 1,
                rows_as_col(Ctr1, Stop).

        uninst_var_as_0([]):- !.
        uninst_var_as_0([H|T]):-
                var(H),
                H is 0,
                uninst_var_as_0(T).
```

```
uninst_var_as_0([1|T]):- uninst_var_as_0(T).

find_elem(DT_Leaves, Leaf1, Leaf2, LeafName1, LeafName2):-
        n_th_elem(1, Leaf1, DT_Leaves, LeafName1),
        n_th_elem(1, Leaf2, DT_Leaves, LeafName2),
        !.

oper(Leaf1, Leaf2, [1, 1, 1, 0], DT_Leaves, [ior(LeafName1, LeafName2)]):
        find_elem(DT_Leaves, Leaf1, Leaf2, LeafName1, LeafName2).
oper(Leaf1, Leaf2, [1, 1, 0, 1], DT_Leaves, [req(LeafName2, LeafName1)]):
        find_elem(DT_Leaves, Leaf1, Leaf2, LeafName1, LeafName2).
oper(Leaf1, Leaf2, [1, 0, 1, 1], DT_Leaves, [nec(LeafName1, LeafName2)]):
        find_elem(DT_Leaves, Leaf1, Leaf2, LeafName1, LeafName2).
oper(Leaf1, Leaf2, [1, 0, 0, 1], DT_Leaves, [req(LeafName2, LeafName1)|[n
        find_elem(DT_Leaves, Leaf1, Leaf2, LeafName1, LeafName2).
oper(Leaf1, Leaf2, [0, 1, 1, 1], DT_Leaves, [eor(LeafName1, LeafName2)]):
        find_elem(DT_Leaves, Leaf1, Leaf2, LeafName1, LeafName2).
oper(Leaf1, Leaf2, [0, 1, 1, 0], DT_Leaves, [onl_o(LeafName1, LeafName2)]
        find_elem(DT_Leaves, Leaf1, Leaf2, LeafName1, LeafName2).
oper(_, _, _, _, []):- !.


add_proc_consts(Ctr1, Ctr2, Poss, DT_Leaves, Const):-
        uninst_var_as_0(Poss),
        oper(Ctr1, Ctr2, Poss, DT_Leaves, Const),
        !.

        poss_match(1, 1, [1, _, _, _]).
        poss_match(1, 0, [_, 1, _, _]).
                poss_match(0, 1, [_, _, 1, _]).
                poss_match(0, 0, [_, _, _, 1]).

        comp_lists([], [], _R):- !.
        comp_lists([H1|T1], [H2|T2], Poss):- !,
                poss_match(H1, H2, Poss),
                comp_lists(T1, T2, Poss), !.

compare_lists(Stp, Stp, _, _, _, []):- !.
compare_lists(N, Stp, Leaf_Id1, L1, DT_Leaves, NewConst):-
        retract(leaf_poss(Leaf_Id2, L2)),
        assertz(leaf_poss(Leaf_Id2, L2)),
        comp_lists(L1, L2, Poss_Combin),
        add_proc_consts(Leaf_Id1, Leaf_Id2, Poss_Combin, DT_Leaves, Const),
        N1 is N + 1,
        compare_lists(N1, Stp, Leaf_Id1, L1, DT_Leaves, NextConst),
        append(Const, NextConst, NewConst).


oth_add_proc_const(Stp, Stp, _, []):- !.
oth_add_proc_const(N, Stp, DT_Leaves, NewConst):-
        retract(leaf_poss(Leaf_Id, L)),
        N1 is N + 1,
        compare_lists(N1, Stp, Leaf_Id, L, DT_Leaves, Const),
        oth_add_proc_const(N1, Stp, DT_Leaves, NextConst),
        append(Const, NextConst, NewConst).


        compul_list(N, N, Compul_List, [1|Compul_List]):- !.
        compul_list(1, N, [], R_Compul_List):-
                compul_list(2, N, [1], R_Compul_List), !.
        compul_list(Ctr, N, [Compul_List], [1|R_Compul_List]):-
                Ctr1 is Ctr + 1,
                compul_list(Ctr1, N, [Compul_List], R_Compul_List).

compar(L, L, Leaf_Id, DT_Leaves, [compul(LeafName)]):-
        n_th_elem(1, Leaf_Id, DT_Leaves, LeafName),
```

```
        !.
compar(_, _, _, _, []):- !.

%       "leaf_poss" will usefull in the next procedure: "oth_add_proc_const" ...
        compulsory_oper(Stp, Stp, _, _, []):- !.
        compulsory_oper(N, Stp, Compul_List, DT_Leaves, NewConst):-
                retract(leaf_poss(Leaf_Id, L)),
                assertz(leaf_poss(Leaf_Id, L)),
                compar(L, Compul_List, Leaf_Id, DT_Leaves, Const),
                N1 is N + 1,
                compulsory_oper(N1, Stp, Compul_List, DT_Leaves, NextConst),
                append(Const, NextConst, NewConst).


show_oper([none]):-
        write(' No procedural constraints exit !...'), nl, !.
show_oper([none|List_oper]):-
        write_list(List_oper), nl.

analysis_of_relations(DT_Leaves, LConst_ARel):-
        rows_as_col,
        leaf_poss(_L_Id, List_Poss),
        length(List_Poss, Nb_TE),
        length(DT_Leaves, Nb_Leaves),
        compul_list(1, Nb_TE, [], Compul_List),
        Nb is Nb_Leaves + 1,
        compulsory_oper(1, Nb, Compul_List, DT_Leaves, CompulConst),
        oth_add_proc_const(1, Nb, DT_Leaves, OthConst),
        append(CompulConst, OthConst, LConst_ARel),
        !.
```

```
transform_H(H, [C, Arg1, Arg2]):-
        H =.. [C, Arg1, Arg2], !.

look_arg(N_Op_Arg, [_, N_Op_Arg, _]):- !.       % argument non operationnel appar
look_arg(N_Op_Arg, [_, _, N_Op_Arg]):- !.       % argument non operationnel appar

%       look_arg: retourne dans LRetract la liste de ProcConst a changer: argumen
%       (lorsque la ProcConst n'implique pas sur une feuille
look_const(_, [], []):- !.                      % plus de constraintes a verifier
look_const(N_Op_Arg, [H|T], [H|LRetract]):-     % l'argument non-operationnel app
        transform_H(H, L),                      % isole le predicat de la 1ere co
        look_arg(N_Op_Arg, L),                  % regarde les arguments
        look_const(N_Op_Arg, T, LRetract).      % verifie la contrainte suivante
look_const(N_Op_Arg, [_|T], LRetract):-         % passe a la contrainte suivante
        look_const(N_Op_Arg, T, LRetract).


make_Lof_Modif(_, _, [], []):- !.               % composer la liste de modificati
make_Lof_Modif(N_Op_Arg, L_Op_Arg, LConst, [N_Op_Arg, L_Op_Arg, LConst]):- !.

%       conversion de la DT en CD
%       propager tous les ET/OU et les ProcConst vers le dernier niveau de la str
dt_cd_s_tree2(t(1, nil), t(Elem, nil), [Elem]):- !.     % feuille presente
dt_cd_s_tree2(t(0, nil), _L, []):- !.                   % feuille absente

dt_cd_s_tree1([], [], [], _, []):- !.                   % fin d'un niveau
dt_cd_s_tree1([t(1, and(StBr_u))|StBr_b], [t(Node, unkn(StBr_under))|StBr_besid],
        look_const(Node, LConst_DSubt, LRetract),
        dt_cd_s_tree1(StBr_u, StBr_under, CD_Br_u, LConst_DSubt, L1of_Modif),   %
        dt_cd_s_tree1(StBr_b, StBr_besid, CD_Br_b, LConst_DSubt, L2of_Modif),   %
        append([and(CD_Br_u)], CD_Br_b, NewCD),                                 %
        make_Lof_Modif(Node, CD_Br_u, LRetract, Lof_Modif),
        append(Lof_Modif, L1of_Modif, L3of_Modif),
        append(L2of_Modif, L3of_Modif, NewLRetract).
dt_cd_s_tree1([t(1, or(StBr_u))|StBr_b], [t(Node, unkn(StBr_under))|StBr_besid],
        look_const(Node, LConst_DSubt, LRetract),
        dt_cd_s_tree1(StBr_u, StBr_under, CD_Br_u, LConst_DSubt, L1of_Modif),   %
        dt_cd_s_tree1(StBr_b, StBr_besid, CD_Br_b, LConst_DSubt, L2of_Modif),   %
        append([or(CD_Br_u)], CD_Br_b, NewCD),                                  %
        make_Lof_Modif(Node, CD_Br_u, LRetract, Lof_Modif),
        append(Lof_Modif, L1of_Modif, L3of_Modif),
        append(L2of_Modif, L3of_Modif, NewLRetract).
dt_cd_s_tree1([t(0, _)|StBr_b], [t(_, _)|StBr_besid], CD_Br_b, LConst_DSubt, LRet
        dt_cd_s_tree1(StBr_b, StBr_besid, CD_Br_b, LConst_DSubt, LRetract).
dt_cd_s_tree1([StN|StBr_b], [StNode|StBr_besid], NewLElem, LConst_DSubt, LRetract
        dt_cd_s_tree2(StN, StNode, Elem),
        dt_cd_s_tree1(StBr_b, StBr_besid, LElem, LConst_DSubt, LRetract),
        append(Elem, LElem, NewLElem).

dt_cd([t(_, and(StL))], [t(_Root, unkn(StList))], [and(CD_List)], LConst_DSubt, L
        dt_cd_s_tree1(StL, StList, CD_List, LConst_DSubt, Lof_Modif).
dt_cd([t(_, or(StL))], [t(_Root, unkn(StList))], [or(CD_List)], LConst_DSubt, Lof
        dt_cd_s_tree1(StL, StList, CD_List, LConst_DSubt, Lof_Modif).

modif(N_Op_Arg, Op_Arg, [Oper, N_Op_Arg, Arg2], Op_Const):-             % modifie
        Op_Const =.. [Oper, Op_Arg, Arg2], !.
modif(N_Op_Arg, Op_Arg, [Oper, Arg1, N_Op_Arg], Op_Const):-             % modifie
        Op_Const =.. [Oper, Arg1, Op_Arg], !.

modification([_, [], _], []):- !.                                       % plus de
modification([N_Op_Arg, [H_Op_Arg|T_Op_Arg], C_to_modif], NewLConst):-  % remplac
        C_to_modif =.. [Oper, Arg1, Arg2],
        modif(N_Op_Arg, H_Op_Arg, [Oper, Arg1, Arg2], Op_Const),        % 1er ou
        modification([N_Op_Arg, T_Op_Arg, C_to_modif], LOp_Const),
```

```
        append([Op_Const], LOp_Const, NewLConst).

make_modif([_, _, []], OldLConst, OldLConst):- !.
make_modif([N_Op_Arg, L_Op_Arg, [H_Modif|T_Modif]], [H_Modif|T_OldLConst], NewLCo
        modification([N_Op_Arg, L_Op_Arg, H_Modif], NewH),
        make_modif([N_Op_Arg, L_Op_Arg, T_Modif], T_OldLConst, NewT),
        append(NewH, NewT, NewLConst).
make_modif([N_Op_Arg, L_Op_Arg, [H_Modif|T_Modif]], [H_OldLConsc|T_OldLConst], Ne
        make_modif([N_Op_Arg, L_Op_Arg, [H_Modif|T_Modif]], T_OldLConst, T_LConst
        append([H_OldLConst], T_LConst, NewLConst).

take_out([], Elem, [Elem]):- !.                          % conserver la ProcConst de AnalR
take_out([Elem|_], Elem, []):- !.                        % ProcConst existe deja dans Proc
take_out([_|T], Elem, NewElem):-                         % verifier chaque ProcConst des P
        take_out(T, Elem, NewElem).

take_out_rep(_, [], []) :- !.                            % plus de ProcConst a verifier da
take_out_rep(L, [H|T], NewL):-
        take_out(L, H, NewL1),                           % verifie que la ProcConst de DSu
        take_out_rep(L, T, NewL2),                       % passe a la Proc'onst DSubt suiv
        append(NewL1, NewL2, NewL).

concept_definition(DT_Nodes, LConst_DSubt, LConst_ARel, [CD, NewLConst]):- % comp
        write('New Domain Theory'), nl,
        write('-----------------'), nl,
        current_theo(CT),                                          % conserver dans
        w_struct(CT, DT_Nodes), nl,                                % dans write_proc
        dt_cd(CT, DT_Nodes, CD, LConst_DSubt, Lof_Modif), nl,      % CD fait a parti
        make_modif(Lof_Modif, LConst_DSubt, NewLConst_DSubt),      % remplacer les a
        write('New Concept Definition'), nl,
        write('----------------------'), nl,
        write_CD(CD, 3),                                           % dans write_proc
        take_out_rep(NewLConst_DSubt, LConst_ARel, NewLConst_ARel), % retire les
        append(NewLConst_ARel, NewLConst_DSubt, NewLConst),        % forme la liste
        write_list(NewLConst), nl, nl,                             % dans write_proc
        !.
```

```
%       write_procedure                                  4 avril 1990

        write_list([]):-  nl, !.                                  % plus d'elements
        write_list([H|T]):-                                       % passe au traver:
                write(H), nl,
                write_list(T).

        sub_tree2(t(Elem, nil), Tab):-                            % feuille
                tab(Tab),
                write(Elem), nl, !.

        sub_tree1([], _Tab):- !.                                  % fin d'un niveau
        sub_tree1([t(Node, and(StBr_under))|StBr_besid], Tab):-   % noeud ET
                tab(Tab),
                write(Node), nl,
                Tab1 is Tab + 4,
                tab(Tab1),
                write('and'), nl,
                sub_tree1(StBr_under, Tab1),                      % sous-arbre d'ab
                sub_tree1(StBr_besid, Tab).                       % autres branches
        sub_tree1([t(Node, or(StBr_under))|StBr_besid], Tab):-    % noeud OU
                tab(Tab),
                write(Node), nl,
                Tab1 is Tab + 4,
                tab(Tab1),
                write('or'), nl,
                sub_tree1(StBr_under, Tab1),                      % sous-arbre d'ab
                sub_tree1(StBr_besid, Tab).                       % autres branches
        sub_tree1([StNode|StBr_besid], _Tab):-
                sub_tree2(StNode, _Tab),                          % feuille
                sub_tree1(StBr_besid, _Tab).                      % autres branches

        write_struct([t(Root, and(StList))]):-                    % arbre ET
                write(Root), nl,
                write('and'), nl,
                sub_tree1(StList, 4).
        write_struct([t(Root, or(StList))]):-                     % arbre OU
                write(Root), nl,
                write('or'), nl,
                sub_tree1(StList, 4).
        write_structure(Str):-
                write_struct(Str), nl.

write_procedure(TE_Id, []):-
        write(TE_Id),
        write(' cannot be learned: INCOMPLETE THEORY ...'), nl, !, fail.
write_procedure(TE_Id, Reg):-
        nl, write('Training Example:   '),
        write(TE_Id), tab(4),write('is not already learned.'), nl, nl,
        write('Generalized explanation'), nl,
        write('------------------------'), nl,
        write_struct(Reg), nl, !.

write_CD([], Tab):-                 % fin d'un niveau
        tab(Tab),
        write(')'), nl, !.
write_CD([and(LArg)], Tab):-        % noeud ET
        tab(Tab),
        write('and('), nl,
        Tab1 is Tab + 3,
        write_CD(LArg, Tab1).
write_CD([or(LArg)], Tab):-         % noeud OU
        tab(Tab),
        write('or ('), nl,
        Tab1 is Tab + 3,
        write_CD(LArg, Tab1).
```

```
write_CD([H|T], Tab):-          % branches d'un meme niveau
        tab(Tab),
        write_CD(H, Tab),
        write_CD(T, Tab).
write_CD(and(LArg), Tab):-      % arbre ET
        write('and('), nl,
        Tab1 is Tab + 3,
        write_CD(LArg, Tab1), !.
write_CD(or(LArg), Tab):-       % arbre OU
        write('or ('), nl,
        Tab1 is Tab + 3,
        write_CD(LArg, Tab1), !.
write_CD(Elem, _Tab):-          % feuille
        write(Elem), nl,
        !.


        s_tree2(t(1, nil), t(Elem, nil), Tab):-                 % feuille present
                tab(Tab),
                write(Elem), nl, !.
        s_tree2(t(0, nil), _L, _):- !.                          % feuille absente

        s_tree1([], [], _):- !.                                 % fin d'un niveau
        s_tree1([t(1, and(StBr_u))|StBr_b], [t(Node, unkn(StBr_under))|StBr_besid
                tab(Tab),
                write(Node), nl,
                Tab1 is Tab + 4,
                tab(Tab1),
                write('and'), nl,
                s_tree1(StBr_u, StBr_under, Tab1),              % sous-arbre d'ab
                s_tree1(StBr_b, StBr_besid, Tab).               % autres branches
        s_tree1([t(1, or(StBr_u))|StBr_b], [t(Node, unkn(StBr_under))|StBr_besid],
                tab(Tab),
                write(Node), nl,
                Tab1 is Tab + 4,
                tab(Tab1),
                write('or'), nl,
                s_tree1(StBr_u, StBr_under, Tab1),              % sous-arbre d'ab
                s_tree1(StBr_b, StBr_besid, Tab).               % autres branches
        s_tree1([t(0, _)|StBr_b], [t(_, _)|StBr_besid], Tab):-  % aucun sous-arbre
                s_tree1(StBr_b, StBr_besid, Tab).               % passe a la bran
        s_tree1([StN|StBr_b], [StNode|StBr_besid], _Tab):-      % liste de branche
                s_tree2(StN, StNode, _Tab),                     % premiere branche
                s_tree1(StBr_b, StBr_besid, _Tab).              % autres branches

        w_str([t(_, and(StL))], [t(Root, unkn(StList))]):-      % arbre ET
                write(Root), nl,
                write('and'), nl,
                s_tree1(StL, StList, 4).
        w_str([t(_, or(StL))], [t(Root, unkn(StList))]):-       % arbre OU
                write(Root), nl,
                write('or'), nl,
                s_tree1(StL, StList, 4).

w_struct(StrCW, Str):-                                          % ecrit la structure avec
        nl, w_str(StrCW, Str), nl, !.
```

```
%       subway (input file pour learn_proc_const)          6 mars 1990

%       q2
%       [learn_proc_const].
%       execution automatique
%       halt.

:- dynamic
        take_subway14/1,
        pay_for3/1,
        enter_station9/1,
        get_access6/1,
        exit_station13/1,
        order/1.


%       the goal concept
goal_concept(take_subway14).

%       domain theory
%       -------------
take_subway14(_Id):-
        pay_for3(Access),
        enter_station9(Access),
        ride_subway10,
        exit_station13(Access).

pay_for3(Access):-
        buy_from1(Access, _Dispenser),
        receive2(Access).

enter_station9(Access):-
        get_access6(Access),
        enter7(_Ent_Machine),
        receive8(Access).

get_access6(Access):-
        put_into4(Access, _Access_Machine1),
        receive5(Access).

ride_subway10.

exit_station13(Access):-
        put_into11(Access, _Access_Machine2),
        exit12(_Exit_Machine).

% training instance
% -----------------

train_ex(bart, [buy_from1(ticket, machine1),
        receive2(ticket),
        put_into4(ticket, machine2),
        enter7(through_gate),
        receive8(ticket),
        ride_subway10,
        put_into11(ticket, machine3),
        exit12(through_gate)]).

train_ex(nyc, [buy_from1(token, teller),
        receive2(token),
        put_into4(token, turnstile),
        enter7(xxxxxxx),
        ride_subway10,
        exit12(through_turnstile)]).

train_ex(paris, [buy_from1(ticket, cashier),
```

```
        receive2(ticket),
        put_into4(ticket, turnstile),
        receive5(ticket),
        enter7(xxxxxxxx),
        ride_subway10,
        exit12(yyyyyyyyy)]).

train_ex(paris_rer, [buy_from1(ticket, cashier),
        receive2(ticket),
        put_into4(ticket, turnstile),
        receive5(ticket),
        enter7(xxxxxxxx),
        ride_subway10,
        put_into11(ticket, turnstile),
        exit12(yyyyyyyyy)]).

train_ex(mexico, [buy_from1(ticket, cashier),
        receive2(ticket),
        put_into4(ticket, turnstile),
        enter7(ppppp),
        ride_subway10,
        exit12(mmmmm)]).

train_ex(toronto, [buy_from1(ticket, teller),
        enter7(through_turnstile),
        receive8(ticket),
        ride_subway10,
        exit12(zzzzz)]).

train_ex(montreal, [buy_from1(ticket, cashier),
        receive2(ticket),
        put_into4(ticket, tunstile),
        enter7(ppppppp),
        ride_subway10,
        exit12(1111111111)]).

% ... incomplete DT
train_ex(moscow, [have1(coin),
        put_into2(coin, turnstile),
        enter7(turnstile),
        ride_subway10,
        exit12(turnstile)]).

train_ex(prague, [put_into1(money, machine),
        receive2(coin),
        put_into4(coin, turnstile),
        enter7(turnstile),
        ride_subway10,
        exit12(special_gate)]).

%       the order in which the TE are entered
% order thesis
order(moscow).
order(nyc).
order(prague).
order(toronto).
order(mexico).
order(bart).
order(montreal).
order(paris).
order(paris_rer).

%       order #1: bart, nyc, paris, paris_rer, toronto, mexico, montreal
%       order(paris_rer).
%       order(toronto).
%       order(bart).
```

```
%       order(nyc).
%       order(mexico).
%       order(montreal).
%       order(paris).

%       order #2:  paris_rer, toronto, bart, nyc, montreal, mexico, paris
%       order(bart).
%       order(nyc).
%       order(paris).
%       order(paris_rer).
%       order(mexico).
%       order(toronto).
%       order(montreal).
```

```
Script started on Sat Apr  7 21:17:53 1990
{ai3}jmorin(1) q2

Quintus Prolog Release 2.4.2 (Sun-3, SunOS 3.4)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- [learn_proc_const].
[consulting /home/prga/usr7/grad/jmorin/ML/ebg/pro/learning_procedural_constraint
 [compiling /pub/Quintus/q2.4.2/library/not.pl...]
  [compiling /pub/Quintus/q2.4.2/library/freevars.pl...]
  [freevars.pl compiled in module free_variables 2.600 sec 2,680 bytes]
  [compiling /pub/Quintus/q2.4.2/library/break.pl...]
  [break.pl compiled in module break 0.350 sec 564 bytes]
 [not.pl compiled in module negation 4.133 sec 5,540 bytes]
 [consulting /pub/Quintus/q2.4.2/library/basics.pl...]
 [basics.pl consulted in module basics 0.683 sec 1,932 bytes]
 [consulting /home/prga/usr7/grad/jmorin/ML/ebg/pro/learning_procedural_constrain
 [subway consulted 0.633 sec 3,928 bytes]
 [consulting /home/prga/usr7/grad/jmorin/ML/ebg/pro/learning_procedural_constrain
 [common consulted 0.283 sec 1,216 bytes]
 [consulting /home/prga/usr7/grad/jmorin/ML/ebg/pro/learning_procedural_constrain
 [ebg consulted 0.567 sec 1,984 bytes]
 [consulting /home/prga/usr7/grad/jmorin/ML/ebg/pro/learning_procedural_constrain
 [already_learned consulted 0.983 sec 4,412 bytes]
 [consulting /home/prga/usr7/grad/jmorin/ML/ebg/pro/learning_procedural_constrain
 [discrim_subt consulted 1.733 sec 7,168 bytes]
 [consulting /home/prga/usr7/grad/jmorin/ML/ebg/pro/learning_procedural_constrain
 [analysis_of_relations consulted 1.267 sec 5,508 bytes]
 [consulting /home/prga/usr7/grad/jmorin/ML/ebg/pro/learning_procedural_constrain
 [concept_definition consulted 1.284 sec 5,304 bytes]
 [consulting /home/prga/usr7/grad/jmorin/ML/ebg/pro/learning_procedural_constrain
 [write_procedure consulted 1.100 sec 5,388 bytes]

List of training examples to learn from:

moscow
nyc
prague
toronto
mexico
bart
montreal
paris
paris_rer

moscow cannot be learned: INCOMPLETE THEORY ...

Training Example:   nyc    is not already learned.

Generalized explanation
-----------------------
take_subway14(_1905)
and
    pay_for3(_2102)
        and
        buy_from1(_2102,_2488)
        receive2(_2102)
    enter_station9(_2102)
        and
        get_access6(_2102)
            and
            put_into4(_2102,_4185)
        enter7(_3821)
    ride_subway10
    exit_station13(_2102)
```

```
        and
        exit12(_7123)

nyc   First justification becomes the current theory ...

prague cannot be learned: INCOMPLETE THEORY ...

Training Example:   toronto    is not already learned.

Generalized explanation
-----------------------
take_subway14(_1905)
and
    pay_for3(_2102)
        and
        buy_from1(_2102,_2488)
    enter_station9(_2102)
        and
        enter7(_3821)
        receive8(_2102)
    ride_subway10
    exit_station13(_2102)
        and
        exit12(_7123)

New Domain Theory
-----------------

take_subway14(_1905)
and
    pay_for3(_2102)
        or
        buy_from1(_2102,_2488)
        receive2(_2102)
    enter_station9(_2102)
        or
        get_access6(_2102)
            and
            put_into4(_2102,_4185)
        enter7(_3821)
        receive8(_2102)
    ride_subway10
    exit_station13(_2102)
        and
        exit12(_7123)


New Concept Definition
----------------------
   and(
     or (
        buy_from1(_2102,_2488)
        receive2(_2102)
        )
     or (
        and(
           put_into4(_2102,_4185)
           )
        enter7(_3821)
        receive8(_2102)
        )
     ride_subway10
     and(
        exit12(_7123)
        )
```

```
compul(buy_from1(_2102,_2488))
compul(enter7(_3821))
compul(ride_subway10)
compul(exit12(_7123))
req(put_into4(_2102,_4185),receive2(_2102))
nec(receive2(_2102),put_into4(_2102,_4185))
onl_o(receive2(_2102),receive8(_2102))
onl_o(put_into4(_2102,_4185),receive8(_2102))
req(receive2(_2102),buy_from1(_2102,_2488))
nec(put_into4(_2102,_4185),enter7(_3821))
req(receive8(_2102),enter7(_3821))


mexico is ALREADY LEARNED !


Training Example:   bart    is not already learned.

Generalized explanation
-----------------------
take_subway14(_1905)
and
    pay_for3(_2102)
        and
        buy_from1(_2102,_2488)
        receive2(_2102)
    enter_station9(_2102)
        and
        get_access6(_2102)
            and
            put_into4(_2102,_4185)
        enter7(_3821)
        receive8(_2102)
    ride_subway10
    exit_station13(_2102)
        and
        put_into11(_2102,_7118)
        exit12(_7123)

New Domain Theory
-----------------

take_subway14(_1905)
and
    pay_for3(_2102)
        or
        buy_from1(_2102,_2488)
        receive2(_2102)
    enter_station9(_2102)
        or
        get_access6(_2102)
            and
            put_into4(_2102,_4185)
        enter7(_3821)
        receive8(_2102)
    ride_subway10
    exit_station13(_2102)
        or
        put_into11(_2102,_7118)
        exit12(_7123)


New Concept Definition
----------------------
```

```
   and(
      or (
         buy_from1(_2102,_2488)
         receive2(_2102)
         )
      or (
         and(
            put_into4(_2102,_4185)
            )
         enter7(_3821)
         receive8(_2102)
         )
      ride_subway10
      or (
         put_into11(_2102,_7118)
         exit12(_7123)
         )
compul(buy_from1(_2102,_2488))
compul(enter7(_3821))
compul(ride_subway10)
compul(exit12(_7123))
req(put_into4(_2102,_4185),receive2(_2102))
nec(receive2(_2102),put_into4(_2102,_4185))
ior(receive2(_2102),receive8(_2102))
req(put_into11(_2102,_7118),receive2(_2102))
ior(put_into4(_2102,_4185),receive8(_2102))
req(put_into11(_2102,_7118),put_into4(_2102,_4185))
req(put_into11(_2102,_7118),receive8(_2102))
nec(put_into11(_2102,_7118),exit12(_7123))
req(receive2(_2102),buy_from1(_2102,_2488))
nec(put_into4(_2102,_4185),enter7(_3821))
req(receive8(_2102),enter7(_3821))


montreal is ALREADY LEARNED !


Training Example:   paris    is not already learned.

Generalized explanation
-----------------------
take_subway14(_1905)
and
    pay_for3(_2102)
        and
        buy_from1(_2102,_2488)
        receive2(_2102)
    enter_station9(_2102)
        and
        get_access6(_2102)
            and
            put_into4(_2102,_4185)
            receive5(_2102)
        enter7(_3821)
    ride_subway10
    exit_station13(_2102)
        and
        exit12(_7123)

New Domain Theory
-----------------

take_subway14(_1905)
and
    pay_for3(_2102)
```

```
        or
        buy_from1(_2102,_2488)
        receive2(_2102)
    enter_station9(_2102)
        or
        get_access6(_2102)
            or
            put_into4(_2102,_4185)
            receive5(_2102)
        enter7(_3821)
        receive8(_2102)
    ride_subway10
    exit_station13(_2102)
        or
        put_into11(_2102,_7118)
        exit12(_7123)
```

New Concept Definition

```
   and(
      or (
         buy_from1(_2102,_2488)
         receive2(_2102)
         )
      or (
         or (
            put_into4(_2102,_4185)
            receive5(_2102)
            )
         enter7(_3821)
         receive8(_2102)
         )
      ride_subway10
      or (
         put_into11(_2102,_7118)
         exit12(_7123)
         )
compul(buy_from1(_2102,_2488))
compul(enter7(_3821))
compul(ride_subway10)
compul(exit12(_7123))
req(put_into4(_2102,_4185),receive2(_2102))
nec(receive2(_2102),put_into4(_2102,_4185))
req(receive5(_2102),receive2(_2102))
ior(receive2(_2102),receive8(_2102))
req(put_into11(_2102,_7118),receive2(_2102))
ior(put_into4(_2102,_4185),receive8(_2102))
req(put_into11(_2102,_7118),put_into4(_2102,_4185))
eor(receive5(_2102),receive8(_2102))
eor(receive5(_2102),put_into11(_2102,_7118))
req(put_into11(_2102,_7118),receive8(_2102))
req(receive5(_2102),put_into4(_2102,_4185))
nec(put_into11(_2102,_7118),exit12(_7123))
req(receive2(_2102),buy_from1(_2102,_2488))
nec(put_into4(_2102,_4185),enter7(_3821))
nec(receive5(_2102),enter7(_3821))
req(receive8(_2102),enter7(_3821))
```

Training Example: paris_rer is not already learned.

Generalized explanation

```
----------------------
take_subway14(_1905)
and
    pay_for3(_2102)
        and
        buy_from1(_2102,_2488)
        receive2(_2102)
    enter_station9(_2102)
        and
        get_access6(_2102)
            and
            put_into4(_2102,_4185)
            receive5(_2102)
        enter7(_3821)
    ride_subway10
    exit_station13(_2102)
        and
        put_into11(_2102,_7118)
        exit12(_7123)

New Domain Theory
-----------------

take_subway14(_1905)
and
    pay_for3(_2102)
        or
        buy_from1(_2102,_2488)
        receive2(_2102)
    enter_station9(_2102)
        or
        get_access6(_2102)
            or
            put_into4(_2102,_4185)
            receive5(_2102)
        enter7(_3821)
        receive8(_2102)
    ride_subway10
    exit_station13(_2102)
        or
        put_into11(_2102,_7118)
        exit12(_7123)


New Concept Definition
----------------------
   and(
      or (
         buy_from1(_2102,_2488)
         receive2(_2102)
         )
      or (
         or (
            put_into4(_2102,_4185)
            receive5(_2102)
            )
         enter7(_3821)
         receive8(_2102)
         )
      ride_subway10
      or (
         put_into11(_2102,_7118)
         exit12(_7123)
         )
compul(buy_from1(_2102,_2488))
```

```
compul(enter7(_3821))
compul(ride_subway10)
compul(exit12(_7123))
req(put_into4(_2102,_4185),receive2(_2102))
nec(receive2(_2102),put_into4(_2102,_4185))
req(receive5(_2102),receive2(_2102))
ior(receive2(_2102),receive8(_2102))
req(put_into11(_2102,_7118),receive2(_2102))
ior(put_into4(_2102,_4185),receive8(_2102))
req(put_into11(_2102,_7118),put_into4(_2102,_4185))
eor(receive5(_2102),receive8(_2102))
req(receive5(_2102),put_into4(_2102,_4185))
nec(put_into11(_2102,_7118),exit12(_7123))
req(receive2(_2102),buy_from1(_2102,_2488))
nec(put_into4(_2102,_4185),enter7(_3821))
nec(receive5(_2102),enter7(_3821))
req(receive8(_2102),enter7(_3821))


    No more training example !!! ..


Incremental Learning of Procedural Constraints took 8.833 sec.


[learn_proc_const consulted 22.917 sec 47,512 bytes]

yes
| ?- halt.
{ai3}jmorin(2) ^D
script done on Sat Apr  7 21:19:02 1990
```

Références

Alterman, R. (1988). "Adaptive Planning", *Cognitive Science*, vol. 12, pp. 393-421.

Bergadano, F. and Giordana, A. (1988) "Knowledge Intensive Approach to Concept Induction", Procs. of *the Fifth International Conference on Machine Learning*, University of Michigan.

DeJong, G.F. and Mooney, R. (1986). "Explanation-Based Learning: An Alternative View", *Machine Learning*, vol. 1, pp. 145-176.

Diday, E., Lemaire, J., Poujet, J. and Testu, F. (1982). *Eléments d'analyse des données*, Dunod, Paris.

Dietterich, T.G. (1986). "Learning at the Knowledge Level", *Machine Learning*, vol. 1, no. (3), pp. 287-316.

Dietterich, T.G. and Flann, N.S. (1988) "An Inductive Approach to Solving the Imperfect Theory Problem", Procs. of *Spring Symposium Series, Explanation-Based Learning*, Stanford University,

Ellman, T. (1989). "Explanation-Based Learning: a Survey of Programs and Perspectives", *ACM: Computing Surveys*, vol. 21, 2, no. 2 (June),

Geldrez, C., Matwin, S., Morin, J. and Probert, R.L. (1989). "Protocol Conformance Testing by Explanation-Based Learning", *IEEE Expert (submitted)*.

Hirsh, H. (1989a) "Combining Empirical and Analytical Learning with Version Space", Procs. of *the Sixth International Workshop on Machine Learning*, Ithaca, N-Y, pp. 29-33.

Hirsh, H. (1989b) "Incremental Version-Space Merging: A General Framework for Concept Learning", PhD, Stanford.

Hoff, W.A., Michalski, R.S. and Stepp, R.E. (1982) "INDUCE 3: A program for Learning Structural Descriptions from Examples", Research Report, Department of Computer Science, University of Illinois at Urbana-Champaign, 82-5.

Matwin, S. and Morin, J. (1989) "Learning Procedural Knowledge in the EBG Context", Procs. of *Sixth International Workshop on Machine Learning*, Ithaca, NY, pp. 197-200.

Michalski, R. (1987) "The Inference-Based Theory of Learning: Principles and Approches", Procs. of *The International Workshop On Knowledge Representation and Organization in Machine Learning*, Geseke, Germany.

Michalski, R.S. (1983). "A Theory and Methodology of Inductive Learning", in *Machine Learning* , vol. 1, R. S. Michalski, J. Carbonell and T. Mitchell, ed. , Morgan-Kaufmann, pp. 83-134.

Michalski, R.S., Carbonell, J. and Mitchell, T. ed. (1986). *Machine Learning*, 2, Morgan-Kaufmann.

Mitchell, T., Keller, R.M. and Kedar-Cabelli, S.T. (1986). "Explanation-Based Generalization: A Unifying View", *Machine Learning*, vol. 1, pp. 47-80.

Mitchell, T.M. (1978) "Version Spaces: An Approach to Concept Learning", PhD, Stanford.

Mooney, R. and Ourston, D. (1989) "Induction Over the Unexplained: Integrated Learning of Concepts with Both Explainable and Conventional Aspects.", Procs. of *Proceedings of the Sixth International Conference on Machine Learning*, Ithaca, N-Y, pp. 5-.

Myers, G.J. (1979). *The Art of Software Testing* , Wiley.

Pazzani, M. (1987) "Creating High Level Knowledge Structures from Simple Elements", Procs. of *The International Workshop on Knowledge Representation and Organization in Machine Learning*, Geseke, Germany.

Pazzani, M. (1988) "Integrated Learning with Incorrect and Incomplete Theories", Procs. of *the Fifth International Conference on Machine Learning*, University of Michigan.

Winston, P.H. (1984). *Artificial Intelligence* , Addition-Wesley (ed.), Addition-Wesley Publishing Company.