Critères de vérité en planification.

Dans le cadre de la planification, nous abordons le problème de la génération de plans avec et sans expertise. De nombreux travaux ont été consacrés à l'élaboration de formalismes de représentation d'actions et de liens temporels. Par delà les qualités respectives de ces planificateurs purs subsiste un écueil commun qui est la difficulté d'extension du mode de résolution de conflits entre actions, due à l'imprécision dans la définition de notions aussi basiques que le "conflit" par exemple.

Contre cela, nous avons mis au point un planificateur pur, Yaps, permettant d'isoler clairement ce contrôle heuristique. La technique de génération de plans d'actions que nous proposons, rejoignant des approches récentes, est fondée sur la définition préalable d'un critère permettant de donner une valeur de vérité à un terme en un noeud d'un graphe non linéaire d'actions exprimées en logique d'ordre 1. Ce critère permet, en déduction, de détecter les conflits et les termes satisfaits, de façon potentielle ou nécessaire ; il permet, en induction, d'identifier les amendements possibles du plan (pose de contraintes de précédence ou d'unification, ajout d'action) résolvant les conflits détectés, délimitant ainsi le rôle du contrôle heuristique. Nous validons cette optique en retrouvant les résultats de micro-mondes classiques (singe / bananes, cubes, construction de maison, ...).

L'approche alternative à la planification pure consiste à déduire l'existence, les caractéristiques et les liens entre tâches à partir d'une expertise, non à partir d'un calcul portant sur les effets des actions effectuées par chaque tâche. Nous présentons également un planificateur, Coplaner, utilisant une expertise dans le domaine des chantiers de bâtiments, dont les résultats en grandeur réelle ont été confirmés par un expert humain.

Mots-clés : Intelligence artificielle, Planification non linéaire, Critère de vérité, Planification experte.

ECOLE NATIONALE SUPERIEURE DES TELECOMMUNICATIONS

Téléphone (1) 45 81 77 77 - Télécopie (1) 45 89 79 06 - Telex SUPTLCM 200160F

FRANCE TELECOM — DIRECTION DE L'ENSEIGNEMENT SUPERIEUR

Philippe MORIGNOT TELECOM Paris 91 E 017

SPECIALITÉ : INFORMATIQUE ET RESEAUX

THESE

TELECOM PARIS

THESE

présentée pour obtenir le titre de Docteur de l'Ecole Nationale Supérieure des Télécommunications

Spécialité : Informatique et Réseaux

Philippe MORIGNOT

CRITERES DE VERITÉ EN PLANIFICATION.

TELECOM Paris 91 E 017

SOUTENUE LE 23 MAI 1991 DEVANT LE JURY COMPOSÉ DE

Henry FARRENY	Président
Alain BONNET Chistophe ROCHE	Examinateurs
Jean-Pierre MULLER Olivier PAILLET	Rapporteurs

ECOLE NATIONALE SUPERIEURE DES TELECOMMUNICATIONS

TELECOM
PARIS

THESE

présentée pour obtenir le titre de Docteur

de l'Ecole Nationale Supérieure
des Télécommunications

Spécialité : Informatique et Réseaux

Philippe MORIGNOT

CRITERES DE VERITÉ EN PLANIFICATION.

SOUTENUE LE 23 MAI 1991 DEVANT LE JURY COMPOSÉ DE

Henry FARRENY	Président
Alain BONNET	Examinateurs
Chistophe ROCHE	
Jean-Pierre MULLER	Rapporteurs
Olivier PAILLET	

ECOLE NATIONALE SUPERIEURE DES TELECOMMUNICATIONS

à Zabou,

Remerciements

Je remercie monsieur le professeur Henry Farreny, de l'IRIT-CNRS, de m'avoir fait l'honneur de présider mon jury. Je remercie également messieurs Jean-Pierre Müller, professeur de l'Université de Neuchâtel, et Olivier Paillet, ingénieur d'Alcatel-Alsthom Recherche, d'avoir accepté de rapporter cette thèse, ainsi que monsieur le professeur Christophe Roche, de l'Université de Savoie, d'avoir participé au jury de thèse.

Je remercie mon directeur de thèse, monsieur le professeur Alain Bonnet, de l'Ecole Nationale Supérieure des Télécommunications, pour son encadrement et ses conseils judicieux pendant ces trois années et demi de recherches.

Je remercie les membres du Département Informatique de Télécom-Paris qui m'ont aidé et plus particulièrement mes camarades de thèse Sophie Midenet, Patrick Constant, Martin Rajman, Lamia Choukair, Frédéric Pigamo, ainsi que l'*Unix wizard* Philippe Dax et ses accolytes Serge Gadret et Georges Salle.

Je remercie monsieur Jean-Michel Truong N'goc, P.D.G. initial de la société Cognitech, de m'avoir accepté en contrat CIFRE, ainsi que monsieur Michel Clerget, directeur de production initial, de m'avoir fourni les moyens nécessaires à mon travail. Que soient remerciés ici les compagnons de route d'alors, Philippe Vernet et surtout Christophe Assémat, valeureux ingénieurs du projet COPLANER, mais aussi Jean-Claude Coriton, Astride Zarka, Pierre Vesoul, Sylvie Fabre, Gilles Ranou, Nadine Ducoudray, François-Marie Lesaffre, Lucia Tavares Farah, Hervé Mahé, Daniel Delmas, ...

Le soutien constant de ma famille a été d'un grand réconfort.

Je serre virilement la pince aux copains d'abord qui, consciemment ou non, m'ont soutenu dans ce travail : Japy, Psy, TouGuy, Yvounet, QuetteBi, Doudou, Punky†, Delphie, PetitGé, Fabi, Christine & Bernard, Pascale & Bel, Papy, CAssemotte, Philippe, Marcus, Jan-Willem, et ceux que j'oublie.

Je remercie enfin Zabou, amie, compagne et épouse, sans qui ce travail, comme tout le reste, n'aurait jamais eu aucun sens.

Ph. M.

Sommaire

Liste des figures

Liste des tables

Introduction Générale

La planification, ou génération de plans d'actions, est probablement une des voies les plus prometteuses à ce jour pour obtenir un comportement autonome et cohérent *dans le temps* d'un système artificiel. Par exemple, pouvoir donner un ordre à un robot mobile, sorte de serviteur humanoïde, et le laisser en détailler et en adapter la réalisation, reste un objectif symbolique important. Outre la connaissance technique propre à l'ordre donné, une source de difficulté pour ce serviteur provient de la coordination entre l'exécution de cet ordre et la réalisation de ses besoins personnels instantanés, ou, si cet ordre comporte plusieurs volets, de la coordination entre les exécutions respectives de chacun de ces sous-ordres.

Si l'on considère la banalité avec laquelle tout un chacun concilie en permanence de tels "ordres", volontés personnelles ou demandes de tiers, on mesure pleinement le saut qualitatif qu'il reste à accomplir, dans ce domaine, pour reproduire chez un être inanimé cette faculté humaine aussi commune que vitale.

Motivation de l'étude

De nombreux systèmes de résolution de problèmes puis de planification ont vu le jour depuis 30 ans pour atteindre cet objectif. Initialement limités à un raisonnement instantané, ils ont démontré par l'absurde l'importance d'une représentation temporelle, pouvant transformer la recherche par *exécution* en une recherche par *simulation*. Ensuite, la nécessité pour un planificateur de manipuler des graphes d'actions *non-linéaires* s'impose depuis 15 ans pour des applications un tant soit peu réalistes.

Que ce soit pour générer un planning traitable ensuite par un PERT (qui, seul, ne fait jamais qu'*évaluer* un planning et non le *générer*), pour superviser la cohérence de l'enchaînement de ses tâches (au moyen d'un formalisme décrivant l'action microscopique de chaque tâche, et plus seulement des inégalités arithmétiques), ou, plus généralement, pour informer la couche de contrôle d'un planificateur du degré d'adéquation du plan en cours de génération, le problème récurrent est de pouvoir déduire l'*état de l'environnement* dans lequel se trouve le planificateur-robot *en un point donné* (un "instant") du graphe d'actions qu'il exécutera.

Ce problème est considéré comme relevant clairement de l'Intelligence Artificielle via les sous-domaines de la représentation des connaissances, du raisonnement temporel et de la résolution de problèmes (souvent agrégés en celui de *planification*).

Choix stratégiques

Bien des travaux ont été consacrés en propre à la planification depuis 20 ans [Fikes & Nilsson 71] [Sacerdoti 77] [Tate 77] [Wilkins 84] [RIP 90]. Ces modèles généraux de planification, *indépendants du domaine*, tentent tous de construire au moins un plan dont l'exécution, à partir d'une situation initiale donnée, réalisera un but convenu d'avance (une situation finale partielle). Ils s'appuient tous sur un formalisme (microscopique) de description de ce qu'effectue une action du plan (variations autour du formalisme STRIPS).

Une des composantes fondamentales de cette planification, le temps, y est souvent représenté comme qualificatif symbolique des faits en terme de *points* [McDermott 82] ou d'*intervalles* [Allen 81] [Allen 84] en relation, améliorant ainsi déjà nettement la représentation numérique canonique implicitement linéaire.

L'approche alternative consiste à déduire l'existence, les caractéristiques et les liens temporels entre tâches à partir d'une expertise, non à partir d'un calcul sur les actions effectuées par chaque tâche [Haren & Neveu 84] [Descottes & Latombe 85]. Ces planificateurs, alors dits à *expertise*, exploitent les connaissances techniques du domaine d'application pour déduire et ordonner ces tâches, et n'ont alors plus besoin d'un des formalismes précédents de description des actions, qui ne ferait au mieux que redémontrer la validité de ces connaissances. Par l'architecture que l'expertise impose, la généralité du mode de résolution est ainsi sacrifiée au profit de l'acuité d'une compétence particulière (une résolution étant une variation autour de la solution-prototype, ou de combinaisons de solutions-prototypes, implicitement définie par l'expertise).

Par delà les qualités respectives de ces planificateurs purs, subsistent plusieurs écueils communs. D'abord, ils fonctionnent tous sur prédicats formés de constantes et ne font que tolérer temporairement des *variables* ; aussi, le moindre exemple en nécessitant, ils introduisent des objets partiellement contraints qui sont rapidement éliminés, car mal inclus dans le raisonnement et interdisant souvent la déduction d'information temporelle en terme de points ou d'intervalles. Ensuite, les types de conflits entre branches en parallèle ayant été initialement élucidés à la main sur des exemples, rien n'indique que tout conflit futur se rangera sagement dans cette typologie empirique (en ce sens, le plan-solution fourni n'est pas *correct* dans le cas général). Enfin, lors de la résolution d'un conflit, on ne sait séparer ce qui relève de l'heuristique, experte dans le domaine de l'exemple mais a priori indifférente ailleurs, de ce qui relève de l'algorithme, générateur en soi de solution effectivement indépendante du domaine ; l'introduction d'une expertise, au sens du paragraphe précédent, n'en est que plus délicate.

La raison sous-jacente à ces imprécisions est qu'aucun de ces planificateurs n'explicite *en fonction de quoi, de quel critère* un "conflit" est dit "résolu" ou non (ces mots n'ayant finalement de sens que dans le contexte de l'exemple). Dire qu'une situation est conflictuelle si et seulement si elle est contradictoire n'a de sens que si le calcul de ce qui existe (vrai ou faux) en un point donné du graphe d'actions a été posé une fois pour toutes (résolution personnelle du problème du cadre), malgré l'éventuelle présence de variables ou l'éventuelle non-linéarité du graphe d'actions.

C'est l'étude de cette interprétation en terme de *critère* déterminant la valeur de vérité d'un terme en un point du graphe d'actions, ou *critère de vérité*, et de ses conséquences pour les planificateurs purs comme pour les planificateurs à expertise, que nous nous proposons d'effectuer dans ce mémoire.

Objectifs

Pour échapper à ce dilemne *macroscopique expert* (pratique, analytique a posteriori) vs. *microscopique calculatoire* (théorique, synthétique a priori), nous proposons d'approcher la planification avec et sans expertise sous l'angle du *critère de vérité*, dont nous présentons quelques exemplaires.

Pour cela, nous présentons un planificateur pur complet respectant un critère à la fois dans la déduction de valeurs de termes que dans l'induction de valeur, isolant ainsi les amendements de plan purement heuristiques (convergence du plan vers un plan-solution), formant les points d'entrée potentiels d'une expertise, des amendements imposés par le critère. Nous présentons aussi un planificateur à expertise (pour des chantiers de bâtiments) dont l'architecture, structurant cette expertise, peut également s'interpréter en terme de critère de vérité.

Conscients de la difficulté, nous avons délibérément opté pour un formalisme simple de description des actions, un formalisme plus élaboré rendant difficile la définition du critère et/ou rédhibitoire les performances des algorithmes l'implémentant.

Enfin, nous nous sommes posés deux contraintes pour le planificateur pur :

1. se situer en *ordre 1*, pour pouvoir manipuler des objets partiellement instanciés sous forme de contraintes sur les variables les représentant ;
2. raisonner sur un plan d'actions *non-linéaire*, pour profiter des influences bénéfiques mutuelles entre branches en parallèle.

Organisation du mémoire

Ce mémoire est organisé en trois parties, respectivement constituées de quatre, trois et deux chapitres.

La première partie dégage progressivement la notion de *critère de vérité* comme cœur de tout raisonnement sur un graphe d'actions, à partir d'une analyse de l'existant : une fois fixé le vocabulaire dans le premier chapitre, les deuxième et troisième chapitres s'appuient sur une analyse bibliographique pour identifier les concepts liés à la représentation du temps et à la planification. Le quatrième chapitre rapproche deux théories homologues (en ordre 0 et en ordre 1) introduisant un critère de vérité.

La deuxième partie est la description du planificateur indépendant du domaine (ou *pur*) que nous proposons : le cinquième chapitre décrit les représentations des objets, du temps et du calcul, que nous avons adoptées, ainsi que les algorithmes permettant la *déduction* d'informations d'un graphe d'actions. Le sixième chapitre décrit les modifications possibles (compatibles avec le critère) de ce graphe et présente les six heuristiques de contrôle implémentant le choix de leur application. Enfin, le septième chapitre commente et critique le traitement d'exemples.

La dernière partie revient à la planification avec expertise à travers l'étude d'un problème concret, la génération de planning pour la construction de bâtiments : après avoir identifié les caractéristiques de cette expertise dans le huitième chapitre, le neuvième chapitre décrit

le système (alors *expert*) implémenté. Cette réalisation nous permet de mettre en rapport la planification *avec* et *sans* expertise, et de montrer qu'une interprétation en terme de critère de vérité est avantageuse non seulement pour un planificateur indépendant du domaine mais aussi pour un planificateur expert.

Partie I

L'approche de l'I.A. sur la planification

Introduction

Cette partie propose un parcours bibliographique concentrique sur la Planification en Intelligence Artificielle se focalisant sur la notion de critère de vérité.

Nous définirons d'abord le vocabulaire que nous utiliserons dans ce mémoire (chapitre 1), ce qui nous permettra de structurer les concepts fondamentaux utilisés et de délimiter les frontières du domaine de la planification vis-à-vis des autres domaines de l'I.A.

Le Temps ayant été dégagé comme un des concepts fondamentaux en planification, nous étudierons ensuite sa représentation dans des domaines tels que la philosophie, la psychologie, les mathématiques et l'informatique.

Nous présenterons ensuite les planificateurs existants de façon à pouvoir rattacher notre approche à la planification traditionnelle (chapitre 3). Ce parcours se centrera finalement sur une théorie de la planification utilisant explicitement un critère de vérité, se déclinant en ordre 0 et surtout en ordre 1 (chapitre 4).

Chapitre 1

Fondements

1.1 Structuration de la planification

La polysémie usuelle du mot "planification" provient des énoncés suivants : *planifier, c'est évaluer un planning ; planifier, c'est dire qui va faire quoi ; planifier, c'est construire un plan.* Le premier sens renvoie à l'interprétation numérique du temps (cf. 2.2), le deuxième, à l'ordonnancement (cf. 3.3.3), le troisième, à la génération de plan d'actions. Selon le *Petit Robert*, planifier, c'est *décider la façon d'organiser des activités* (cf. l'annexe B) ; le concept de *planification* englobe en fait les trois concepts d'*objet*, de *temps* et de *calcul*.

Objet, temps et calcul Par *objet* est compris toute entité physique, réalité sensible, tactile, manipulable. Sa modélisation est maintenant connue, grâce aux *frames* par exemple [Minsky 75] ou encore aux Langages Orientés Objets [Cointe 87]. Un modèle d'objet peut exprimer statiquement un contexte, un réseau d'objets en relation, ou un réseau sémantique (même *augmenté*), mais ne peut guère exprimer leur évolution.

Nous ne nous risquerons pas à tenter de définir le *temps* (cf. 2.1) et nous nous bornerons à nous référer à sa caractérisation issue du sens commun : le temps est ce qui permet une évolution, il est représenté formellement par les logiques temporelles et réifiées (cf. chapitre 2).

Par *calcul* est compris le moyen de décrire une transformation indépendamment de l'objet sur lequel elle s'applique et indépendamment de sa correspondance avec notre temps réel. Il est modélisé mathématiquement par exemple par le lambda-calcul, la récursivité ou la machine idéale de Turing (cf. 2.3.1) ; en I.A., ce calcul peut être représenté au moyen de règles de production [Davis 77] [Hayes-Roth et coll. 83].

Associations En isolant objet, temps et calcul, la notion d'*action* (transformation d'objets dans le temps) ne peut guère être exprimée que de façon extérieure, macroscopique : un objet (son nom) et un couple de dates (son début et sa durée). Leur jonction permet de retrouver des concepts classiques (voir figure 1.1).

L'association *objet/temps* fait émerger l'*événement*, considéré comme la conjonction d'un *objet* et d'une *date* (cf. la modélisation d'une action macroscopique en 2.2). En associant *objet* et *calcul*, on peut représenter une singularité spatiale (*changement atemporel*), i-e la transformation d'un objet indépendamment du temps : évaluation d'une fonction avec ses arguments, ou transmission (et l'évaluation) d'une méthode à un objet (au sens des Langages Objets), ...De l'association *temps/calcul* naît la notion de *processus* (cf. 2.6) ou vieillissement : calcul qui peut s'effectuer dans le temps mais qui n'a pas d'objet sur lequel s'appliquer (processus-réflexe). Enfin, l'association globale *objet/temps/calcul* permet de représenter un événement avec le calcul qu'il effectue, ou une singularité spatiale temporalisée ou encore un vieillissement et l'objet sur lequel il s'applique : c'est le concept d'*action*, description du contenu d'un calcul transformant des objets dans le temps.

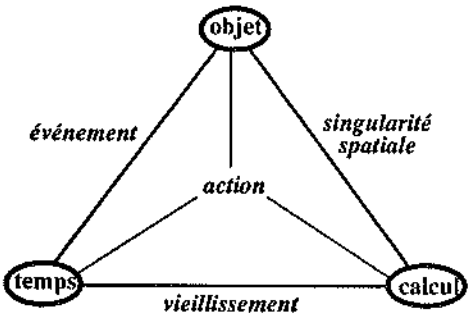


Figure 1.1: Objet, temps et calcul pour la planification

Répartition de la complexité Confronté à un problème comprenant des objets se transformant dans le temps, le premier réflexe consiste à situer ce problème dans ce référentiel : doit-on prendre en compte le temps ? ai-je besoin d'une représentation poussée des objets manipulés ? comment dois-je transformer ces objets ? L'expression d'un problème selon ces trois axes tente de mesurer sa complexité et de la répartir dans des zones où son traitement est connu.

Une façon usuelle d'échapper à la complexité d'un problème de planification consiste à éviter de modéliser une ou plusieurs de ces trois dimensions : les outils de gestion de projets du commerce, par exemple, se basent sur une représentation numérique du temps, modélisent à peine le *calcul* (réduit au nom de l'action) et les *objets* (seuls les plus importants sont vus comme des ressources qu'une tâche consomme plus ou moins ; ce qui est effectivement changé par la tâche n'est pas du tout représenté ...).

Aux limites du modèle, un *objet* peut représenter non seulement des objets réels, mais aussi les notions plus abstraites de temps et de calcul, en interprétant l'application du motif objet/temps/calcul de façon *réflexive*[1] : lorsque l'objet transformé est un *calcul*, le calcul sur ce calcul correspond aux transformations effectuées sur une action, règles d'inférences de formules d'un système logique ou dérivation d'un schéma d'actions en une action. Lorsque l'objet transformé est le temps, le calcul correspond au raisonnement temporel (cf. chapitre 2). Dans les deux cas, le temps correspond au temps réel passé par un humain ou une machine pour réaliser le-dit calcul sur la représentation.

[1] La mise en évidence de la réflexivité d'un modèle démontre sa capacité à *passer au niveau méta* (cf. [Cointe 87] pour les langages objets et [Ferber 89] pour les langages réflexifs en général).

1.2 La Planification en I.A.

1.2.1 Formulation

Un *problème de planification* se pose dans les termes suivants (cf. figure 1.2) : étant donnés

- une description de la *situation initiale*,
- une description de la *situation finale*,
- l'exhaustivité des *schémas d'actions* disponibles,

comment obtenir un ensemble d'actions[2] appelé plan ou *planning* (*to plan* : prévoir), tel que l'exécution transforme la situation initiale en la situation finale (formulation identique à celle de GPS [Newell & Simon 63]) ? Le plan généré est censé pouvoir être ensuite exécuté par un ou plusieurs agents : robot mobile (auquel on identifie le planificateur informatique), système de fabrication, corps de métiers, ...

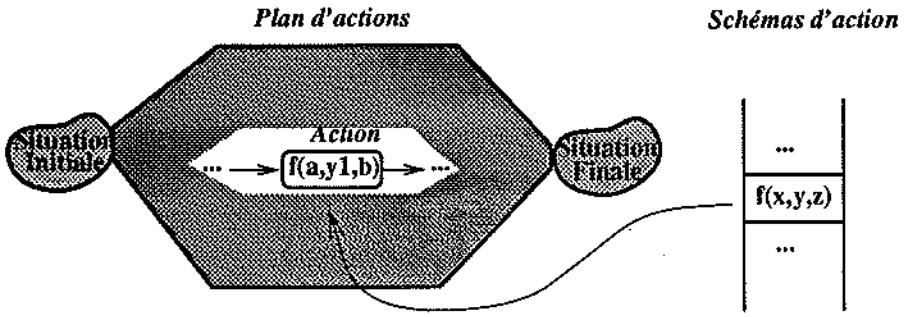


Figure 1.2: Définition d'un problème de planification

Le monde dans lequel évolue le planificateur (le domaine sémantique des schémas d'actions) est appelé *domaine d'application*. Une *situation* (ou *état*) est définie, à un instant donné, par *l'ensemble des faits qui ont une valeur connue (vraie ou fausse)* ; un fait est généralement un prédicat.

Un *schéma d'actions* définit la transformation d'une situation possible en une autre ; une action est une instance d'un schéma d'actions, l'application d'un schéma d'actions à une situation particulière. Pour pouvoir définir la transformation la plus générale possible, un schéma d'actions est souvent basé sur la logique des prédicats d'ordre 1 ; une action, par contre, peut ou non contenir des variables (toutes les variables du schéma d'actions générateur ne sont pas a priori instanciées).

[2] On trouve dans la littérature des termes de sens très proche de celui de "action" :

Opérateur Transformation du calcul des situations.

Règle Structure de connaissance définissant une inférence élémentaire. Terme plutôt utilisé dans le domaine des systèmes experts.

Tâche Travail déterminé qu'on doit exécuter. Terme utilisé dans les méthodes d'évaluation numérique en Recherche Opérationnelle (C.P.M., PERT — *Program Evaluation and Review Technique*).

La description de la situation initiale est explicitement la liste des prédicats connus dans cette situation (c'est effectivement une *situation*). La description de la situation finale n'est que partielle : les effets de bords induits par l'application des actions ne peuvent pas toujours être tous connus au moment où le problème est posé, ce sont précisément les degrés de liberté du planificateur. Cette description de la situation finale est composée de plusieurs prédicats, sinon on parle de *résolution de problèmes* plutôt que de *planification*.

L'ensemble des actions constituant le plan est ordonné au moins par la relation d'ordre *"est-exécuté-avant"* (lien de précédence). Dans le cas d'un ordre total, on parle de *planification linéaire*, dans celui d'un ordre partiel, de *planification non linéaire*. Comme on peut toujours identifier une action initiale[3] (au besoin, en rajoutant une action fictive dont la situation sortante est la situation initiale) et une action finale (id. avec la situation sortante), un plan est représenté par un graphe acyclique orienté à source et puits unique (réduit à un *chemin* en planification linéaire).

Un plan est une solution (ou une *famille* de solutions) d'un problème de planification.

1.2.2 Problématique

Degrés de liberté Un planificateur linéaire sans variable cherche un plan en choisissant les schémas d'actions (sans variable) applicables à partir de la situation initiale (chaînage avant) ou depuis la situation finale (chaînage arrière). Un plan ainsi construit représente *une seule* solution du problème de planification posé. Ce planificateur doit chercher autant d'autres chemins, dans l'arbre de décision implicite formé par ces schémas d'actions d'ordre 0, que de plans désirés.

Tous ces parcours peuvent être évités en exploitant les degrés de liberté constructibles selon nos trois axes objet / temps / calcul :

- objet : manipuler plusieurs objets en une fois s'effectue au moyen de variables (ordre 1) pouvant s'instancier a priori en plusieurs constantes (objets) ;
- temps : manipuler plusieurs liens de précédence *"est-exécuté-avant"* en une fois s'effectue en interprétant l'incomparabilité de deux actions a et a' d'un ordre partiel comme "a est-exécutée-avant a', ou bien a' est-exécutée avant a" ;
- calcul : une description suffisamment simple du calcul n'oblige pas le planificateur à exécuter l'action correspondante, pour connaître l'effet de ce calcul ; cette simplicité permet au planificateur de raisonner directement sur l'expression de ce calcul (simulation).

Le dernier point permet de représenter le plan non comme une trace d'exécution mais comme un graphe d'actions. Le deuxième point montre qu'un plan non-linéaire de n_a actions peut représenter jusqu'à $n_a!$ plans linéaires ; le premier point montre qu'un plan avec n_v variables et n_c constantes peut représenter jusqu'à $n_c^{n_v}$ plans d'ordre 0.

[3] Un graphe orienté sans circuit possède au moins un sommet appelé source (resp. puits) qui n'a pas de prédécesseur (resp. successeur) [Gondran & Minoux 79].

Esquisse d'une taxonomie La structuration ternaire proposée suggère d'articuler certains sous-domaines de l'I.A de la façon suivante :

- un seul but : résolution de problèmes ;
- plusieurs buts :
  - exécuteur : systèmes experts (moteurs d'inférences) ;
  - simulateur :
    - linéaire : planification linéaire ;
    - non-linéaire :
      - sans variable : planification non linéaire d'ordre 0 ;
      - avec variables : planification non linéaire d'ordre 1 ;

La distinction *exécuteur / simulateur* provient des deux points de vue sur le temps (cf. 2.1 : temps que l'on parcourt du passé vers le futur (exécuteur), ou temps-environnement dans lequel on distingue des instants, des durées (simulateur). Dans la figure 1.3, les exécuteurs progressent d'état en état (passé/futur), alors que les simulateurs considèrent la future exécution dans son ensemble.

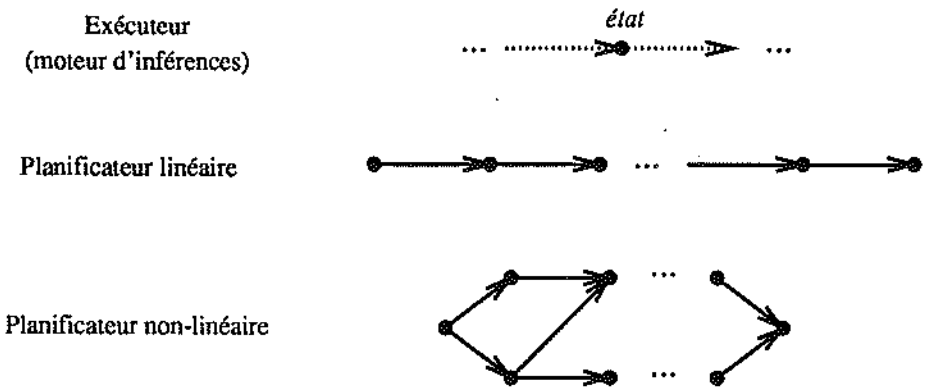


Figure 1.3: Passage d'un exécuteur à un planificateur non-linéaire

Notions connexes Les notions suivantes et leur conjugaison dressent une liste de questions ouvertes :

- Linéarité : chercher une séquence d'actions solution d'un problème peut s'effectuer en ne raisonnant à un instant donné que sur un seul chemin, celui qui mène à l'état courant, l'algorithme de contrôle guidant ce planificateur-exécuteur. Obtenir toutes les solutions autrement qu'en les exécutant exhaustivement nécessite de pouvoir représenter ces alternatives et leurs mises en facteur. Cet accroissement d'expressivité par la non-linéarité d'un plan se paie par la difficulté de gestion des interactions entre branches concurrentes.
- Hiérarchie : confronter d'emblée le planificateur à tous les détails du problème augmente sa combinatoire. Le tri des données par hiérarchisation permet à un instant donné de n'en

considérer qu'une partie, de combinatoire moins forte et quand même cohérente[4]. Cette hiérarchisation se paie par une difficulté de gestion des interactions entre ces niveaux hiérarchiques[5].

- Métaplanification : le choix de la prochaine action à entreprendre est du ressort de l'algorithme de contrôle, souvent simple, aveugle et syntaxique. En considérant un planificateur comme un agent disposant d'actions connues pour modifier un état (un graphe d'actions), le problème du contrôle peut être considéré comme ... un problème de planification du second ordre ! Pour éviter un empilement infini de planificateurs (méta-méta-planificateur, ...), il faut soit trouver un point fixe (faire boucler la chaîne des planificateurs méta), soit contrôler d'autorité un méta^n-planificateur par un algorithme simple, aveugle et syntaxique.

- Réactivité : la modélisation d'un problème réel en un micro-monde résoluble par un planificateur se paie toujours par une perte d'information. Aussi certains planificateurs sont munis de capteurs pour suivre ces phénomènes réels non modélisés, rectifiant constamment leur monde simulé par le monde réel. Leur capacité à réagir à une incohérence soudaine de leur modèle, résultant d'une rectification, mesure leur réactivité.

- Temps : beaucoup de planificateurs ne considèrent le temps que comme l'ordre partiel "est-après" entre des actions de durée infinitésimale. Inversement, une gestion plus complète associée à la description des modifications continues et internes effectuées par ces actions se paie par une diminution de l'attention portée par le planificateur sur les problèmes purement liés à la gestion du graphe d'actions.

[4] Comme cela a été remarqué ([Stefik 81] parmi d'autres), cette technique de (dé-)composition hiérarchique en résolution de problème est une lecture informatique des deuxièmes et troisièmes préceptes du *Discours de la Méthode* de René Descartes (1637).

[5] Il suffit d'observer le fonctionnement de n'importe quelle administration pour s'en convaincre.

Chapitre 2

Représentation du temps

Si l'on en croit le *Petit Robert*, le temps serait ce *"milieu indéfini où paraissent se dérouler irréversiblement les existences dans leur changement, les événements et les phénomènes dans leur succession"*. Curieux concept que celui défini d'emblée par une entité *indéfinie* et dont les caractéristiques ne sont, elles non plus, pas certaines ! La gêne évidente du *Petit Robert* montre bien la difficulté qu'il y a à donner une *définition* du temps, alors que son utilisation semble si immédiate à chacun.

Cette contradiction humaine entre la grande facilité de perception et l'extrême difficulté de formalisation a stimulé l'activité des plus grands penseurs, dont les conceptions ont été le support de toute recherche scientifique depuis des siècles. Pouvant difficilement ignorer ces "hénaurmes" tentatives de formalisation, nous retraçons extrêmement brièvement l'évolution de la conception du temps chez ces philosophes. Pour comprendre l'origine commune des recherches sur le temps en I.A., nous indiquons ensuite l'aspect basé sur l'expérimentation quotidienne de la notion de temps, tel qu'il est tracé par les deux courants de la psychologie, le behaviourisme et le cognitivisme.

Devant bien concilier formalisation et perception, l'informatique, sans chercher non plus à définir le temps, le fait utiliser par la machine en jouant sur l'ambiguïté du statut de l'objet physique "machine" : acteur plongé dans le temps (intégration du temps de la machine dans notre temps humain), ce qui conduit à l'informatique *temps réel*, ou bien environnement porteur d'un micro-monde clos (séparation du temps auquel est soumis la machine de celui du phénomène observé, comportement dit autistique), ce qui conduit à l'informatique *de simulation*.

Que le temps du micro-monde soit connecté ou non au temps réel auquel est soumis la machine, sa modélisation influe directement sur le type de planification possible. Nous présentons d'abord la modélisation numérique du temps, intuitivement la plus immédiate (chacun croyant ce que lui indique sa montre). La nécessité de raisonner sur ce temps a conduit aux modélisations symboliques : les logiques classiques s'y prêtant mal, les logiques modales et temporelles introduisent très proprement le temps en mathématique formelle. Avec les logiques réifiées, l'intelligence artificielle cherche un moyen terme entre un raisonnement formel pur et une utilisation pratique immédiate, à partir d'une modélisation des points, due à Mc Dermott, et surtout d'une modélisation des intervalles, due à Allen [Allen 81].

2.1 Fondements

Pendant une expérience, le temps du modèle est mis en correspondance avec le temps observé et ressenti par l'expérimentateur. C'est l'observateur qui donne une signification au temps du mobile, aussi nous présentons l'aspect proprement humain du sens et de la structure du temps, dans le domaine de la philosophie et de la psychologie.

2.1.1 Temps et philosophie

Nombre de philosophes, historiens, physiciens et chercheurs de toutes disciplines se sont attachés à vouloir expliquer et traduire le temps en schémas conceptuels. Nous tentons ici d'en esquisser les grandes tendances, réparties entre l'objectivité et la subjectivité du temps [Mosnier 90], en espérant ne pas trop défigurer la pensée de leur auteur.

Une théorisation embryonnaire du temps objectif provient de Zénon d'Elée, qui tente de nier le mouvement, donc le temps, en présentant ses paradoxes, dont le fameux *Achille et la Tortue*[1]. Pour Aristote, la conception du temps dérive directement de celle de l'espace par le biais du mouvement, dont le temps est inséparable. Pour résoudre les paradoxes de Zénon d'Elée, qui nécessitent de trouver dans le continu des grandeurs divisibles à l'infini, Aristote introduit un infiniment petit dans les mathématiques antiques (uniquement discrètes), l'*instant*, qui est exclusivement réservé à la mesure de ce temps. Cette conception a régné depuis le IVe siècle av.JC (Aristote) jusqu'au XVIIIe siècle en Philosophie (Kant) et jusqu'au XIXe siècle en Physique (Einstein). Par exemple, le second principe de Thermodynamique a confirmé l'aspect destructeur du temps (le temps aristotélicien est lié au mouvement qui, par définition, détruit son état initial) ; ou encore l'aspect non dénombrable de l'ensemble des instants (isomorphe à $\mathcal{R}$) prolonge le continu et le divisible à l'infini du temps aristotélicien.

La temporalité est la constitution subjective, proprement humaine, du temps : notre temps vécu tel qu'il se constitue dans notre conscience, par opposition au temps spatialisé (objectif) d'Aristote. Pour Bergson, cette temporalité est la durée, qui contient l'aspect interne de nos états de conscience, notre histoire. Notre conscience s'accroit sans cesse sans séparer ses nouvelles acquisitions, ce qui nous suppose une mémoire continue et pouvant comprendre la succession.

La phénoménologie cherche à dépasser cette notion de temporalité (le temps constitué *dans* et *pour* notre conscience) pour montrer que la temporalité est la conscience même. L'opposition n'est plus à faire entre un temps objectif et un temps subjectif, mais entre un temps *constituant* la conscience (Husserl) et un temps *constitué* par la conscience (Heidegger : l'être *est* temporalité). La conception commune du temps (infini à partir du présent, successif et irréversible) est bien sûr légitime, mais ne reste qu'une représentation dérivée de cette temporalité phénoménologique. Merleau-Ponty pense le temps comme quelque chose de fluide et en constitution, catalysé à la fois par un présent transcendant (en son aspect fugitif, sans coïncidence possible, en constant devenir) et par l'être qui existe dans ce temps, et qui devient par là-même l'agent de cette constitution.

[1] La Tortue peut toujours fuir pendant qu'Achille effectue ce qu'il croyait être sa dernière enjambée sur la position de la Tortue : Achille ne devrait alors jamais la rattraper (cf. [Hofstädter 86] pour une superbe variation sur ce thème).

Kant s'oppose d'abord à la conception objective du temps d'Aristote (selon laquelle le temps existe en soi) en arguant que (1) si ce temps était inhérent aux objets réels, alors il ne pourrait être donné avant ces objets comme leur condition de possibilité, et (2) si, par contre, il existait indépendemment de ces objets, notre perception humaine ne nous permettrait pas de le connaitre. Ensuite, le temps en tant que concept relève de l'intellectuel, non du sensible, et ne peut donc pas non plus dériver de l'expérience. Il doit donc nous être donné a priori, cablé en nous en tant que *catégorie de l'entendement*[2]. Plus généralement, la structuration kantienne délimite le champs du connaissable : toute connaissance est subjective, en ce qu'elle dépend des catégories du sujet connaissant, ou au mieux localement objective, pour un groupe de sujets connaissants possédant des catégories identiques, mais jamais objective, en tant que connaissance en soi du point de vue de l'objet. Condition de perception et non objet soumis à la perception, le temps ne peut nous être figuré, lacune que nous comblons par l'analogie de la droite aristotélicienne continue et infinie. Enfin, Kant démontre que la proposition "le monde est fini dans le temps et dans l'espace" est à la fois vraie et fausse (première antinomie de la raison).

De ce survol philosophique, nous nous bornerons à relever que le temps irréversible du devenir (passé / présent / futur) trouve une certaine interprétation dans la planification / exécution linéaire, alors que le temps considéré comme milieu immobile baignant tous les changements (simultanéité / succession / durée) possède quelques résonances avec la planification non linéaire. Ceci confirmerait la supériorité de la non linéarité sur la linéarité en planification pour représenter le temps subjectif et, par suite, les processus mentaux rendant possible toute connaissance, tels que ces philosophes les ont élucidés.

2.1.2 Temps et psychologie

Comme précédemment, nous tentons maintenant de présenter les principales approches situant le temps dans le domaine de la psychologie, sans trop déformer la pensée de leur auteur (présentation dans [Bergadaã 89])

Deux modes de pensée s'affrontent généralement en psychologie et en particulier sur la définition de la structure du temps : le behaviourisme et le cognitivisme.

Pour les behaviouristes (approche *holiste*[3] dirait-on en I.A.), l'individu ne peut agir qu'en réaction à son environnement. Toute action est la réponse à un état de manque, satisfaisant un besoin biologique créé par des stimuli externes. L'impulsion vient exclusivement du milieu extérieur, une action n'est jamais la cristallisation d'une idée.

Pour les cognitivistes (approche *réductionniste*[4] dirait-on èn I.A.), le choix d'une action par un individu est basé non sur son environnement mais sur sa structure cognitive. Des stimuli *internes*, (buts, idées, projets) sont à la base de l'action humaine. Les stimuli externes n'incitent

[2]Dans la pensée kantienne, une catégorie de l'entendement est ce qui permet d'effectuer la synthèse (transcendantale) de l'intuition sensible (qui reçoit le chaos des impressions successives) et des concepts de l'entendement (qui reconnait dans ces impressions des phénomènes ordonnés) pour obtenir une connaissance.

[3]Le *holisme*, ou *Gestalt*, est l'approche selon laquelle le tout ne peut pas être connu par la seule observation des parties (*"le tout est plus que la somme des parties"*) : il y a *émergence* de phénomènes inattendus, qui ne semblent pas provenir a priori des parties du système complexe observé. Ces phénomènes collectifs s'expliquent cependant en prenant en compte l'*interaction* entre ces parties [Minsky 88].

[4]Le *réductionnisme* est l'approche selon laquelle on peut connaître le tout par la seule observation des parties.

pas directement à une action : ils ne sont que des possibilités de choix. Le temps y est interne, subjectif ; il est ponctué d'événements, de souvenirs qui rendent l'individu acteur conscient (c'est le *volontarisme cognitique* s'opposant au *déterminisme behaviouriste*).

La *perception* de la durée d'une activité chez un sujet fait également l'objet de deux interprétations : l'interprétation behaviouriste postule l'existence d'un temps externe sur lequel l'individu se modèle. L'individu serait, paraît-il, doté d'un mécanisme d'évaluation du temps, lui permettant de réagir à ces stimuli temporels sous forme d'action, de décision. L'interprétation cognitive postule également l'existence d'un temps externe, mais là ce sont les émotions, les sentiments, la subjectivité de l'individu qui déterminent la perception de sa durée (rejoignant ainsi l'analyse bergsonienne). Ceci explique le décalage entre le temps externe (objectif) et le temps interne (subjectif) : l'exemple courant est l'impression qu'un même intervalle de temps est plus grand en situation d'attente que lorsqu'on effectue une activité que l'on aime.

Les analyses réalisées par des psychologues montrent que les individus se représentent le temps comme une droite orientée du passé vers l'avenir en passant par le présent, sur laquelle sont placés les événements, souvenirs ou prévisions. Les mesures effectuées portent sur : l'orientation temporelle (préférence à visualiser le passé, le présent ou le futur), la longueur de l'horizon temporel futur (corrélation entre la longueur de cet horizon et des variables socio-démographiques — age, sexe, revenu, ... — ou des facteurs de personnalité — identité, relations parents-enfants, ...), le réalisme des projets (divergence entre le contexte extérieur et le monde abstrait que se crée l'individu et dans lequel il place ses projets), la cohérence de l'individu à ordonner ses actions (influence de l'organisation des projets dans l'horizon temporel futur sur la motivation présente), la motivation liée au projet (corrélation entre la motivation et la distance subjective séparant le sujet de la réalisation de son projet).

2.1.3 Temps et intelligence artificielle

L'introduction du temps en Intelligence Artificielle date de 1957, année où Mc Carthy définit le *calcul des situations* (*"situation calculus"*) et Prior, la logique temporelle [Prior 57]. Plus que le temps lui-même, c'est l'évolution des phénomènes que Mc Carthy propose de modéliser : un événement particulier opère une transition depuis une description complète du monde vers une autre description complète du monde. Cette approche a l'inconvénient de conduire au *problème du cadre* (*Frame Problem*) : pour décrire une transformation, il faut non seulement décrire ce qui a changé mais aussi ce qui n'a pas changé[5], problème d'épistémologie [McCarthy & Hayes 69] se traduisant par une complexité algorithmique et qui fait toujours l'objet de recherche : sont distingués le *problème de la qualification*[6], [Ginsberg & Smith 88b] le *problème de la ramification*[7]

[5] Une façon agréable d'imaginer ce problème est de se représenter le travail du dessinateur dans un studio de dessins animés [Nilsson 80] : pour dessiner la prochaine image d'un cartoon, il faut dessiner le mouvement suivant du personnage principal (ce qui a changé) et le décor dans lequel évolue le-dit personnage (ce qui n'a pas changé). Ces deux fonctions sont d'ailleurs dissociées dans un studio, la patte du concepteur étant plus valorisée dans le dessin du personnage central.

[6] Qui est que le nombre de conditions effectivement préalables à la réalisation d'une action est beaucoup trop grand dans la réalité. Pour *"démarrer une voiture"*, il faut bien sûr *"tourner la clé de contact"* (seule précondition a priori), mais aussi s'assurer de la véracité d'autres préconditions : *"Par exemple, il doit y avoir de l'essence dans le réservoir, la batterie doit être branchée, le câblage doit être correct, et il n'y a pas de pomme-de-terre dans le tuyau d'échappement"* ! [Ginsberg & Smith 88b, p. 312].

[7] Qui est que le nombre de conclusions à lister explicitement dans la formalisation d'une action est beaucoup trop grand dans la réalité : si j'ai déplacé un carton de a à b, alors bien sûr le carton est en b, b n'est plus libre,

[Ginsberg & Smith 88a] et le *problème de la prédiction étendue*[8] [Shoham & McDermott 88].

Prior a introduit un raisonnement formel complètement centré sur le temps, dérivant des logiques modales : l'introduction des modalités P (passé) et F (futur) permet de caractériser temporellement des formules issues de la logique des propositions ; la construction de modèles passe par la définition d'un graphe de mondes possibles, qui n'est pas sans rappeler le graphe des états de Mc Carthy. Les extensions de la logique temporelle de base sont utilisées en informatique théorique (preuve de programmes, ...) et ont été appliquées à l'étude du comportement des programmes parallèles [Pnuelli 71].

Une autre approche symbolique, plus applicable en I.A. que le formalisme pur de Prior ou les logiques standardes brutes, est constituée par les logiques réifiées, qui formalisent certains types temporels (événements, propriété, processus, ...) permettant de qualifier le type atemporel (proposition ou prédicat) dans lequel se pose pratiquement le problème. Les logiques réifiées les plus célèbres sont la logique des intervalles due à Allen [Allen 84], qui a donné lieu à des applications pratiques, au moins au niveau du formalisme, et la logique, duale, des instants de Mc Dermott [McDermott 82].

Enfin, les systèmes d'I.A. en raisonnement temporel utilisent parfois en bout de chaîne la modélisation numérique (canonique) du temps : l'ensemble des instants est identifié à la droite des réels $\Re$ et des algorithmes de propagations issus de la Recherche Opérationnelle sont utilisés pour évaluer les fenêtres temporelles des événements manipulés par ailleurs.

2.2 Temps numérique

Les systèmes numériques empruntent leurs techniques à la Recherche Opérationnelle (*Critical Path Method*, PERT, *méthode des potentiels*, ... [Carlier & Chrétienne 82]). Pour eux, la seule vérité importante est d'ordre comptable : tout l'aspect sémantique d'un événement est écrasé sur l'axe des nombres réels $\Re$, représentant le temps, et renaït sous la forme unique de "date", mesure unidimensionnelle finale du fragment d'espace sémantique initial.

A partir d'un ensemble de contraintes, ce type de système essaie d'attribuer une date à chaque événement. Si, pour chaque événement, l'ensemble des dates possibles est non vide, cet ensemble de contraintes est dit *consistant* et il n'y a plus qu'à choisir une date pour chaque événement. Aussi ces systèmes manipulent des *fenêtres temporelles*, représentant ce degré de liberté sur le choix d'une date.

De tels modules d'évaluation sont généralement intégrés en bout de chaîne dans des systèmes d'I.A. à part entière, qui traitent surtout la partie sémantique des événements et des relations temporelles[9].

---

a est devenu libre, mais aussi les objets dans le carton ont été déplacés, leurs positions relatives ont changé, ...

[8] Qui consiste à savoir jusqu'à quand les conclusions faites sont valides (une version plus générale de la *persistance*) : si une boule de billard roule, on peut déduire qu'elle roulera encore ...jusqu'à ce qu'elle rencontre une autre boule ou une bande.

[9] Dans le cas contraire, ce type de système conserve quand même un intérêt industriel certain (il suffit de considérer le nombre de systèmes de PERT ou de "gestion de projet" disponibles, de celui sur Macintosh pour quelques centaines de francs aux monstres sur Cray-x pour quelques milliers de dollars ...).

2.2.1 Propagation de dates sous contraintes linéaires

Pratiquement, un événement i (alors appelé *tâche*) est modélisé par t_i, d_i et f_i, (resp.) sa date de début, sa durée, sa date de fin. Une contrainte entre deux tâches i et j est modélisée par une égalité ou inégalité arithmétique entre les extrémités de ces tâches. Une telle contrainte est un triplet (α, β, γ) qui signifie (voir figure 2.1) : $e_i + \alpha \times d_i + \beta \preceq e_j + \gamma \times d_j$, où l'extrémité e_k d'une tâche k vaut t_k ou f_k, $\preceq$ vaut $=$, $\leq$ ou $<$, α (resp. γ) est une constante de [0,1[qui est nulle lorsque $e_i = f_i$ (resp. $e_j = f_j$), β est une constante quelconque.

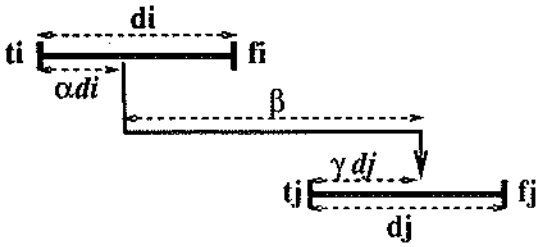


Figure 2.1: Lien type d'une représentation numérique

Les durées d_i étant fixées, l'évaluation des dates de début au plus tôt t_i et de fin au plus tard f_i nécessite d'abord la réécriture de l'ensemble des contraintes pour éliminer celles du type (t_i, f_j) (réécrit en $(t_i, t_j + d_j)$) et (f_i, t_j) (réécrit en $(f_i, f_j - d_j)$). Les dates de début au plus tôt sont propagées par l'algorithme PROPAGER-DÉBUT :

Procédure PROPAGER-DÉBUT (i)
Pour toute contrainte $c_{i,j}$ entre une date de début t_i et une date de début t_j,
- identifier $c_{i,j}$ comme une égalité
- calculer la nouvelle valeur t'_j de t_j :
$$t'_j = f_{c_{i,j}}(t_i) = t_i + \alpha d_i + \beta - \gamma d_j$$
- **si** $t'_j \leq t_j$, **alors** SORTIE (pour cet appel uniquement)
sinon, $t_j \leftarrow t'_j$ et PROPAGER-DEBUT(j)

Les variables t_i et f_i ayant été initialisées à la date de début au plus tôt du graphe, cet algorithme doit être lancé sur les tâches les plus à gauche (que l'on peut repérer initialement en $O(n)$, les contraintes étant orientées), ou, par défaut, sur l'ensemble des tâches. Il propage les valeurs forcées des dates de début au plus tôt en parcourant le graphe de gauche à droite. On peut considérer PROPAGER-DÉBUT comme une implémentation du calcul d'une date de début t_j

$$t_j = \max_{c_{i,j}} \{ f_{c_{i,j}}(t_i) \} \; avec \; f_{c_{i,j}}(x) = x + (\alpha d_i + \beta - \gamma d_j)$$

qui est progressivement calculé jusqu'à stabilisation du réseau (cf. [Laurière 86], §IV). Cet algorithme de propagation est un cas particulier de l'algorithme d'optimisation de Bellman (présentation dans [Clément 75]), qui se réduit ici à un tri[10] en $O(n^2)$, dont la complexité peut

[10] Ce type d'algorithme s'intègre à merveille dans des représentations où une *tâche* n'est pas réduite à quelques nombres : lorsque la tâche est un objet de la représentation, une propagation s'implémente sous la forme d'un *réflexe* [Moon & Weinreb 87] de type SI_MODIFIÉ, ce qui correspond à un contrôle *décentralisé* de l'algorithme précédent.

être ramenée à $O(n\log(n))$ par hachage. Davis propose une étude générale de la propagation de *contraintes* numériques (algébriques) dans [Davis 87].

L'algorithme sur les dates de fin f_i PROPAGER-FIN est obtenu par dualité (échanger i et j, remplacer t par f) et propage les valeurs de dates de fin au plus tard en parcourant le graphe de droite à gauche. Il est lancé sur les tâches les plus à droite (maintenant calculables) ou, toujours par défaut, sur toutes les tâches, en ayant au préalable initialisé leur date de fin au plus tard à une valeur arbitrairement grande (en général, la date de fin au plus tôt du graphe D est obtenue par $D = \max_{i\in\mathcal{P}}(t_i + d_i)$).

Il reste à vérifier que les fenêtres temporelles ainsi déterminées $[t_i, f_i]$ forment effectivement une solution en vérifiant la *contrainte d'intégrité* : $f_i - t_i \geq d_i$. La violation d'une des contraintes d'intégrité indique l'inconsistance de l'ensemble des contraintes $c_{i,j}$ (il faut alors en desserrer une et relancer les algorithmes PROPAGER-DEBUT et PROPAGER-FIN).

Si le graphe est consistant, on démontre qu'il existe un plus long chemin, appelé *chemin critique*, qui détermine la durée globale du graphe (notée D précédemment). Les tâches, critiques, appartenant au chemin critique sont telles que $f_i - t_i = d_i$ (leur *marge* $(f_i - t_i) - d_i$ est nulle). Une tâche à marge positive peut encore *déraper* (i-e démarrer plus tard que prévu) sans faire déraper le plan global, ce qui est impossible pour une tâche critique sous peine de faire reculer la date de fin au plus tôt du graphe.

2.2.2 Systèmes à modules numériques

Cette technique de propagation de dates s'implémente bien de façon répartie. Vere l'utilise pour déterminer les intervalles pendant lesquels des photographies d'une planète peuvent être prises depuis une sonde interplanétaire [Vere 83]. Les systèmes de gestion de carte du temps (TMM)les intègrent également [Dean 86]. Certains algorithmes polynomiaux d'admissibilité d'un ensemble de contraintes temporelles ont été mis à jour dans [Rit 88].

Ces systèmes seront plus amplement détaillés en 3.3.1.

2.3 Logiques classiques

2.3.1 Logique des prédicats du premier ordre

Axiomatique La logique des prédicats du premier ordre se construit à partir d'une *signature* Σ (composée de symboles de *constantes*, de *fonctions* et de *prédicats*) et d'un ensemble de *variables* X, ce qui permet de définir par induction l'ensemble des *termes* $T_\Sigma[X]$ (composé des constantes, des variables et des termes composés $f(t_1, \ldots, t_n)$ où f est une fonction n-aire de Σ et les t_i sont des termes) et l'ensemble des *formules* $L_\Sigma[X]$ (termes reliés par les prédicats de la signature, par les *opérateurs* : $\wedge$, $\vee$, $\Rightarrow$, $\neg$ et par les *quantificateurs* $\forall$ et $\exists$). La transformation d'une formule s'effectue au moyen des treize *règles d'inférences* de Gentzen [Lalement & Saint 86], ce qui permet de construire des *dérivations* (applications successives des règles d'inférences) aboutissant à des *tautologies* (formules cibles obtenues sans axiomes). Une *théorie* se construit

à partir d'un ensemble A d'*axiomes* (formules) et est composée de toutes les formules dérivables à partir de ces axiomes (*théorèmes*), i-e les formules φ de $L_\Sigma[X]$ telles que $A \vdash \varphi$. Le *calcul des prédicats du premier ordre* est la théorie sans axiome, i-e l'ensemble des formules φ de $L_\Sigma[X]$ telles que $\vdash \varphi$.

Sémantique Pour donner une sémantique à une signature Σ, on définit une Σ-*algèbre* A qui associe (resp.) aux symboles de constantes, aux symboles de fonctions et aux symboles de relations de Σ, (resp.) les éléments de A, les fonctions de $A^n \to A$ (n étant l'arité du symbole de fonction) et les relations $A^n \to \{vrai, faux\}$ (ou dans $\underline{2} = \{0,1\}$) qui les *réalisent*. La réalisation (*dénotation*) d'un terme, puis d'une formule, dans A est définie par induction à partir de la réalisation d'une variable, qui associe à toute *instanciation* (application $\xi : X \to A$) un élément de A (le *calcul* du terme ou de la formule à partir d'une instanciation sur A). La Σ-algèbre A est un *modèle* d'une formule φ (A *satisfait* φ, noté $A \models \varphi$) ssi la réalisation de φ dans A est vraie pour toute instanciation sur A. Une formule φ est *valide* ssi elle est satisfaite dans toute Σ-algèbre.

Limitations Les résultats mettant en rapport le syntaxique (les tautologies $\vdash \varphi$ ou plus généralement les théorèmes $A \vdash \varphi$) et le sémantique (les formules valides $\models \varphi$) sont constitués par la série des théorèmes de limitation bien connus [Lalement & Saint 86] :

- *une formule φ est un théorème ssi φ est valide* (premier théorème de Gödel en 1930).
- *si l'arithmétique formelle est consistante, alors elle est incomplète* (deuxième théorème de Gödel en 1931). Gödel ayant également démontré que tout système formel est équivalent à l'arithmétique de Peano (codage de Gödel), le résultat précédent s'étend à tout système formel. Une des interprétations classiques de cette bombe mathématique consiste à dire qu'il existe des pensées non verbalisables, incommunicables (inatteignables par un système quelconque de signes) [Hofstädter 86].
- *il existe des systèmes formels pour lesquels toute interprétation conduit à des énoncés à la fois vrais et non dérivables* (Tarski en 1935). Ce qui est interprété par : *"ce qui est vrai n'est pas toujours démontrable"* ou *"la notion de vérité n'est pas formalisable"* [Laurière 86].
- *la logique des prédicats du premier ordre est indécidable* (Church en 1936). En fait, elle est semi-décidable. Ces preuves se basent sur une définition de la notion de *calcul* : pour Gödel (1934), le calcul se définit à partir de *fonctions récursives* ; pour Church et Kleene, le calcul se définit par le *lambda-calcul* ; pour Turing (1936), le calcul se définit à partir des *états d'une machine idéale* (*machine de Turing*, déterministe ou non). Church a démontré que ces trois définitions sont de toutes façons équivalentes et a proposé d'identifier calculabilité et récursivité (thèse de Church).

Systèmes en logique classique Le principe de systèmes basés uniquement sur la logique classique (des prédicats du premier ordre) consiste à augmenter d'une unité l'arité de tout prédicat manipulé pour représenter le temps : des termes du type $f(t, a_1, \ldots, a_n)$ sont substitués aux termes $f(a_1, \ldots, a_n)$, où f est initialement un symbole fonctionnel d'arité n. L'idée immédiate de l'ajout d'un argument représentant le temps a l'énorme défaut de ne pas privilégier le temps.

Introduire des axiomes tirant parti de ce premier argument temporel revient à modifier la logique des prédicats du premier ordre et à renvoyer dans l'atemporel (réifier) les autres arguments : c'est exactement la base de l'approche utilisée dans les logiques réifiées (cf. 2.6).

2.3.2 Logique des propositions

L'ensemble des propositions sur un ensemble A se définit comme l'ensemble des termes $T_\Sigma[A]$ où A est l'ensemble (quelconque) des *variables propositionnelles* et où la signature Σ ne contient que les connecteurs $\wedge$, $\vee$, $\neg$, $\Rightarrow$. La seule réalisation possible s'effectue sur la Σ-algèbre $\{vrai, faux\}$, ces valeurs représentant les *valeurs de vérité* des variables propositionnelles. Si l'ensemble initial A est fini, le nombre d'instanciations (*interprétations*) possibles est de $2^{Card(A)}$, ce qui assure la décidabilité de la théorie (parcours exhaustif ou *table de vérité*).

2.4 Logiques modales

C'est C. Lewis qui, au début du siècle (1912), donna naissance à ce qui s'appellera les *logiques modales aléthiques*. Etudiant le *paradoxe de l'implication matérielle*, en logique des propositions, $\varphi \Rightarrow (\psi \Rightarrow \varphi)$ (si on a φ, alors on peut la déduire de n'importe quoi) et $\neg\varphi \Rightarrow (\varphi \Rightarrow \psi)$ (si $\neg\varphi$ est démontrée, φ permet de déduire n'importe quoi), il définit l'*implication stricte* $\varphi \rightarrow \psi$ à partir de l'impossibilité logique $\neg\Diamond(\varphi \wedge \neg\psi)$ (il est impossible que φ soit vraie et ψ fausse lorsque φ implique strictement ψ). La possibilité $\Diamond$ ainsi introduite, la nécessité $\Box$ est implicitement définie par dualité, et Lewis étudia divers systèmes modaux S_1, S_2, ...

En présentant les logiques modales, nous présenterons les systèmes S_4 (rebaptisé $KT4$, du nom de leurs axiomes) et S_5 [Audureau et coll. 90].

Axiomatisation Une logique modale est une extension d'une logique classique (logique des propositions ou logique des prédicats du premier ordre) qui contient un plus grand nombre de modalités que la logique source. Une modalité étant une suite quelconque d'opérateurs, les logiques classiques contiennent au moins deux modalités : "." (absence de modalité) et "$\neg$" (négation) ; les autres modalités syntaxiquement constructibles ($\neg\neg$, $\neg\neg\neg$, ...) se ramènent aux deux précédentes ($\forall\varphi, \forall n,\ \neg^{2n}\varphi \Leftrightarrow \varphi$ et $\forall\varphi, \forall n,\ \neg^{2n+1}\varphi \Leftrightarrow \neg\varphi$). Introduire de nouveaux opérateurs dans une logique, donner des axiomes les définissant et des règles d'inférences les manipulant crée une logique modale particulière : logique aléthique (cf. ci-après), mais aussi logique épistémique, logique déontique, ... (cf. [Audureau et coll. 90]).

Les modes les plus connus, et initiateurs de l'extension modale des logiques [Audureau et coll. 90], sont la *nécessité* et la *possibilité*. Ce type de logique modale (*aléthique*) se construit en ajoutant à la logique des propositions les deux opérateurs $\Box$ (mode nécessité) et $\Diamond$ (mode possibilité) Leur signification à titre indicatif est la suivante[11] :

[11] Par analogie avec *"nécessairement"*, on emploiera parfois le néologisme *"possiblement"* (au lieu de *"il est possible que"*) : la proposition φ est *possiblement* vraie si ...

$\Box\varphi$: la proposition φ est nécessairement vraie ;
$\Diamond\varphi$: il est possible que la proposition φ soit vraie.

Les règles de construction des formules sont alors étendues pour intégrer des formules de forme $\Box\varphi$ et $\Diamond\varphi$ (φ étant une formule). Ces opérateurs sont duaux : $\Box\varphi = \neg\Diamond\neg\varphi$ et $\Diamond\varphi = \neg\Box\neg\varphi$. Le choix des axiomes à ajouter ensuite à une logique classique définit une logique modale aléthique particulière. Par exemple, les axiomes suivants :

$$\Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi) \quad (1)$$
$$\Box\varphi \Rightarrow \varphi \quad (2)$$
$$\Box\varphi \Rightarrow \Box\Box\varphi \quad (3)$$

ajoutés à ceux de la logique des propositions, définissent l'axiomatique KT4. Le premier axiome (appelé K) traduit la distributivité de $\Box$ sur $\Rightarrow$. L'axiome (2) (appelé T) servira à modéliser la réflexivité et l'axiome (3) (appelé 4), la transitivité. En remplaçant l'axiome (3) par l'axiome (4) (appelé 5) : $\Diamond\varphi \Rightarrow \Box\Diamond\varphi$, on définirait l'axiomatique KT5, extension de K-T4 [Audureau et coll. 90]. Dans les deux cas, les règles d'inférences supplémentaires sont :

modus ponens :

$$\frac{\vdash \varphi \qquad \vdash \varphi \Rightarrow \psi}{\vdash \psi}$$

nécessitation :

$$\frac{\vdash \varphi}{\vdash \Box\varphi}$$

Si une logique classique possède deux modalités (à une équivalence près, cf. ci-dessus), Audureau démontre que KT4 possède 14 modalités alors que son extension KT5 n'en possède que 6 (des modalités de KT4, distinctes à une équivalence près, sont démontrées équivalentes grâce à l'axiome (4) de KT5).

Sémantique La construction habituelle des modèles de logique classique (cf. 2.3) ne s'appliquent pas ici, car la vérité d'une formule dépend maintenant des circonstances : la formule P est nécessairement vraie si elle est vraie *en toutes circonstances*. L'intérêt de la logique modale est de fournir un formalisme permettant de définir précisément ces "circonstances" jugées acceptables, et dans lesquelles l'interprétation habituelle s'applique. On parle alors de *mondes possibles* au lieu de circonstances, un monde possible étant un ensemble de formules.

Un modèle $\mathcal{M}$ d'une logique modale se construit avec un *graphe de mondes possibles* (sémantiques de Kripke) : un ensemble de mondes possibles M, une relation d'accessibilité $\rightsquigarrow$ entre mondes possibles et une fonction $v : M \rightarrow \wp(T)$, qui fournit, pour chaque monde possible, l'ensemble des formules (déclarées vraies) qui le composent (*valuation* du monde possible).

Une formule φ est *satisfaisable* s'il existe un monde possible m dans lequel φ est vraie : $\mathcal{M} \models \varphi \overset{\text{def}}{\Longleftrightarrow} \exists m \in M, \varphi \in v(m)$ (ce qui est alors noté : $\mathcal{M}, m \models \varphi$). Concernant les modalités, la formule φ est *nécessairement vraie* dans un monde possible m de M ssi elle est vraie dans tous les mondes possibles directement accessibles depuis m : $\mathcal{M}, m \models \Box\varphi \overset{\text{def}}{\Longleftrightarrow} \forall m'$, ($(m \rightsquigarrow m') \Rightarrow \mathcal{M}, m' \models \varphi$). De même, *il est possible* qu'une formule φ soit vraie dans un monde

possible m ssi elle est vraie dans au moins un monde possible directement accessible depuis m : $\mathcal{M}, m \models \Diamond\varphi \stackrel{\text{def}}{\Longleftrightarrow} \exists m',\ ((m \rightsquigarrow m') \Rightarrow \mathcal{M}, m' \models \varphi)$.

Les différents modèles de logiques modales sont obtenus en jouant sur les propriétés de la relation d'accessibilité $\rightsquigarrow$. Par exemple, l'axiome (2) précédent est vrai dans tout modèle où $\rightsquigarrow$ est réflexive. Si $\rightsquigarrow$ est en plus transitive, ce modèle est un modèle de KT4. Si $\rightsquigarrow$ est en plus symétrique (relation d'équivalence), ce modèle devient un modèle de KT5.

Le premier ordre L'extension modale de la logique des prédicats du premier ordre s'effectue de façon analogue à celle de logique des propositions : ajout d'opérateurs, extension de la construction des formules pour les intégrer, axiomes et règles d'inférences pour les manipuler.

Cependant, dans la définition d'un modèle d'une logique modale du premier ordre, rien ne garantit que la réalisation d'une constante, par exemple, ne varie pas au fil des mondes possibles ou même disparaisse pendant certaines périodes. L'individu dénoté par une constante peut varier au cours du temps : l'individu dénoté par la constante `Francois-Mitterrand` n'est pas le même selon l'instant considéré (ses propriétés, ou opinions, changent au cours du temps, ou des décennies, ...), et finit (ou finira) même par disparaître un jour[12]. On considère ici le cas où, précisément, l'interprétation des constantes et des variables ne varie pas au fil des mondes possibles (sémantique *rigide*).

Un modèle $\mathcal{M}$ d'une logique modale du premier ordre se compose, comme précédemment, d'un graphe de mondes possibles $(M, \rightsquigarrow)$ mais aussi d'un domaine D d'individus, d'une fonction $d : M \rightarrow \wp(I)$ qui associe à chaque monde possible un sous-ensemble d'individus, et d'une fonction $m : T_\Sigma[X] \rightarrow D$ qui asssocie à tout terme un individu (éternel).

Comme précédemment, les différentes logiques modales du premier ordre sont obtenues en modulant la relation d'accessibilité $\rightsquigarrow$: si elle est réflexive, le modèle est modèle de l'axiome (2) ; si en plus elle est transitive, c'est un modèle d'une axiomatique KT4 du premier ordre ; si en plus elle est symétrique, c'est un modèle d'une axiomatique KT5 du premier ordre.

On peut aussi jouer sur le principe de variation du domaine d'individu en fonction du monde possible (qui n'est a priori pas contraint non plus). Si m et m' sont deux mondes possibles consécutifs ($m \rightsquigarrow m'$), la définition de la relation entre $d(m)$ et $d(m')$ est laissée libre. On distingue trois principes de variation, en comparant $d(m)$ et $d(m')$ par la relation d'inclusion $\subset$:

$\forall m, m' \in M, (m \rightsquigarrow m' \Rightarrow d(m) \subset d(m'))$ (*domaine cumulatif*). Les individus se transportent de monde possible en monde possible.

$\forall m, m' \in M, (m \rightsquigarrow m' \Rightarrow d(m) = d(m'))$ (*population uniforme*). Tous les domaines sont identiques (*"les individus sont éternels"* ... [Bestougeff & Ligozat 90]) Ce cas est le plus sûr : les formules $\Box\forall x, \varphi$ (formule de type *de re*) et $\forall x, \Box\varphi$ (formule de type *de dicto*) sont équivalentes et les constantes ne risquent pas de référencer des individus devenus inexistants.

[12] Selon la règle d'inférence bien connue (cf. [Laurière 86], parmi beaucoup d'autres) $humain(x) \rightarrow mortel(x)$, par laquelle la mortalité de *Socrate* est affirmée.

$\forall m, m' \in M, (\, m \rightsquigarrow m' \Rightarrow d(m) \supset d(m')\,)$ (*extension imaginaire*). Réduction monotone des individus au cours des mondes possibles.

2.5 Logiques temporelles

Il n'y a pas de différence essentielle entre logique temporelle et logique modale. La logique temporelle est censée utiliser un langage plus riche et est intuitivement plus claire : au lieu de *monde possible* et de relation d'*accessibilité* $\rightsquigarrow$, on parle plutôt de *date* et de relation de *précédence* $\preceq$.

Nous présentons succintement la logique temporelle de base et ses logiques dérivées utilisées en informatique (cf. [Audureau et coll. 90], [Bestougeff & Ligozat 90] et [Turner 86] pour une présentation détaillée).

2.5.1 Logique temporelle de base

On introduit les modalités *"passé"* et *"futur"* au moyen des opérateurs P et F interprétés par :

$P\varphi$: la proposition φ a été vraie à un instant du passé ;
$F\varphi$: la proposition φ sera vraie à un instant du futur.

Pour s'intégrer aux logiques modales, on définit leurs opérateurs duaux H (passé fort, *has been*) et G (futur fort, *going to*) :

$H\varphi =_{\text{def}} \neg P \neg \varphi$ *(la proposition φ a toujours été vraie) ;*
$G\varphi =_{\text{def}} \neg F \neg \varphi$ *(la proposition φ sera toujours vraie).*

Les modalités $\Box$ et $\Diamond$ de la logique modale sont récupérées de deux manières, selon le point de vue adopté par rapport au *maître argument* de Diodore dit Cronos [Rescher 71] :

- l'approche d'Aristote, puis des *Stoïciens*, résolument optimiste, situe le possible et le nécessaire dans l'avenir : $\Diamond\varphi =_{\text{def}} F\varphi$ et $\Box\varphi =_{\text{def}} G\varphi$.
- l'approche des *Mégariques*, rejetant le présent, trace un possible et un nécessaire éternels : $\Diamond\varphi =_{\text{def}} P\varphi \wedge F\varphi$ et $\Box\varphi =_{\text{def}} H\varphi \wedge \varphi \wedge G\varphi$. Est nécessaire ce qui a été, est ou sera toujours.

Axiomatisation L'axiomatisation minimale est la suivante :

$$H(\varphi \Rightarrow \psi) \Rightarrow (H\varphi \Rightarrow H\psi)$$
$$\varphi \Rightarrow HF\varphi$$

et avec les deux axiomes duaux concernant G et P. Le premier axiome, alter ego de celui de 2.4, exprime la distributivité de H sur $\Rightarrow$. Le deuxième axiome est dicté par l'intuition : si

φ est vraie maintenant, alors dans toutes les dates antérieurs il a été vrai que φ serait vraie un jour.

Les règles d'inférences supplémentaires, outre le *modus ponens*, comprennent celles de *généralisation temporelle* vers le passé et le futur : si $\vdash \varphi$, alors $\vdash H\varphi$; si $\vdash \varphi$, alors $\vdash G\varphi$.

Sémantique On donne un modèle à cette axiomatisation, de façon analogue aux logiques modales, avec un graphe de dates et une fonction de valuation. Une formule φ est vraie dans un modèle $\mathcal{M}(M, \preceq, v)$ ssi elle est vraie dans l'un de ses mondes possibles : $\mathcal{M} \models \varphi \stackrel{\text{def}}{\Longleftrightarrow} \exists m \in M, \varphi \in v(m)$, ce qui est noté $\mathcal{M}, m \models \varphi$; $F\varphi$ (resp. $P\varphi$) est vraie à une date ssi φ est vraie à *au moins* une date immédiatement postérieure (resp. antérieure) ; $G\varphi$ (resp. $H\varphi$) est vraie à une date ssi φ est vraie à *toute* date immédiatement postérieure (resp. antérieure).

Jouer sur les contraintes de la relation de précédence $\preceq$ définit la conception du temps adoptée pour une logique temporelle. La séparation du passé d'avec le futur semble naturelle, aussi la contrainte intuitivement minimale est l'*antisymétrie* : $\forall m, m', ((m \preceq m') \wedge (m' \preceq m)) \Rightarrow m = m'$; l'antisymétrie ne se pas modélise pas sous forme d'axiomes, mais les modèles construits la satisfont quand même [Audureau et coll. 90].

L'intuition suggère également que $\preceq$ soit :

- ou bien un *ordre partiel*, ce qui s'axiomatise par $H\varphi \Rightarrow \varphi$ (réflexivité, cf. 2.4) et $H\varphi \Rightarrow HH\varphi$ (transitivité) sans oublier les deux axiomes duaux pour G ;
- ou bien un *ordre total*, ce qui axiomatise par les axiomes de l'ordre partiel augmentés de $F\varphi \wedge F\psi \Rightarrow F(\varphi \wedge F\psi) \vee F(F\varphi \wedge \psi)$ (si φ et ψ seront vraies, soit on aura φ puis ψ, soit on aura ψ puis φ).

La logique temporelle minimale est valide et complète (démonstration compréhensible de la complétude dans [Bestougeff & Ligozat 90]).

2.5.2 Logiques temporelles dérivées

L'ajout d'axiomes à la logique temporelle de base, signifiant l'ajout de contraintes sur la relation de précédence $\preceq$, permet de créer les diverses logiques temporelle étendues utilisées en informatique. Nous indiquons ici les principales, qui sont toutes valides et complètes [Rescher 71] :

logique à extrémité(s) La relation $\preceq$ possède un plus petit ou un plus grand élément, ce qui s'exprime en remplaçant les axiomes de réflexivité et transitivité de $\preceq$ dans la logique temporelle de base par $H\mathit{faux} \vee PH\mathit{faux}$ dans le cas du plus petit élément (origine) : soit les dates antérieures à l'origine satisfont *faux*, i-e sont inexistantes (auquel cas on se trouve à cette origine), soit $H\mathit{faux}$ est vraie à une date précédente (auquel cas on se trouve à une date valide différente de l'origine). Axiome et interprétation analogue pour l'extrémité supérieure (utiliser les modalités duales vers le futur).

logique arborescente La conception du temps représenté est celle pour laquelle le passé est connu (linéaire) et le futur, encore indéterminé, peut se développer et redévelopper en plusieurs voies (arborescent), ce qui peut correspondre à l'exécution de programmes parallèles. La formalisation de la logique arborescente comprend les axiomes correspondant à un ordre total pour le passé et ceux correspondant à un ordre partiel pour le futur.

logique discrète Cette logique modélise un temps à étapes et est utilisée pour la modélisation de l'exécution de programmes. Elle se formalise avec les axiomes modélisant la transitivité et l'anti-symétrie de $\preceq$, et avec l'axiome $\varphi \wedge H\varphi \Rightarrow FH\varphi$ (toute date a au moins un prédecesseur et un successeur) et son dual avec G et P.

Bestougeff [Bestougeff & Ligozat 90] esquisse un panorama des diverses logiques temporelles dérivées (toutes valides et complètes) et donne leurs relations d'inclusion.

On notera cependant que c'est précisément l'ordre 1, le moins bien maîtrisé dans les logiques modales et temporelles, qui nous intéressera dans la suite.

2.6 Logiques réifiées

Face aux logiques modales ou temporelles centrées sur le raisonnement sur le temps, se sont développées des logiques dites *réifiées* séparant nettement le raisonnement sur le temps du raisonnement atemporel habituel. Un prédicat d'une telle logique est un couple $< t, \varphi >$, où φ est un prédicat atemporel (la chosification, ou réification du prédicat initial) et t le qualificatif temporel de φ. Ces logiques délaissent tout raisonnement sur le prédicat atemporel (dépendant de l'environnement d'application) et se focalisent sur la nature de la qualification temporelle selon deux approches : l'objet temporel élémentaire est soit un intervalle [Allen 84], soit un instant [McDermott 82].

Formellement, les logiques réifiées de Allen et Mc Dermott sont plus proches des théories du premier ordre que des logiques temporelles. Nous avons déjà signalé en 2.4 les difficultés inhérentes aux logiques modales du premier ordre : existence ou non des objets à différents instants, interprétation différente des termes suivant l'instant, ...Les logiques réifiées prennent comme postulat que le domaine d'interprétation ne varie pas au cours des instants : les objets considérés sont éternels (population uniforme, cf. 2.4). Par contre, les objets dénotés par les termes peuvent changer au cours du temps (les "flots" de Mc Dermott), ce qui interdit l'utilisation d'une sémantique rigide.

Les objets représentés par ces logiques sont grossièrement les suivants (types atemporels) :

- les *faits* ou *propriétés* qui ont une valeur de vérité donnée et qui ne changent pas au cours du temps ("*la Terre tourne*"[13]).

[13] Les faits sont supposés ne pas changer de valeur de vérité *dans une portion de temps raisonnable*, correspondant par exemple au temps d'observation des phénomènes modélisés. Sinon, d'aucuns argueraient que la rotation de la Terre n'a de sens qu'entre le Big Bang et Big Crunch ou alors contraindraient le modèle de Friedmann à intégrer une Terre dans chacun des univers successifs en expansion/contraction ...

- les *événements* qui représentent des transformations macroscopiques du monde et dont le déroulement interne n'est pas pris en compte : si *"le plâtre s'est solidifé dans la baignoire de Gaston"*, seul le résultat de l'action est représenté.
- les *processus* qui représentent aussi des transformations du monde, mais dont le déroulement (microscopique, continu) est analysé : "le plâtre *est en train* de se solidifier dans la baignoire de Gaston".

2.6.1 Logique des intervalles de Allen

Ayant travaillé sur la représentation du dialogue en langage naturel dans son système ARGOT, Allen avait besoin d'un système temporel présentant une certaine imprécision au niveau de l'échelle des temps et de la nature des relations entre les faits. Son système ne pouvait alors pas être numérique ni manipuler explicitement des dates (trop pauvre sémantiquement : $<$, $=$ *ou* $>$). Allen s'est alors intéressé à la manipulation des *intervalles* via les *relations* qu'ils entretiennent[14] [Allen 81], et a ensuite formalisé cette approche en une logique réifiée [Allen 83], [Allen 84].

Système de base Pour construire ces relations formellement (termes de relation binaires entre des termes du type temporel), on modélise une dernière fois un intervalle i_1 par ses dates de début et de fin (d_1, f_1) et on cherche à placer deux dates (d_2, f_2) (représentant un autre intervalle i_2) en utilisant les cinq zones définies par i_1 ($t < d_1$, $t = d_1$, $d_1 < t < d_2$, $t = d_2$, $d_2 > t$). Sur les 25 possibilités de placement de d_2 et f_2, dix sont éliminées car elles violent la contrainte d'intégrité ($d_2 < f_2$), deux sont éliminées car elles réduisent i_2 à une date (d_1 ou d_2). Les relations restantes sont les *treize relations de Allen*, constituées de l'égalité ($=$), de 6 relations (notées[15] $<, m, o, s, d, f$, cf. figure 2.2) et des 6 relations transposées (notées $>, m', o', s', d'$ et f').

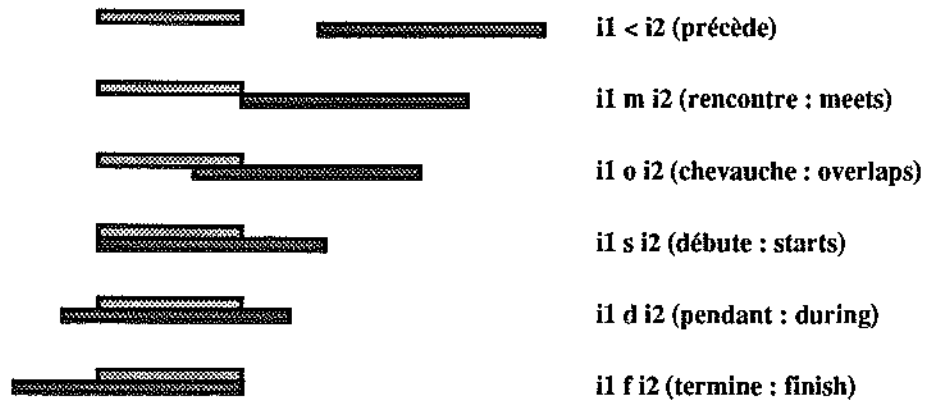


Figure 2.2: Représentation linéaire des treize relations élémentaires de Allen (réduites à 6)

Une représentation planaire originale (cf. figure A.1) est présentée dans [Rit 88].

[14] L'exemple typique de problème traité est celui de la synchronisation du flash en photographie : l'intervalle comprenant le déclenchement du flash et l'arrivée des photons dans l'objectif, de durée très faible, doit être inclus dans l'intervalle pendant lequel le diaphragme de l'appareil est ouvert (synchro-flash X au 250e, en général).

[15] La dernière relation *finish* (f) est parfois appelée *ends* (e) [Bestougeff & Ligozat 90], ce qui est mnémotechniquement meilleur : les six relations sont $<, m, o, d, e, s$ ...

Pour exprimer la composition de relations élémentaires de Allen, on introduit la notation de disjonction suivante : soit I l'ensemble des intervalles et Σ l'ensemble des treize symboles de relations précédents.

$$\forall i_1, i_2 \in I, \forall (r_i)_{i=1,n} \in \Sigma^n : \; i_1 \, (r_i)_{i=1,n} \, i_2 \stackrel{\text{def}}{\Longleftrightarrow} \bigvee_{i=1}^{n} (i_1 \, r_i \, i_2)$$

Ces 2^{13} relations composées permettent d'exprimer une certaine incertitude que ne permettent pas toujours d'exprimer les (déjà imprécises) relations initiales. Par exemple, l'inclusion au sens large de deux intervalles i et j se définit par : $i \, in \, j \stackrel{\text{def}}{\Longleftrightarrow} i \, (sdf) \, j$. Tout l'intérêt du formalisme de Allen provient de la définition de la composition de relations élémentaires : $i_1 \, r_1 \, i_2 \wedge i_2 \, r_2 \, i_3 \Rightarrow i_1 \, (r_1 \circ r_2) \, i_3$. Mais l'ensemble des treize relations élémentaires Σ n'est pas stable par la loi de composition $\circ$ ($(< \circ f) \notin \Sigma$, par exemple, cf. figure 2.3).

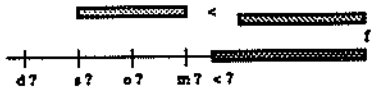


Figure 2.3: Exemple de composition non stable

Cependant, cette composition peut s'exprimer par une *disjonction* de quelques relations élémentaires, en remarquant que la relation la plus générale est $\{=, <, m, o, s, d, f, >, m', o', s', d', f'\}$, c'est-à-dire Σ ! La relation $< \circ f$ ci-dessus était en fait la relation composée $(< mosd)$ (cf. la table A.1). La composition de relations composées $R_1 \circ R_2$ (et non plus de relations élémentaires comme $r_1 \circ r_2$) se définit simplement en considérant la composition de deux relations élémentaires $r_i \circ r_j$ comme un ensemble (éventuellement réduit à un seul élément) de relations élémentaires. Avec $R_1 = (r_{1,i})_{i \in [1,n_1]} \in \Sigma^{n_1}$ et $R_2 = (r_{2,j})_{j \in [1,n_2]} \in \Sigma^{n_2}$:

$$R_1 \circ R_2 = (r_{1,i})_{i \in [1,n_1]} \circ (r_{2,j})_{j \in [1,n_2]} =_{\text{def}} \bigcup_{i,j \in [1,n_1] \times [1,n_2]} (r_{1,i} \circ r_{2,j})$$

Axiomatisation Le système d'Allen est une logique du premier ordre à trois types atemporels : *propriété, événement* et *processus*. Le seul type temporel est *intervalle*. Dans la suite, les variables $i, i_1, \ldots$ seront implicitement suppposées appartenir à ce dernier type *intervalle*.

Une *propriété* (ou un *fait*) p est vraie sur un intervalle i, noté $vrai(i,p)$, ssi elle est vraie sur tout sous-intervalle de i : $vrai(i,p) \stackrel{\text{def}}{\Longleftrightarrow} \forall j \in I, \, (\, j \, (sdf) \, i \Rightarrow vrai(j,p) \,)$ (simplification de l'axiome d'Allen, qui sous-entend la *densité* de la structure des intervalles). Selon 2.3, cette notation de satisfaction devrait en toute rigueur être $p(i)$, mais elle est plus pratique lorsque le terme réifié n'est plus une proposition simple mais un terme complexe avec fonctions et prédicats : l'écriture $vrai(i, \textit{se-solidifie-dans}(\textit{plâtre}, \textit{baignoire-de-Gaston}))$ est préféré à $(\textit{se-solidifie-dans}(\textit{plâtre}, \textit{baignoire-de-Gaston}))(i)$. Les interprétations donnés à la conjonction, à la négation et à la disjonction sont les suivantes :

$$\begin{array}{rcl}
vrai(i,p_1) \wedge vrai(i,p_2) & \stackrel{\text{def}}{\Longleftrightarrow} & \forall j \in I, \, (\, j \, (sdf) \, i \Rightarrow vrai(j,p_1) \,) \wedge \forall j \in I, \, (\, j \, (sdf) \, i \Rightarrow vrai(j,p_2) \,) \\
\neg vrai(i,p) & \stackrel{\text{def}}{\Longleftrightarrow} & \forall j \in I, \, (\, j \, (sdf) \, i \Rightarrow \neg vrai(j,p) \,) \\
vrai(i,p_1) \vee vrai(i,p_2) & \stackrel{\text{def}}{\Longleftrightarrow} & \forall j \in I, \, (\, j \, (sdf) \, i \Rightarrow \exists k, \, k \, (sdf) \, j \wedge (vrai(k,p_1) \vee vrai(k,p_2)) \,)
\end{array}$$

L'interprétation de la conjonction est évidente. Allen interpète la négation d'une propriété p sur un intervalle i comme : la propriété n'est vraie sur aucun sous-intervalle. Il interprète la disjonction de deux propriétés sur un même intervalle comme : on peut trouver une partie de tout sous-intervalle où la disjonction est vraie.

Pour Allen, un *événement* ne se décompose pas sur ses sous-intervalles :

$$occur(i, e) \wedge j\ (sdf)\ i \Rightarrow \neg occur(j, e)$$

Au contraire, un *processus* est décomposable, mais pas forcément sur tous ses sous-intervalles :

$$occuring(i, proc) \Rightarrow \exists j,\ j\ (sdf)\ i \wedge occuring(j, proc)$$

Les processus et les événements sont reliés par la relation $occur(i, x) \Rightarrow occuring(i, x)$, qui traduit la possibilité de focalisation sur un fait : *"le plâtre s'est solidifié dans la baignoire de Gaston"* peut être vu ponctuellement en tant qu'événement, ou comme processus si l'on veut exprimer le fait que le plâtre est *en train de* se solidifier dans la baignoire de Gaston. Cette relation doit être explicitement spécifiée, puisque les événements et processus ne sont pas définis mais caractérisés. Pour utiliser une propriété, un processus ou un événement, on décrit les intervalles pendant lesquels cette propriété, processus ou événement est vrai et les relations qu'il entretient avec ses sous-intervalles. L'implication résultante doit vérifier l'implication de son type (décomposition totale, partielle ou nulle sur les sous-intervalles).

L'expression de la causalité sous-entend celle d'un lien de déclenchement entre événements "si tel événement se produit, alors tel autre événement se produira". Allen introduit pour cela la notation $ECAUSE(e_1, i_1, e_2, i_2)$. formellement définissable, qui se lit *"l'événement e_1, qui a eu lieu en i_1, a causé l'événement e_2, qui a eu lieu en i_2"*. Allen exprime également la notion d'*agent*, qui engendre des événements ou des processus.

Maintien de la cohérence Un cas réel (plus de deux intervalles) se modélise par un *graphe de Allen*, représentation des informations temporelles portées par un énoncé quelconque : à chaque intervalle correspond un nœud du graphe, les relations entre deux intervalles sont représentées par un arc (orienté) étiqueté par un ensemble de symboles représentant ces relations.

A partir d'un graphe composé d'un ensemble fini de nœuds $\{N_i\}_{i\in[1,n]}$ et d'un ensemble fini de relations entre ces nœuds $\{R_{i,j}\}_{i,j\in[1,n]^2}$, la modification $R'_{a,b}$ des relations $R_{a,b}$ entre deux intervalles particuliers a et b nécessite un recalcul des toutes les relations entre intervalles (deux intervalles sont au moins en relation par la relation la plus générale Σ). L'algorithme PROPAGE-SYMBOLIQUE suivant effectue cette sorte de fermeture transitive :

Procédure PROPAGE-SYMBOLIQUE
Initialisations
$P \leftarrow \{(a,b)\}$
$R_{a,b} \leftarrow R_{a,b} \cap R'_{a,b}$
Tant que $P \neq \emptyset$, **faire**
Dépiler (i,j) de P (ou prendre un élément quelconque de P)
Pour tout $k \in [1,n], k \neq i,j$, **faire**
- Calculer les nouvelles valeurs $R'_{i,k}$ et $R'_{k,j}$ de $R_{i,k}$ et $R_{k,j}$:
$R'_{i,k} \leftarrow (R_{i,j} \circ R_{j,k}) \cap R_{i,k}$
$R'_{k,j} \leftarrow (R_{k,i} \circ R_{i,j}) \cap R_{k,j}$
- si $R'_{i,k} = \emptyset \vee R'_{k,j} = \emptyset$, alors SORTIE (incohérence)
- effectuer la mise à jour (si elle a eu lieu) et la propager :
si $R'_{i,k} \neq R_{i,k}$, alors $R'_{i,k} \leftarrow R_{i,k}$ et $P \leftarrow P \cup \{(i,k)\}$
si $R'_{k,j} \neq R_{k,j}$, alors $R'_{k,j} \leftarrow R_{k,j}$ et $P \leftarrow P \cup \{(k,j)\}$

Tant que le réseau n'est pas stabilisé, cet algorithme ajoute de proche en proche des contraintes cohérentes $R'_{m,n}$ sur les arcs $R_{m,n}$ (en restreignant les symboles étiquetant les arcs, puisque $R'_{m,n} \subset R_{m,n}$ par construction) et détecte au passage l'incohérence gloable des relations grâce au test $R_{m,n} = \emptyset$. La possibilité d'une mise à jour non triviale nécessite le stockage des couples d'intervalles attendant d'être traités, via la pile P. Allen montre par un contre-exemple que cet algorithme ne permet pas de détecter *toutes* les incohérences.

La complexité en temps de l'algorithme est de $O(n^3)$ [Vilain & Kautz 86]. Le problème général de la détermination de la cohérence (ou de la détermination de toutes les inférences : la fermeture transitive) est un problème NP-complet [Vilain & Kautz 86], ce qui était prévisible puisque ce problème est équivalent à celui de la satisfaisabilité d'une formule par un modèle [Bestougeff & Ligozat 90]. Le résultat de complétion initialement annoncé sur l'algorithme de fermeture de l'algèbre des points par Vilain, équivalent à une algèbre des intervalles restreinte, a été ensuite réfuté [VanBeek 89] (en fait, seule la relation d'égalité pose problème). En raisonnant sur les points (la sous-partie $\{=, <, m, d, >, m', d'\}$ de Σ), Malik [Malik & Bindford 83] détermine la fermeture transitive avec l'algorithme du simplexe exponentiel, ce qui le limite à de petites bases d'intervalles.

Conscient du fort ordre de complexité de son algorithme (a priori, un intervalle est relié à tout autre intervalle du réseau), Allen propose de séparer le réseau d'intervalles en sous-réseaux par des considérations d'ordre sémantique (i-e *user-defined*) : à chaque intervalle est associé une référence, autre intervalle de niveau sémantique plus élévé, l'*intervalle de référence*. Sont regroupés sous un même intervalle de référence (relation *during* d) des intervalles supposés "fortement interdépendants", par opposition aux intervalles sous les autres intervalles de référence supposés "faiblement interdépendants". Allen espère ainsi que l'algorithme ci-dessus aura le bon goût de propager les modifications à l'intérieur d'un même intervalle de référence (celui auquel a et b appartiennent), divisant ainsi la complexité par le nombre de groupes de références créés.

Critique D'un point de vue axiomatique, Allen néglige d'exprimer rigoureusement son système selon les termes habituels des logiques temporelles (cf. 2.5) et se contente du méta-langage, ce qui, associé à une certaine lourdeur, peut faire douter de sa consistance [Turner 86]. Il s'agit clairement d'une logique du première ordre pour une certaine description, mais quelle est la forme de ses modèles éventuels [Turner 86] ? Il faut attendre [Bestougeff & Ligozat 90] pour obtenir ce rattachement propre aux structures et logiques temporelles.

D'un point de vue algorithmique, la NP-complétude de la propagation assurant la consistance [Vilain & Kautz 86] rend impossible l'utilisation telle quelle du système, sous peine de temps de mise à jour prohibitif. Allen s'en était apperçu dès son article initial [Allen 81], puisqu'il y introduit déjà la notion d'*intervalle de référence*, dérive sémantique visant à isoler des groupes d'intervalles faiblement couplés, qui compartimente la propagation.

Dans une base de données, fût-elle temporelle, s'affrontent deux types de complexité :

- *la complexité inductive*, qui mesure la difficulté de stocker de nouveaux faits ;
- *la complexité déductive*, qui mesure la difficulté d'extraire une information ;

La complexité globale d'un système de représentation (fixe une fois le système posé) ne peut que se répartir sur ces deux types de complexités. Le système de Allen est l'exemple typique d'une très faible complexité inductive (ajout d'un relation temporelle) naturellement contrebalancée par une très grande complexité déductive (inférences internes à la base pour déceler les contradictions).

Mais, même s'il souffre d'une NP-complétude de propagation, le système des treize intervalles de Allen est intuitivement évident et est générateur d'idées depuis 10 ans. De même, le système de Allen est souvent utilisé pour caractériser des faits en ordre 0 (à la rigueur, des prédicats sans variables) : la moindre variable, pourtant vitale pour un système complet de planification (comme on le verra), rend difficile les inférences de relations temporelles. Peut-être l'apport principal d'Allen est d'avoir montré qu'un raisonnement temporel pouvait s'envisager autrement qu'avec les dates et les instants de la Recherche Opérationnelle, et donc d'avoir cherché à asseoir le raisonnement temporel en I.A. comme domaine propre et autonome.

2.6.2 Logique des instants de Mc Dermott

Pour des raisons obscures, Mc Dermott emploie la notation "polonaise de Cambridge" pour axiomatiser son système [McDermott 82]. Turner [Turner 86] s'est chargé de reformuler son axiomatique en une théorie typée du premier ordre, alors que Bestougeff [Bestougeff & Ligozat 90] la reformule (et la complète) en une théorie typée du second ordre (l'apparition du second ordre venant des *chroniques*, voir ci-dessous).

Système de base La logique temporelle de Mc Dermott est basée sur une représentation arborescente des états et sur une représentation temporelle linéaire des dates (assimilée à la droite des réels $\Re$). L'ensemble des états E, instantanés de l'univers, est ordonné par $\leq$ (relation de précédence $\preceq$ de 2.5), ordre total pour le passé et partiel pour le futur (cf. les logiques

temporelles arborescentes en 2.5.2), dense (on peut toujours trouver un *état* entre deux *états* donnés) sans extrémités. Le lien entre ces états et leur date s'effectue par une fonction de datation $date : E \rightarrow \Re$, de l'ensemble des états dans l'ensemble des réels $\Re$, strictement croissante ($\forall e_1, e_2 : e_1 < e_2 \Rightarrow date(e_1) < date(e_2)$), ce qui assure la compatibilité de l'ordre entre états avec celui des dates.

Une *chronique* (voir figure 2.4) est une histoire du monde, i-e un ensemble d'états totalement ordonnés et qui s'étend à l'infini dans le temps (à toute date de $\Re$, on peut associer un état dans la chronique : la fonction *date*, réduite à la chronique, est surjective).

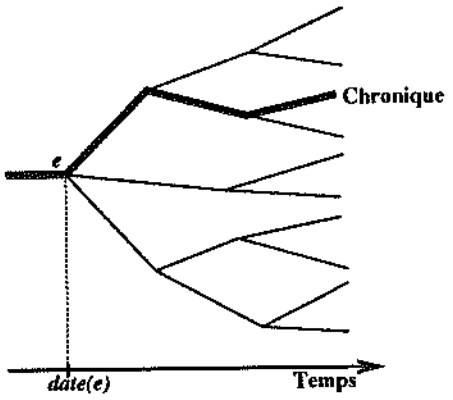


Figure 2.4: Représentation arborescente du temps de Mc Dermott

Une chronique est une copie de la droite des réels qui passe par les états [Bestougeff & Ligozat 90] ; deux chroniques se rencontrant en un état ont un passé commun ; il existe toujours une et une seule chronique passant par deux états donnés.

Bestiaire de Mc Dermott A partir de la structure temporelle précédente (temps arborescent, états, chroniques), Mc Dermott construit les notions de *fait*, d'*intervalle* et d'*événement*, et traduit les liens de causalité.

Pour Mc Dermott, un *fait* est l'ensemble des états dans lequel il est vérifié (la réciproque de la fonction de valuation v en logique modale, cf. 2.4). Comme pour Allen, sa valeur de vérité est notée par $vrai(e, f)$, interpreté par *"le fait f est vrai en l'état e"* : $vrai(e, f) \stackrel{def}{\Longleftrightarrow} e \in f$, ce qui permet de reconstruire formellement le vrai (le fait E) et le faux (le fait $\emptyset$). Un *intervalle* d'états, noté $i(e_1, e2)$, est la portion de l'unique chronique (cf. ci-dessus) qui contient les états e_1 et e_2 ($i(e_1, e_2) \subset E$). Un *événement* evt est l'ensemble des intervalles pendant lequel il se produit. Sa valeur de vérité est notée par $occ(i(e_1, e_2), evt)$ (*to occur*, se produire), qui est vraie ssi $i(e_1, e_2) \in evt$. On la note également en étendant la fonction *vrai* de satisfaisabilité des faits au cas où son premier argument est un intervalle : $vvrai(i(e_1, e_2), evt)$.

Mc Dermott raffine la notion de causalité de Allen en exprimant le *retard de déclenchement* de l'événement cible. Ne pouvant adopter de retard numérique (on retomberait dans des algorithmes tels ceux de 2.2), il définit des *échelles de temps* qui permettent d'exprimer des relations sur les durées d'un niveau sémantique plus élevé.

La causalité entre les événements evt_1 et evt_2 s'exprime par la notation $ECAUSE(f, evt_1, evt_2, r, d_1, d_2)$ qui signifie *"l'événement evt_1 est toujours suivi de l'événement*

evt_2 après un retard se trouvant à l'intérieur de l'intervalle de durées $[d_1, d_2]$, à moins que le fait f ne devienne faux entre temps". Le paramètre r est un réel de l'intervalle $[0, 1]$ qui indique à partir de quel pourcentage de l'exécution de l'événement evt_1 le fait f est testé (rôle analogue au paramètre α de 2.2.1). Cette causalité est complétée par le *"principe de paranoïa"* qui permet d'inférer la présence d'un événement a priori inconnu, s'il est quand même connu pour être la seule cause possible d'un autre événement qui, lui, est connu (*si certaines choses se passent, c'est qu'il y a une raison cachée* ...).

Mc Dermott exprime l'inférence d'un *fait* par un événement (et non plus d'un événement par un événement) grâce à la notion de *persistance* : un fait persiste pendant un temps donné ssi, dans toutes les chroniques, il reste vrai tant que le temps limite n'est pas dépassé, ou bien si le garder vrai rendrait le système incohérent. Plus que l'inférence du fait même, c'est l'inférence de sa persistance qui est intéressante et qui est notée $PCAUSE(f_1, evt_1, f_2, r_1, d_1, d_2, r_2)$ et se lit : *"un événement evt_1 est toujours suivi par un fait f_2 avec un retard à l'intérieur de l'intervalle (d_1, d_2), à moins que F_1 ne devienne faux avant l'expiration du délai"*. Le paramètre r_1 est l'analogue du paramètre r ci-dessus (caractérisation du début du test), le paramètre r_2 est utilisé pour la persistance.

Mc Dermott introduit ensuite nombre de prédicats définissant, entre autres, la notion de "flot" (modélisant une transformation continue) et celle de "canal" (mécanisme responsable des changements spontanés de valeur).

Critique Critiquer le système de Mc Dermott est aisé puisque son formalisme n'est supporté par aucune implémentation (formellement, sa théorie papier n'admet pour l'instant aucun modèle informatique). Comme Allen, Mc Dermott néglige de formaliser son système selon les termes habituels de la logique temporelle [Turner 86] : seul le méta-langage de 2.5 est conservé. L'axiomatique peut alors se permettre d'être particulièrement lourde (44 axiomes en tout), ce qui laisse un doute quant à sa consistance [Turner 86] et rend quasi impossible la construction explicite d'un modèle [Bestougeff & Ligozat 90] (ces deux difficultés étant fortement liées d'après Gödel, cf. 2.3.1). Enfin, pourquoi ne pas autoriser les branches de temps à se rejoindre (structure temporelle en forme de graphe et non d'arbre avec une extrémité supplémentaire à droite qui serait la fin du temps d'observation), ce qui permettrait de définir des relations de comptabilité et de mélange entre chroniques partielles. C'est précisément ce cas qui fait l'objet de toutes les applications intéressantes en planification.

Chapitre 3

Planificateurs existants

Nous présentons dans ce chapitre quelques uns des systèmes de planification qui ont vu le jour depuis 1971 et dont nous nous sommes inspirés pour notre approche. Plusieurs synopses existent dans la littérature : celle de Tate [Tate et coll. 90] (ou [Hendler et coll. 90] pour une version plus introductive) balaie en historien 30 ans d'idées et d'influences mutuelles en planification (et fournit une bibliographie conséquente) ; celle de Georgeff [Georgeff 87] s'intéresse plus à l'aspect formel, théorique (voire philosophique) de ces mêmes auteurs ; celle de Swartout [Swartout 88] s'attache à identifier les problèmes ouverts en résumant et comparant (de façon très argumentée) les techniques existantes ; celle de Brown [Brown 84] considère les débouchés industriels de ces outils de recherche ; celle de Levitt définit une typologie permettant d'intégrer planification avec et sans expertise et ordonnancement [Levitt & Kunz 87] ; celle (très rapide) de Chapman [Chapman 85] attaque un angle théorique original (cf. chapitre 4).

La première vague a *formalisé* le problème : le triplet de STRIPS de Fikes et Nilsson [Fikes & Nilsson 71] a corrigé le principal défaut de GPS [Newell & Simon 63], mais restait dans le cadre trop strict de la linéarité, aboutissant aux contorsions de Waldinger [Waldinger 75]. Le système NOAH d'Earl Sacerdoti [Sacerdoti 77] a inauguré la planification *non-linéaire*, deuxième vague constituée par NONLIN d'Austin Tate [Tate 77] et surtout par SIPE de David Wilkins [Wilkins 86] (parmi beaucoup d'autres systèmes). Le problème du contrôle de ces planificateurs se résolvait soit par un algorithme global lâche et beaucoup d'heuristiques (NOAH, NONLIN et surtout SIPE), soit en s'orientant vers l'abstrait par la mise en évidence de la *méta-planification*, avec le système MOLGEN de Stefik [Stefik 81], soit en se tournant vers le concret par la prise en compte de la *réactivité*, toujours latente dans tous mais érigée en principe vital dans les *plans universels* de Schoppers [Schoppers 87].

L'aspect numérique sera représenté par DEVISER de Vere [Vere 83], les approches types "base de données temporelle", par le TMM de Dean [Dean 85] et IxTeT de Ghalab [Ghallab & Mounir Alaoui 89].

Enfin, nous discuterons rapidement de l'ordonnancement via le système ISIS-X / OPIS-X de Mark Fox et ses collègues [Fox 83], principalement pour combattre l'idée reçue selon laquelle *planifier* et *ordonnancer* sont deux activités identiques.

3.1 Planification linéaire

3.1.1 Exécuteurs (Genèse)

Le premier exécuteur GPS Le *General Problem Solver* (GPS) de Newell et Simon [Newell & Simon 63] était capable, à partir d'un état courant et d'un but, d'évaluer une différence entre cet état et ce but et de la minimiser progressivement jusqu'à son annulation, par l'application successive d'opérateurs permettant d'atteindre un meilleur état (algorithme *moyens-fins*). GPS devait répéter à chaque application d'opérateur non seulement ce que cet opérateur a changé mais également ce qu'il n'a *pas* changé, mettant ainsi en évidence le *problème du cadre* (cf. 2.1.3).

La succession des opérateurs qui se sont activés peut être considéré comme un plan linéaire : GPS est un système exécuteur.

Les systèmes à règles de production Dans un moteur d'inférences (d'ordre 0, 0^+ ou 1, fonctionnant en chaînage avant, arrière ou mixte [Davis 77] [Hayes-Roth et coll. 83]), l'ensemble des faits connus à un instant donné est incrémentalement augmenté par le déclenchement de règles. La succession des règles qui se sont déclenchées au cours d'une session, peut être considérée comme une succession d'actions transformant une situation initiale en une situation finale. Il ne s'agit cependant que d'exécution (le plan est visible a posteriori uniquement). De plus, dans un moteur d'inférences d'ordre 0 et 0^+, la valeur d'un prédicat, une fois connue, ne peut généralement pas changer (monotonie), ce qui simplifie fortement les capacités planificatrices du moteur.

3.1.2 Simulateurs

Levée de voile sur STRIPS Pour lutter contre le problème du cadre de GPS, Fikes et Nilsson ont conçu le programme STRIPS [Fikes & Nilsson 71] [Nilsson 80], qui présente les concepts de 1.2.1 permettant de définir un *problème de planification linéaire*. Les actions, en particulier, sont formalisées par un triplet, noté (π, α, δ) :

- liste de préconditions[1] (*precondition list* π) : liste de formules qui doivent être vraies dans la situation entrante pour que l'action s'effectue ;
- liste de formules ajoutées[2] (*add list* α) : liste des formules qui doivent être ajoutées à la situation entrante, lors du calcul de la situation sortante ;
- liste de formules enlevées[3] (*delete list* δ) : liste des formules de la situation entrante qui sont détruites, lors du calcul de la situation sortante.

[1] Ou *prémisse*, ou *sous-but*, ou *prérequis*.
[2] Ou *conclusions positives*.
[3] Ou *conclusions négatives*, ou *limitations*.

Pouvant ainsi simuler l'exécution d'une action, STRIPS recherche un chemin-solution en parcourant le graphe des états en chaînage avant ou arrière tout en ne mémorisant à un instant donné que le chemin qui mène de la racine (situation initiale ou finale) à l'état courant (planification linéaire). En appelant I la situation initiale et F la situation finale, l'algorithme (classique) de contrôle de STRIPS est le suivant [Nilsson 80] :

Procédure RÉGRESSER1 (F)
jusqu'à ce que I subsume F faire
- **choisir** une formule f de F ne s'unifiant à aucune formule de I
- **choisir** un schéma d'action s dont la liste de retrait contient une formule s'unifiant à g
- soit p la liste des préconditions de l'action a, l'instance de s générée ci-dessus
- appeler RÉGRESSER1(p) pour résoudre récursivement le sous-problème
- affecter l'état sortant de l'action a à I

Les deux **choisir** sont des nœuds "OU" du graphe des états. Cet algorithme est un cas particulier de l'algorithme MOYENS-FINS (cf. paragraphe précédent) dans lequel la *différence* est définie par *"l'ensemble des formules de F qui ne s'unifient pas à des formules de I"*. Les couples *formule postcondition d'une action / même formule précondition d'une action postérieure* (appelés aussi *liens de dépendances causales*) sont stockés dans la *table triangulaire*, structure permettant d'assurer une réactivité minimale lorsque le planificateur se fait robot-exécuteur (système PLANEX [Fikes et coll. 72]) ; cette technique a été récemment réutilisée pour produire des explications sur le déclenchement de règles [Picardat 87].

Selon le classement de 1.2.2, STRIPS est un planificateur-exécuteur linéaire : les appellations *"formules ajoutées"* et *"formules enlevées"* se réfèrent à la situation courante (effectivement définissable en planification linéaire) en des termes similaires au `assert` et au `retract` de Prolog[4] [Clocksin & Mellish 81]. La formulation STRIPS constitue l'un des principaux paradigmes de la planification (presque tous les systèmes décrits dans le présent chapitre s'y réfèrent explicitement) et même de la résolution de problème : une des (multiples) façons d'étendre STRIPS est, par exemple, de le rendre *hiérarchique* (c'est le programme ABSTRIPS, sur lequel avait initialement travaillé Sacerdoti, avant de concevoir NOAH [Sacerdoti 77], cf. 3.2).

Par rapport à GPS, son énorme avantage face au problème du cadre vient d'une hypothèse que l'on peut formuler ainsi :

Hypothèse STRIPS :

Tout ce qui n'est pas explicitement déclaré faux est considéré comme étant vrai.

[4] Formulation manifestement dans l'air du temps, puisque Colmerauer concevait Prolog à la même époque (rapporté dans [Colmerauer 84]).

C'est-à-dire : par défaut, rien ne change ; une action n'effectue qu'une modification *incrémentale* de son environnement.

En considérant STRIPS comme une théorie en logique des prédicats du premier ordre (au sens de 2.3.1), la sémantique ainsi définie n'est cependant pas *valide* pour deux raisons : (1) la véracité de formules *non atomiques* n'est pas assurée [Lifschitz 86] et (2) le micro-monde dans lequel évolue le robot-planificateur obéit lui aussi à sa propre théorie[5], que ne modélise pas la définition initiale de l'effet d'une action sur une situation ; la modélisation du comportement propre du micro-monde nécessite d'écrire un démonstrateur formel supplémentaire (le programme Disprover de [Siklòssy & Roach 75]) ou tout au moins de choisir quel sur-monde *cohérent* est le plus proche d'une situation sortante (théorie des mondes possibles de [Ginsberg 86]).

Modification du squelette de Waldinger Waldinger a développé dans [Waldinger 75] une technique de modification d'un plan linéaire pour achever plusieurs buts, technique issue de la génération automatique de programmes (un programme est considéré comme un plan linéaire[6]). Devant réaliser deux buts A et B, son planificateur expérimental construit d'abord un plan linéaire $P\ \{P_1, \ldots, P_n\}$ réalisant A, puis cherche à modifier P pour qu'il réalise B tout en continuant à réaliser A. Pour cela, le but B est placé après P_n ; le planificateur cherche des conditions *suffisantes* successives réalisant B (*régression de B le long de P*) et pouvant être placées avant $P_n, \ldots, P_1$ et ce tant que la condition suffisante courante détruit la logique du plan P i-e sans violer les *prédicats protégés* de $P_1, \ldots, P_n$ (un prédicat p protégé dans $[A_i, A_{i+1}]$ interdit tout ajout d'action entre A_i et A_{i+1} qui rend p faux).

L'algorithme est le suivant :

Procédure RÉGRESSER2(B_i)
  si B_i détruit un prédicat protégé de A_{n-i},
  alors si $i = n+1$, **alors** ECHEC (il n'y a plus d'action A_k sur laquelle régresser B_{n+1}),
    sinon calculer B_{i+1} condition nécessaire avant A_{n-i} pour que B_i soit vrai après A_{n-i}.
      appeler RÉGRESSER2(B_{i+1}).
    sinon ajouter B_i entre A_{n-i} et A_{n-i+1}. SUCCES

La régression est initialement lancée par RÉGRESSER2(B_0). Si B ne peut pas être régressé le long de P (construit pour réaliser A), le planificateur échange B et A et reprend la construction du plan linéaire initial (satisfaisant B et non plus A, cette fois). L'échec global n'est prononcé que lorsque toutes les régressions sur tous les buts ont échoué.

La critique du système de Waldinger (1975) est aisée à faire aux lumières des concepts actuels. D'abord la linéarité du plan crée une combinatoire pour le choix du plan linéaire initialement développé, car la symétrie de traitement entre les buts initiaux A et B ne peut pas être conservée (aucun moyen de noter une *protection* entre les actions B_i). Ensuite, la *protection* d'un prédicat n'a pas d'extension dans le temps : elle ne court que sur un intervalle unité $[A_i, A_{i+1}]$, alors

[5] Dans l'exemple de 7.1, nous avons précisément pris soin de formaliser le comportement propre du micro-monde.

[6] Ni Ada, ni Occam n'existaient à l'époque.

qu'on peut tout à fait trouver des exemples pour lesquels un intervalle plus long est requis (cf. les intervalles de valeur de [Tate 77]). De plus, il y a manifestement confusion entre *protection* et *précondition* : toute précondition doit être *protégée* au moins sur un intervalle de longueur unité (le précédant), mais il existe des prédicats non préconditions immédiates qui doivent être protégés sur de grandes longueurs (cf. le problème de la préservation des prédicats de haut niveau hiérarchique de NOAH [Sacerdoti 77]). Enfin et surtout, Waldinger reste muet sur la façon de régresser effectivement un prédicat par dessus une action (quand cette régression est différente de l'égalité) : comment peut-on déduire les conditions nécessaires B_{i+1} à partir des conditions B_i et de l'action A_{n-i} [Chapman 85], puisqu'on ne sait pas comment deux prédicats sont affectés l'un par l'autre ?

3.2 Planification non linéaire

Deux types de planificateurs non-linéaires sont distingués :

- ceux qui visent à une certaine généralité (*planificateurs indépendants du domaine*). Le raisonnement de ces planificateurs se base sur une description fine des actions intervenantes (preque toujours héritée du triplet STRIPS (π, α, δ)) ;
- ceux qui sont dédiés à un problème donné (*planificateurs dépendants du domaine, ou à expertise*). La description des actions et la structure de contrôle sont particulières au problème.

Nous présentons brièvement les principaux représentant de chaque type.

3.2.1 Planificateurs indépendants du domaine

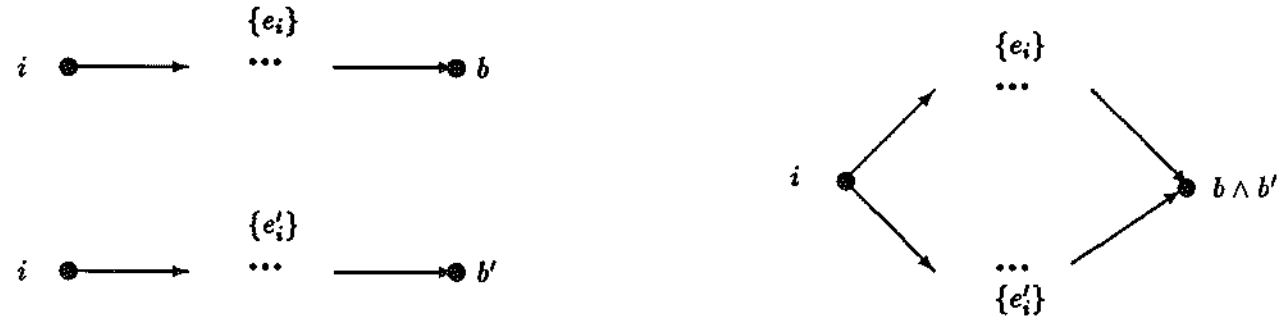
La lignée des planificateurs indépendants du domaine a été inaugurée par le système NOAH de Sacerdoti. Nous présentons deux de ses principaux successeurs directs : le système NONLIN de Tate et le système, plus récent, SIPE de Wilkins.

La non-linéarité dans NOAH Ayant mis en évidence l'intérêt de la *hiérarchie* dans ABSTRIPS, Sacerdoti (fondateur du domaine de la planification en I.A.) s'est ensuite intéressé à la *non-linéarité* avec NOAH (*Net Of Action Hierarchies*), planificateur opérationnel aidant un quidam faillible à démonter une machine [Sacerdoti 77] (ou [Sacerdoti 75] pour une version compactée).

Jusqu'alors, on ne connaissait que les systèmes exécuteurs (systèmes de résolution de problèmes comme GPS en 3.1.1) et on venait de trouver un moyen de simuler une exécution linéaire en représentant un chemin (plan linéaire) menant d'un état initial à un but b (la table triangulaire STRIPS, cf 3.1.2). Pour atteindre un but conjonctif $b \wedge b'$, on remarquait d'abord que la recherche d'un chemin y menant directement est exponentiellement plus complexe[7] que la recherche d'un

[7] Remarquons que ce surplus exponentiel de complexité n'est valable que pour la conjonction $\wedge$ de buts : la recherche d'un chemin menant à la *négation* d'un but $\neg b$ (resp. *disjonction* de buts $b \vee b'$) est de même complexité

chemin menant à un but atomique (b, ou b') ; aussi le chemin menant à $b \wedge b'$ était recherché, à partir du chemin menant à b ou du chemin menant à b', par modifications successives, en tentant d'intégrer l'un dans l'autre et en conservant toujours un plan linéaire (cf. la discussion de 3.1.2 sur la régression de Waldinger). L'originalité de Sacerdoti fut précisément de représenter le chemin menant au (sous-)but b et celui menant au (sous-)but b' *dans un même graphe* (voir figure 3.1), le plan devenant du même coup *non-linéaire* (*réseau procédural*).



Deux plans linéaires — Un seul plan non-linéaire

Figure 3.1: Passage de la planification linéaire à la planification non-linéaire

Chez Sacerdoti, et dans tous les planificateurs non-linéaires, des actions (ou des branches) en parallèle doivent toujours être exécutées, mais *on ne sait pas encore dans quel ordre.* C'est une manifestation d'un principe plus général[8] :

> *Principe du moindre engagement* :
>
> Ne pas choisir, tant que l'on n'en sait pas assez pour le faire judicieusement.

Ce principe[9] (*least commitment approach*) prône de représenter le plus longtemps possible une *absence de choix*, et de ne choisir effectivement que lorsque l'information disponible est jugée suffisante (pour choisir en connaissance de cause) ou lorsqu'on y est forcé (auquel cas la maigre information disponible est déclarée suffisante). Le parallélisme de deux actions, par exemple, représente ici l'absence de choix sur l'ordre d'exécution de ces deux actions.

La difficulté de la représentation non-linéaire d'un plan provient de l'inter-dépendance des sous-buts b et b', qui se traduisent par des interactions contradictoires entre leurs chemins respectifs (conflit de ressources, ...). Sacerdoti utilise pour cela des règles de résolution (*critics*), empiriquement déduites d'exemples simples, qui se basent sur la comparaison des prédicats ajoutés et détruits entre les deux branches (*TOME, Table Of Multiple Effects*).

(resp. complexité double) que celle concernant un terme atomique b. Le dernier opérateur $\Rightarrow$ (cf. 2.3.1) se ramène aux opérateurs précédents ($b \Rightarrow b' \Leftrightarrow \neg b \vee b'$) est donc aussi de complexité double.

[8] On le retrouve aussi bien en manipulation formelle d'équations [Laurière 86] qu'en traitement du Langage Naturel (désambiguïsation par "synchronisation" [Constant 91]).

[9] Appelé aussi *comportement paresseux*.

Le principe de la stratégie de résolution est grossièrement basé sur un cycle de chaînage arrière :

```
Procédure PLANIFIER1
   tant que il existe un sous-but non satisfait, faire
      choisir un sous-but b
      satisfaire ce sous-but particulier par ajout d'action ou rebranchement
      tant que il existe un conflit, faire
         choisir un conflit à résoudre
         choisir une règle de résolution
         appliquer cette règle à ce conflit
```

La procédure PLANIFIER1 tente de résoudre un à un tous les sous-buts et de résoudre un à un tous les conflits qu'entraine cette résolution. Résoudre un conflit, résultant de la satisfaction d'un sous-but, génère à son tour d'autres sous-buts, aussi la convergence (et la terminaison) de PLANIFIER1 n'est assurée qu'empiriquement (i-e ne l'est pas, rigoureusement), en espèrant que la structure de contrôle profitera des liberté de choix (les termes "**choisir**") offertes par PLANIFIER1 :

- trier les sous-buts (resp. les conflits) pour choisir de satisfaire (resp. résoudre) les plus urgents en premier (par *hiérarchie*, par exemple) ;
- trier les schémas d'actions (resp. règles de résolution de conflits) satisfaisant (resp. résolvant) le sous-but (resp conflit) élu.

NOAH privilégie la satisfaction des sous-buts qui résolvent en même temps des conflits (et inversement). Nous verrons au chapitre 4 que ces deux activités sont effectivement identiques.

Par rapport à l'algorithme RÉGRESSER1 de STRIPS, PLANIFIER1 a la liberté de choisir le sous-but à satisfaire (parce que le plan est complètement représenté et non exécuté), peut satisfaire un sous-but par rebranchement et non uniquement par ajout d'action (parce que le plan est non-linéaire), et surtout doit inclure un cycle complet de résolution de conflits (ce qui n'a aucun sens pour STRIPS).

L'algorithme PLANIFIER1 est souvent résumé par le cycle :

1. Choisir un sous-but
2. Satisfaire ce sous-but
3. Résoudre les conflits générés

Que ce soit pour choisir le sous-but à satisfaire ou le conflit à résoudre, le planificateur doit être capable de déduire des informations du plan : le chapitre 5 y sera consacré. De même satisfaire un sous-but ou résoudre un conflit exige de pouvoir induire des informations sur le plan : le chapitre 6 sera consacré à cette structure de controle.

NOAH fonctionne principalement en ordre 0 et tente d'introduire des pseudo-variables (*formel objects*), mal gérées donc vite instanciées. La mise en application du principe du moindre engagement suggèrerait d'interpréter une variable comme l'absence de choix d'une constante, censée représenter un objet. Mais Sacerdoti n'a pas l'outil théorique pour gérer des variables (cf. 4.3). Nous présenterons au chapitre 4 une formalisation avec variable de NOAH, qui nous permettra de construire en partie II un planificateur d'ordre 1 à part entière.

Les modifications lentes de NONLIN Le planificateur NONLIN[10] (*NON LINear*) de Tate [Tate 77] est un des héritiers directs de NOAH (et a été récemment révisé dans OPLAN [Tate 86]). Visant des applications industrielles (la révision de turbines électriques), NONLIN améliore les performances de NOAH grâce à deux structures de données supplémentaires.

Une première structure concerne le lien entre un sous-but et l'action introduite initialement pour le satisfaire (lien de causalité ou *protection* selon 3.1.2), lien que NOAH ne conserve pas, aussi il doit recalculer à chaque cycle la satisfaction de tous les sous-buts ... et souvent redécouvrir qu'il y avait bien cette même action satisfaisant ce même sous-but. Cet option (recalcul) suppose que le graphe évolue tellement rapidement que le planificateur passerait plus de temps à gérer l'ajout et le retrait de ces liens qu'à se préoccuper de planification proprement-dite. NONLIN suppose au contraire que le plan évolue lentement et que l'information qu'apporte ce lien est supérieure à la perte de place qu'il entraîne. Il garde donc explicitement ces liens de causalité (intervalle de valeur, *value range*) dans un deuxième graphe (GOST, *GOal STructure*), extrait du graphe représentant le plan. Ce réseau des influences facilite en outre le repérage des conflits (selon le vocabulaire du chapitre 4, savoir si un masqueur s'intercale entre un établisseur et le sous-but s'effectue en regardant si ce masqueur appartient ou non à un des liens causaux) en subsumant la notion de *protection* de Waldinger (cf. 3.1.2).

Pour lutter contre la *"tyrannie du détail"*, NOAH distingue des *niveaux hiérarchiques*, correspondant en pratique à un tri des sous-buts à satisfaire et des conflits à résoudre : plus le niveau hiérarchique est élevé, moins il y a de détails. Mais faire fonctionner PLANIFIER1 *de façon autonome* sur un même niveau hiérarchique peut détruire les constructions prévues au niveau hiérarchique supérieur (exemple dans [Sacerdoti 77] du singe, des clés et des bananes). NOAH gère cette interaction entre niveaux hiérarchiques par la pose, sur chaque niveau hiérarchique, de points de contrôle (*hierarchical kernels*) issus du niveau hiérarchique supérieur.

Cette gestion hiérarchique rigidement descendante, pyramidale, nie l'importance de la remontée de l'information, pourtant révélatrice des conflits liés au détail du terrain, dans le processus global de résolution. NONLIN propose une vision, socialement plus acceptable, distinguant différents types de points de contrôle : *supervised* (immédiatement vérifié par le niveau hiérarchique inférieur), mais aussi *holds* (immédiatement vérifié, mais par n'importe quelle action), et *unsupervised* (non contrôlé par le niveau supérieur, juste vérifié à la fin du processus).

Enfin, NOAH ne peut pas effectuer de retour-arrière (choix irréversibles), ce qui explique la présence de son *principe du moindre engagement*, alors que NONLIN, tout en conservant ce principe, note les points de choix et explore systématiquement toutes les branches de recherche par retour-arrière.

[10] Le code source Common-Lisp de NONLIN est depuis peu dans le domaine public. Il peut être obtenu via le réseau Internet par `ftp` anonymous (directory /pub/nonlin) à l'université de Maryland (`cs.umd.edu`, avec serveur de noms, ou au 128.8.128.8 directement) [Hendler 91].

Le pragmatisme de SIPE Wilkins a développé dans SIPE (*System for Interactive Planning and Execution monitoring*) un système planifiant la mission de porteurs d'engins [Wilkins 84], [Wilkins 85], [Wilkins 86], et un résumé synthétique en [Wilkins 88]. SIPE améliore aussi notablement les performances de NOAH grâce aux concepts suivants :

- Wilkins définit les conflits à partir des ressources : une ressource est un objet ne pouvant être utilisé que par *une seule action* à la fois ; les conflits sont détectés uniquement par la consommation multiple de ces ressources.
- D'un point de vue formel, les pseudo-variables de NOAH (*formal objects*) sont implicitement quantifiées existentiellement (ce point sera discuté en 5.2.2) : $sur(a, x)$ signifie $\exists x$, $sur(a, x)$. Wilkins montre sur des exemples l'intérêt de la quantification *universelle* explicite des variables, et définit une batterie de contraintes prenant en compte ce $\forall$.
- Wilkins remarque ensuite la rigidité de la description des actions dans NOAH et en particulier le manque de *conclusions conditionnelles* : en NOAH, une conclusion est soit positive (c), soit négative ($\neg c$), mais est toujours présente (i-e n'est jamais du type *contexte* $\Rightarrow c$) ; cette absence nécessite d'écrire, à partir d'un schéma d'actions sémantiquement identifié, autant de schémas d'actions que de conjugaisons de conclusions possibles. Wilkins propose l'option inverse, dans laquelle une action unique en réalité correspond aussi à un schéma d'actions unique, mais dont certaines conclusions, jugées dépendantes du contexte, sont précisées par les *opérateurs déductifs* (cf. un exemple en 7.1).

Le point de vue de Wilkins est résolument pratique et heuristique, la démonstration formelle de la calculabilité de la valeur d'un terme après une action n'est jamais effectuée. Nous verrons au chapitre 4 que la prise en compte, par exemple, des conclusions conditionnelles, via les opérateurs déductifs, rend le problème du *calcul d'une situation* NP-complet (dans le cas général).

3.2.2 Planification à expertise

L'heuristique stabilisatrice de GARI Descottes a développé dans GARI un planificateur qui indique la suite d'opérations (coupes, mais aussi taraudage, polissage, ...) à effectuer dans un bloc de métal pour obtenir une pièce usinée finie, à partir de la donnée de sa géométrie (une représentation du plan de la pièce, avec *faces, trous tauraudés ou non, fentes*, ...)
[Descottes & Latombe 85].

GARI génère d'abord un plan faiblement contraint, composé de coupes grossières et fines. Puis il *ordonne* et *détaille* ces actions en utilisant l'expertise de planification représentée sous forme de règles d'ordre 1 : la partie gauche de ces règles concerne le but, i-e la géométrie de la pièce, les machines disponibles, ... ; la partie droite fournit une indication sur l'ordre des actions ou sur les attributs de ces actions.

Le fonctionnement global de GARI consiste à contraindre le plan initial grossier, en déclenchant toutes les règles de planification applicables. A un instant donné, plusieurs règles de planification sont déclenchables, ce qui peut mener GARI à des contraintes contradictoires : une action doit être à la fois avant et après une autre, ou bien l'ensemble des machines pouvant

réaliser une action est vide. Il faut alors trouver une contrainte à relaxer, i-e la règle de planification coupable à désactiver. Un retour-arrière chronologique, bien que très facilement codable, désactiverait non seulement la règle coupable, mais aussi toutes celles qui se seraient déclenchées *après* cette règle jusqu'à l'instant du conflit. Inversement, utiliser un retour-arrière dirigé par les dépendances serait certes très efficace pour trouver la règle coupable, mais serait beaucoup trop coûteux en place mémoire[11] (nous retrouverons ce dilemne en 6.2).

L'originalité de GARI repose sur une heuristique de choix de cette règle de planification coupable de conflit : l'expert (humain) attribue à chaque règle de planification un coefficient traduisant l'importance qu'il accorde à l'action de la partie droite (le *coefficient de vraisemblance* des systèmes-experts, mais sans l'aspect pseudo-probabiliste) dans l'ensemble des règles déclenchables, la règle de coefficient maximale (i-e la plus importante) est déclenchée ; Descottes prouve alors que la règle responsable d'un conflit appartient à l'ensemble des règles déclenchées à *coefficient minimum* ; GARI désactive la règle la plus récente de cet ensemble en construisant sa négation, en calculant son coefficient pour ne pas boucler et en la déclenchant en lieu et place de la cette règle coupable. Le conflit est alors levé, le processus de planification peut continuer. Cette heuristique de choix des règles coupables de conflits a été systématisée dans le système PROPEL [Tsang 87].

Comparer GARI aux planificateurs indépendants du domaine de 3.2.1 mesure sur un exemple l'effet de l'introduction d'une expertise dans un planificateur : d'abord la modélisation fine des actions en prémisses/conclusions, permettant de calculer la transformation de la situation entrante, n'a plus lieu d'être ici puisque c'est maintenant l'*expertise* qui indique *pourquoi* il faut effectuer telle action avant telle autre, ou *pourquoi* il faut donner telle valeur à tel attribut de telle action (l'expertise remplace ou plutôt compile le calcul des situations). Ensuite, la décision d'intégrer ou non une action dans le plan ne provient pas de la satisfaction d'un sous-but (puisqu'il n'y a plus ni premisse, ni conclusion), mais directement de la représentation de la pièce usinée à construire, le problème étant d'ordonner et de préciser ces actions. Enfin, ces règles de planification peuvent s'interpréter comme des règles de résolution de conflits (*critics*) de NOAH (qui laisse d'ailleurs la porte ouverte à des *critics* particuliers au problème).

Nous retrouverons ces différences entre planificateur à expertise et planificateur indépendant du domaine (remplacement du calcul sur les actions avec prémisses et conclusions par l'expertise, intégration des actions basée sur la description de l'objet à construire, ...) dans le planificateur (à expertise) COPLANER, que nous présenterons en partie III.

3.3 Autres principes de planification

3.3.1 Planification numérique

Imagerie temporelle (TMM) La partie déductive des planificateurs de 3.2 est isolée et formalisée par Thomas Dean, qui considère un plan comme une *base de données temporelle* ([Dean 85], publication originale, filtrée et corrigée dans [Dean & McDermott 87] ; présentation

[11] *"En fait, lorsque nous sommes passé du retour-arrière dirigé par les dépendances à la méthode courante, nous avons négocié un grand graphe d'inférences contre un risque raisonnable de revenir sur des points de choix non impliqués dans le conflit."* [Descottes & Latombe 85, p. 202].

"grand public" dans [Dean 86]). L'aspect uniforme d'une dimension spatiale et d'une dimension temporelle[12] justifie l'appellation de *carte du temps*[13], ou d'*imagerie temporelle*, d'une telle base, le système de requète étant alors un *gestionnaire de carte du temps* (TMM, *Time Map manager*).

Un entrée y est non plus un fait, mais un évènement (fait *et* relation temporelle, délimitant l'intervalle de temps pendant lequel l'événement est valide). Les liens de précédence sont ici valués ; un instant numérique (ou, pour prendre en compte l'imprécision, un intervalle numérique le contenant à coup sûr) peut être évalué pour chaque événement, et l'utilisateur peut définir des règles de comportement (conflits numériques de type PERT) à partir de ces évaluations. Les dépendances positives ou négatives entre événements sont représentées en pratique par des actions de type STRIPS ; l'utilisateur dispose cependant d'un sur-langage lui évitant ce formalisme strict. La difficulté (et l'intérêt) d'un TMM provient de la façon de calculer la *persistance* d'un événement (atomique ou composé), i-e l'intervalle de temps numérique pendant lequel l'événement considéré est vrai[14].

L'aspect algorithmique d'un TMM est présenté dans [Dean & Boddy 88] (comme nous le verrons en 4, ces algorithmes sont basés sur un *critère de vérité* en ordre 0).

Le système IxTeT de Ghallab [Ghallab 86] [Ghallab et coll. 88] [Ghallab & Mounir Alaoui 89], héritier des TMMs de Dean, évalue les relations temporelles entre les tâches décrivant les mouvements d'un robot (caméras d'une sonde spatiale de type Spot) de façon suffisamment performante pour être réactif : la déduction ou l'induction de relations temporelles entre instants (les relations entre intervalles, de type Allen, sont transformées en des relations relatifs à leurs extrémités) s'effectue de façon linéaire, grâce à la gestion d'un emploi du temps (*Time Table*) référençant les prédicats introduits par les tâches. Comme le système de Dean, IxTeT fonctionne en ordre 0 et ne se concentre pas sur la *génération* proprement dite du plan.

Fenêtres temporelles de Deviser Le système DEVISER de Vere détermine les fenêtres de temps pendant lesquelles une sonde Voyager peut prendre des photographies d'une planète [Vere 83]. Le concept central de DEVISER est la *fenêtre temporelle*, constituée d'une date de début et d'une durée (persistance), qui caractérise l'aspect temporel d'une *action* (transformation du monde), d'un *événement* (ajout de faits déduits des lois de comportement du monde) ou de *règles d'inférences* (ajout de faits issus des lois de comportement statique du monde) ; le plan généré est alors directement au format PERT (cf. 2.2).

L'aspect purement causal de la dépendance entre tâches dans DEVISER est traité comme en NONLIN (inspiration explicite). Sur ce planificateur de base est greffé le module de traitement des fenêtres temporelles (4000 lignes d'Interlisp pour DEVISER vs. 3300 lignes utiles de Common-Lisp pour NONLIN), qui fait l'originalité de ce système. A chaque fait temporel (action, événement

[12] *"L'image intuitive qui doit venir à l'esprit à propos du processus d'imagerie temporelle est celle de la construction de cartes, cartes du type de celles qui sont utilisées pour tracer des informations spatiales, et celle du balayage de ces cartes pour en extraire des informations et y repérer des formes. Selon beaucoup de points de vue, le temps se comporte comme l'espace. Les événements dans le temps sont comme des objets d'un espace à une dimension."* [Dean 85, p. 3], ou [Dean & McDermott 87, p. 2].

[13] Dean pousse l'analogie jusqu'au bout : *"C'est comme si tout ce que vous savez du passé, du présent et du futur était mis à plat devant vous."*, [Dean & McDermott 87, p. 3]. Il retrouve en cela l'approche, beaucoup plus ancienne, des Mégariques (cf. 2.5).

[14] Influence évident de la logique des instants de McDermott, cf. 2.6.2.

ou inférence) est associé une fenêtre temporelle (initialement à $(0, +\infty)$) qui est progressivement diminuée en fonction des autres contraintes. Une inconsistance du plan se produit lorsqu'une fenêtre est inférieure à la durée de son fait temporel hôte : le système effectue alors un retour-arrière.

3.3.2 Méta-planification

Les interprètes emboités de MOLGEN Le système MOLGEN (*MOLecular GENetics*) de Mark Stefik aide les généticiens d'un laboratoire à planifier leurs expériences [Stefik 81]. MOLGEN construit un plan en tirant parti de la hiérarchisation des objets et des opérations manipulés, les interactions (verticales) entre niveaux hiérarchiques étant gérées par la propagation verticale de contraintes : c'est la technique de l'*attachement des contraintes* (*constraint posting*), qui consiste à poser physiquement l'objet représentant une contrainte[15] sur chacun des objets qu'elle relie. Stefik utilise également la propagation de contraintes pour gérer les interactions entre sous-problèmes à l'intérieur d'un niveau hiérarchique. Mais l'aspect qui nous concerne ici est la décomposition du planificateur en plusieurs niveaux de planification.

La couche la plus basse (espace du laboratoire) représente les objets (hélice d'ADN, enzyme, ...) et opérations (hybrider, amplifier, faire-réagir, trier) physiques du laboratoire.

La couche supérieure (espace de conception) décide de l'organisation des objets et des opérations dans l'espace du laboratoire, au moyen d'opérateurs de *comparaison* (calculer la différence entre les moyens et le but), d'*extension temporelle* (étendre le plan courant vers la situation initiale ou vers la situation finale) et de *spécialisation* (détailler une opération ou un objet et propager les contraintes vers les niveaux hiérarchiques inférieurs).

La troisième couche (espace stratégique) détermine la stratégie instantanée du planificateur : d'abord principe du moindre engagement, au moyen des opérateurs de *focalisation* (demande à chaque opérateur de décision de trouver un point d'application dans le plan, en choisit un et lui demande de s'éxecuter), de *reprise* (choisit un opérateur sans lui demander de s'évaluer, et lui demande de s'exécuter) ; mais aussi stratégies heuristiques, lorsqu'il faut effectivement s'*engager* pour faire progresser la résolution, au moyen des opérateurs d'*engagement* (demande aux opérateurs de conception d'évaluer leurs choix, en choisit un et l'exécute) et de *retour-arrière* (cherche les opérateurs de conception dont l'application peut avoir généré le conflit, en choisit un et l'exécute).

La dernière couche est constituée par un automate d'états (interprète aveugle) qui exécute les opérateurs de la couche stratégique : tant qu'une inférence est possible, enchaîner les action de *focalisation* et *reprise* (cycle du moindre engagement) ; sinon ajouter heuristiquement une contrainte par l'action *engagement*, ou en retirer une, lors de conflits, par l'action *retour-arrière*, et reprendre le cycle de moindre engagement dans les deux cas.

En interprétant les niveaux de planification MOLGEN selon le vocabulaire du chapitre 1, on retrouve le fonctionnement de type "moindre engagement" de l'algorithme PLANIFIER1, avec en plus un retour-arrière lorsqu'un sous-but (resp. un conflit) ne peut pas être satisfait (resp.

[15] Voir SketchPad [Steele & Sussman 79] pour un exemple de raisonnement sur des contraintes considérées comme des objets à part entière.

résolu) (cf. 3.2.1) : la première couche définit le problème de planification (schémas d'actions, situation initiale, situation finale) en ordre 1 (un *objet partiellement instancié* est une variable sous contraintes). Après avoir considéré les prémisses non satisfaites et les conflits non résolus (*"comparaison"*), ce planificateur peut soit ajouter une action ou des contraintes de précédences (*"extension temporelle"*), soit ajouter des contraintes sur les variables (*"spécialisation"*). Le cycle standard de PLANIFIER1, qui observe / modifie le plan courant (exécution alternée de *"focalisation"* et *"reprise"*), peut nécessiter un retour-arrière (*"retour-arrière"*) lorsqu'il n'y a plus aucun moyen de satisfaire (resp. résoudre) une prémisse (resp. un conflit). L'importance du choix de l'action satisfaisant une prémisse ou résolvant un conflit est ici clairement exhibée, puisqu'elle dépend de l'opérateur "engager" remonté jusqu'à la couche stratégique[16].

MOLGEN décompose l'algorithme PLANIFIER1 et répartit ses composants *dans un même formalisme* sur deux couches (n. 2 et n. 3, la 4^e couche n'étant qu'un artifice de présentation). Aucune couche de méta-planification n'est *réflexive*[17], mais Stefik s'est au moins débarrassé des *agendas* avec ses interprètes virtuels emboités. Le contrôle procédural figé de PLANIFIER1 disparaît au profit d'une hiérarchie d'entités de contrôle déclaratives souples, qui s'échangent des messages ...L'influence des *sources de connaissances* des *tableaux noirs* (*blackboard*) est évidente [Hayes-Roth 85], et on voit même poindre la notion d'*acteur* [Ferber 89].

3.3.3 Ordonnancement

La série ISIS / OPIS L'archétype du système d'ordonnancement[18] est ISIS (*Intelligent Scheduling and Information Systems*), construit par Mark Fox et ses collègues [Fox 83], [Fox et coll. 83], qui indique, dans une usine de fabrication de pales de turbines, la pale que doit traiter chaque machine à chaque instant pour honorer les commandes arrivantes. Il s'agit bien ici d'*ordonnancement*[19] et non de *planification* : le problème n'est pas de découvrir une séquence d'actions qui réalise une pale de turbine finie, mais, à partir d'un processus industriel connu (i-e d'une séquence d'actions déjà élucidée), d'allouer une machine à une pale partielle à chaque instant, de façon à optimiser des critères variés, tels que le retard de certaines commandes urgentes, le retard moyen des commandes passées à l'usine sur plusieurs années ou le temps d'inactivité de chaque machine. La planification se situe en amont de l'ordonnancement [Smith 87].

Le cycle de fonctionnement d'ISIS est composé des quatres étapes suivantes :

1. une priorité est d'abord associée à chaque commande non traitée et la commande la plus prioritaire est traitée.
2. Puis, à partir de la séquence d'actions (connues) réalisant la commande, une première passe sur les machines détecte les futures machines surchargées (*goulots d'étranglement*)

[16] Importance également repérée par Wilkins, qui essaie aussi de ne choisir l'action détaillée à appliquer que le plus tard possible avec le maximum d'informations : une action est en fait un squelette d'action, dont la description est complétée, lorsque demandée, par les opérateurs déductifs. Mais même ce choix de squelette est litigieux, puisqu'il est éclairé par le champs *déclencheur* [Wilkins 84].

[17] Mais Stefik y a évidemment pensé : *"L'idée des couches [qui pilotent l'exécution de la couche immédiatement inférieure] ne se limite pas à deux niveaux ; elle peut être appliquée récursivement. Pour réduire la complexité apparente d'un système, des couches peuvent être empilées jusqu'à ce que la connaissance subsistant dans l'interprète de la plus haute couche soit évidente [à traiter]."* [Stefik 81, p. 148].

[18] *Job schop scheduling problem.*

[19] "Ensemble des processus de mise en œuvre et de contrôle d'une commande", *Petit Robert, 1987.*

et détermine grossièrement les dates de début au plus tôt et de fin au plus tard de la commande.

3. ISIS cherche ensuite à associer une machine à chaque action de la séquence réalisant la commande déterminée : il détermine d'abord l'action qui sera le point de départ de la recherche, le sens de cette recherche (vers le futur ou vers le passé) et la stratégie d'allocation (réservation au plus tôt d'une machine, ou principe du moindre engagement et laisser choisir l'étape suivante) ; puis s'effectue la recherche proprement-dite, recherche top-down *par faisceau* sur des états action / machine / position (dans la file d'attente de cette machine) ; l'évaluation d'un état est basé sur la fonction d'évaluation des contraintes de tout type qui s'appliquent sur cette action / machine / position ; ISIS diagnostique ensuite la qualité de l'état déterminé grâce à un système de règles, ce qui peut modifier les paramètres de la recherche (sens et stratégie) et peut même amener à un retour-arrière (soit par une interprétation du graphe de recherche en terme de lien de causalité, soit en tirant parti de l'aspect *hiérarchique* des contraintes).

4. une dernière phase effectue la réservation pratique des machines pour la commande traitée et en profite pour minimiser le temps d'attente d'une machine par une pale de turbine.

Etant donné l'énorme volume d'informations traitées, ISIS utilise le langage de frame SRL [Fox et coll. 85] (ancêtre de Knowledge Craft) pour représenter tout concept : machine, commande ou pale de turbine, mais aussi contrainte (*attachement de contraintes* à la MOLGEN [LePape & Smith 87]) et temps (basé sur la logique réifiée de Allen [Smith 83], cf. 2.6.1). Plusieurs variations ont tenté de remédier au temps prohibitif de traitement[20] : ISIS-2, ISIS-3 rebaptisé OPIS-0 (*Opportunistic Intelligent Scheduler*, [Smith et coll. 86], [Si Ow 86]).

Dérivés d'ISIS Le système SOJA (*Système d'Ordonnancement Journalier d'Atelier*), fortement inspiré d'OPIS[21], prend en compte un niveau de détail très élevé (jusqu'au mouvement des chariots dans l'atelier) et construit progressivement un ordonnancement admissible avec des contraintes temporelles sous forme d'inégalités [LePape 85].

Le système OPAL est un outil général d'ordonnancement, qui détermine l'ordre dans lequel vont être effectuées, sur les moyens de production, les opérations nécessaires à la réalisation de la production souhaitée, de façon à satisfaire les contraintes de ces moyens de production et de cet objectif de production. Son originalité réside dans l'introduction de règles expertes (utilisant la théorie des ensembles flous [Dubois & Prade 85]), qui complètent la solution admissible issue de la satisfaction de contraintes PERT étendues (cf. 2.2) [Bel et coll. 86] (description plus complète dans [Bel et coll. 88] ou [Bensana et coll. 88]).

Esquirol considère un problème d'ordonnancement comme un problème purement numérique d'analyse de contraintes, et modélise en Prolog des limitations sur la consommation des ressources et sur les temps alloués [Esquirol 87]. Enfin, Parello analyse le cas de l'ordonnancement du montage d'options sur une chaîne de fabrication de voitures (et exhibe quelques heuristiques simples) pour valider son démonstrateur de théorèmes ITP (clauses de Horn) [Parello et coll. 86].

[20] 30 mn. sur un VAX 780 pour ordonnancer un ordre de fabrication d'une dizaine d'opérations (rapporté dans [Bel et coll. 88, p. 511]).

[21] C. Lepape a aussi travaillé sur OPIS [LePape & Smith 87], [Smith et coll. 86].

Enfin, les systèmes successifs PLATFORM I, II et III, de Levitt intègrent en un outil unique des capacités hybrides de planification et d'ordonnancement, pour la gestion de projets. Quatre niveaux sont explicitement distingués, la pose des objectifs, la planification du projet, son ordonnancement et son contrôle. Cependant, seuls des exemples sont présentés, constituant sans doute de bons tests de l'environnement de développement KEE (et, en particulier, de ses facilités graphiques) mais ne remplaçant pas une application en vraie grandeur.

3.3.4 Planification réactive

La difficulté de la planification réactive réside dans la gestion de l'incohérence du monde modélisé (simulation), résultant de la prise en compte du monde réel (exécuté). Les perturbations issues du monde réel obligent ces planificateurs réactifs à *replanifier une séquence d'actions*, en évitant la pire solution : tout replanifier depuis ce qui a été effectivement exécuté (l'instant présent, voir en 5.1.3). Au contraire, la manière d'identifier ce que ces perturbations rendent obsolètes dans le plan qualifie les différents types de réactivités.

Mettant en rapport l'exécution et la simulation des planificateurs, la rubrique "réactivité" pourrait concerner presque tous les systèmes de parcours de graphe, depuis les exécuteurs non-planificateurs jusqu'aux planificateurs non-exécuteurs : un moteur d'inférences de système-expert, par exemple, relève de l'exécution pure (non-planificatrice : adaptation à un cas réel à partir de la décomposition cognitive de cas typiques) ; un résolveur de problèmes (avec un algorithme de recherche de type A^*) est du type exécuteur / légèrement planificateur ; les systèmes de planification linéaires et non-linéaires précédents sont tous du type planificateur / légèrement exécuteurs ; d'autres systèmes théoriques (cf. chapitre 4) sont du type planificateur pur.

Planificateurs à replanification totale Ce type de planificateur autorise une réactivité en remettant en cause presque tout le plan construit.

Les *plans universels* de Schoppers [Schoppers 87] permettent de faire face à des actions de sabotage du monde réel[22]. A partir d'un but désiré, ce planificateur linéaire régresse des actions jusqu'à obtenir un sous-but réalisé (vérifié par des capteurs). Cette régression s'adapte à toute situation initiale, et en fait ne requiert pas de situation initiale figée, d'où l'aspect réactif. Etant donné la simplicité de ce mécanisme, ce système peut fonctionner en temps réel dans un environnement perturbé.

Le "monde" et sa logique propre ne sont pas modélisés : seules les réactions par rapport à des déviations le sont. Le contrôle n'existe pas : il est reporté sur la logique de ce monde extérieur, supposé cohérent. Ensuite, on voit mal quelle est la différence avec le fonctionnement d'un moteur d'inférences d'ordre 1 en chaînage arrière ... Lorsqu'aucune règle ne conclut sur un terme, ce n'est pas à l'utilisateur qu'une question est posée (comme pour les moteurs d'ordre 0) mais à un capteur. Pour preuve, Schoppers réinvente l'algorithme de Rete, en construisant son arbre de décision. C'est une notion un peu dégradée de la planification : aucun plan à long terme

[22] *"Du côté de la table opposé au robot, se trouve un bébé farceur qui va faire s'écrouler les tours de cubes, s'emparer des cubes que tient le bras du robot, et même jeter des cubes sur le robot."*, [Schoppers 87, p. 1039], (rigoureusement sic !).

n'est construit, le système se modèle sur les perturbations. Enfin, prétendre représenter toutes les réponses à tous les états possibles nie l'existence d'une explosion combinatoire [Ginsberg 89].

STRIPS, associé à son système exécuteur PLANEX, réagit à une incohérence (un prédicat qui n'a pas la valeur attendue) en déterminant l'action qui n'a pas fonctionné grâce à sa table triangulaire. Il sait ainsi exactement depuis où il faut reprendre l'exécution de son plan ... qui ne sera pas modifié !

Systèmes à replanification partielle Des plans tels que ceux construits par NOAH, NONLIN ou SIPE d'une part sont non linéaires, et d'autre part contiennent suffisamment d'informations pour pouvoir détecter ce qu'une perturbation détruit dans un plan initialement construit.

NOAH replanifie en exploitant sa notion de hiérarchie : lorsqu'un prédicat (sous-but) n'a pas la valeur attendue lors de l'exécution[23], NOAH construit un morceau de plan du niveau hiérarchique inférieur qui correspond à la réalisation de ce sous-but ("re-développement d'un nœud-problème hâtivement catalogué en nœud-fantôme"). La gestion des interactions avec le reste du réseau, qu'entraîne cette re-planification, est alors du ressort de la gestion des interactions entre niveaux hiérarchiques (cf. 3.2.1).

SIPE est le seul planificateur incluant un système complet de replanification [Wilkins 85]. Une perturbation du monde réel, se traduisant par une valeur d'un terme, se représente par l'ajout d'une action dont le terme-perturbation est conclusion (action libellée *Mother Nature*). Wilkins construit une série d'heuristiques pour chacunes des phases (détermination des interférences entre la perturbation et la logique initiale du plan, recherche de satisfaction du nouveau sous-but représenté par la perturbation, gestion des interactions entre le sous-plan résultant de cette satisfaction et le plan initial), qui viennent s'ajouter aux (déjà nombreuses) heuristiques initiales de SIPE (cf. 3.2.1). Ces heuristiques initiales sont parfois réutilisées pour la replanification, le SIPE-initial devenant un sous-programme du SIPE-replanificateur !

Les heuristiques de SIPE supposent toujours que la perturbation a une "faible" influence sur le graphe, i-e que quelques modifications suffisent (ce qui est le cas pour tous les planificateurs, où les perturbations sont liés à des problèmes d'exécution de dernière minute). En cas de trop grosse perturbation, le problème risque d'avoir trop changé pour pouvoir adapter le plan courant obsolète : Wilkins suggère alors de recommencer le processus de planification (ce n'est plus le même problème).

Systèmes à réallocation Dans un système tel qu'ISIS/OPIS (cf. 3.3.3), les manifestations possibles du monde extérieur sont déjà intégrées comme des contraintes particulières : la casse de machine s'exprime par une contrainte sur les pièces devant y être traitées, l'arrivée d'une commande très urgente s'exprime par sa priorité, ...

La notion de perturbation issue du monde extérieur a ici un sens moins dramatique que précédemment, car cette perturbation ne remettra jamais en cause la *logique* de la séquence d'actions, mais l'*allocation* des machines qui les réalisent : que ce soit pour ordonnancer une nouvelle commande à partir d'une chaîne de fabrication déjà chargée, ou pour ré-ordonnancer

[23] La manière effective de transformer une image du monde réel (des pixels d'une image, par exemple) en une valeur de terme sort complètement du cadre de cette étude.

partiellement une commande perturbée, il faut dans les deux cas ordonnancer à partir d'un état initial déjà fortement contraint.

Chapitre 4

Consistance de situations

L'activité de planification, vue comme la génération d'un graphe d'actions, peut se résumer ainsi : *modifier incrémentalement un graphe d'actions par ajout de contraintes ou d'actions, en espérant une convergence vers un but fixé d'avance, principalement grâce à un raisonnement sur les interactions entre branches en parallèle.* Pour assurer leur convergence, les systèmes de planification présentés au chapitre 3 construisent, à partir de l'observation de quelques exemples, des heuristiques qui devront sortir le planificateur des ornières successives dans lesquelles il tombe à chaque pas. Ces approches pragmatiques et novatrices ont cependant en commun un empirisme paralysant : rien ne prouve que de telles heuristiques, se substituant presque à tout contrôle, assureront également la convergence pour d'autres problèmes de la même famille. Dans l'approche inverse (rationnelle), la caractéristique principale d'une heuristique consiste à arbitrer plus intelligemment que ne le ferait le hasard chaque fois que la théorie laisse une possibilité de choix [Laurière 76]. La méticuleuse observation d'exemples typiques (cf. 7.1) peut faire germer des idées de résolution qui combleront les trous théoriques, mais qui ne doivent pas se substituer à la théorie elle-même[1].

Après avoir identifié le nœud du problème de la détection d'interactions entre actions concurrentes, nous donnons une présentation unifiée des deux théories principales, et fondamentalement identiques, délimitant les aires d'application des futures heuristiques : la théorie de Thomas Dean [Dean & Boddy 88] en logique des propositions et la théorie de David Chapman [Chapman 85] en logique des prédicats d'ordre 1.

4.1 Résolution locale du problème du cadre

Planification et problème du cadre Tenter de formaliser un comportement planificateur pour prouver sa correction mathématique est un exercice très prisé [RAP 86]. Mais fournir des critères simples, traitant exactement du problème de l'interaction des branches en parallèle et ayant des applications pratiques, l'est en revanche beaucoup moins [RIP 90].

L'activité de génération de graphe d'actions se compose du cycle (dit *analyse moyens-fins*)

[1] En droit, un tribunal rendant un jugement faisant jurisprudence complète la loi, mais ne la remplace pas.

consistant à (1) observer la solution partielle courante (déduction), (2) évaluer sa différence avec un but fixé à l'avance et (3) modifier cette solution partielle (induction). L'observation fine de l'état courant est (entre autres) déterminante pour la convergence du processus. Mais cet état courant (ou solution courante) est ici un graphe d'actions entier, aussi son observation globale ne se réduit pas à l'unique observation de sa dernière situation (comme c'est le cas dans beaucoup de systèmes de résolution de problèmes), mais comprend le sondage de plusieurs (sinon toutes) situations du graphe. Calculer explicitement toutes les situations à chaque modification serait soit très couteux en place (en ordre 0), soit quasi-impossible (en ordre 1, cf. l'exemple introductif de 4.3). C'est pourtant ce que font les moteurs d'inférences de systèmes experts [Hayes-Roth et coll. 83], car (1) il n'y a pas de variables dans une situation (même dans les moteurs d'ordre 1, la base de faits ne contient pas de variable[2]) et (2) à un instant donné de la résolution et quel que soit le mode de contrôle, *une seule* situation (la situation courante) a besoin d'être explicitée. **En planification, on ne peut que raisonner directement sur les conclusions des actions (situations duales) sans jamais expliciter une seule situation du graphe des actions** (bien que la mesure fine de la complexité d'un problème de planification nécessite, *en théorie seulement*, d'expliciter exhaustivement le graphe des situations [Joslin & Roach 89]).

Le problème de la détermination des interactions entre branches en parallèle a été ramené en 1985 à ce raisonnement sur les conclusions d'actions en adaptant le problème du cadre (cf. 2.1.3) : il s'agit de définir ce qu'une action transforme et laisse intact, en tenant compte non seulement des effets *propres* de l'action, mais aussi des effets des actions *situées en parallèle*[3]. Tout raisonnement se base sur la définition du calcul de la valeur d'un terme juste après une action (i-e dans la situation qui suit l'action) : après telle action ce terme est-il vrai, faux ou inconnu ? Cette définition est appelée *critère de vérité* dans [Chapman 85] et *instance de projection temporelle* dans [Dean & Boddy 88]. Le problème des interactions entre branches en parallèle est ainsi subsumé sous celui du calcul d'une situation juste après une action.

Une fois ce critère de vérité introduit, se pose le problème du rebranchement des situations : à quelle condition peut-on calculer la conjonction de deux situations en parallèle (définie comme une situation succèdant immédiatement à ces deux situations) ? C'est le problème de la *cohérence* (ou consistance) des situations : une situation est dite *conflictuelle* ou *inconsistante* ssi il existe un terme de cette situation qui peut être à la fois vrai et faux. Deux situations sont rebranchables de façon licite si la situation suivante (commune) est consistante. Par extension, un graphe d'action est dit *consistant* si toutes les situations le composant le sont. Tout le problème du processus de génération de graphe d'actions est de représenter suffisamment ces conflits pour pouvoir les résoudre (si possible intelligemment) par l'ajout de contraintes de précédence ou d'unification, en espérant en plus une convergence globale.

On peut considérer un graphe d'action en terme de logique modale (cf. 2.4) grâce au graphe (dual) des situations : une situation est appelée *monde possible*, la précédence temporelle $\prec$ est appelée *relation d'accessibilité* $\rightsquigarrow$. La planification vue comme la génération d'un graphe d'action est un modèle de la théorie levant localement le problème du cadre. Ceci justifiera l'existence de plusieurs critères de vérité, le modèle d'une théorie n'ayant aucune raison d'être unique.

[2] A l'exception notable de Prolog, où base de faits et base de règles sont mélangées sous forme de clauses.

[3] Ce qui revient à admettre l'influence d'autrui sur le devenir de chacun : ce que sera demain est déterminé par l'action que j'effectue maintenant, par celles que j'ai effectuées dans le passé et par celles que les autres effectuent aussi aujourd'hui.

Démarche commune Pour calculer une situation, Dean raisonne sur la situation prise au global (point de vue ensembliste) alors que Chapman s'intéresse à la valeur d'un seul terme d'une situation (point de vue particulier). Dean appelle *projection temporelle* cette détermination de toutes les conséquences d'un ensemble d'actions, ce qui est exactement le calcul d'une situation juste après une action que Chapman définit par un *critère de vérité*. Dans les deux cas, la démarche est la suivante :

1. formaliser les conclusions des actions du graphe. Délimiter la façon dont une action peut intervenir sur son environnement : un ensemble de termes ajoutés et un ensemble de termes retranchés, comme dans STRIPS, mais aussi des conclusions conditionnelles, numériques, ...

2. définir le calcul d'une situation dans un ordre total. Comment savoir ce qui est vrai et ce qui est faux à la sortie d'une action, connaissant ce qui y entre et ce que fait l'action (*fonction de transfert*) ? Plus l'expressivité du langage des actions est grande, moins le calcul d'une situation est facile.

3. définir l'extension de la définition du calcul précédant à un ordre partiel. Comment savoir ce qui est vrai (ce qui peut être vrai, ce qui est nécessairement vrai) et ce qui est faux (id.) à la sortie d'une action, en ajoutant maintenant les perturbations provenant des actions en parallèle (*fonction de transfert bruitée*, cf. figure 4.1) ?

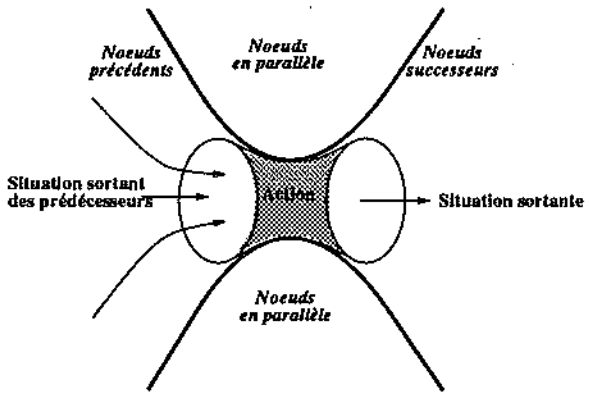


Figure 4.1: Les trois zones d'actions dans un ordre partiel

4. exhiber un algorithme polynomial déterminant la valeur (possible ou nécessaire) d'un terme dans une situation. Version calculatoire de la *projection temporelle* de Dean ou du *critère de vérité* de Chapman.

5. définir l'utilisation de la différence entre situations *attendues* et *observées* (contrôle). Les conditions (prémisses) d'une action servent à indiquer aux actions précédentes ce vers quoi elles doivent tendre (situation partielle attendue), alors que les conclusions servent à indiquer ce qui existe actuellement (situation partielle observée). Quelle modification du graphe le planificateur envisage-t-il à partir de l'observation de ces différences avant chaque action ?

4.2 Consistance d'ordre 0

4.2.1 Théorie de Dean

Initialement intéressé par les bases de données temporelles (imagerie temporelle ou *Time Map Management*, cf. [Dean 85]), Dean avait besoin de calculer ce qui est connu en un point donné de sa base. Ce problème est exactement celui du calcul formel d'une situation après une action, aussi Dean a du développer le critère de vérité (curieusement peu connu) correspondant à sa formalisation. Son TMM, vu comme une base de données, est modifié par l'utilisateur qui y ajoute des événements complétement instanciés : ce n'est pas le programme qui instancie des pseudo-règles d'inférences d'ordre 1 en des évènements sans variable. Aussi son raisonnement se situe en logique des propositions.

Nous présentons une version allégée de sa théorie pour la rendre facilement comparable à celle de l'ordre 1.

Axiomatisation Une instance de projection temporelle est spécifiée par un environnement général, définissant les actions possibles dans l'absolu, et un environnement particulier, définissant l'ordre partiel (pré-ordre) courant sur des actions particulières.

L'environnement général est un triplet $(\mathcal{T}, \mathcal{A}, \mathcal{P})$ où $\mathcal{T}$ est l'ensemble des types d'événements $(\tau_1, \ldots, \tau_n)$, $\mathcal{A}$ est l'ensemble des actions (*règles causales*) disponibles $(a_1, \ldots, a_m)$, $\mathcal{P}$ est un ensemble de propositions (en logique des propositions). Respectant le formalisme de STRIPS, une action a_i est de la forme (τ, π, γ) où τ est un type d'événement $(\tau \in \mathcal{T})$ correspondant à la dénomination de l'action a_i, π est un ensemble de propositions $(\pi \in \mathcal{P})$ représentant les préconditions de l'action a_i, γ est l'ensemble de conclusions de l'action a_i, composé de termes de la forme p (ajoutés) et de termes de la forme $\neg p$ (détruits).

Un environnement particulier d'une instance de projection temporelle est composé d'une situation initiale $\mathcal{I}$, ensemble de propositions $(\mathcal{I} \subset \mathcal{P})$ et du graphe courant des événements $(\mathcal{E}, \prec)$ où $\mathcal{E}$ est un ensemble d'événements de type connu $(\forall e \in \mathcal{E}, type(e) \in \mathcal{T})$ et $\prec$ un ordre partiel quelconque sur ces événements.

Ordre total Pour calculer ce qui existe après un événement dans un ordre partiel (les propositions vraies et les propositions fausses juste après un événement, i-e une situation), Dean se ramène au cas de l'ordre total. On note $ext(\prec)$ l'ensemble des ordres totaux $\prec_i$ qui sont des extensions de l'ordre partiel $\prec$ (i-e $\forall \prec' \in ext(\prec), \forall e, e' \in \mathcal{E},\ e \prec e' \Rightarrow e \prec' e'$).

Soit $\prec'$ une extension totalement ordonnée de $\prec$, on numérote selon $\prec'$ les événements du plan courant $(\mathcal{E}, \prec)$ par $(e_1, \ldots, e_{\text{Card}(\mathcal{E})})$. On appelle $post_{\prec'}(e_i)$, où $i \in [1, Card(\mathcal{E})]$, la situation juste après l'événement e_i dans l'ordre total $\prec'$ (voir figure 4.2).

La situation $post_{\prec'}(e_i)$ est composée de toutes les propositions p telles que :

- ou bien p vient d'être ajoutée par une action applicable correspondant à e_i (*condition*

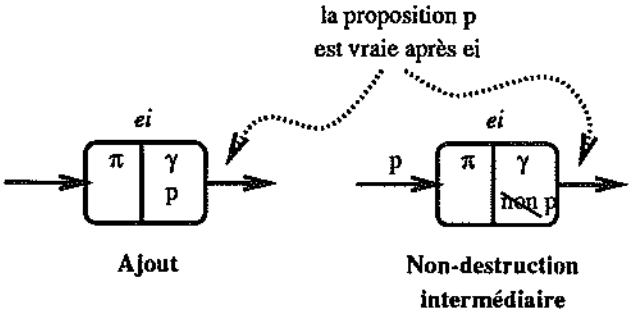


Figure 4.2: Critère de vérité d'ordre 0

d'ajout) :

$$\exists(\tau,\pi,\gamma)\in\mathcal{E},\begin{cases}\tau = type(e_i) & \textit{(l'action correspond à } e_i\textit{)}\\ \pi \subset post_{\prec'}(e_{i-1}) & \textit{(l'action est applicable en } e_i\textit{)}\\ p\in\gamma & \textit{(l'action ajoute } p\textit{)}\end{cases}$$

- ou bien p était déjà vraie juste avant e_i (donc juste après e_{i-1}) et il n'y a pas d'ultime action applicable correspondant à e_i qui détruit p (*condition de non-destruction intermédiaire*) :

$$p\in post_{\prec'}(e_{i-1})\ \wedge\ \neg\left(\exists(\tau,\pi,\gamma)\in\mathcal{E},\begin{cases}\tau = type(e_i) & \textit{(id.)}\\ \pi\subset post_{\prec'}(e_{i-1}) & \textit{(id.)}\\ (\neg p)\in\gamma & \textit{(l'action détruit } p\textit{)}\end{cases}\right)$$

Cette définition suppose que les conclusions γ d'une règle sont initialement consistantes : $\forall p\in\mathcal{P},\ \neg(p\in\gamma\ \wedge\ \neg p\in\gamma)$. Les deux conditions sont également valables pour le calcul des propositions fausses p' (remplacer p par $\neg p'$ dans les expressions précédentes).

La situation $post_{\prec'}(e_i)$ est définie récursivement à partir de la situation précédente $post_{\prec'}(e_{i-1})$, en posant $post_{\prec'}(e_0)=\mathcal{I}$ pour un événement e_0 virtuel :

$$(e_0 \xrightarrow{\mathcal{I}})\ e_1 \xrightarrow{post_{\prec'}(e_1)} e_2\ \ldots\ e_{i-1} \xrightarrow{post_{\prec'}(e_{i-1})} e_i \xrightarrow{post_{\prec'}(e_i)} e_{i+1}\ \ldots\ e_{\mathrm{Card}(\mathcal{E})-1} \xrightarrow{post_{\prec'}(e_{\mathrm{Card}(\mathcal{E})-1})} e_{\mathrm{Card}(\mathcal{E})}$$

La situation $post_{\prec'}(e_{Card(\mathcal{E})})$ est aussi appelée *situation finale*.

Ordre partiel On distingue deux manières de définir le calcul d'une situation dans un ordre partiel à partir de celle du calcul d'une situation dans un ordre total :

$$Nécessaire(e) = \bigcap_{\prec'\in ext(\prec)} post_{\prec'}(e)$$

$$Possible(e) = \bigcup_{\prec'\in ext(\prec)} post_{\prec'}(e)$$

Dans l'optique forte (*Nécessaire*(e)), la situation juste après un événement e ne contient que ce qui est vrai dans *toutes* les extensions totalement ordonnées de $\prec$. Dans l'optique faible

(*Possible*(e)), la situation juste après un événement e contient ce qui est vrai dans au *moins* une extension totalement ordonnée de $\prec$.

Ces deux cas se retrouvent dans la définition de la valeur de vérité d'une proposition p et se notent avec les modalités de nécessité $\Box$ (cf. 2.4) et de possibilité $\Diamond$:

$$\Box_e p \overset{\text{def}}{\Longleftrightarrow} p \in \textit{Nécessaire}(e) \Leftrightarrow \forall \prec' \in \textit{ext}(\prec), p \in \textit{post}_{\prec'}(e)$$
$$\Diamond_e p \overset{\text{def}}{\Longleftrightarrow} p \in \textit{Possible}(e) \Leftrightarrow \exists \prec' \in \textit{ext}(\prec), p \in \textit{post}_{\prec'}(e)$$

Une proposition est *nécessairement vraie* juste après e ssi elle y est vraie dans *toutes* les extensions totalement ordonnées $\prec'$ de $\prec$. Il est *possible* qu'une proposition soit vraie juste après e ssi elle y est vraie dans au *moins une* de ses extensions totalement ordonnées $\prec'$ de $\prec$.

Pour un problème de projection temporelle arbitraire ($(\mathcal{T}, \mathcal{A}, \mathcal{P})$ et $(\mathcal{E}, \mathcal{I}, \prec)$), déterminer si une proposition peut être vraie après un événement ($p \in \textit{Possible}(e)$) est un problème NP-complet [Dean & Boddy 88], ce qui est du au fait que le nombre d'extensions totalement ordonnées $\textit{Card}(\textit{ext}(\prec))$ croît exponentiellement avec le nombre d'événements courants $\textit{Card}(\mathcal{E})$ (le calcul d'une situation $\textit{post}_{\prec'}(e_i)$ dans un ordre total $\prec'$ étant de complexité linéaire).

4.2.2 Algorithme de calcul d'une situation

Il est évident par construction que : $\forall e, \emptyset \subset \textit{nécessaire}(e) \subset \textit{possible}(e) \subset \mathcal{P}$. Le calcul direct de $\textit{nécessaire}(e)$ et de $\textit{possible}(e)$ est de trop forte complexité (cf. ci-dessus), aussi il est préférable (i-e polynomial) d'évaluer $\textit{nécessaire}(e)$ par une borne inférieure $\textit{fort}(e)$ (l'ensemble des propositions qui sont vraies après e dans *toutes* les extensions totalement ordonnées de $\prec$), et $\textit{possible}(e)$ par une borne supérieure $\textit{faible}(e)$ (l'ensemble des propositions qui sont vraies après e dans au *moins* une extension totalement ordonnée de $\prec$) : $\forall e, \emptyset \subset \textit{fort}(e) \subset \textit{nécessaire}(e) \subset \textit{possible}(e) \subset \textit{faible}(e) \subset \mathcal{P}$.

N'utilisant pas l'ordre 1, Dean use du succédané des types d'événements pour distinguer les actions des schémas d'actions (égaux à une instanciation ou à un renommage près), ce qui complique fortement son algorithme (cf. l'algorithme simplifié en 5.3.1). Le principe de son algorithme est le suivant :

- Pour calculer $\textit{fort}(e)$ et $\textit{faible}(e)$, on considère le graphe vu depuis l'événement e, qui est composé de trois zones (voir figure 4.1): E_-, l'ensemble des événements avant e ($e' \prec e$), E_+, l'ensemble des événements après e ($e' \succ e$), et $E_{//}$, l'ensemble des événements en parallèle de e ($e' \not\prec e \wedge e' \not\succ e$).
- Comme pour l'ordre total, $\textit{fort}(e)$ et $\textit{faible}(e)$ se calculent récursivement à partir de leur valeur juste avant e, i-e par réunion des $\textit{fort}(e')$ et $\textit{faible}(e')$ issus de E_-. Toute la difficulté est qu'on ne peut plus ignorer les événements de $E_{//}$ (qui n'est plus vide, comme en ordre total), qui peuvent maintenant perturber les valeurs des propositions arrivant directement sur e (E_-).
- Ce qui sort nécessairement de e ($\textit{fort}(e)$) est ce qui y entre ($\textit{fort}(e')$) moins ce qui peut être détruit par des événements en parallèle ($E_{//}$), plus ce qui est éventuellement ajouté par tous les événements de $E_{//}$.

- Ce qui peut sortir de e ($\mathit{faible}(e)$) est principalement ce qui peut y entrer directement ($\mathit{faible}(e)$), plus tout ce qui peut être ajouté par un événement en parallèle ($E_{//}$), moins ce qui est nécessairement détruit par tous les événements de $E_{//}$.

En terme de complexité, le nombre d'opérations élémentaires sur les ensembles, bien que non exponentiel, est de l'ordre de $O(n^{5+3c})$ où n est le nombre d'événements ($\mathit{Card}(\mathcal{E})$) et c le nombre maximum de propositions différentes mentionnées en partie condition d'une action (champs π). Il signale que cette complexité peut être ramenée à $O(n^{c+1})$ en utilisant un module de *dépendance de données*.

Les faibles performances de son algorithme poussent Dean à trouver d'autres idées pour faire chuter la complexité ; il s'interroge en particulier sur l'intérêt d'une étude *probabiliste* de $\mathit{nécessaire}(e)$ et $\mathit{possible}(e)$: on saurait ainsi par exemple que $\mathit{sur}(a, b)$ est vrai avec une probabilité de 0.7, alors que $\mathit{sur}(a, c)$ est vrai avec une probabilité de 0.6. Cette idée est au moins originale et peut être intéressante si elle ne fait pas que reporter le problème sur l'aspect numérique, mais Dean ne fait que la mentionner sans l'explorer.

Critique Dans le cas de Dean, la complexité moyenne est en $O(n^4)$ (trois conditions maximum par règle, cf. 7.1, dans l'implémentation *"data dependency"* — $O(n^{14})$ dans l'implémentation naïve), ce qui, allié à une forte complexité en espace[4], peut faire douter de son utilité pratique : si l'intérêt d'un logiciel se mesure à son efficacité et à la fréquence d'utilisation par son auteur [Ferber 89], on voit mal qui pourrait s'intéresser au très boulimique algorithme de Dean. De plus, la constante multiplicatrice du terme principal de l'expression d'une complexité, n^4 dans notre cas, est également forte : cr^{c+1}, où r est le nombre maximal d'actions par type d'événement. Il est souvent préférable d'utiliser un bon algorithme exponentiel qu'un algorithme polynomial à forte constante multiplicatrice [Laurière 86].

Ensuite, l'influence des travaux de Dean à propos de son *Time Map Management* [Dean 85] sur la présente formalisation se manifeste sur les points suivants :

- Absence totale de variables : les faits de la base de données temporelle sont des relations entre constantes (logique des propositions). Or le moindre exemple en planification a besoin de variables (cf. 4.3), aussi Dean introduit une distinction entre *événement* et *action* (à un événement peut correspondre plusieurs actions via leur *type*), qui revient grossièrement à introduire le "OU" dans un graphe d'actions : le terme $\mathit{couleur}(\mathit{téléphone}, x)$ de l'ordre 1 se traduit par exemple par la conjonction d'ordre 0 $\mathit{couleur}(\mathit{téléphone}, \mathit{rouge}) \vee \mathit{couleur}(\mathit{téléphone}, \mathit{crème}) \vee \mathit{couleur}(\mathit{téléphone}, \mathit{marron})$. L'absence de gestion explicite des variables au profit de l'ersatz de la liste exhaustive des valeurs complique sérieusement ses algorithmes[5].

[4] Dean reste muet sur ce point, mais on peut raisonnablement supposer que le gain annoncé en complexité de calcul se traduit par une perte (équivalente ?) en complexité de stockage (*"data dependency"*). C'est précisément cette complexité de stockage qui suggère généralement de pervertir le modèle théorique propre initial par l'introduction d'heuristiques, ce qui peut conduire à des algorithmes originaux [Descottes & Latombe 85].

[5] Sacerdoti avait exactement le même problème pour la même raison : la gestion imparfaite des variables via ses *objets formels* l'obligeait à introduire un OU dans son graphe, ce qui augmente immédiatement d'un degré la complexité des algorithmes [Sacerdoti 77].

- Aucune réflexion sur la façon de résoudre un problème d'inconsistance ou d'ajouter des informations cohérentes au graphe : Dean ne s'intéresse qu'à la déduction d'informations de sa base de données temporelle, son système est incapable de suggérer une modification levant une inconsistance (c'est l'utilisateur qui est seul habilité à modifier l'agencement des événements). La présence d'une partie condition π dans une action suggérait pourtant un raisonnement à partir de ces préconditions, mais il n'y a pas de situation finale, donc pas de but global à atteindre. De même, si Dean signale que le problème de la détection d'inconsistances dans une situation est en général NP-complet, il n'inverse pas sa définition de la valeur d'un prédicat pour suggérer des méthodes de levée d'inconsistance (cf. 4.3).

4.3 Consistance d'ordre 1

Il semble a priori simple d'introduire des termes du premier ordre dans des raisonnements tels que ceux de Dean : on pourrait naïvement croire qu'il suffit de remplacer les symboles de propositions p précédents par les termes composés habituels $sur(a,b)$; comme le nombre de constantes est généralement fini et les termes, de profondeur 1 uniquement[6] ($f(g(x,y),y)$ interdit), il est facile de numéroter (ou "indexer" [Agre 87] et [Ghallab & Mounir Alaoui 89]) ces soi-disant termes du premier ordre, de les appeler $p_1, \ldots, p_n$ ...et de retrouver l'ordre 0 !

La plupart des planificateurs ne fonctionnent qu'en logique des propositions. Cependant, le moindre exemple (cf. 7.1) montre que la présence de variables est nécessaire : par exemple, Sacerdoti introduit ses *objets formels*[7], pseudo-variables qui sont instanciées dès que possible (i-e dont il se débarrasse au plus vite) [Sacerdoti 77].

Dans la figure 4.3 par exemple, deux événements e_1 et e_2 (au sens de la section précédente) concluant respectivement sur $sur(a,b)$ et sur $sur(a,c)$ (ces deux événements sont incompatibles selon la sémantique de l'exemple de 7.1) et un événement successeur e_3 ($e_1 \prec e_3 \wedge e_1 \prec e_3$) qui conclut sur $\neg sur(a,\underline{x})$.

La question est : *"sur quoi est a après e_3 ?"* Intuitivement, si on sait que x vaut b, alors a doit rester sur c ; si on sait que x vaut c, alors a doit rester sur b ; si on sait que x ne vaut ni b ni c, e_1 et e_2 doivent être déclarés incompatibles ; et si on ne sait rien sur x, comment représenter les trois cas précédents ?

---

[6] Plus généralement, ces symboles dits "fonctionnels" ne sont pas itérables : un terme tel que $f(f(f(x)))$ est interdit. Ces symboles sont en réalité des symboles **relationnels**, bien que respectant la syntaxe habituelle des symboles fonctionnels d'une signature. Un tel système formel, sans symbole fonctionnel, ne serait d'ordre 1 qu'avec un nombre infini de constantes, ce qui est difficile à réaliser sur un ordinateur ...

[7] NOAH s'est révélé d'une grande finesse en ce qui concerne le raisonnement sur les termes constants (très peu de ses heuristiques ont été contredites par ses successeurs, y compris le sévère Chapman) mais chute de plusieurs niveaux de raisonnement en ce qui concerne les *objets formels* : ils sont juste définis en termes informatiques sans justification conceptuelle, ce qui est à l'origine de la curieuse gymnastique d'instanciation / désinstanciation de ces pseudo-variables. On peut raisonnablement deviner que Sacerdoti a conçu son système sans penser à ce problème et a introduit en catastrophe ce "patch" de dernière minute (une indirection supplémentaire pour pointer successivement sur plusieurs constantes).

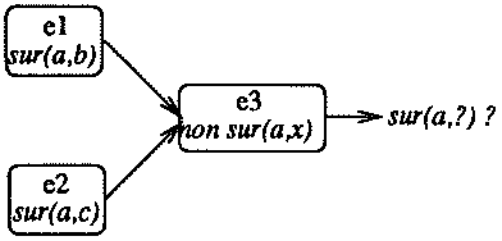


Figure 4.3: Exemple d'indécision linéaire en ordre 1 avec incompatibilité

4.3.1 Théorie de Chapman

Axiomatisation Comme précédemment, une instance de projection temporelle est spécifiée par un environnement général, définissant les schémas d'actions disponibles et un environnement particulier définissant l'ordre partiel courant sur des actions particulières.

Un environnement général est un couple $(\mathcal{A}', T_\Sigma[X])$ où $\mathcal{A}'$ est l'ensemble des schémas d'actions et $T_\Sigma[X]$ l'ensemble des termes construits à partir de la signature Σ (par exemple, $\Sigma = \{sur_2, libre_1, a_0, b_0, c_0, d_0, sol_0\}$, les indices indiquant les arités des symboles). Reprenant aussi le formalisme de STRIPS, un schéma d'actions est un couple (π, γ) où π est l'ensemble des termes préconditions du schéma ($\pi \subset T_\Sigma[X]$) et γ l'ensemble des termes conlusions du schéma ($\gamma \subset T_\Sigma[X]$). On notera aussi π et γ sous forme d'extracteur de champs d'un schéma d'actions a : $\pi = \pi(a)$ et $\gamma = \gamma(a)$. Le connecteur $\neg$ permet de compacter en un seul champs (γ) les champs habituels *ajout* et *détruit* de STRIPS : un terme de la forme $f(a_1, \ldots, a_n, x_1 \ldots, x_m)$ correspond à une conclusion *ajoutée* ; un terme de la forme $\neg g(a_1, \ldots, a_n, x_1 \ldots, x_m)$ signifie que $g(a_1, \ldots, a_n, x_1 \ldots, x_m)$ est une conclusion *détruite*[8].

Un environnement particulier est composé d'une situation initiale $\mathcal{I}$, d'une situation finale $\mathcal{F}$ et du graphe courant des actions : un couple $(\mathcal{A}, \prec)$ où $\mathcal{A}$ est l'ensemble des actions du plan ($\mathcal{A} \subset \mathcal{A}'$ à une substitution près) et $\prec$ un ordre partiel sur ces actions. Une *action* est un *schéma d'actions* dans lequel les variables ont été substituées soit par des constantes (instanciation), soit par d'autres variables (pour éviter des conflits de noms de variables)[9].

Unification réduite L'ordre 1 pose le problème suivant : si une action conclut qu'un terme $f(x)$ est vrai, que vaut le terme $f(y)$ juste après cette action : vrai, faux ou inconnu ? Les deux termes sont différents (au sens de l'égalité de termes), pourtant il est intuitivement évident que, si rien ne contraint les variables x et y, ces deux termes devraient être identiques à un renommage près. $f(y)$ serait alors vrai bien que n'ayant pas été explicitement déclaré en conclusion. L'égalité de l'algèbre des termes ne suffit plus en ordre 1.

Ce problème est traditionnellement résolu par la théorie de l'unification [Lalement & Saint 86] : à partir de l'ensemble $\mathcal{S}$ des substitutions de variables par un terme ($\sigma : X \rightarrow T$), on définit un préordre sur les termes ($t \preceq t' \Leftrightarrow \exists \sigma \in \mathcal{S}, \sigma(t) = t'$), permettant de comparer la quantité d'informations contenue dans deux termes. Les classes d'équivalence de la relation d'équivalence

[8] En toute rigueur, $\neg f(a_1, \ldots, x_m)$ n'est pas un *terme* de $T_\Sigma[X]$ mais plus généralement une *formule*, à cause du connecteur $\neg$ (cf. 2.3.1). On continuera cependant à appeler "terme" ce type de formule pour éviter une trop grande lourdeur.

[9] Il s'agit de substitutions réduites à $X \cup \Sigma_{cst}$, au sens de [Lalement & Saint 86].

$\sim$ associée à ce préordre sont constituées par des ensembles de termes égaux par renommage. Deux termes t et t' sont *unifiables* (noté $t \approx t'$) ssi il existe une substitution σ, appelée unificateur, telle que $\sigma(t) = \sigma(t')$. Si deux termes sont unifiables, alors il existe un unificateur principal (*most general unifier*), unificateur minimal selon $\preceq$ (les autres unificateurs en sont des substitutions).

L'existence et le calcul de cet unificateur principal se détermine facilement mais naïvement selon une complexité en temps de $O(n^2)$ [Lalement & Saint 86], ce coefficient 2 provenant du test de non auto-référence d'un terme (*occur check* : vérifier que, pour tout terme t, $t \not\sim f(t)$ — le *problème du point fixe* n'a pas de sens dans l'algèbre des termes !). Cette complexité est couramment inférieure ($O(n\log(n))$ pour implémentations commerciales de Prolog, mais généralement sans *occur check*) et a été récemment rendue presque linéaire et sans importantes structures de données annexes [Escalada-Imaz & Ghallab 88].

Les trois égalités $=$, $\approx$ et $\sim$ sont reliées par : $\forall\varphi,\psi,\ \varphi = \psi \Rightarrow \varphi \sim \psi \Rightarrow \varphi \approx \psi$

Dans le cas précis de Chapman, seule une version réduite de l'unification suffit, puisque les termes utilisés ne sont que de profondeur 2 et que les variables ne peuvent s'unifier qu'avec des constantes ou d'autres variables.

Unification modale S'ajoute cependant au problème de l'unification celui de la *négation* (voir figure 4.4) : si une action a conclut sur un terme $f(x_0)$ et qu'une action suivante a' conclut sur $\neg f(a)$, que vaut $f(x_1)$ après a' ?

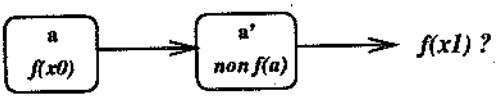


Figure 4.4: Autre exemple d'indécision linéaire en ordre 1

Si x_1 vaut a, $f(x_0)$ est faux, sinon, $f(x_1)$ s'unifie avec $f(x_0)$ et est vrai, mais, dans aucun cas, $f(x_1)$ n'est inconnu. Si l'on veut contraindre $f(x_1)$ à être vrai, il faut pouvoir exprimer à la fois $x_1 \approx x_0$ et $x_1 \not\approx a$. Pour cela, on définit la fonction de contrainte $unif : X \longrightarrow \wp(X \cap \Sigma_{cst})^2$, qui associe à toute variable x de X l'ensemble $unif^+(x)$ des termes (variable ou constante) avec lesquels elle doit s'unifier et l'ensemble $unif^-(x)$ des termes (variable ou constante) avec lesquels elle ne doit pas s'unifier : $unif(x) = (unif^+(x), unif^-(x))$. Dans l'exemple ci-dessus, obliger $f(x_1)$ à être vrai revient à poser $unif^+(x_1) = unif^+(x_0) \setminus \{a\}$ et $unif^-(x_1) = unif^-(x_0) \cup \{a\}$.

La procédure ci-dessus de test et de calcul de l'unificateur de deux termes $\sigma = \{(x_i, t_i)\}$ doit, en plus, tenir compte des tests de cohérence sur ces contraintes d'unification : $\forall i,\ t_i \notin unif^-(x_i)$.

Deux égalités sont utilisées, selon que φ est ajoutée ou testée :

- lorsque φ est ajoutée dans une situation, il faut trouver une conclusion ψ (de l'action qui génère cette situation) qui signifie essentiellement[10] la même chose que φ. L'égalité

[10] En terme de substitution σ, la condition d'ajout peut s'écrire :

$$\exists\psi \in \gamma(a_i), \forall\sigma \in \mathcal{S},\ \sigma(\varphi) = \sigma(\psi)$$

de l'algèbre des termes $\varphi = \psi$ serait trop forte, comme on a pu le voir dans l'exemple donné dans l'introduction du paragraphe précédent. L'unification $\varphi \approx \psi$ serait trop faible, puisque rien n'indique que toute instanciation de φ soit effectivement ajoutée.

- lorsque la négation de φ est testée, il faut trouver une conlusion χ dont la négation peut signifier[11] la même chose que φ (une destruction partielle suffit), ce qui correspond à l'égalité la plus faible $\approx$.

Cette fonction de contraintes *unif* (qui sera implémentée comme une *base de contraintes* en partie II) peut s'intégrer élégamment dans ces égalités grâce aux qualificatifs modaux de nécessité $\Box$ et de possibilité $\Diamond$: deux termes t et t' de $T_\Sigma[X]$ s'unifient nécessairement ($\Box(t \approx t')$) ssi ils s'unifient au sens standard (non modal) et s'ils sont contraints à le faire via leurs ensembles $unif^+$; deux termes t et t' de $T_\Sigma[X]$ s'unifient possiblement ($\Diamond(t \approx t')$) ssi ils s'unifient au sens standard (non modal) et si leur unification n'est pas en contradiction avec les contraintes déjà présentes dans la base[12].

Par rapport à l'ordre 0 précédent, ces modalités de possibilité et de nécesité interviennent ici une deuxième fois pour introduire les contraintes (positives et négatives) d'unification, en plus de leur première introduction qui visait à ramener l'ordre partiel à un ordre total (voir l'ordre 0 précédent ou les deux paragraphes suivants).

Ordre total Pour calculer une situation dans un ordre partiel $\prec$, Chapman commence par définir le calcul d'une situation dans un ordre total $\prec'$.

Soit $\prec'$ une extension totalement ordonnée de $\mathcal{A}$ ($\prec' \in ext(\prec)$), on peut numéroter les actions de $\mathcal{A}$ selon $\prec'$: $a_1, \ldots, a_{\mathrm{Card}(\mathcal{A})}$. En notant comme précédemment $post_{\prec'}(a_i)$ la situation située juste après l'action a_i dans cet ordre total $\prec'$, le terme φ est vrai dans la situation $post_{\prec'}(a_i)$ (i-e $\varphi \in post_{\prec'}(a_i)$) ssi :

- ou bien φ vient d'être ajouté par a_i (*condition d'ajout*) :

$$\exists \psi \in \gamma(a_i),\ \varphi \sim \psi$$

- ou bien φ était déjà vrai juste avant a_i (donc juste après a_{i-1}) et la dernière action a_i ne détruit pas φ (*condition de non-destruction intermédiaire*) :

$$\varphi \in post_{\prec'}(a_{i-1}) \ \wedge\ \neg\,(\exists \chi \in \gamma(a_i), \varphi \approx \neg\chi)$$

Les deux conditions précédentes servent également à calculer des termes φ faux (en remplaçant φ par $\neg\varphi$ dans les expression précédentes). Ces deux conditions supposent que les conclusions de toute action sont déjà consistantes.

[11] En terme de substitution σ, la condition de non-destruction peut s'écrire :

$$\varphi \in post_{\prec'}(a_{i-1}) \ \wedge\ \neg\,(\exists \chi \in \gamma(a_i), \exists \sigma \in \mathcal{S}, \sigma(\varphi) = \neg\sigma(\chi))$$

[12] La vision modale de l'unification sous contraintes sera définie correctement en 5.2.1.

La situation $post_{\prec'}(a_i)$ est définie récursivement à partir de la situation précédente $post_{\prec'}(a_{i-1})$, en posant $post_{\prec'}(a_0) = \mathcal{I}$ pour une action virtuelle a_0. Un terme φ appartient pour la première fois à une situation s'il est égal à un renommage prés à une conlusion de l'action qui génère cette situation.

Ces deux conditions ressemblent fortement à celles de l'ordre 0. Hormis la correspondance *événement* $\leftrightarrow$ *action*, la différence principale provient du changement d'égalité : l'égalité des termes $=$ devient l'unification $\approx$. La table 4.1 compare terme à terme les conditions d'ajout et de non-destruction intermédiaire en ordre 0, en ordre 0 sans OU (une seule règle (τ, π, γ) par événement, alors appelé action) et en ordre 1. Indiquer que le terme φ est ajouté, par exemple, s'écrit $\varphi \in \gamma$, i-e $\exists \psi \in \gamma,\ \varphi = \psi$, en ordre 0 et s'écrit $\exists \varphi \in \gamma,\ \varphi \sim \psi$ en ordre 1.

Ordre 0	Ordre 0 sans OU	Ordre 1
Condition d'ajout		
$\exists (\tau, \pi, \gamma) \in \mathcal{E}, \begin{cases} \tau = type(e_i) \\ \pi \subset post_{\prec'}(e_{i-1}) \\ p \in \gamma \end{cases}$	$p \in \gamma(e_i)$	$\exists \psi \in \gamma(a_i),\ \varphi \sim \psi$
Condition de non-destruction intermédiaire		
$p \in post_{\prec'}(e_{i-1})$ $\wedge$ $\neg \left(\exists (\tau, \pi, \gamma) \in \mathcal{E}, \begin{cases} \tau = type(e_i) \\ \pi \subset post_{\prec'}(e_{i-1}) \\ (\neg p) \in \gamma \end{cases} \right)$	$p \in post_{\prec'}(e_{i-1})$ $\wedge$ $(\neg p) \notin \gamma(e_i)$	$\varphi \in post_{\prec'}(a_{i-1})$ $\wedge$ $\neg (\exists \chi \in \gamma(a_i), \varphi \approx \neg \chi)$

Table 4.1: Comparaison des critères d'ordre 0 et 1 (en ordre total)

Ce calcul de la vérité d'un terme dans une situation à partir de celui fait dans la situation immédiatement précédente peut se généraliser à une situation précédente quelconque (version plus forte[13] que le théorème initial de Chapman) : un terme φ est vrai juste après l'action a_i ssi il existe une action a_j avant a_i (au sens large) qui conclut sur un terme égal à φ et s'il n'existe aucune action entre a_j et a_i qui conclut sur un terme ψ, de signe opposé à φ, qui peut s'unifier à φ :

$$\varphi \in post_{\prec'}(a_i) \iff \begin{cases} & \exists j \leq i, \exists \chi \in \gamma(a_j),\ \varphi = \chi \\ \wedge & \\ & \neg \left(\exists k, j \leq k \leq i, \exists \psi \in \gamma(a_k),\ \varphi \approx \neg \psi \right) \end{cases}$$

On peut là encore calculer des termes φ faux en remplaçant φ par $\neg\varphi$. Les deux conditions précédentes ne supposent pas que les conclusion des actions soient déjà consistantes : les conclusions χ et $\neg\chi$ d'une même action seraient détectées lorsque $k = j$.

Ordre partiel Comme en ordre 0, on définit le calcul d'une situation après une action dans un ordre partiel à partir de celui en ordre total. Les modalités de nécessité $\Box$ et de possibilité $\Diamond$ sont introduites pour qualifier la valeur d'un terme φ dans une situation juste après une action

[13] Par rapport au théorème présenté, le théorème initial de Chapman [Chapman 85] présente des approximations : il indique que la condition de non-destruction de φ doit être $\varphi = \neg\chi$, ce qui est trop fort puisqu'il suffit seulement que ces deux termes *puissent* s'unifier pour que la valeur globale de φ ne soit plus conservée.

a :

$$\Box_a\varphi \stackrel{\text{def}}{\Longleftrightarrow} \forall \prec' \in ext(\prec),\ \varphi \in post_{\prec'}(a)$$
$$\Diamond_a\varphi \stackrel{\text{def}}{\Longleftrightarrow} \exists \prec' \in ext(\prec),\ \varphi \in post_{\prec'}(a)$$

Un terme est *nécessairement vrai* juste après a ssi il est vrai juste après a dans *toutes* les extensions $\prec'$ de $\prec$. Il est *possible* qu'un terme φ soit vrai juste après a ssi il y est vrai dans au *moins une* extension totalement ordonnée $\prec'$ de $\prec$.

4.3.2 Algorithme de calcul d'une situation

Caractérisation Tout l'intérêt de la théorie de Chapman vient de la mise à jour d'une caractérisation (que Chapman appelle *critère de vérité*) de la définition de la valeur de vérité d'un terme dans un ordre partiel $(\mathcal{A}, \prec)$.

Un terme φ est nécessairement vrai dans la situation $post(a)$ (i-e $\Box_a\varphi$) ssi :

$$\exists a_0 \in \mathcal{A},\ \Box(a_0 \preceq a) \wedge \exists \chi \in \gamma(a_0),\ \Box(\varphi \approx \chi)$$
$$\wedge$$
$$\forall a_1 \in \mathcal{A}, \Diamond(a_1 \preceq a),\ \forall \psi \in \gamma(a_1), \Diamond(\varphi \approx \neg\psi),\ \exists a_1' \in \mathcal{A}, \Box(a_1 \prec a_1' \prec a),\ \exists \psi' \in \gamma(a_1'), \Box(\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi')$$

Le critère pour une valeur nécessairement fausse de φ est obtenu par dualité en remplaçant φ par $\neg\varphi$; de même, les critères pour une valeur *possible* vraie et fausse sont obtenus en échangeant les modalités $\Box$ et $\Diamond$. En remplaçant les termes $\Box(x \preceq y)$ et $\Diamond(x \preceq y)$ par $x \prec y$, on retrouve la formulation précédente en ordre total.

Le critère concernant la véracité de φ dans la situation *avant* a serait construit par une démarche analogue : il faudrait pour cela définir une situation juste *avant* une situation (définir $prec_{\prec'}(a_i$ pour une extension totalement ordonnée $\prec'$ de $\prec$, puis l'étendre en $prec(a_i)$) pour un ordre partiel avec les modalités possible et nécessaire), ce qui ne poserait aucun problème. On aboutirait au critère précédent dans lequel les inégalités $\Box(a_0 \preceq a)$ et $\Diamond(a_1 \preceq a)$ sont strictes. Une autre façon de faire (moins lourde) consiste à remarquer que $prec(a_i)$ ne dépend pas de $\gamma(a_i)$ (il n'y a que ce qui est avant ou en parallèle d'une situation qui peut influer[14]), mais uniquement de sa place selon l'ordre partiel $\preceq$; on peut changer ce qui est nécessairement après a_i (au sens large) sans changer la valeur de $prec(a_i)$: annulons les conclusions de a_i ($\gamma(a_i) = \emptyset$), qui devient a_i' ; on a $prec(a_i) = prec(a_i')$; mais puisque $\gamma(a_i') = \emptyset$, $prec(a_i') = post(a_i')$, et on se ramène au critère ci-dessus.

Comme en ordre total, pour que le terme φ soit nécessairement vrai juste après a, il faut d'abord qu'il ait été nécessairement vrai au moins une fois avant (au sens large : $\Box(a_0 \approx a)$

[14] Démonstration facile par récurrence sur le nombre d'actions du plan.

inclut $a_0 = a$). Au lieu de renvoyer récursivement à un $post(a_0)$, le critère fixe le point de départ de cette récursion : φ doit se trouver explicitement en conclusion d'une action a_0 nécessairement avant a. L'action a_0 est un *établisseur* de φ. Il faut ensuite que toute action a_1 *potentiellement* avant a ($\Diamond(a_1 \preceq a)$), qui détruise, même partiellement, la valeur de φ avec une conclusion ψ ($\Diamond(\varphi \approx \neg\psi)$), soit rattrapée à temps par une troisième action a'_1 ($\Box(a_1 \prec a'_1 \prec a)$), grâce à une de ses conclusions ψ' qui reconstruise exactement ce qu'a détruit, même partiellement, ψ ($\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi'$). L'action a_1 (resp. a'_1) est appelée un *masqueur* (resp. *démasqueur* ou *rétablisseur*).

L'établisseur a_0 peut être a lui-même ($\Diamond(a_0 \preceq a)$), ainsi que le masqueur a_1 ($\Diamond(a_1 \preceq a)$), ce qui permet de détecter le cas des actions inconsistantes.

Sous ce formalisme, la planification avec une infinité de constantes est indécidable (une machine de Turing peut s'y coder). Déterminer la valeur d'un terme après une action dans un ordre partiel est NP-complet lorsque les conclusions d'une action dépendent de la situation d'entrée (cas des opérateurs déductifs de SIPE [Wilkins 86]).

Induction Ce théorème étant nécessaire et suffisant, il suffit d'en inverser la lecture pour obtenir toutes les façons d'influer sur la valeur d'un terme : pour rendre un terme φ nécessairement vrai juste après une action a, il faut d'abord l'établir ; pour cela, trouver ou créer une action a_0 qui peut se trouver avant a et dont une conclusion χ peut s'unifier avec φ ; contraindre le-dit établisseur a_0 à se trouver nécessairement avant a et contraindre sa conclusion χ à s'unifier nécessairement avec φ ; pour rendre un terme φ vrai juste après une action a, il faut aussi s'assurer qu'il n'est pas masqué, même partiellement ; pour cela, identifier toutes les actions a_1 éventuellement situées avant a et dont la négation d'une conclusion ψ peut s'unifier avec φ ; pour éviter de transformer chaque a_1 en masqueur nécessaire, essayer d'abord de le repousser après a (*promotion*) ou de contraindre sa conclusion destructrice ψ à ne nécessairement pas s'unifier avec φ (*séparation*) ; sinon, trouver un rétablisseur potentiel a'_1, le contraindre à se situer nécessairement entre a_1 et a, et contraindre sa conclusion salvatrice ψ' à s'unifier avec φ chaque fois que ψ s'unifie avec φ.

On a là les moyens de modifier un graphe d'action que nous retrouverons et détaillerons en partie II :

- ajouter des contraintes de précédence entre actions ;
- ajouter des contraintes d'unification entre variables ;
- ajouter des actions quand celles présentes ne sont pas adéquates.

Une structure de contrôle globale se baserait sur les *préconditions* de chaque action pour identifier les valeurs attendues de chaque terme (un problème particulier de planification contient une *situation finale*, cf. l'axiomatisation de 4.3.1). Toute la difficulté vient évidemment du choix du moyen à utiliser pour donner une valeur à un terme (cf. chapitre 6) : c'est le problème de la structure de contrôle globale, que Chapman ne traite pas, rendant son programme TWEAK pratiquement inutilisable tel quel dans le monde réel[15]. Dans l'application

[15] *"Je supposerai que la structure de contrôle non-déterministe devine toujours correctement la première*

Pengi [Agre 87], une expertise simple acquise par la pratique du jeu d'arcade *Pengo* dont est tiré Pengi, le faible nombre d'actions envisageables et la technique de l'indexation des propositions (cf. l'introduction de 4.3) permettent de définir un contrôle a posteriori et en temps réel, en cablant le comportement du pingouin-planificateur sous formes de pseudo-règles d'inférences d'ordre 1 : la planification de son comportement est abandonnée et n'est plus perceptible qu'a posteriori, comme effet de bord de sa réactivité.

Critique Etant donné la clarté et la simplicité de sa théorie et de l'outil pratique résultant, Chapman peut utiliser son critère comme outil de mesure pour détecter les cas non traités par les planificateurs antérieurs (généralement empiriques) ; attitude certes instructive mais qui le conduit parfois à des jugements un peu sévères, ce qui justifie les tirs adverses nourris[16].

Si l'on considère le critère en lui-même, on peut mettre en défaut la rigueur dont se réclame Chapman : l'unification n'est pas l'unification *sous contraintes*, qui nécessite l'introduction de l'unification possible et nécessaire. Sans cette précision, le critère en ordre *total* que donne Chapman est ou bien trop flou (i-e faux) si on le considère en logique des prédicats du premier ordre, ou bien se ramène à celui que donne Dean en ordre 0 (toutes les variables sont instanciées). La même imprécision sur l'unification possible et nécessaire nuit à l'expression du critère en ordre partiel : sans unification possible ni nécessaire, que signifierait[17] le rétablissement $\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi'$? Chapman évacue tout simplement ce problème[18]. Manque de précision également à propos de la quantification des variables libres d'une formule : pour une conclusion $f(a,x)$, doit-on comprendre $f(a,x)$ pour *une seule* instanciation de x ($\exists x, f(a,x)$) ou pour *toute* instanciation de x ($\forall x, f(a,x)$) [Wilkins 86] ? Toujours à propos de variable, Chapman suppose le nombre de constantes infini, ce qui interdit l'introduction de règles de déduction simples du type : $x \in \{a,b,c\} \wedge x \neq a \wedge x \neq b \Rightarrow x = c$ ou encore de règles plus astucieuses du type (cf. figure 4.5) :

$$f(x,y) \in \gamma(a_1) \wedge f(y,x) \in \gamma(a_2) \wedge a_1 \prec a_2 \wedge x,y \in \{a,b\} \wedge x \not\approx y \Rightarrow f(a,b) \in post(a_1)$$

---

fois." ! [Chapman 85, p. 15]. Chapman s'en justifie un peu avant : *"Puisqu'aucune des stratégies de recherche développées jusqu'à présent ne semble très bonne, j'utilise simplement dans* TWEAK *une recherche en "largeur d'abord" dirigée par les dépendances. Je n'argumenterai pas en faveur du "profondeur d'abord" ; c'est certainement trop coûteux dans le cas général"* [Chapman 85, p. 13].

[16] Exemple d'échange musclé (débat connu entre Théorie et Pratique) :

Chapman (TWEAK) : *"Le traitement que fait* SIPE *[des effets de bords] est incomplet et incorrect en général."* [Chapman 85, p. 23].

Wilkins (SIPE) : *"Alors qu'il est facile de critiquer les algorithmes non linéaires à cause des hypothèses contradictoires qu'ils posent, on devrait considérer l'alternative liée à la résolution de tout problème NP-complet : si l'on peut se permettre d'attendre un temps arbitraire pour qu'un planificateur décide si un prédicat est déjà vrai ou non, Chapman (...) fournit un algorithme valide. Par contre, si l'on veut qu'un planificateur résolve des problèmes concrets, l'algorithme [que je propose] (...) est d'un certain intérêt."* [Wilkins 88, p. 83].

[17] En utilisant l'unification possible et nécessaire, ce rétablissement s'écrirait :

$$\Diamond(\varphi \approx \neg\psi) \wedge (\mathit{ajout\text{-}contraintes}(\varphi, \neg\psi) \wedge \Box(\varphi \approx \phi'))$$

en notant $\mathit{ajout\text{-}contraintes}(\varphi, \neg\psi)$ la modification des ensembles $unif^+$ et $unif^-$ des variables de φ et de $\neg\psi$, de façon que $\Box(\varphi \approx \neg\psi)$.

[18] *"Faire que [$\varphi \Rightarrow \neg\psi$] implique [$\varphi \Rightarrow \psi'$] est délicat ; on ne peux pas l'exprimer directement avec une contrainte"* [Chapman 85, p. 12].

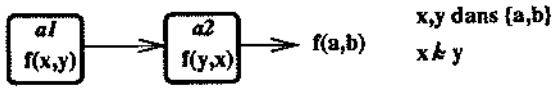


Figure 4.5: Cas linéaire non détecté par le critère

Ensuite, les critères de Chapman en ordre total ou partiel exigent que pour chaque masqueur a_1 on puisse trouver un rétablisseur intermédiaire a_1' (cf. le paragraphe précédent), ce qui est nécessaire mais pas suffisant : dans la figure 4.6, par exemple, les actions en séquence a_0, a_1, a_2 et a_3 concluent respectivement sur $f(a)$, $\neg f(x_1)$, $\neg f(x_2)$ et $f(x_3)$; si l'on veut que $f(a)$ soit vrai après a_3, a_1 est un établisseur, a_2 et a_3 sont des masqueurs et a_3 est un rétablisseur, mais pour qui ?

Chaque masqueur doit avoir son rétablisseur, mais un rétablisseur ne doit pas pouvoir servir plusieurs fois [Wilkins 88]. L'attribution d'un rétablisseur à chaque masqueur est ensuite d'une complexité de l'ordre de $O(n^2)$ (n est le nombre d'actions) et l'ordre de la réalisation de ces attributions (par ajout de contraintes) augmente également la complexité lorsque le graphe n'est pas bien parenthésé [Morignot 86].

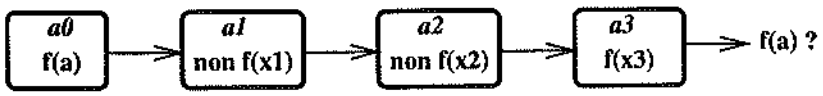


Figure 4.6: Autre cas linéaire non détecté le critère

Certains comportements simples ne sont pas recouverts pas ce critère. La durée des actions est implicitement nulle, ce qui exclut de pouvoir représenter la *simultanéité* : pour déplacer mon bureau, il faut lever simultanément deux bords et non un bord puis l'autre (la synergie des déménageurs). De même, ce critère ne peut pas représenter l'aspect cumulatif d'actions en parallèle : si une action (bancaire) coûte 28 FF, deux actions coûteront toujours 56 FF, que les dépenses soient faites en série ou en parallèle, alors que le critère de Chapman ne décèlerait toujours qu'un coût de 28 FF si les dépenses ont lieu en parallèle[19] ! Ces comportements nécessitent l'écriture d'autres critères (Chapman évalue leur nombre à une vingtaine pour couvrir tous les domaines).

Finalement, l'apport principal de Chapman est d'avoir clairement ramené le problème de la planification théorique au *problème du cadre*, imposant ainsi un vocabulaire (standardisation) et outil de jugement d'un système de planification.

[19] Les logiciels de PERT distinguent les ressources consommables (non exclusives, globalement cumulatives) des non-consommables (exclusives, prêtées / rendues).

Partie II

Planification sans expertise

Introduction

Nous avons présenté dans la partie précédente les techniques connues de planification en I.A. et en avons retenu (chapitre 4) qu'il est fondamental de rechercher d'abord à *quelle condition* on peut déduire quelque chose d'un plan d'actions, si l'on veut trouver une certaine logique dans le développement de ce plan (observation également basée sur l'expérimentation, voir ci-dessous). Ce désir de prudence, tant dans l'examen du plan que dans sa modification, s'est traduit pratiquement par la nécessité de définition préalable d'un critère de vérité.

Cette partie décrit le *critère de vérité* choisi et son utilisation, au moyen d'un *planificateur pur* complet, baptisé YAPS[20], pouvant raisonner sur des graphes *non-linéaires* d'actions en *ordre 1*.

Après avoir rappelé le formalisme de description des objets, du temps et du calcul, nous présenterons (chapitre 5) les algorithmes, en ordre 0 puis 1, implémentant la déduction d'information d'un graphe d'actions à l'aide de ce critère (*propagation de la vérité* dans notre planificateur). Puis (chapitre 6), toujours en nous basant sur notre critère de vérité, nous identifierons les amendements de plan possibles et leurs conditions d'application, et nous présenterons les heuristiques profitant des degrés de liberté restants. Enfin (chapitre 7), nous testerons ce planificateur sur quelques exemples issus des résolveurs de problèmes.

Historique Le planificateur YAPS présenté dans cette partie est le troisième planificateur théorique que nous avons été amenés à développer, l'absence de critère de vérité explicite pour les deux premiers s'étant traduit par une grande difficulté d'extension (du pouvoir expressif, par exemple).

Le premier, basé sur NOAH [Sacerdoti 77], empruntait les idées d'intervalles de valeur à NONLIN [Tate 77] et d'opérateur déductif à SIPE [Wilkins 84] ; bien qu'utilisant ces concepts forts, il ne pouvait guère évoluer en pratique, en raison d'une implémentation particulièrement naïve [Morignot 86].

Le deuxième, basé également sur NOAH, se concentrait sur la recherche d'heuristiques et tirait parti des facilités de programmation offertes par le langage orienté objet d'implémentation et par son moteur 1 intégré (TG2 [Assémat & Morignot 87]) ; cette fois, ce fut le câblage trop profond de certaines heuristiques qui empêcha toute possibilité d'évolution de ce code déjà volumineux (5000 lignes utiles en `lelisp15.x`).

Le troisième et dernier planificateur théorique, YAPS, codé chronologiquement après COPLANER, est, lui, basé sur une théorie d'ordre 1 (cf. 4.3), ne dépend d'aucune sur-couche de Lelisp15.2x et bénéficie d'une implémentation condensée et soignée (2000 lignes utiles en `lelisp15.x`) ; l'évolutivité pratique et conceptuelle est de ce fait conservée : l'implémentation d'heuristiques, dont le rôle n'est plus que de combler les trous théoriques, s'effectue (enfin) aisément.

[20] Yet Another Planning System.

Chapitre 5

Représentation et critères de vérité

Les planificateurs existants, que nous avons rapidement passés en revue au chapitre 3, peuvent être grossièrement répartis par volume et qualité de leur formalisation : certains planificateurs formalisent beaucoup, mais se justifient peu et démontrent des généralités[1] (formalisation riche et malhonnête) ; d'autres formalisent peu, mais savent prouver l'exactitude d'un comportement simple et précis[2] (formalisation pauvre et honnête) ; l'espoir d'une formalisation riche et honnête étant, depuis Gödel[3], abandonné.

Notre démarche est, à partir de représentations simples (i-e *pauvres*) des 3 entités du chapitre 1, de préciser la manière de déduire des informations d'un plan comprenant un ordre partiel sur la relation de précédence (cf. 5.1.3) et des variables (cf. 5.2). Plusieurs façons de déduire ces informations seront étudiées (cf. 5.3) et nous indiquerons à chaque fois une évaluation de la complexité des algorithmes les implémentant.

5.1 Représentation des intervenants

La génération de plan met en jeu les concepts d'*objet*, de *calcul* et de *temps* (d'après 1.1). Dans cette section, nous nous proposons d'étudier leur représentation ainsi que les algorithmes qu'elle nécessite.

[1] Tout ce qui fait la difficulté du problème est reporté dans une "fonction f", censée supporter tout le poids des connaissances utiles et pratiques en général, et qui sert de base à, par exemple, une preuve de NP-difficulté d'un problème. Le seul inconvénient est que la-dite fonction f est *incalculable en pratique*, ce qui est d'ailleurs parfois explicitement signifié ("il serait très difficile de calculer la fonction f" ... cette fois, nous ne citerons personne).

[2] La preuve de NP-difficulté du problème n'est alors que la prémisse à la présentation d'un algorithme polynomial dans un cas particulier (cf. les théories présentées au chapitre 4).

[3] L'arithmétique formelle étant incomplète si elle est consistante, tout système formel contenant l'arithmétique est à son tour incomplet s'il est consistant (cf. 2.3) [Lalement & Saint 86].

5.1.1 Représentation des objets

Est *objet* ce qui est transformé par l'application d'une action. Dans notre planificateur, leur représentation a été simplifiée à l'extrême, pour les rendre facilement manipulables, et se ramène à l'algèbre des termes (cf. 2.3.1) : les objets sont représentés par des constantes de Σ_{cst} ; leurs relations sont représentées par les symboles fonctionnels de la signature Σ au moyen de termes composés.

En pratique, un terme $f(a, x, b)$ s'écrit en notation (lispienne[4]) préfixée `(f a x b)`, un terme $\neg f(a, x, b)$ s'écrit en notation préfixée `(non (f a x b))`. Par exemple :

```
homme(Romeo)                          femme(Juliette)
amour(Romeo, Juliette)                amour(Juliette,Romeo)
clan-de(Romeo, Montaigu)              clan-de(Juliette, Capulet)
haine-ancestrale(Montaigu, Capulet)   haine-ancestrale(Capulet, Montaigu)
```

La tension dramatique de ce classique shakespearien provient, en terme de langage orienté objet, de l'ambiguïté de l'algorithme de résolution de conflit d'héritage et de la signification de son application au conflit particulier entre la relation (directe) `amour` et la relation (issue de l'héritage) `haine-ancestrale`, sous-relation de `haine` qui est justement la *négation* de `amour`.

Par rapport à l'algèbre des termes, nos termes auront une profondeur maximale de 3 (négation d'un terme composé de variables ou de constantes) ; les variables seront implicitement quantifiées existentiellement (justification détaillée en 5.2.2) ; une liste de tels termes est implicitement une formule conjonctive (ajouter des "$\wedge$" reliant les exemples ci-dessus).

Plusieurs arguments nous évitent toute jalousie mal placée vis-à-vis des autres systèmes de représentation (très évolués) que l'on peut trouver en I.A. : d'abord, les difficultés, que nous rencontrerons, trouveront leurs homologues dans toute autre représentation, aussi autant gagner en facilité de manipulation (pour le calcul de 5.3.2) en choisissant la représentation la plus souple ; ensuite, le pouvoir expressif de cette représentation "simple" est quand même suffisant pour traiter bien des problèmes (voir tous les développements effectués en Prolog) ; enfin et surtout, il est évident d'indexer des objets effectifs, par exemple en `leloo` (cf. annexe D), avec ces constantes, considérant ainsi le planificateur comme une couche basse d'une représentation moins fruste (ce qui sera précisément fait en partie III avec un critère bien particulier).

5.1.2 Représentation du calcul

Est *calcul* dans un planificateur tout ce qui permet de décrire la transformation que subissent les objets (indépendamment de la notion de temps).

Définition du calcul Depuis STRIPS, le calcul est présent dans un planificateur sous forme particulière d'*action* ou sous forme générique de *schéma d'actions*, qui (cf. 4.3.1) est un couple

[4] YAPS étant écrit en `lelisp15.2x`.

(π, γ) où π est l'ensemble des termes préconditions de l'action ($\pi \subset T_\Sigma[X]$) et γ l'ensemble des termes conlusions de l'action ($\gamma \subset T_\Sigma[X]$).

Pour définir la façon dont ces termes-conclusions γ transforment la situation entrante s_{entrante} en situation sortante s_{sortante}, nous décomposons chaque ensemble de termes manipulé (γ, s_{entrante}, s_{sortante}) en un sous-ensemble de termes positifs (de la forme $f(x, y, z)$), indicé positivement, et un sous-ensemble de termes négatifs (de la forme $\neg f(x, y, z)$), indicé négativement :

$$\begin{array}{lcll} \gamma & = & \gamma^+ & \cup \ \gamma^- \\ s_{\text{entrante}} & = & s^+_{\text{entrante}} & \cup \ s^-_{\text{entrante}} \\ s_{\text{sortante}} & = & s^+_{\text{sortante}} & \cup \ s^-_{\text{sortante}} \end{array}$$

Le calcul élémentaire effectué par l'action (π, γ), transformant la situation s_{entrante} en situation s_{sortante} s'exprime alors par les deux équations (voir figure 5.1):

$$\begin{cases} s^+_{\text{sortante}} = (s^+_{\text{entrante}} \setminus \gamma^-) \cup \gamma^+ \\ s^-_{\text{sortante}} = (s^-_{\text{entrante}} \setminus \gamma^+) \cup \gamma^- \end{cases}$$

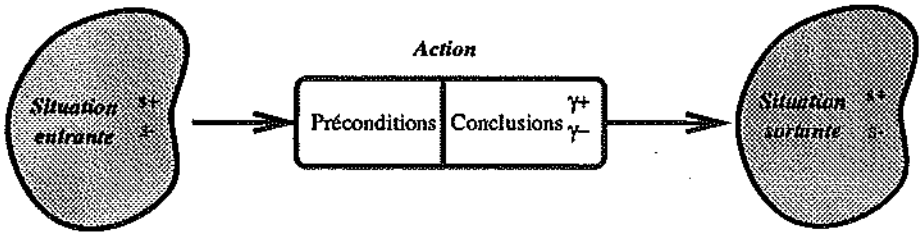


Figure 5.1: Situations entrantes et sortantes

L'hypothèse du *monde clos* n'est plus conservée : par défaut, un terme non mentionné n'est plus faux, mais inconnu. Mentionner un terme, c'est précisément lui donner une valeur *vrai* ou *faux*. Par contre, l'hypothèse STRIPS (cf. 3.1.2) tient toujours, puisque seul une action peut changer quelque chose à une situation entrante.

En pratique, cette définition pourrait suggérer que les termes positifs et négatifs d'une situation sont effectivement calculés. Ce serait beaucoup trop coûteux en place (cf. 4.1) : comme il y aurait autant de situations utiles à calculer que d'actions (la situation avant ou après chaque action), l'espace nécessaire pour stocker la situation initiale serait multiplié par la taille du graphe ! Dans le vocabulaire des systèmes experts, par exemple, expliciter les situations s'apparenterait à figer et stocker une base de faits *après chaque déclenchement de règle* ...

De plus, expliciter une situation n'a de sens que lorsque cette situation est consistante (sans conflits), or, dans sa progression, le planificateur ne peut que générer des situations conflictuelles (qu'il tente de résoudre ensuite) : s'il se basait sur l'explicitation des situations, il ne pourrait rien inférer de ces situations conflictuelles, pour tenter de lever ces conflits.

Comme pour STRIPS (cf. 3.1.2), les préconditions π d'une action déterminent son applicabilité : *ce qui doit exister* et *ce qui ne doit pas exister* (termes commençant par $\neg$ ou *non* dans

les figures) pour que l'exécution de cette action puisse s'effectuer[5].

Notre planificateur utilise des *variables* pour représenter classiquement une absence de choix sur les objets manipulés (planificateur d'ordre 1, selon 4.3) : pour obtenir une action d'un schéma d'actions particulier, il n'est pas besoin d'instancier *toutes* les variables, certaines n'étant que re-nommées (de nouvelles variables libres pour éviter des conflits de nom).

Les termes, intervenant en précondition π ou en conclusion γ d'une action, sont soit basés sur la signature Σ correspondant à la sémantique du domaine d'application, soit sont des égalités ou des inégalités entre variables et constantes (qui se traduisent d'emblée par des contraintes d'unification, comme on le verra en 5.2). La liste π avec variables crée un environnement de liaison des variables de l'action, les liaisons variables/valeurs restant bien sûr valables en partie γ (à la façon de tout moteur d'inférences d'ordre 1). La liste π sert donc non seulement à spécifier la situation entrante (rôle initial des préconditions), mais aussi à poser des contraintes sur les variables internes de l'action (voir un exemple complet en 7.1), au moyen des symboles évidents $\neg$ et $=$ (supposés toujours appartenir à la signature Σ).

5.1.3 Représentation du temps

Temps discret et non numérique Le temps, dernière des trois composantes de la planification selon 1.1, est représenté sous forme symbolique, de la façon la plus simple, par la relation de précédence "est-exécutée-avant" entre deux actions. Le domaine d'application est donc modifié de façon discrète[6], les transformations discrètes que représentent les actions s'effectuent instantanément.

Tout aspect numérique (durée non nulle d'une action, lien de précédence valué à la façon d'un PERT, ..., cf. 2.2) est volontairement évacué, notre problème étant plus de découvrir au moins un plan-solution (i-e dont la logique interne prouve qu'il réalise le but recherché, même mal) que d'en affiner l'exécution[7]. Cependant, une interprétation numérique du temps, par exemple en terme de liens valués de type PERT, reste possible une fois la phase de génération du plan effectivement terminée : les actions de notre planificateur possèdent les attributs "date de début au plus tôt", "durée[8]", "date de fin au plus tard" (cf. 2.2) qui sont utilisées lorsqu'une durée n'est pas nulle.

Cette relation de précédence définit, suivant le planificateur, un ordre total ou partiel $\preceq$ sur l'ensemble des actions du plan ; on peut alors parler d'un plan comme d'un *graphe* d'actions, acyclique orienté dont la *source* (initialement unique, ou ajoutée et donc unique) est la situation

[5] Ce champ *précondition* ne doit pas être confondu avec le champ *déclencheur* (*trigger* de [Wilkins 84]), qui conditionne la décision d'*ajouter* ou non l'action dans le plan, alors que le champ *précondition* n'entre en jeu qu'après, une fois l'action présente dans le plan.

[6] Les relations possibles entre les pseudo-instants que sont les actions sont $<, =, >$, ce qui peut rappeler le point de départ de la logique des instants de Mc Dermott (cf. 2.6.2).

La seule façon de représenter une transformation continue serait de la discrétiser, i-e de lui substituer un grand nombre de transformations discrètes (tout passage à la limite, à la façon de l'intégrale de Riemann, étant évidemment exclu !). Pour éviter de sombrer dans la combinatoire correspondante, un traitement hiérarchique pourrait alors être utile.

[7] Respectant en cela une des plus célèbres *lois de Murphy* : *"il ne sert à rien d'optimiser un programme [ou un plan] qui ne marche pas car il est mal conçu"*.

[8] Etant entendu que cette durée n'a aucun effet sur le calcul de la persistance des conclusions des actions.

initiale et le *puits* la situation finale (cf. 1.2.1).

Dans notre cas, il s'agit d'un ordre partiel sur les actions, puisque l'ordre partiel est un moyen simple de manipuler tous les ordres totaux qui en sont des extensions. Nous conviendrons que le parallélisme induit par cet ordre partiel représente une *absence de choix* sur l'ordre d'exécution effectif des actions (une formule conjonctive, cf. 3.2.1) : si a et a' sont deux actions en parallèle, elles devront toutes les deux être exécutées, mais on ne sait pas (encore) si $a \prec a'$ ou[9] si $a \succ a'$.

Non fermeture transitive Lors de la représentation de l'ordre partiel sur les actions par un graphe d'actions, la transitivité de $\preceq$ est gênante car elle peut d'une part conduire à une inutile multiplication des liens et surtout créer des erreurs de jugement lors de l'évaluation de la vérité de termes.

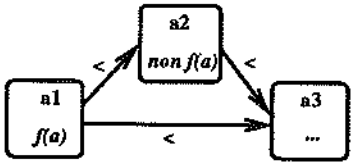


Figure 5.2: Exemple d'erreur sur la valeur d'un terme, due à la transitivité de $\prec$

Dans la figure 5.2 par exemple, l'action a_1, concluant sur $f(a)$, est avant l'action a_2, concluant sur $\neg f(a)$, qui est avant l'action a_3. Si le lien $a_1 \rightsquigarrow a_3$ est présent, on pourrait croire qu'il y a un conflit sur le terme $f(a)$ juste avant a_3, puisqu'il y arrive négativement via a_2 et positivement via a_1. Il n'y a pourtant pas de conflit, puisque la *dernière* action avant a_3 est a_2, et non a_1 : $f(a)$ est faux juste avant a_3 (cf. la définition et l'explication du critère de calcul en 5.3.2). Ce sont donc les liens *directs* (sans intermédiaires) entre actions qui ont un sens vis-à-vis de la vérité des termes.

Ce *filtrage* sur la relation de précédence est réalisé par la condition de minimalité suivante (en notant $\rightsquigarrow$ le lien du graphe d'actions) :

$$\forall a, a' \in \mathcal{A},\ a \rightsquigarrow a' \ \Rightarrow\ \neg(\exists a'' \in \mathcal{A},\ a \prec a'' \prec a')$$

Un lien entre deux nœuds $a \rightsquigarrow a'$ signifie non seulement que les actions des nœuds se précèdent $a \prec a'$, mais aussi qu'il n'y aucune action intermédiaire. Ce point est vital pour le futur critère de vérité (cf. 5.3.2), comme on a pu partiellement le constater sur l'exemple précédent, puisqu'une action doit pouvoir accéder directement aux *dernières* actions avant elle.

En pratique, pour ajouter une contrainte de précédence entre deux actions a_1 et a_2, à partir d'un graphe vérifiant déjà l'équation ci-dessus, il faut vérifier les points suivants :

[9] Attention sur la signification de ce "OU" : notre interprétation du parallélisme des actions a et a' par "$a \prec a' \vee a \succ a'$" ne doit surtout pas être confondue avec celle, classique, des parcoureurs de graphe d'états : "$a \vee a'$". Cette dernière signifie que le chemin-solution passe par a ou passe par a', mais pas par les deux (seule l'une des deux actions doit être effectuée), ce qui n'a rien à voir avec ce que nous proposons.

1. **d'abord s'assurer que cette contrainte n'existe pas déjà implicitement (par fermeture transitive), i-e qu'il n'existe pas déjà de chemin allant de a_1 à a_2 ;**
2. **ensuite s'assurer que cette contrainte est consistante avec le graphe présent, i-e qu'il n'y a pas déjà $a_2 \prec a_1$, ce qui revient à vérifier qu'il n'existe pas de chemin allant de a_2 à a_1 ;**
3. **poser effectivement $a_1 \rightsquigarrow a_2$;**
4. **rendre le graphe minimal : enlever les éventuels chemins directs que le lien $a_1 \rightsquigarrow a_2$ a pu rendre obsolète.**

Bien que devant être vérifié a priori, le premier point sera implicitement déjà traité par le futur critère de 5.3.2. Ce dernier point, conservant la minimalité du graphe, est en fait plus basé sur la contre-apposée de la relation de minimalité que la relation de minimalité elle-même :

$$\forall a, a' \in \mathcal{A},\ \exists a'' \in \mathcal{A},\ a \prec a'' \prec a' \ \Rightarrow\ a \not\rightsquigarrow a'$$

Si une action a'' est entre deux autres actions a et a', il ne doit pas y avoir de lien $\rightsquigarrow$ entre a et a' bien que $a \preceq a'$. Le chemin direct entre deux nœuds du graphe doit être effacé s'il existe un autre chemin (plus long) entre eux.

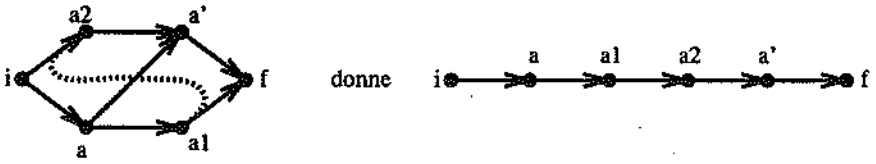


Figure 5.3: Exemple d'effacement de lien dû à la minimalité

Dans l'exemple de la figure 5.3 (4 actions a, a', a_1, a_2, action initiale i représentant la situation initiale en conclusion γ, action finale f représentant la situation finale en précondition π), le lien $a_1 \prec a_2$ n'est ni déjà implicite, ni générateur d'inconsistances, et peut donc être posé par $a_1 \rightsquigarrow a_2$. Pour re-rendre le graphe minimal, il faut supprimer les liens $i \rightsquigarrow a_2$ (chemin $i \rightsquigarrow a \rightsquigarrow a_1 \rightsquigarrow a_2$), $a_1 \rightsquigarrow f$ (chemin $a_1 \rightsquigarrow a_2 \rightsquigarrow a' \rightsquigarrow f$), et aussi le lien $a \rightsquigarrow a'$ (chemin $a \rightsquigarrow a_1 \rightsquigarrow a_2 \rightsquigarrow a'$). Dans l'exemple, l'ajout d'un seul lien a linéarisé le graphe initial.

Ceci aboutit à l'algorithme suivant, qui efface les liens préservant la minimalité du graphe, suite à l'ajout du lien $a_1 \rightsquigarrow a_2$:

Procédure NFT-EFFACE(a_1, a_2)
- **marquer les actions $a_{\succeq}$ après a_2 au sens large ($a_2 \preceq a_{\succeq}$) avec la marque $m_{\succeq}$**
- ***Recherche d'un début de lien avant a_1***
- **pour toute action $a_{\preceq}$ avant a_1 au sens large ($a_{\preceq} \preceq a_1$)**
  si il existe un successeur direct $a'_{\preceq}$ de $a_{\preceq}$ ($a_{\preceq} \rightsquigarrow a'_{\preceq}$) qui est marqué avec $m_{\succeq}$, alors
  détruire le lien $a'_{\preceq} \rightsquigarrow a_{\preceq}$
- **démarquer les actions $a_{\succeq}$**

L'algorithme NFT-EFFACE est simplement basé sur le fait que le graphe vérifie initialement la condition de minimalité : les seuls liens directs à effacer ne peuvent qu'être dus à un nouveau long chemin qui passe par $a_1 \rightsquigarrow a_2$. Les extrémités de ce long chemin sont évidemment les mêmes que celles de l'éventuel lien à effacer, aussi les éventuels liens à effacer ne peuvent donc concerner qu'une action $a_{\preceq}$ avant a_1 au sens large et une action $a_{\succeq}$ après a_2 au sens large. Sont finalement recherchées deux actions $a_{\preceq}$ et $a_{\succeq}$ telles que : $a_{\preceq} \preceq a_1 \wedge a_2 \preceq a_{\succeq} \wedge a_{\preceq} \rightsquigarrow a_{\succeq}$. L'algorithme NFT-EFFACE détecte ces couples $(a_{\preceq}, a_{\succeq})$ en effectuant une intersection entre les successeurs directs des actions avant a_1 et les actions après a_2.

D'un point de vue complexité, soit $n_{\succeq}$ le nombre d'actions après a_2, soit $n_{\preceq}$ le nombre d'actions avant a_1, soit n_{succ} le nombre maximal de successeur : le premier parcours construit l'ensemble $\mathcal{A}_{\succeq}$ des actions $a_{\succeq} \succeq a_2$, en les marquant ; le nombre d'actions considérées est borné supérieurement par $n_{\succeq}$. Le deuxième parcours cherche une action $a_{\preceq} \preceq a_1$ telle que $\exists a'_{\preceq} \in \mathcal{A}_{\succeq}$, $a_{\preceq} \rightsquigarrow a'_{\preceq}$ (l'intersection de ci-dessus) ; le nombre d'actions considérées est borné supérieurement par $n_{\preceq} + n_{\preceq} n_{\text{succ}}$. Le dernier parcours réinitialise proprement les marques des actions $a_{\succeq}$ de $\mathcal{A}_{\succeq}$; comme dans le premier parcours, le nombre d'actions considérées est borné supérieurement par $n_{\succeq}$. Le nombre total d'actions vues est donc $2(n_{\preceq} + n_{\succeq}) + (n_{\text{succ}} - 1)n_{\preceq}$; la complexité de parcours de NFT-EFFACE est donc légèrement supérieure à $O(2n)$.

Ce résultat n'est pas symétrique (échanger $n_{\preceq}$ et $n_{\succeq}$), car il faut bien choisir entre rechercher l'intersection entre les actions après a_2 et les successeurs des actions avant a_1, et rechercher l'intersection entre les actions avant a_1 et les prédécesseurs des actions après a_2.

Remarquons enfin que cette non fermeture transitive a l'avantage (initialement non voulu) d'améliorer la complexité de stockage du graphe, puisqu'elle supprime certains liens $\rightsquigarrow$ au cours de son évolution.

L'instant présent Les actions ne sont repérées temporellement que de façon relative : les seules "dates", absolues, s'obtiennent en interprétant la pseudo-action initiale comme la date de début ("maintenant") ou la pseudo-action finale comme la date de fin (*date maximale de réalisation du plan*). On peut pourtant faire progresser l'instant présent (le "maintenant") selon une démarche voisine de celle de Mc Dermott (cf. 2.6.2) : si l'instant présent n'est plus seulement l'extrémité gauche du graphe, la représentation du *passé* doit être *linéaire* : les actions du passé ont été exécutées dans un certain ordre qu'il suffit de noter a posteriori (le temps réel est inplicitement supposé linéaire).

La progression du temps déterminant la position de l'instant présent dans le graphe nécessite de (voir figure 5.4) :

1. déterminer les actions du passé : ces actions du passé $\mathcal{P}$ ne doivent pas encadrer une action non passée $(\neg(\exists a', a'' \in \mathcal{P}, \exists a \in \mathcal{A},\ a' \preceq a \preceq a''))$;
2. étendre l'ordre partiel $\preceq$ en un ordre total, relativement aux actions passées de $\mathcal{P}$;
3. (éventuellement) exécuter ces actions passées totalement ordonnées en une seule action, la nouvelle pseudo-action initiale.

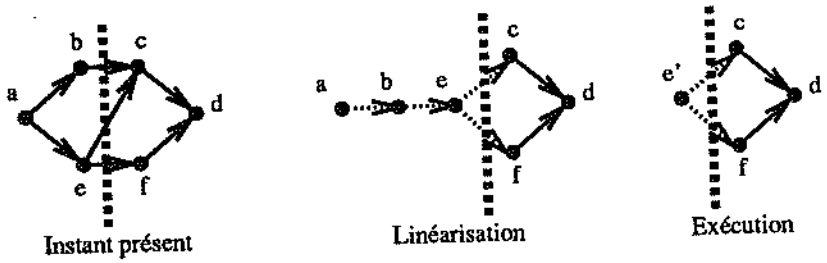


Figure 5.4: Progression de l'instant présent

Le premier point fixe le futur instant présent désiré (voir l'exemple de la figure 5.4). Le deuxième force le choix de l'ordre d'exécution de ces futures actions passées ; tous les liens passant à travers l'instant présent posé sont rebranchés sur l'action présente ($b \prec e$ dans l'exemple). Le troisième point exécute ces actions passées (a, b et e), rejetant l'instant présent dans une nouvelle situation initiale (e').

Ce compactage par exécution n'est possible que s'il n'y a aucun conflit parmi ces actions passées.

Cette nouvelle pseudo-action initiale se calcule en ne conservant que les derniers prédicats, pour les termes sans variable, et en compactant listant tous les termes avec variables puisqu'ils sont sans conflit.

5.2 Contraintes sur les variables

Pour représenter les contraintes d'unification et de non-unification, nous définissons la fonction $unif : X \longrightarrow \wp(X \cap \Sigma_{cst})^2$, qui, pour toute variable de X, fournit un couple $unif(x) = (unif^+(x), unif^-(x))$, où $unif^+(x)$ est l'ensemble des termes (variables de X ou constantes de Σ_{cst}) avec lesquels x doit nécessairement s'unifier, et où $unif^-(x)$ est l'ensemble des termes (variables de X ou constantes de Σ_{cst}) avec lesquels x ne doit nécessairement pas s'unifier (cf. 5.2.1). La relation $unif^+$ étant une relation d'*équivalence*, on peut en effectuer la fermeture transitive : on convient que, pour toute variable x de X, $unif^+(x)$ contient *tous* les termes avec lesquels x doit nécessairement s'unifier : $\forall x, y, z \in X,\ y \in unif^+(x) \wedge z \in unif^+(y) \Rightarrow z \in unif^+(x)$.

Cette fonction est une extension de la substitution σ solution d'une unification, le cas des contraintes sur les constantes et les termes composés n'a pas à y être représenté.

Les fonctions *unif* de chaque variable manipulée sont regroupées en pratique dans la *base de contraintes d'unification*. Tous les algorithmes et définitions présentés maintenant visent à préserver la consistance de cette base, i-e à aucun moment, on ne doit avoir :

$$\exists x, x' \in X,\ x \in unif^+(x') \wedge x' \in unif^-(x)$$

5.2.1 L'unification modale

Une façon élégante d'intégrer ces contraintes d'unification dans l'unification standard consiste à utiliser les qualificatifs modaux de nécessité $\Box$ et de possibilité $\Diamond$.

Justification intuitive Il est *possible* que deux termes t et t' de $T_\Sigma[X]$ s'unifient ($\Diamond t \approx t'$) ssi ils s'unifient au sens standard (non modal) et il n'y a aucune contre-indication à leur unification provenant des contraintes $unif^-$ (i-e l'ajout de la contrainte d'unification ne rend pas la base incohérente). Deux termes t et t' de $T_\Sigma[X]$ s'unifient *nécessairement* ($\Box t \approx t'$) ssi ils s'unifient au sens standard (non modal) et ils sont contraints à s'unifier (l'un est dans l'ensemble $unif^+$ de l'autre). Intuitivement, l'unification nécessaire correspond à ce qui existe effectivement ; l'unification possible permet d'obtenir quand même des informations sur des variables non encore instanciées, i-e qui ne s'unifient pas encore nécessairement avec une constante (*objets partiellement contraints*, selon [Stefik 81]).

En pratique, dans la définition de l'unification nécessaire, le test d'unification standard des deux termes est redondant, puisqu'on ne contraindra deux termes à s'unifier *nécessairement* qu'en s'étant au préalable assuré qu'ils peuvent bien s'unifier *possiblement*, ce qui inclut les tests d'unification standard. On ne fait en cela que mettre en pratique la relation suivante entre les unifications standard, possible et nécessaire :

$$\forall t, t' \in T_\Sigma[X],\ \Box(t \approx t') \Rightarrow \Diamond(t \approx t') \Rightarrow t \approx t'$$

Cette relation permet de conserver à tout moment une base de contraintes d'unification *consistante*, i-e qui vérifie (en adaptant la définition du paragraphe précédent) :

$$\neg(\exists t, t' \in T_\Sigma[X],\ \Box t \approx t' \wedge \Box t \approx (\neg t'))$$

Définitions détaillées par cas D'après la relation ci-dessus, les définitions de l'unification possible et nécessaire de deux termes n'ont de sens que lorsque les termes s'unifient effectivement. Ces termes étant soit des constantes, soit des variables, soit composés de sous-termes constantes ou variables, cette définition doit distinguer trois cas pour les termes atomiques (2×2, moins un par symétrie) et un cas pour les termes composés. Le détail de ces quatre cas est le suivant :

- pour deux constantes a et b de Σ_{cst}, les unifications nécessaire, possible et standard sont toutes équivalentes à l'égalité de l'agèbre des termes : $\Box(a \approx b) \Leftrightarrow \Diamond(a \approx b) \Leftrightarrow (a \approx b) \Leftrightarrow (a = b)$

- pour une variable x de X et une constante a de Σ_{cst} :

$$\begin{array}{lcll} \Box(x \approx a) & \stackrel{\text{def}}{\Longleftrightarrow} & x \approx a \ \wedge & \exists x' \in \{x\} \cup (unif^+(x) \cap X),\ a \in unif^+(x') \\ \Diamond(x \approx a) & \stackrel{\text{def}}{\Longleftrightarrow} & x \approx a \ \wedge & \forall x' \in \{x\} \cup (unif^+(x) \cap X),\ a \notin unif^-(x') \end{array}$$

  Une variable s'unifie nécessairement à une constante ssi elle y est contrainte, directement (par elle-même) ou indirectement (par une variable nécessairement équivalente). Il est possible qu'une variable s'unifie à une constante ssi cette variable, ou toute autre variable nécessairement équivalente, n'est pas contrainte à ne pas le faire.

- pour deux variables x et y de X :

$$\Box(x \approx y) \stackrel{\text{def}}{\Longleftrightarrow} x \approx y \wedge (x \in \mathit{unif}^+(y) \vee y \in \mathit{unif}^+(x))$$

$$\Diamond(x \approx y) \stackrel{\text{def}}{\Longleftrightarrow} \left\{ \begin{array}{l} x \approx y \\ \wedge \\ \quad \neg \left(\begin{array}{l} \exists x' \in \{x\} \cup (\mathit{unif}^+(x) \cap X), \\ \exists u \in \mathit{unif}^-(x'), \\ \exists y' \in \{y\} \cup (\mathit{unif}^+(y) \cap X), \\ \exists v \in \mathit{unif}^-(y'), \end{array} \quad x' \approx v \vee u \approx y' \right) \end{array} \right.$$

Deux variables s'unifient nécessairement ssi l'une au moins est contrainte à le faire. Le "$\vee$" vient du fait qu'on n'a fait aucune hypothèse sur la fonction *unif*, en particulier on n'a pas imposé la symétrie : $\forall x, y \in X,\ x \in \mathit{unif}^+(y) \Rightarrow y \in \mathit{unif}^+(x)$. Il est possible que deux variables s'unifient ssi il n'existe pas de variable ou constante avec laquelle l'une de ces deux variables (ou une de ses variables nécessairement équivalentes) doit nécessairement s'unifier, et avec laquelle l'autre variable (ou une de ses variables nécessairement équivalentes) ne doit nécessairement pas s'unifier.

- pour deux termes composés $f(t_1, \ldots, t_n)$ et $f'(t'_1, \ldots, t'_{n'})$, le possible et le nécessaire se définissent par induction :

$$\Box(f(t_1, \ldots, t_n) \approx f'(t'_1, \ldots, t'_{n'})) \stackrel{\text{def}}{\Longleftrightarrow} f = f' \wedge n = n' \wedge \forall i \in [1, n], \Box(t_i \approx t'_i)$$
$$\Diamond(f(t_1, \ldots, t_n) \approx f'(t'_1, \ldots, t'_{n'})) \stackrel{\text{def}}{\Longleftrightarrow} f = f' \wedge n = n' \wedge \forall i \in [1, n], \Diamond(t_i \approx t'_i)$$

Deux termes composés s'unifient nécessairement (resp. possiblement) ssi les symboles fonctionnels sont identiques, les nombres d'arguments sont identiques et les sous-termes s'unifient nécessairement (resp. possiblement).

On peut facilement démontrer que la relation de dualité entre $\Diamond$ et $\Box$ (i-e $\forall t, t', \Diamond(t \approx t') \Leftrightarrow \neg\Box(t \not\approx t')$) est bien vérifiée dans chacun des cas ci-dessus.

5.2.2 Complétion de variables

Nous avons vu au chapitre 4 que les notions de précédence *possible* et *nécessaire* se définissent par rapport à l'extension d'un ordre partiel $\prec$ en un ordre total $\prec'$: un terme φ est nécessairement (resp. possiblement) vrai juste après la situations a ssi φ est vrai juste après a pour toute (resp. au moins une) extension totale $\prec'$ de $\prec$. Les notions d'unification *possible* et *nécessaire* peuvent s'interpréter d'une façon analogue.

Quantification existentielle implicite Toute formule manipulée φ a ses variables libres implicitement quantifiées de façon existentielle :

$$\varphi(x, y, z) \text{ doit être lue comme } \exists x, y, z \in X, \varphi(x, y, z)$$

Et en particulier : $\neg\varphi(x, y, z)$ doit être lue comme $\exists x, y, z \in X, (\neg\varphi(x, y, z))$

Si $\neg\varphi(x,y,z)$ se lisait comme $\neg(\exists x,y,z \in X, \varphi(x,y,z))$, c'est-à-dire $\forall x,y,z \in X, \neg\varphi(x,y,z)$, un quantificateur universel apparaîtrait. Comme aucun autre axiome ou règle de réécriture de la logique des prédicats d'ordre 1 ne génère de quantificateur universel (cf. 2.3.1), une première conséquence de la quantification existentielle implicite est que *le quantificateur universel ne peut pas apparaître dans une formule, sauf s'il y est explicitement introduit.* L'ensemble des formules $L'_\Sigma[X]$, sous-ensemble de $L_\Sigma[X]$ uniquement construit sur le quantificateur $\exists$, est donc stable par dérivation selon les règles d'inférences usuelles.

La quantification universelle Si le système n'introduit pas intempestivement de $\forall$ et que les variables sont sagement quantifiées existentiellement, l'utilisateur pourrait volontairement en introduire en pré- ou en post-condition d'un schéma d'action, sous forme d'une terme $\forall x, \varphi(x)$.

Le traitement d'un tel terme pourrait se ramener aux cas discutés ici grâce à l'équivalence opératoire suivante :

$$\forall x, \varphi(x) \Leftrightarrow \bigwedge_{c_i \in \Sigma_{cst}} \varphi(c_i)$$

Le traitement du $\forall$ se ramène à une boucle sur les constantes de la signature, ce qui imposerait aux constantes de Σ d'être en nombre fini. Par exemple, si $\varphi(x)$ est un terme litéral, le terme $\forall x, \varphi(x)$ *en précondition* signifierait, en mode déductif, que, pour toute constante c_i de la signature, il faudrait vérifier la valeur de $\varphi(c_i)$, et qu'en mode inductif (voir chapitre 6) il faudrait donner la valeur de $\forall x, \varphi(x)$ à chaque $\varphi(c_i)$. Le terme $\forall x, \varphi(x)$ en postcondition signifierait simplement que tous les termes $\varphi(c_i)$ sont en postcondition.

L'assimilation du quantificateur universel à une macro-expansion sous forme de boucle conjonctive fonctionnerait correctement pour des termes litéraux ($\forall x, \varphi(x)$ ou $\forall x, \neg\varphi(x)$). Dans ces termes, les constantes sur lesquelles s'effectue la boucle *ne dépendent pas de la situation courante*, mais de caractéristiques atemporelles des constantes (*"pour tous les cubes rouges"* voire *"pour tous les cubes de plus de 3 cm de haut"*, mais pas *"pour tous les cubes sur le cube* a" qui dépend de la situation). Sinon, le terme manipulé serait plus complexe, par exemple de la forme $\forall x, P(x) \Rightarrow \varphi(x)$ avec un terme $P(x)$ dépendant de la situation, et on retomberait sur l'écueil des conditions conditionnelles [Wilkins 84] qui demandent un critère de vérité différent de celui que nous avons utilisé (le problème de la détermination de la valeur de vérité d'un tel terme est alors NP-complet [Chapman 85, p. 45]).

Etant donné le critère de vérité utilisé (cf. 5.3), le traitement général du quantificateur universel n'a pas été introduit dans notre planificateur, bien que le *quantificateur universel réduit* (macro-expansion indépendante de la situation sur des constantes supposées en nombre fini) puisse s'implémenter sans difficulté.

Instanciation et skolemisation On peut maintenant interpréter la vérité d'une formule $\varphi(x)$, ou plutôt $\exists x \in X, \varphi(x)$, par : il existe(ra) une constante a de Σ_{cst} telle que $\varphi(a)$ soit vraie (*skolemisation* [Lalement & Saint 86]). Une formule contenant une variable est vraie ssi il existe une constante, en laquelle la variable peut s'instancier, qui rende la formule vraie. Dans notre formalisme de contraintes d'unification, une formule $\varphi(x)$ est vraie ssi $\exists a \in \Sigma_{cst}$, $\Diamond(x \approx$

a) [Nilsson 80]. Autrement dit, pour toute variable manipulée x de X, il doit exister une constante a de Σ_{cst} telle que, à tout instant du processus de planification, $\Diamond(x \approx a)$.

Considérer une variable comme un moyen commode de remplacer temporairement une constante future aurait permis de donner directement une définition de l'unification possible et nécessaire, au moyen de l'*instanciation* : une instanciation ι est une *susbsitution* de la théorie de l'unification (cf. le deuxième paragraphe de 4.3.1) réduite au cas où les variables ne peuvent être remplacées que par des constantes : si une substitution σ est une application de l'ensemble des variables dans l'ensemble des termes fermés ($\sigma : X \to T$), une instanciation ι est une substitution dont l'image est réduite aux constantes ($\iota : X \to \Sigma_{cst}$). On aurait alors pu reconstruire une "théorie de l'instanciation" en se calquant sur la construction de la théorie de l'unification [Lalement & Saint 86], aboutissant à la définition de la relation "$\approx$" : soit $\mathcal{I}$ l'ensemble des instanciations, pour toute formule φ et ψ de $L'_\Sigma[X]$,

$$\varphi \approx \psi \stackrel{\text{def}}{\Longleftrightarrow} \forall \iota \in \mathcal{I},\ \iota(\varphi) = \iota(\psi)$$

Cette définition peut s'étendre de façon modale[10] :

$$\begin{aligned} \Diamond(\varphi \approx \psi) &\stackrel{\text{def}}{\Longleftrightarrow} \exists \iota \in \mathcal{I},\ \iota(\varphi) = \iota(\psi) \\ \Box(\varphi \approx \psi) &\stackrel{\text{def}}{\Longleftrightarrow} \forall \iota \in \mathcal{I},\ \iota(\varphi) = \iota(\psi) \end{aligned}$$

Les définitions ci-dessus permettent de reconstruire facilement une version amoindrie de l'unification classique et d'en retrouver toutes les propriétés connues.

Aspect pratique Un plan peut être complété non seulement en étendant l'ordre partiel des actions en ordre total (ajout de contraintes de précédence, cf. le chapitre 4) mais aussi en fixant les instanciations des variables utilisées dans le plan courant (ajout de contraintes d'instanciation).

Dans cette vision, une variable est un moyen de représenter une *absence de choix* dans un jeu de constantes donné. L'ensemble des contraintes représentant un problème de planification est alors *cohérent* ssi l'ensemble des instanciations de toute variable du problème est non vide (toute variable possède au moins une instanciation). Si l'ensemble des instanciations possibles d'une variable devient vide, c'est que l'ensemble des contraintes posées est inconsistant ; il faut alors desserrer une de ces contraintes (explorer une autre branche de recherche) et recommencer le processus.

Cette utilisation des variables est l'analogue symbolique, ou discret, du cas continu des *fenêtres temporelles* de 2.2.

En fait, une constante représente un objet (ou une de ses caractéristiques) du micro-monde modélisé : l'ensemble des constantes Σ_{cst} d'un problème de planification sera donc en pratique fini[11].

[10] Les deux définitions ci-dessus ne sont que partielles puisque les contraintes ne sont pas représentées.

[11] Chapman a un besoin impératif d'un ensemble *infini* de constantes pour son curieux "codesignation consistency lemma" [Chapman 85, p. 40], à cause, semblerait-il, de sa gestion des contraintes d'unification, qui devrait autoriser des contraintes *explicites* sur les constantes. Cette gestion des contraintes d'unification et de précédence ne semble pas très claire dans l'esprit de Chapman, qui sous-estime, en tout cas, nettement le problème (id., p.

Considéré comme un programme de résolution de problèmes, le planificateur est dans la situation du programme ALICE [Laurière 76] face à un problème de nombres entiers (en numérotant arbitrairement les constantes).

5.2.3 Gestion des contraintes d'unification

L'unification possible et nécessaire, interprétation de l'unification sous contraintes, est l'une des couches basses de YAPS les plus sollicitées, aussi nous indiquons le versant algorithmique avec une évaluation de la complexité.

Circularité et aplatissement des classes d'équivalences Si $\mathcal{C}$ est une classe d'équivalence pour la relation d'équivalence "s'unifie-nécessairement-à", on a par définition la relation $\forall x, y \in \mathcal{C}, y \in \mathit{unif}^+(x)$, ce qui se traduit par $\forall x \in \mathcal{C}, \mathit{unif}^+(x) = \mathcal{C} \setminus \{x\}$. Stocker effectivement $\mathit{unif}^+(x)$ sous cette forme serait trop coûteux, aussi on préfère utiliser une version relationnelle plutôt qu'ensembliste de unif^+, noté $\overline{\mathit{unif}^+}$, et vérifiant une condition moins forte :

$$\forall x, y \in \mathcal{C}, \exists n \in [0, \mathrm{Card}(\mathcal{C}) - 1],\ y \in (\overline{\mathit{unif}^+})^n(x)$$

Cette version permet de limiter $\overline{\mathit{unif}^+}(x)$: $\forall x \in \mathcal{C}$, $\mathrm{Card}(\overline{\mathit{unif}^+}(x)) = 1$. La relation $\overline{\mathit{unif}^+}$ indique en fait pour chaque variable de la classe une autre variable de la classe (éventuellement elle-même pour une nouvelle classe à un seul élément), de façon que toute variable de $\mathcal{C}$ soit accessible en $\mathrm{Card}(\mathcal{C})$ coups maximum. On notera parfois directement $\overline{\mathit{unif}^+}(x)$ cet unique élément qu'il est censé *contenir*.

Si $\mathcal{C} = \{x_0, \ldots, x_{\mathrm{Card}(\mathcal{C})-1}\}$, cette propriété se traduit par une contrainte de circularité :

$$\begin{cases} \forall i \in [1, \mathrm{Card}(\mathcal{C}) - 1], \mathit{unif}^+(x_{i-1}) = \{x_i\} \\ \mathit{unif}^+(x_{\mathrm{Card}(\mathcal{C})-1}) = \{x_0\} \end{cases}$$

La deuxième équation assure la circularité.

Dans les deux cas, la complexité de parcours est la même : $O(n)$ pour parcourir l'ensemble $\mathit{unif}^+(x)$ initial ou la nouvelle relation unif^+. Par contre, pour les ensembles $\mathit{unif}^+(x)$, la complexité de stockage est passé de $O(n^2)$ à $O(n)$; pour les ensembles $\mathit{unif}^-(x)$, la complexité de stockage est passée de $O(nm)$ à $O(m)$, où m est le nombre maximal d'éléments d'un $\mathit{unif}^-(x)$ et n le nombre d'éléments de la classe.

La relation unif^- étant symétrique, mais ni réflexive ni transitive, on ne peut guère optimiser l'espace consommé par les ensembles $\mathit{unif}^-(x)$. De façon évidente, on notera $\mathit{unif}^-(x) \cap X$ par $\mathit{unif}^-_{\mathrm{var}}(x)$ et $\mathit{unif}^-(x) \cap \Sigma_{\mathrm{cst}}$ par $\mathit{unif}^-_{\mathrm{cst}}(x)$.

48) : *"[The plan representation and the constraint maintenance] are long and messy, but completely straightforward"*. Les principes sous-jacent à l'unification sous contraintes ainsi que les algorithmes (optimisés) présentés dans le présent chapitre **ne font aucune hypothèse** sur l'ensemble des constantes : il peut être fini ou non, une constante peut être un symbole ou un objet d'un système de représentation évolué, ...

Une deuxième convention sur les ensembles *unif* permet de faire chuter d'un degré la complexité du calcul de l'unification possible dans le cas d'une variable et d'une constante. Lorsqu'une variable x de X est contrainte à s'unifier à une constante a de Σ_{cst}, toutes les variables x' de la classe d'équivalence de x sont, d'après la définition de 5.2.1, contraintes à s'unifier aussi avec a, mais de façon *indirecte* : pour savoir si $\Box(x' \approx a)$, il faut d'abord trouver $\Box(x' \approx x)$, ce qui nécessite de parcourir a priori toute la classe (complexité en $O(n)$), puis $\Box(x \approx a)$, ce qui est immédiat (complexité $O(1)$).

Une façon d'éviter cette complexité linéaire consiste à *aplatir* (ou *compiler*) la classe de x sur a (presque une fermeture transitive). Lorsqu'une variable x est contrainte à s'unifier avec une constante, on convient que toutes les variables x' de la classe de x sont aussi contraintes à s'unifier *directement* avec cette constante (voir figure 5.5), ce qui s'exprime par :

$$\forall x \in X,\ \Box(x \approx a) \Rightarrow \forall x'/\Box(x' \approx x),\ \overline{unif^+}(x') = a$$

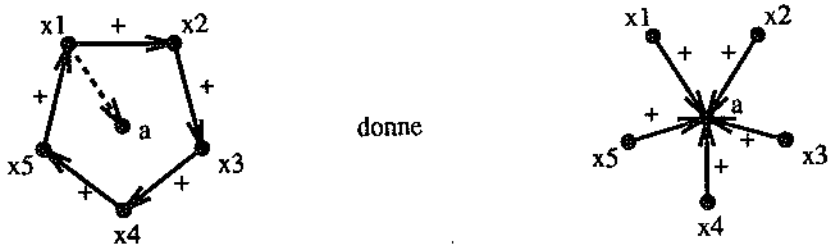


Figure 5.5: Aplatissement d'une classe d'équivalence sur une constante

Remarquons que la complexité de calcul de $\Box(x \approx y)$, dans le cas où les deux variables appartiennent à une classe aplatie C, est passée dans la foulée de $O(n)$, où $n = \text{Card}(C)$, à $O(1)$.

Le gain en complexité qu'apporte cet aplatissement se paie par la perte de circularité explicite de la classe d'équivalence : on ne peut plus lister exhaustivement les variables d'une classe aplatie.

En troisième et dernière convention, on suppose que la relation "est-contraint-à-ne-pas-s'unifier-avec", lorsqu'elle est réduite à $X \times X$, est symétrique : $\forall x, y \in X,\ x \in unif^-(y) \Rightarrow y \in unif^-(x)$.

Avec les conventions précédentes sur les classe d'équivalences (une classe d'équivalence est soit circulaire, soit aplatie), un terme atomique t est soit une constante ($t \in \Sigma_{cst}$), soit une variable ($t \notin \Sigma_{cst}$, donc $t \in X$) d'une classe aplatie ($\overline{unif^+}(t) \in \Sigma_{cst}$), soit une variable ($t \notin \Sigma_{cst}$) d'une classe circulaire ($\overline{unif^+}(t) \notin \Sigma_{cst}$). Les algorithmes suivants devront distinguer a priori $3 \times 3 = 9$ cas, selon la table 5.1, mais se ramènent immédiatement à 6 par symétrie.

Algorithme d'unification nécessaire Avec les conventions précédentes, le calcul booléen de l'unification *nécessaire* de deux termes atomiques $\Box(t \approx t')$ est donné par l'algorithme évident NÉCESSAIRE-UNIF suivant :

	t' constante	t' classe aplatie	t' classe circulaire
t constante	cas 1		
t classe aplatie	cas 2	cas 4	
t classe circulaire	cas 3	cas 5	cas 6

Table 5.1: Numérotation des neuf cas de l'unification

Procédure NÉCESSAIRE-UNIF(t, t')
si $t \in \Sigma_{cst}$ **alors si** $t' \in \Sigma_{cst}$ **alors** SORTIE$(t = t')$; *CAS 1*
sinon NÉCESSAIRE-UNIF(t', t) ; *voir les cas 2 ou 3 (symétrie)*
sinon si $t' \in \Sigma_{cst}$ **alors** NÉCESSAIRE-UNIF$(\overline{unif^+}(t), t')$; *CAS 2 & 3*
sinon si $\overline{unif^+}(t) \in \Sigma_{cst}$, **alors** NÉCESSAIRE-UNIF$(\overline{unif^+}(t), t')$; *CAS 4*
sinon si $\overline{unif^+}(t') \in \Sigma_{cst}$, **alors** NÉCESSAIRE-UNIF$(t, \overline{unif^+}(t'))$; *CAS 5*
sinon si $\exists n$, $(unif^+)^n(t) = t'$, **alors** SORTIE(**vrai**), **sinon** SORTIE(**faux**) ; *CAS 6*

La procédure NÉCESSAIRE-UNIF est simplement basée sur le fait que (cas 1 de la table 5.1) deux constantes s'unifient nécessairement ssi elles sont égales (cf. 5.2.1) : les cas 2, 3, 4 et 5 se ramènent à ce cas 1, moyennant parfois un ou deux appels récursif avec la valeur $\overline{unif^+}$ d'un terme au lieu du terme initial ...Le seul cas non trivial est celui où les deux termes appartiennent à une classe circulaire ; il faut alors vérifier qu'il s'agit de la même classe, i-e que l'on peut atteindre une variable à partir de l'autre par application récursive de $\overline{unif^+}$.

A cause de ce dernier cas, la complexité de parcours de NÉCESSAIRE-UNIF mesurée par le nombre de tests d'égalité, passe de $O(1)$ à $O(n^+)$, où n^+ est le nombre maximal de variables d'une classe circulaire.

Algorithme d'unification possible Le calcul booléen de l'unification *possible* de deux termes atomiques $\Diamond(t \approx t')$ est donné par l'algorithme POSSIBLE-UNIF suivant :

Procédure POSSIBLE-UNIF(t, t')
  si $t \in \Sigma_{cst}$ **alors si** $t' \in \Sigma_{cst}$ **alors** SORTIE$(t = t')$; *CAS 1 (constante × constante)*
    sinon POSSIBLE-UNIF(t', t) ; *voir les cas 2 ou 3 (symétrie)*
  sinon si $t' \in \Sigma_{cst}$ **alors**
    si $\overline{unif^+}(t) \in \Sigma_{cst}$ **alors** ; *CAS 2 (classe aplatie × constante)*
    si $t' \in unif^-_{cst}(t)$ **alors** SORTIE(**faux**)
      POSSIBLE-UNIF$(t', \overline{unif^+}(t))$; *voir le cas 1*
    sinon ; *CAS 3 (classe circulaire × constante)*
      si $\exists n,\ t' \in unif^-(\overline{unif^+}^n(t))$ **alors** SORTIE(**faux**)
      pour tout t^- **de** $unif^-_{var}(t)$, **si** $\overline{unif^+}(t^-) = t'$ **alors** SORTIE(**faux**)
      SORTIE(**vrai**)
  sinon si $t \in \overline{unif^-_{var}}(t') \vee t' \in unif^-_{var}(t)$ **alors** SORTIE(**faux**)
    si $\overline{unif^+}(t') \in \Sigma_{cst}$ **alors**
      si $\overline{unif^+}(t) \in \Sigma_{cst}$ **alors** ; *CAS 4 (classe aplatie × classe aplatie)*
        si $unif^+(t) \in \overline{unif^-_{cst}}(t')$ **alors** SORTIE(**faux**)
        POSSIBLE-UNIF$(t, \overline{unif^+}(t'))$; *voir le cas 2*
      sinon ; *CAS 5 (classe circulaire × classe aplatie)*
          si $\exists n,\ unif^+(t') \in unif^-((\overline{unif^+})^n(t))$, **alors** SORTIE(**faux**)
          SORTIE(**vrai**)
    sinon si $\overline{unif^+}(t) \in \Sigma_{cst}$ **alors** POSSIBLE-UNIF(t', t) ; *voir cas 5 (symétrie)*
      sinon ; *CAS 6 (classe circulaire × classe circulaire)*
          ; *Inutile de tester l'inverse (symétrie)*
          si $\exists n,\ unif^+(t') \in unif^-((\overline{unif^+})^n(t))$, **alors** SORTIE(**faux**)
          SORTIE(**vrai**)

La procédure POSSIBLE-UNIF réduit les neuf cas de la table 5.1 à 6 par symétrie et même à 4 : le cas *classe aplatie* × *classe aplatie* (CAS 4) se ramène au cas *classe aplatie* × *constante* (CAS 2), moyennant un test ; ce dernier cas se ramène à son tour au cas *constante* × *constante* (CAS 1), moyennant un deuxième test.

En appelant n^+ le nombre maximal de variables d'une classe d'équivalence ($n^+ = \max_{x \in X}\{\text{Card}(unif^+(x))\}$), n^- le nombre maximal de termes d'un ensemble $unif^-$ ($n^- = \max_{x \in X}\{\text{Card}(unif^-(x))\}$), le nombre de test d'égalité[12] entre deux termes atomiques est borné supérieurement par : 1 pour le cas 1, $n^- + 1$ pour le cas 2, $n^- + n^+$ pour le cas 3, $4n^- + 1$ pour le cas 4, $2n^- + n^-n^+$ pour le cas 5, $2n^- + n^-n^+$ pour le cas 6. Le nombre de tests d'égalité de POSSIBLE-UNIF est borné supérieurement par n^-n^+ (cas 5 ou 6).

Algorithme de pose de contrainte L'algorithme FORCE-UNIF forçant deux termes atomiques t et t' à s'unifier lorsque c'est possible (i-e $\Diamond(t \approx t')$) s'obtient à partir de POSSIBLE-UNIF en remplaçant "SORTIE(**faux**)" par "ECHEC" et "SORTIE(**vrai**)" par les actions indiquées ci-dessous (pour chaque cas) :

[12] Les tests du type $u \in \Sigma_{cst}$ ou $u \in X$ se basent sur l'observation du nom du terme u et ne consomment pas de test d'égalité.

Procédure FORCE-UNIF(t, t')
  Cas 1 :
    ; implicitement déjà fait ou impossible
  Cas 2 :
    ; implicitement déjà fait ou impossible
  Cas 3 :
    APLATIR(t, t')
  Cas 4 :
    ; implicitement déjà fait ou impossible
  Cas 5 :
    APLATIR$(t, \overline{unif^+}(t'))$
    $unif^-_{\text{var}}(t') \leftarrow unif^-_{\text{var}}(t) \cup unif^-_{\text{var}}(t')$
  Cas 6 :
    ; mise en commun des contraintes négatives
    $unif^-(t) \leftarrow unif^-(t) \cup unif^-(t')$
    $unif^-(t') \leftarrow unif^-(t)$
    ; fusion en "8" des deux cycles
    $tmp \leftarrow \overline{unif^+}(t)$
    $\overline{unif^+}(t) \leftarrow \overline{unif^+}(t')$
    $\overline{unif^+}(t') \leftarrow tmp$

La procédure FORCE-UNIF a besoin d'un algorithme d'aplatissement de la classe d'équivalence d'une variable v sur une constante c :

Procédure APLATIR(v, c)
  ; calcul des contraintes de non-unification
  soit $vars^- \leftarrow \bigcup_{i \in [1, \text{Card}(unif^+(v))]} unif^-_{\text{var}}((\overline{unif^+})^i(v))$
  soit $u \leftarrow v$
  marque u
  faire si u est marqué **alors** SORTIE(boucle)
    next $\leftarrow \overline{unif^+}(u)$
    $\overline{unif^+}(u) = c$
    $unif^-_{\text{cst}}(u) \leftarrow \emptyset$ *; contraintes devenues inutiles*
    $unif^-_{\text{var}}(u) \leftarrow var^-$ *; mise en commun*
    $u \leftarrow unif^+(u)$
  demarque v

Avec les notations précédentes, le nombre de tests d'égalité utilisés par la procédure APLATIR est borné supérieurement par $n^- n^+ + n^+$ (en supposant le calcul de l'union var^- par marquage), ce qui donne pour l'algorithme FORCE-UNIF des nouvelles bornes supérieurs de : $n^- n^+ + n^- + 2n^+$ pour le cas 3, $2n^- n^+ + 4n^-$ pour le cas 5, et $n^- n^+ + 4n^-$ pour le cas 6. La complexité de

l'algorithme FORCE-UNIF, mesurée par le nombre de tests d'égalités, est finalement de $O(2n^2)$ (principalement à cause de la procédure APLATIR).

Signalons que les cas 1 à 5 de FORCE-UNIF ne sont par définition appelés qu'une seule fois par classe d'équivalence. Le cas 6 est appelé à chaque ajout de variable dans une classe, ce qui donne une complexité de $n^- \frac{n^+(n^+ +1)}{2} + 4n^-$ pour la construction d'une classe complète et la complexité au pire du cas 5 ci-dessus pour son aplatissement. Le nombre total de test d'égalité, couvrant la construction complète la plus complexe puis l'aplatissement le plus complexe d'une classe, est finalement borné par $n^-(\frac{n^+(n^+ + 1)}{2} + 2n^+) + 8n^-$, c'est-à-dire est de l'ordre de $O(n^3)$.

Exemples Les algorithmes POSSIBLE- et NÉCESSAIRE-UNIF précédents ont été testés sur tous les cas d'implémentation (64 en tout). Nous en enchaînons trois en exemple. Pour cela, nous utiliserons les fonctions `lelisp` suivantes : "==" implémente l'algorithme FORCE-UNIF et affiche la base de contraintes (lignes qui commencent par ">"), "n==?", NÉCESSAIRE-UNIF, et "p==?", POSSIBLE-UNIF. Ces fonctions ont un troisième argument (optionnel) qui indique s'il s'agit d'une unification ($\approx$) ou d'une non-unification ($\not\approx$).

Commençons par créer une classe simple :

```
? (== ?x ?y)     ; □(x ≈ y) est posé.
|-> t            ; C'est compatible. La base courante vaut:
> ?y -> ?x       ; unif+(y) = x
> ?x -> ?y       ; unif+(x) = y
= t

? (== ?y ?z)     ; □(y ≈ z) est posé ...
|-> t
> ?z -> ?x       ; ... et la variable z est ...
> ?y -> ?z       ; ... intégrée dans ...
> ?x -> ?y       ; ... la classe.
= t
```

```
? (== ?z ?t)     ; Idem pour t
|-> t
> ?t -> ?x
> ?z -> ?t
> ?y -> ?z
> ?x -> ?y
= t

? (=/= ?x a ())  ; □(x ≉ a) est posé.
|-> t            ; Cet ajout est compatible avec la base.
> ?t -> ?x       ; Chacune des variables de la classe ...
   <> a          ; ... en hérite.
> ?z -> ?t
   <> a
> ?y -> ?z
   <> a
> ?x -> ?y
   <> a
= t
```

On rajoute une deuxième classe disjointe de la première :

```
? (== ?y ?u ())  ; □(y ≈ u) est posé.
|-> t            ; Ajout compatible.
> ?u             ; La variable u est ajoutée.
   <> ?y
> ?t -> ?x       ; Elle est dans chaque unif⁻.
   <> ?u
   <> a
> ?z -> ?t
   <> ?u
   <> a
> ?y -> ?z
   <> ?u
   <> a
> ?x -> ?y
   <> ?u
   <> a
= t
```

```
? (== ?u ?v)     ; □(u ≈ v)
|-> t
> ?v -> ?u       ; Ajout de la variable v ...
   <> ?y         ; ... qui hérite de sa classe.
> ?u -> ?v
   <> ?y
> ?t -> ?x       ; Le reste est inchangé.
   <> ?u
   <> a
> ?z -> ?t
   <> ?u
   <> a
> ?y -> ?z
   <> ?u
   <> a
> ?x -> ?y
   <> ?u
   <> a
= t
```

Posons quelques questions à cette base :

```
? (n==? ?x ?z)   ; □(x ≈ z) ?
= t              ; car □(x ≈ y) et □(y ≈ z)
? (n==? ?z a)    ; □(z ≈ a) ?
= ()             ; car □(z ≈ x) et □(x ≉ a)
? (p==? ?x ?z)   ; ◇(x ≈ z) ?
= t              ; car □(x ≈ z).
? (p==? ?x ?z ()) ; ◇(x ≉ z) ?
= ()             ; car □(x ≈ z) ci-dessus.
? (p==? ?x ?v)   ; ◇(x ≈ v) ?
= ()             ; car □(x ≈ y), □(y ≉ u)
                 ; et □(u ≈ v)
```

```
? (p==? ?u a)    ; ◇(u ≈ a) ?
= t              ; car □(u ≉ y) et □(y ≉ a)
? (p==? ?x ?a)   ; Tout est possible pour ...
= t
? (p==? ?x ?a ()) ; ... une nouvelle variable ...
= t
? (p==? ?a b)    ; ... y compris avec ...
= t
? (p==? ?a b ())  ; ... des constantes !
= t
```

Modifions cette base et reposons quelques questions :

```
? (p==? ?u b)    ; ◇(u ≈ b) ?
= t              ; Aucune contrainte ...

? (== ?u b)      ; Allons-y :
|-> t
> ?v -> b       ; La classe de u est écrasée
   <> ?y
> ?u -> b       ; sur la constante b.
   <> ?y
> ?t -> ?x
   <> ?u
   <> a
> ?z -> ?t
   <> ?u
   <> a
> ?y -> ?z
   <> ?u
   <> a
> ?x -> ?y
   <> ?u
   <> a
= t
```

```
? (p==? ?u a)    ; ◇(u ≈ a) encore ?
= ()             ; car □(u ≈ b) !
? (p==? ?x b)    ; ◇(x ≈ b) ?
= ()             ; car □(x ≉ u) et □(u ≈ b)
```

5.3 Critères de vérité

Un *critère de vérité* est l'*énoncé d'une condition* que doit remplir un terme pour avoir une certaine valeur dans une certaine situation (cf. chapitre 4). Cette condition, toujours implicite dans un parcoureur de graphe, est ici vitale pour déterminer ce qui existe et ce qui n'existe pas, (les caractéristiques des objets de 5.1.1), à un "instant" donné dans le graphe d'actions. Sans critère de ce type, un planificateur serait incapable de déduire quoi que ce soit d'un plan d'actions, vu comme une base de contraintes de précédence et d'unification !

Le critère que nous utilisons repose sur l'enchaînement des hypothèses suivantes :

Hypothèse 1 *Le seul calcul possible (calcul au sens de 1.1) est celui de 5.1.2.*

La seule façon de modifier quelque chose dans le monde modélisé passe par le biais d'une conclusion d'action dans le plan ; un terme non mentionné en conclusion d'une action ne change pas de valeur par cette action. C'est une variante de la notion de *persistance* de STRIPS (cf. 3.1.2). Une autre formulation équivalente : aucun agent extérieur au monde modélisé ne peut intervenir (*hypothèse* STRIPS).

Hypothèse 2 *Un terme qui n'a jamais été mentionné en conclusion d'une action possède la valeur "inconnu".*

La valeur par défaut d'un terme est non pas *faux* mais *inconnu* (l'*hypothèse du monde clos* n'est pas respectée). Pour être déclaré faux (resp. vrai), un terme doit avoir été mentionné avec (resp. sans) "$\neg$" en conclusion d'une action. Pour rendre homogène les objets manipulés, la situation initiale est alors spécifiée en conclusion d'une action, également déclarée *initiale*, dont les prémisses (vides) sont toujours vraies.

Cette deuxième hypothèse nous situerait dans le cadre des logiques à 3 valeurs (vrai, faux ou inconnu) [Kauffmann 68]. Mais comme nous ne manipulerons que des termes atomiques (i-e pas de formules *inconnu* $\wedge$ *faux*) et mentionnés en partie conclusion d'une action, nous n'aurons besoin en pratique que des valeurs vrai et faux. De plus, l'introduction des modalités $\Box$ et $\Diamond$ rendra en pratique l'énoncé *"p est inconnu"* synonyme de *"il est possible que p soit vrai et il est possible que p soit faux"*.

Nous présentons les critères en ordre 0 et 1 que nous utilisons, et qui modélisent le comportement d'un agent unique personnifiant le planificateur. Pour chaque critère, nous considérerons l'algorithme direct (*naïf*, directement issu du critère), un algorithme optimisé (avec une gestion de marques) et, à titre indicatif[13], un algorithme très optimisé (incluant une structure de données supplémentaires, que l'on peut interpréter grossièrement par les liens de dépendance causale conclusion/prémisse).

[13]Cet algorithme dit "très optimisé" *n'a pas été implémenté*, car (1) il met en jeu une structure de donnée dont la gestion est relativement lourde et surtout (2) *cette structure de données fige le choix du critère à utiliser*. Nous avons préféré échanger un système très efficace, mais lourd et, du coup, fermé, contre un système un peu moins efficace, mais souple et ouvert.

5.3.1 Critère d'ordre 0

Algorithme direct Nous présentons l'alter ego en logique des propositions du critère que nous emploierons en ordre 1 (en 5.3.2), obtenu en remplaçant l'égalité au sens de l'unification possible ou nécessaire par l'égalité au sens de l'algèbre des termes (c'est en fait l'algorithme de 4.2 sans le "OU"). Les modalités de possibilité et de nécessité n'ont plus que leur signification vis-à-vis de l'extension d'un ordre partiel en un ordre total.

Une proposition p est nécessairement vraie dans la situation $post(a)$ (i-e $\Box_a p$) ssi :

$$\exists a_0 \in \mathcal{A}/\ \Box(a_0 \preceq a) \wedge p \in \gamma(a_0),$$
$$\wedge$$
$$\forall a_1 \in \mathcal{A}/\ \Diamond(a_1 \preceq a) \wedge (\neg p) \in \gamma(a_1),$$
$$\exists a'_1 \in \mathcal{A}, \Box(a_1 \prec a'_1 \prec a) \wedge p \in \gamma(a'_1)$$

Pour que la proposition p soit vraie juste après l'action a, il faut d'abord qu'il y ait une action avant a qui conclue sur p ; il faut ensuite que la dernière action concernant p pouvant être avant a conclue absolument sur p et non $\neg p$.

En appelant n le nombre d'actions dans le plan, c^+ (resp. c^-) le nombre maximal de conclusions positives (resp. négatives) d'un schéma d'actions, c le max de c^+ et c^- :

- pour le premier terme (établissement), le nombre de tests d'égalité entre propositions est borné supérieurement par c^+n ;
- pour le second terme (non-destruction intermédiaire), le nombre de tests d'égalité entre propositions est borné supérieurement par $c^+c^-n^2$.

Pour déterminer une valeur (vrai ou faux) d'une proposition, le nombre total de tests d'égalité entre propositions est borné par $c^+c^-n^2 + c^+n$; une implémentation directe du critère en ordre 0 conduit donc a une complexité de parcours de $O(c^2n^2)$, ce qui est compatible avec le résultat de Dean (complexité en $O(n^{c+1})$ pour un algorithme avec le "OU", cf. 4.2.2). Enfin, pour savoir si une proposition a une valeur inconnue, la complexité est double (il faut démontrer que cette proposition n'est ni vraie, ni fausse).

Première optimisation (simple) Des techniques simples de marquages permettent de réduire cette complexité. Elles se basent sur le fait que ce qui influe sur la valeur d'une proposition juste avant une action donnée ne dépend que des *dernières* conclusions (dernières avant l'action et dernières en parallèle de l'action) relatives à cette proposition ou à sa négation : le deuxième terme du critère (ci-dessus) indique précisément que ce n'est pas a_1 mais a'_1, avec $\Box(a_1 \prec a'_1 \prec a)$ qui donne sa valeur à la proposition p en a. Une fois une *dernière* action trouvée, la recherche sur la branche courante peut s'arrêter.

En conservant la structure du graphe et des actions, peu de parcours suffisent à déterminer la valeur (vraie, fausse ou inconnue) d'une proposition p (supposée positive) après l'action a, selon l'algorithme en pseudo-français suivant :

Procédure VALEUR0(p, a)
- *avant-vrai* $\leftarrow$ *faux*, *avant-faux* $\leftarrow$ *faux*, *parall-vrai* $\leftarrow$ *faux*, *parall-faux* $\leftarrow$ *faux*

; *parcours décroissant*
- pour toute action $a_\prec$ nécessairement avant a,
  marquer $a_\prec$ avec $m_\prec$
  si *avant-vrai* = *faux* et $p \in \gamma(a_\prec)$, alors *avant-vrai* $\leftarrow$ *vrai*
  si *avant-faux* = *faux* et $(\neg p) \in \gamma(a_\prec)$, alors *avant-faux* $\leftarrow$ *vrai*

; *parcours croissant*
- pour toute action $a_\succ$ nécessairement après a, marquer $a_\succ$ avec $m_\succ$

; *parcours décroissant depuis le puits*
- pour toute action $a_{//}$, si $a_{//}$ n'a pas la marque $m_\succ$, alors
  si $a_{//}$ a la marque $m_\prec$, alors arrêter le parcours sur cette branche
  sinon si *parall-vrai* = *faux* et $p \in \gamma(a_{//})$, alors
    parall-vrai $\leftarrow$ *vrai*
    si *parall-faux* = *vrai*, alors arrêter totalement ce parcours
      sinon arrêter le parcours sur cette branche uniquement
  si *parall-faux* = *faux* et $(\neg p) \in \gamma(a_{//})$, alors
    parall-faux $\leftarrow$ *vrai*
    si *parall-vrai* = *vrai*, alors arrêter totalement ce parcours
      sinon arrêter le parcours sur cette branche uniquement

; *parcours croissant depuis l'origine*
- pour toute action a', enlever les marques éventuelles $m_\prec$ et $m_\succ$ de a'

; *renvoi du résultat*
- p est possiblement / nécessairement vrai / faux juste avant a selon les formules suivantes :
  $\Box_a p \Leftrightarrow$ *avant-vrai* $\wedge \neg$*avant-faux* $\wedge \neg$*parall-faux*
  $\Box_a(\neg p) \Leftrightarrow$ *avant-faux* $\wedge \neg$*avant-vrai* $\wedge \neg$*parall-vrai*
  $\Diamond_a p \Leftrightarrow \neg$*avant-faux* $\vee$ *avant-vrai* $\vee$ *parall-vrai*
  $\Diamond_a(\neg p) \Leftrightarrow \neg$*avant-vrai* $\vee$ *avant-faux* $\vee$ *parall-faux*

Le but de la procédure VALEUR0 est de donner une valeur aux variables *avant-vrai*, *avant-faux*, relatifs aux actions avant a et aux variables *parall-vrai*, *parall-faux*, relatifs aux actions en parallèle de a.

- Le premier parcours traite les actions (nécessairement) avant a (variables *avant-vrai* et *avant-faux*) et recherche, si elles existent, les premières (dans le sens du parcours, "dernières" selon $\prec$) actions qui concluent sur p et $\neg p$. Une fois trouvée une action de chaque type, ce parcours continue quand même jusqu'à la première action uniquement pour marquer les actions nécessairement avant a avec la marque $m_\prec$, marque qui sera utilisée ensuite pour détecter les actions en parallèle de a.

- Le deuxième parcours ne fait que marquer les actions nécessairement après a avec la marque $m_\succ$, ce qui, en plus du premier parcours, fait se révéler les actions en parallèle de a : ce sont celles qui ne sont marquées par aucun des deux (ni $m_\prec$, ni $m_\succ$).

- Le troisième parcours est l'analogue du premier mais pour ces actions en parallèle de a : les premières conclusions à p ou $\neg p$ sont recherchées ; lorsque qu'une conclusion d'un type a été trouvée, le parcours sur la branche courante est arrêté ; lorsqu'une conclusion de chaque type a été trouvé, le parcours peut s'arrêter avant terme.

- Le quatrième et dernier parcours ne sert qu'à réinitialiser proprement les marques.

L'évaluation de la complexité ne pose pas de problème : on la mesure par le nombre de conclusions observées, en convenant qu'une action sur laquelle un parcours "passe" sans en observer les conclusions compte quand même pour une conclusion observée. Soit c le nombre maximal de conclusions d'une action, soit $n_\prec$ le nombre maximal d'actions nécessairement avant a, soit $n_{//}$ le nombre maximal d'actions en parallèle de a, soit $n_\succ$ le nombre maximal d'actions nécessairement après a : pour le premier parcours, la borne supérieure[14] est $cn_\prec$; pour le deuxième parcours, la borne supérieure est $n_\succ$; pour le troisième parcours, la borne supérieure est $cn_{//} + n_\succ$; pour le quatrième et dernier parcours, le nombre d'actions vues est exactement $n_\prec + n_{//} + n_\succ$. Le nombre total de conclusions observées est donc borné par : $(c+1)(n_\prec + n_{//} + n_\succ) + (2-c)n_\succ$. Compte tenu du fait que c est effectivement proche de 2 en pratique et en appelant n le nombre d'actions du graphe, le nombre total de conclusions observées est de l'ordre de $O((c+1)n)$ ou, dit autrement[15], le nombre d'actions considérées est de l'ordre de $O(3n)$.

La complexité a donc chuté d'un degré par rapport à l'implémentation directe.

Deuxième optimisation (moins simple) Il n'est guère possible d'améliorer notablement l'algorithme précédent en conservant les mêmes structures de données[16]. Le choix des structures de données supplémentaires se déduit souvent des calculs les plus fréquents : stocker le résultat de ces calculs (technique dite de *caching*), dont la complexité d'obtention passe à $O(1)$ dès la seconde fois, et ne les recalculer que lorsqu'il y a un changement et que l'on a besoin du résultat. Le prix à payer pour ce faible temps d'accès est (1) l'augmentation de la consommation en espace-mémoire et (2) l'augmentation de la complexité de mise à jour. Le premier point dépend de la machine-hôte, de l'espace-mémoire effectivement consommé et surtout de la stratégie de développement du programmeur. Le deuxième point dépend du rapport entre le nombre de requêtes adressées au système et le nombre de mise à jour : si ce rapport est grand, alors la base change "lentement" par rapport à son utilisation et ces structures de données supplémentaires deviennent rentables.

Pour l'algorithme VALEUR0, les données intéressantes à connaître sont les actions qui concluent sur p et $\neg p$ et leur positions relatives. Une structure de donnée envisageable serait construite de la façon suivante :

[14] Toutes ces bornes supérieures sont atteintes dans les plus mauvais cas.

[15] De menues optimisations, liées à l'anticipation du résultat sont encore possibles.

[16] Selon la loi de Murphy (parfois visité par Heisenberg) bien connue *place*×*temps-calcul* = *cst* : pour un problème donné (une *cst* donnée), on peut trouver deux familles d'algorithmes : ceux qui sont lents mais qui consomment peu de espace-mémoire, et ceux qui sont rapides, mais qui consomment beaucoup d'espace-mémoire ...

1. **faire correspondre à chaque prédicat p ou $\neg p$ une entrée dans la table**
2. **pour chaque entrée, stocker la liste des actions qui concluent positivement et stocker la liste des actions qui concluent négativement**
3. **pour chaque action stocker la liste des actions se trouvant en parallèle (selon une extension totale quelconque de l'ordre partie $\prec$)**

Le premier point consiste à indexer tout nouveau prédicat dans une table (la numérotation de 4.3). Le deuxième point implémente une extraction virtuelle du sous-graphe relatif au prédicat p (seules les actions relatives à p seront considérées). Retrouver les actions avant ou après une action donnée s'effectue à un coût relativement faible (cf. paragraphe précédent), aussi le stockage des actions en parallèle (troisième point) est un compromis efficace [Ghallab & Mounir Alaoui 89].

En reprenant l'algorithme précédent, le gain en complexité de parcours induit par cette structure de données est évident : d'abord, le premier parcours s'effectue directement sur le sous-graphe extrait (et non sur le graphe entier), ainsi que son parcours de réinitialisation (morceau du 4^{eme} parcours concernant la marque $m_\prec$) ; ensuite, le deuxième parcours, déterminant les actions en parallèle de a, devient inutile, ainsi que son parcours de réinitialisation (le morceau du 4^{eme} parcours concernant la marque $m_\succ$) ; de même, le troisième parcours s'effectue directement sur les actions en parallèle stockées (les tests sur les marques $m_\prec$ et $m_\succ$ disparaissent). En appelant $n'_\prec$, $n'_{//}$, $n'_\succ$ le nombre maximal d'actions avant, en parallèle, après a *concernant le prédicat* p, n' le nombre maximal d'actions concernant un terme ($n = n'_\prec + n_{//} + n_\succ + 1$), c' le nombre de prédicats différents (à une négation près), le nombre de tests d'égalités effectués est le suivant : le nouveau premier parcours est borné supérieurement par $c' + n'_\prec$ (au lieu de $cn_\prec$), le terme c' venant de la recherche initiale du bon item dans la table des prédicats ; le nouveau deuxième parcours n'existe plus (borne supérieure à 0 au lieu de $n_\succ$) ; le nouveau troisième parcours est borné supérieurement par $n'_{//}$ (au lieu de $cn_{//} + n_\succ$) ; le nouveau quatrième et dernier parcours est borné supérieurement par $n'_\prec$ (au lieu de $n_\prec + n_{//} + n_\succ$). Une borne supérieure pour VALEUR0 est alors : $2n'_\prec + n'_{//}$. La complexité en parcours de VALEUR0 avec cette structure de données passe à $O(n')$.

Le prix de ce gain est l'augmentation de l'espace-mémoire consommé pour cette structure de données, qui vaut $c'n$ (n est le nombre total d'actions du graphe, au sens du paragraphe précédent), i-e qui duplique, dans le pire des cas, le graphe par le nombre de prédicats différents (à une négation près).

5.3.2 Critère d'ordre 1

Algorithme direct L'algorithme implémentant le critère d'ordre 1 combine celui de l'ordre 0 VALEUR0 avec les algorithmes NÉCESSAIRE-, POSSIBLE-UNIF et FORCE-UNIF gérant globalement les variables (ou plutôt la base de contraintes d'unification, cf. 5.2).

Le critère de vérité d'ordre 1 que nous utilisons est le suivant (cf. 4.3.2) :

Un terme φ est nécessairement vrai dans la situation *post*(a) (i-e $\Box_a\varphi$) ssi :

$$\exists a_0 \in \mathcal{A},\ \Box(a_0 \preceq a) \wedge \exists \chi \in \gamma(a_0),\ \Box(\varphi \approx \chi)$$
$$\wedge$$
$$\forall a_1 \in \mathcal{A}, \Diamond(a_1 \preceq a),$$
$$\forall \psi \in \gamma(a_1), \Diamond(\varphi \approx \neg\psi),$$
$$\exists a_1' \in \mathcal{A}, \Box(a_1 \prec a_1' \prec a),$$
$$\exists \psi' \in \gamma(a_1'),\ \Box(\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi')$$

Les critères correspondant à φ nécessairement faux, possiblement vrai et possiblement faux s'obtiennent par dualité (il suffit de combiner l'échange de $\Diamond$ et $\Box$ avec celui de φ et $\neg\varphi$).

Ce critère peut aussi se formuler en pseudo-français par :

Un terme p est nécessairement vrai dans la situation *post*(a) juste après l'action a si et seulement si les deux points suivants sont vrais :

1. **il existe une action a_0 possiblement située avant a dont une conclusion s'unifie nécessairement avec φ.**
2. **pour toute action a_1 située possiblement avant a, pour toute conclusion ψ de a_1 qui s'unifie possiblement avec $\neg\varphi$, il existe une action a_1' possiblement entre a_1 et a dont une conclusion ψ' s'unifie avec φ chaque fois que ψ s'unifie avec $\neg\varphi$.**

Dit simplement (sans modalité), une proposition est donc vraie si elle a d'abord été déjà déclarée vraie par un *établisseur* et si, ensuite, aucun *masqueur* ne vient entre temps détruire cette valeur, ou alors il faut qu'un *démasqueur* vienne rétablir la valeur initiale du prédicat concerné.

L'algorithme direct de calcul d'une valeur d'un terme juste après une situation est l'exacte programmation de ce critère : chaque symbole $\exists$ se code par un parcours (de la liste des actions du plan courant, de la liste des conclusions de l'action courante, ...) ; chaque symbole $\forall$ se code par une boucle "**pour tout** ..." (sur ces mêmes listes).

En appelant n le nombre d'actions dans le plan, c le nombre maximal de conclusions d'une action :

- pour le premier terme (*établissement*), le nombre de tests d'unification possible ou nécessaire entre termes est borné supérieurement par cn ;
- pour le second terme (*non-destruction intermédiaire*), le nombre de tests d'unification possible ou nécessaire est borné supérieurement par c^2n^2.

Pour déterminer une valeur (vraie ou fausse, possible ou nécessaire), le nombre total de tests d'unification sous contraintes entre termes est borné supérieurement par $c^2n^2 + cn$; d'autre part, nous avons vu en 5.2.3 que, pour un test d'unification sous contraintes, le nombre de tests d'égalité est borné supérieurement par v^+v^-, où v^+ est le nombre maximum de variables d'une classe d'équivalence et v^- le nombre maximum de contraintes négatives sur une variable. L'implémentation directe du critère d'ordre 1 a donc une complexité, mesurée selon le nombre de tests d'égalité, de $O(v^+v^-c^2n^2)$.

Cet algorithme doit être appliqué au pire 4 fois pour déterminer la valeur de toutes les combinaisons possible/nécessaire vrai/faux. Enfin, il doit être appliqué $4cn$ fois, pour déterminer la valeur de *chaque* conclusion du plan.

La *structure des buts* (GOST) de NONLIN peut s'interpréter comme un *cache* des résultats de ce parcours (l'hypothèse de *modification lente* de la base, voir 3.2.1).

Première optimisation Comme en ordre 0, une optimisation simple est obtenue par l'utilisation de techniques de marquage des actions du graphe. L'algorithme VALEUR1 calcule la valeur nécessaire ou possible, vraie ou fausse du terme φ juste après l'action a :

Procédure VALEUR1(φ, a)
; Initialisation des variables d'établissement et de masquage
- $\mathit{etabl\text{-}vrai}_{\prec} \leftarrow \mathit{faux}$, $\mathit{etabl\text{-}faux}_{\prec} \leftarrow \mathit{faux}$,
$\mathit{masq\text{-}vrai}_{//} \leftarrow \mathit{faux}$, $\mathit{masq\text{-}faux}_{//} \leftarrow \mathit{faux}$, $\mathit{masq\text{-}vrai}_{\prec} \leftarrow \mathit{faux}$, $\mathit{masq\text{-}faux}_{\prec} \leftarrow \mathit{faux}$
; recherche d'un établisseur (parcours décroissant)
- pour toute action $a_{\prec}$ nécessairement avant a,
marquer $a_{\prec}$ avec $m_{\prec}$
si $\mathit{etabl\text{-}vrai}_{\prec} = \mathit{faux}$ et $\exists \chi \in \gamma(a_{\prec})$, $\Box(\varphi \approx \chi)$, alors
$\mathit{etabl\text{-}vrai}_{\prec} \leftarrow \mathit{vrai}$
si $\exists \psi \in a_{\prec}$, $\Diamond(\psi \approx \varphi)$ et DÉMASQUÉ?$(\neg\varphi, a, \chi, a_{\prec}) = \mathit{faux}$, alors
$\mathit{masq\text{-}faux}_{\prec} \leftarrow \mathit{vrai}$
si $\mathit{etabl\text{-}faux}_{\prec} = \mathit{faux}$ et $\exists \chi \in \gamma(a_{\prec})$, $\Box(\varphi \approx \neg\chi)$, alors
$\mathit{etabl\text{-}faux}_{\prec} \leftarrow \mathit{vrai}$
si $\exists \psi \in a_{\prec}$, $\Diamond(\psi \approx \neg\varphi)$ et DÉMASQUÉ?$(\varphi, a, \chi, a_{\prec}) = \mathit{faux}$, alors
$\mathit{masq\text{-}vrai}_{\prec} \leftarrow \mathit{vrai}$
; parcours croissant
- pour toute action $a_{\succ}$ nécessairement après a, marquer $a_{\succ}$ avec $m_{\succ}$
; recherche d'un masqueur en parallèle (parcours décroissant depuis le puits)
- pour toute action $a_{//}$, si $a_{//}$ n'a pas la marque $m_{\succ}$, alors
si $a_{//}$ a la marque $m_{\prec}$, alors arrêter le parcours sur cette branche
sinon si $\mathit{masq\text{-}vrai}_{//} = \mathit{faux}$ et $\exists \chi \in \gamma(a_{//})$, $\Box(\varphi \approx \chi)$, alors
$\mathit{masq\text{-}vrai}_{//} \leftarrow \mathit{vrai}$
si $\mathit{masq\text{-}faux}_{//} = \mathit{vrai}$, alors arrêter totalement ce parcours
sinon arrêter le parcours sur cette branche uniquement
si $\mathit{masq\text{-}faux}_{//} = \mathit{faux}$ et $\exists \chi \in \gamma(a_{//})$, $\Box(\varphi \approx \neg\chi)$, alors
$\mathit{masq\text{-}faux}_{//} \leftarrow \mathit{vrai}$
si $\mathit{masq\text{-}vrai} = \mathit{vrai}$, alors arrêter totalement ce parcours
sinon arrêter le parcours sur cette branche uniquement
; parcours croissant depuis l'origine
- pour toute action a', enlever les marques éventuelles $m_{\prec}$ et $m_{\succ}$ de a'
; renvoi du résultat
- φ est possiblement ou nécessairement, vrai ou faux juste avant a selon les formules suivantes :
$\Box_a\varphi \Leftrightarrow \mathit{etabl\text{-}vrai}_{\prec} \wedge \neg\mathit{masq\text{-}faux}_{\prec} \wedge \neg\mathit{masq\text{-}faux}_{//}$
$\Box_a(\neg\varphi) \Leftrightarrow \mathit{etabl\text{-}faux}_{\prec} \wedge \neg\mathit{masq\text{-}vrai}_{\prec} \wedge \neg\mathit{masq\text{-}vrai}_{//}$
$\Diamond_a\varphi \Leftrightarrow \neg\mathit{etabl\text{-}faux}_{\prec} \vee \mathit{masq\text{-}vrai}_{\prec} \vee \mathit{masq\text{-}vrai}_{//}$
$\Diamond_a(\neg\varphi) \Leftrightarrow \neg\mathit{etabl\text{-}vrai}_{\prec} \vee \mathit{masq\text{-}faux}_{\prec} \vee \mathit{masq\text{-}faux}_{//}$

La procédure VALEUR1 est construite à partir de la procédure VALEUR0. Les seules différences sont (cf. la table 4.1) le passage de tests d'égalités simples "$\varphi \in \gamma(a)$" à des tests sous contraintes d'unification "$\exists \chi \in \gamma(a)$, $\Box(\varphi \approx \chi)$" et le traitement particulier que requiert le démasquage "$\exists \psi' \in a'_1$, $\Box(\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi')$", grâce à un appel à la fonction DÉMASQUÉ? (cf. ci-dessous). Par rapport á la procédure VALEUR0, seul le premier parcours de VALEUR1 a changé, le but reste le même : donner une valeur aux variables $\mathit{etabl\text{-}vrai}_{\prec}$, $\mathit{etabl\text{-}faux}_{\prec}$, relatives à l'existence d'établisseur pour φ ou $\neg\varphi$, aux variables $\mathit{masq\text{-}vrai}_{\prec}$, $\mathit{masq\text{-}faux}_{\prec}$, relatives à l'existence de masqueurs pour φ ou $\neg\varphi$ nécessairement avant a, et aux variables $\mathit{masq\text{-}vrai}_{//}$, $\mathit{masq\text{-}faux}_{//}$, relatives à l'existence de masqueurs pour φ ou $\neg\varphi$ en parallèle de a.

La fonction DÉMASQUÉ? renvoie une valeur booléenne indiquant *ssi* la conclusion ψ de l'action a_1 masque ou non le terme φ juste après l'action a :

```
Procédure DÉMASQUÉ?(φ, a0, ψ, a1)
    pour toute action a'1 nécessairement après a1,
        ; Vérifier que □(a'1 ⪯ a0)
        si a'1 n'a pas la marque m≺, alors arrêter le parcours sur cette branche
        si ∃ψ' ∈ a'1, □(φ ≈ ¬ψ ⇒ φ ≈ ψ'), alors
            ; (ψ, a1), masqueur de (φ, a0), est démasqué par (ψ', a'1)
            SORTIE(vrai)
    SORTIE(faux)
```

Comme les actions nécessairement avant a (ou, en tout cas, nécessairement entre a_1 et a_0) sont supposées avoir été marquées par $m_\prec$, les actions a'_1 vérifiant $\Box(a_1 \preceq a'_1 \preceq a_0)$ sont trouvées par un parcours croissant à partir de a_1 sur les marques $m_\prec$. La fonction renvoie *vrai* dès qu'un démasqueur a été trouvé.

Pour l'évaluation de la complexité de VALEUR1, nous reprenons les notations $n_\prec, n_{//}, n_\succ$ et c qui nous ont servi pour celle de VALEUR0. Le nombre de tests d'unification sous contraintes effectués par DÉMASQUÉ? est borné supérieurement par $cn_\prec$. Une borne supérieure pour le premier parcours de VALEUR1 est donc $(cn_\prec)^2$ (en groupant les tests d'unification). En reprenant les bornes supérieures des autres parcours de VALEUR0, le nombre de tests d'unification de VALEUR1 est borné supérieurement par $c^2n_\prec^2 + n_\prec + (c+1)n_{//} + 2n_\succ$. Le nombre de test d'égalité d'un test d'unification sous contraintes est borné par v^+v^-. La complexité de VALEUR1 mesurée en nombre de tests d'égalité est en $O(v^+v^-c^cn_\prec^2)$.

Le polynome mesurant la complexité est toujours de degrés 2, comme dans l'implémentation directe, mais on a en *une seule passe* les valeurs possibles/nécessaires vraies/fausses de φ (terme multiplicatif moins fort).

Deuxième optimisation Là encore, comme pour VALEUR0, on pourrait abaisser le degré de la complexité de parcours de cet algorithme en augmentant (d'autant ?) celui de la complexité de stockage, i-e en ajoutant des structures de données qui permettent une analyse du plan selon le sens "terme → action" et non "action → terme".

Mais en ordre 1, la table n'indexerait plus les *termes*, mais les *classes de termes* à une négation près. Cela reviendrait à étendre la base de contraintes d'unification aux termes composés et non plus seulement aux variables, et à stocker, pour chaque classe, la liste des actions qui concluent positivement dessus et la liste des actions qui concluent négativement dessus. La *structure des buts* (GOST) de NONLIN joue exactement ce rôle, comme l'a remarqué [Chapman 85]. NONLIN maintient, en plus, pour chaque prémisse de chaque action la liste des *contributeurs* positifs et négatifs (actions nécessairement avant l'action citée, dont une conclusion s'unifie nécessairement ou nécessairement pas avec cette prémisse).

Mais, ayant considéré que l'introduction d'une telle extension de la base diminuerait la

souplesse de notre planificateur (et surtout empêcherait d'adopter un autre critère), nous n'avons pas implémenté cette deuxième optimisation.

5.3.3 Autres critères de vérité

Critère numérique Nous avons vu que les critères ci-dessus ne convenaient pas pour des constantes numériques (l'exemple des actions bancaires, dans le dernier paragraphe de 4.3.2), parce que ce critère est incapable de prendre en compte un *cumul* sur les conclusions *en parallèle* relatives à un terme (pour en faire la somme, dans l'exemple cité).

Il est pourtant simple d'exhiber un critère cumulatif concernant des valeurs numériques (implémentant l'analogue de la *loi de Kirshoff* sur l'intensité des courants électriques). Pour cela, on introduit la conclusion *ajoute*, d'arité 2. Le terme $ajoute(x, 3)$ en conclusion de l'action a s'interprète comme : "l'action a ajoute la valeur 3 à la variable x". Le critère numérique correspondant est le suivant :

La variable x a la valeur v juste après l'action a ssi $v = \sum_{\substack{(a_i, v_i) \in \mathcal{A}_x \\ a_i \preceq a}} v_i$

avec $\mathcal{A}_x = \{a_i \in \mathcal{A} / \exists v_i, ajoute(x, v_i) \in \gamma(a_i)\}$

L'algorithme dérivé ne ferait que repérer les actions nécessairement avant a dont une conclusion est du type $ajoute(x, \ldots)$. On retrouve très exactement l'algorithme PROPAGER-DÉBUT de 2.2.1, avec $d_i = v_i$, $\alpha = 1$, $\beta = \gamma = 0$. Deux suppositions supplémentaires doivent être faites (qui définissent en fait le *calcul*, au sens de 5.1.2, sur lequel se base le critère) :

- l'absence d'une conclusion $ajoute(z, \ldots)$ équivaut à $ajoute(z, 0)$;
- avant la pseudo-action initiale, toute variable est initialisée à 0 (par exemple).

Variantes du critère logique Les critères logiques présentés ne peuvent inclure la notion de *ressource* de SIPE [Wilkins 88], car ils sont trop généraux (exemple de cas non traité à la figure 4.5 de 4.3.2). Ces ressources relèvent d'une particularisation du critère logique, qui précise que :

- une conclusion d'une action ne peut être un établisseur que pour *une seule* prémisse ;
- une conclusion d'une action ne peut être un démasqueur que pour *un seul* masqueur.

L'application "lien de dépendance causale", qui fournit un établisseur à chaque prémisse, doit être injective ; de même pour l'application qui fournit son démasqueur à chaque masqueur.

Cette variante en un "critère à un coup", plus restrictive, nécessite, à chaque passe sur toutes les prémisses du plan, de chercher en plus une *correspondance admissible* entre conclusions et

prémisses d'une part, et entre masqueur et démasqueur d'autre part. Ces deux correspondances ne sont jamais redécouvertes ex nihilo à chaque passe (problème NP-complet), mais établies progressivement au cours de l'évolution du graphe : chaque prémisse stocke son établisseur attitré et ses suppléants potentiels (la GOST de NONLIN peut, encore une fois, s'interpréter sous cet angle) ; chaque masqueur stocke son démasqueur, qui le démasque pour une prémisse donnée, et ses suppléants potentiels.

En ordre 0, ce dernier lien implémente la notion de *ressource* de Wilkins : une ressource est un objet monobloc qui ne peut "appartenir" qu'à une seule action à la fois (ressource exclusive ou *non consommable*, en termes de PERT). Un masqueur s'approprie temporairement la ressource représentée par le terme litigieux, qui doit être rendue par le démasqueur avant que l'action propriétaire du terme litigieux ne s'exécute. Cette interprétation en terme de ressource justifie le besoin d'unicité du démasqueur : un démasqueur ne rend qu'une quantité 1 de la ressource, ce qui ne suffit pas si plusieurs masqueurs préexistaient.

Ce "critère logique à un coup" n'a pas été implémenté dans notre planificateur, car il sous-entend rapidement un *raisonnement numérique sur les ressources*, ce qui nous aurait fait dériver vers des techniques de Recherche Opérationnelle, nous écartant de l'aspect logique (symbolique) de la planification.

Intérêt de la symétrie Le critère numérique diffère des critères logiques à propos de la symétrie. Soit $\prec$ la relation binaire "puis" entre termes (telle qu'elle est fixée par l'ordre partiel $\preceq$ entre les actions auxquelles ces termes appartiennent) : dans le cas des critères logiques, on a $V \prec F \overset{\text{def}}{\Longleftrightarrow} F$ et $F \prec V \overset{\text{def}}{\Longleftrightarrow} V$ (en ordre 0), i-e $\prec$ est *non symétrique* en logique, alors que pour les critères numériques, on a $(x+a) \prec (x+b) \overset{\text{def}}{\Longleftrightarrow} (x+(a+b))$ et $(x+b) \prec (x+a) \overset{\text{def}}{\Longleftrightarrow} (x+(b+a)) \Leftrightarrow (x+(a+b))$, i-e $\prec$ est *symétrique*.

Pour un critère de vérité où $\prec$ est symétrique, $\prec$ n'est pas un ordre et un algorithme de propagation, calqué sur PROPAGER-DÉBUT de 2.2.1 pour des valeurs numériques ou sur PROPAGE-SYMBOLIQUE de 2.6.1, fonctionnera, car le fait que des termes soient en parallèle ou en série n'a aucune influence sur le résultat ; le choix de parcourir une branche plutôt qu'une autre est laissé à cet algorithme de propagation pour son optimisation personnelle. Dans le cas où la relation $\prec$ entre termes est symétrique, le critère de vérité s'exprime très simplement.

Pour un critère de vérité où $\prec$ n'est plus symétrique, l'ordre dans lequel sont considérés deux termes en parallèle est générateur potentiel de conflits : p puis $\neg p$ est différent de $\neg p$ puis p. Il faut établir une *table de vérité*.

$\prec$	V	F	I
V	V	F	V
F	V	F	F
I	V	F	I

Table 5.2: Table de vérité de $\prec$ en ordre 0

La table de vérité 5.2 donne les valeurs de vérité d'une expression $p \prec q$ où p et q peuvent avoir les valeurs Vrai, Faux ou Inconnu. Cette table définit le *calcul* (au sens de 5.1.2) pour

un ordre total entre actions en logique des propositions. Le présent chapitre a été entièrement consacré aux façons d'étendre cette table de vérité pour un ordre partiel entre actions et en logique des prédicats du premier ordre.

Chapitre 6

Contrôle et heuristiques

Notre planificateur est maintenant capable de déduire d'un plan donné des informations concernant l'unification positive/négative et possible/nécessaire des variables, la précédence/succession possible/nécessaire des actions et surtout la valeur vraie/fausse/inconnue possible/nécessaire d'un terme avant/après une action. A ce stade, le plan est considéré comme une *base de données temporelle* (une version en ordre 1 d'un TMM [Dean 85]), puisque des requêtes peuvent lui être adressées (le module *"Question Answering"* de [Tate 77]), le critère utilisé étant une condition suffisante. En particulier, la modification de la base est de la responsabilité de l'utilisateur.

Pour pouvoir résoudre un *problème de planification* (au sens de 1.2.1), le planificateur doit être en plus doté de capacités inductives (la deuxième partie du cycle déduction/induction), lui permettant de faire évoluer le graphe. Nous présentons en 6.1 la façon dont le planificateur peut donner une valeur vraie/fausse possible/nécessaire à un terme avant/après une action. Le choix instantané d'un terme du plan et de sa valeur, ainsi que la façon de la lui donner, n'étant pas unique, nous présentons en 6.2 le contrôle imposé par le critère (celui-ci étant également une condition nécessaire) et les heuristiques complémentaires utilisées.

6.1 Induction sur le critère de vérité

Parmi toutes les modifications que peut subir le graphe, l'identification des leviers valides, influant effectivement sur la valeur d'un terme est (encore) basée sur le critère de vérité d'ordre 1 de 5.3.2 : ce critère étant non seulement suffisant mais nécessaire (cf. 4.3.2), il suffit d'en inverser la lecture pour obtenir *tous* les moyens élémentaires de donner une valeur à un terme.

Les exemples que nous présentons nécessitent parfois l'enchaînement de plusieurs de ces moyens élémentaires, ce qui peut se faire sans entrer dans les détails du contrôle (voir 6.2), puisque, à un instant donné, *un seul* terme aura une valeur réelle (i-e obtenue par les algorithmes du chapitre 5) différente de sa valeur attendue (i-e celle qu'il a en partie *prémisse* π d'une action).

Le libellé du critère d'ordre 1 est le suivant (cf. 5.3.2) :

Un terme φ est nécessairement vrai dans la situation *post*(a) (i-e $\Box_a\varphi$) ssi :

$$\exists a_0 \in \mathcal{A},\ \Box(a_0 \preceq a) \wedge \exists \chi \in \gamma(a_0),\ \Box(\varphi \approx \chi)$$
$$\wedge$$
$$\forall a_1 \in \mathcal{A}, \Diamond(a_1 \preceq a),$$
$$\forall \psi \in \gamma(a_1), \Diamond(\varphi \approx \neg\psi),$$
$$\exists a_1' \in \mathcal{A}, \Box(a_1 \prec a_1' \prec a),$$
$$\exists \psi' \in \gamma(a_1'),\ \Box(\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi')$$

L'action a est appelée action *demandeuse*, l'action a_1, action *masqueuse*, l'action a_1', action *démasqueuse* ; le terme φ est appelé terme *demandeur*, le terme ψ, terme *masqueur*, le terme ψ', terme *démasqueur* ; lorsqu'il n'y aura pas d'ambiguïté, on appellera *demandeur*, *masqueur* ou *démasqueur* le terme ou l'action correspondante.

Une lecture procédurale de ce critère identifie chacun des composants (voir figure 6.1).

La signification de chacun des termes isolés est considérée : nous étudions successivement les inductions élémentaires (ajout de contraintes d'unification ou de précédence) spécifiées par le premier terme (condition d'ajout) en 6.1.1 et celles spécifiées par le second terme en 6.1.2.

6.1.1 Etablissement d'un terme

Pour qu'un terme φ soit vrai juste après l'action a, il faut qu'il existe une action a_0 (éventuellement égale à a) nécessairement inférieure à a, dont une des conclusions χ s'unifie nécessairement à γ ($\exists a_0 \in \mathcal{A}, \Box(a_0 \preceq a) \wedge \exists \chi \in \gamma(a_0), \Box(\varphi \approx \chi)$, d'après la première condition du critère rappelé ci-dessus). Nous détaillons la signification pratique de chacun des trois termes composant cette condition : $\exists a_0 \in \mathcal{A}$ (existence d'un établisseur), $\Box(a_0 \preceq a)$ (précédence de l'établisseur) et $\exists \chi \in \gamma(a_0)$, $\Box(\varphi \approx \chi)$ (adéquation de la conclusion établissante).

Existence d'un établisseur ($\exists a_0 \in \mathcal{A}$) Soit cet établisseur existe déjà *potentiellement* dans le graphe, soit il n'existe pas (s'il existe déjà *nécessairement*, le problème de l'existence est bien sûr déjà résolu).

S'il n'existe aucun établisseur potentiel, le graphe est alors incapable par lui-même de générer un établisseur par ajout de contraintes sur les actions ou variables existantes. Le seul moyen de l'aider est d'incorporer une nouvelle action issue d'un schéma d'actions, ce qui s'effectue en deux étapes :

1. Construire d'abord un jeu de schémas d'actions *candidats*, dont chacun possède une conclusion qui peut s'unifier au terme φ. Dans un moteur d'inférences, il s'agirait typiquement d'une phase de *chaînage arrière* (détermination des règles candidates).

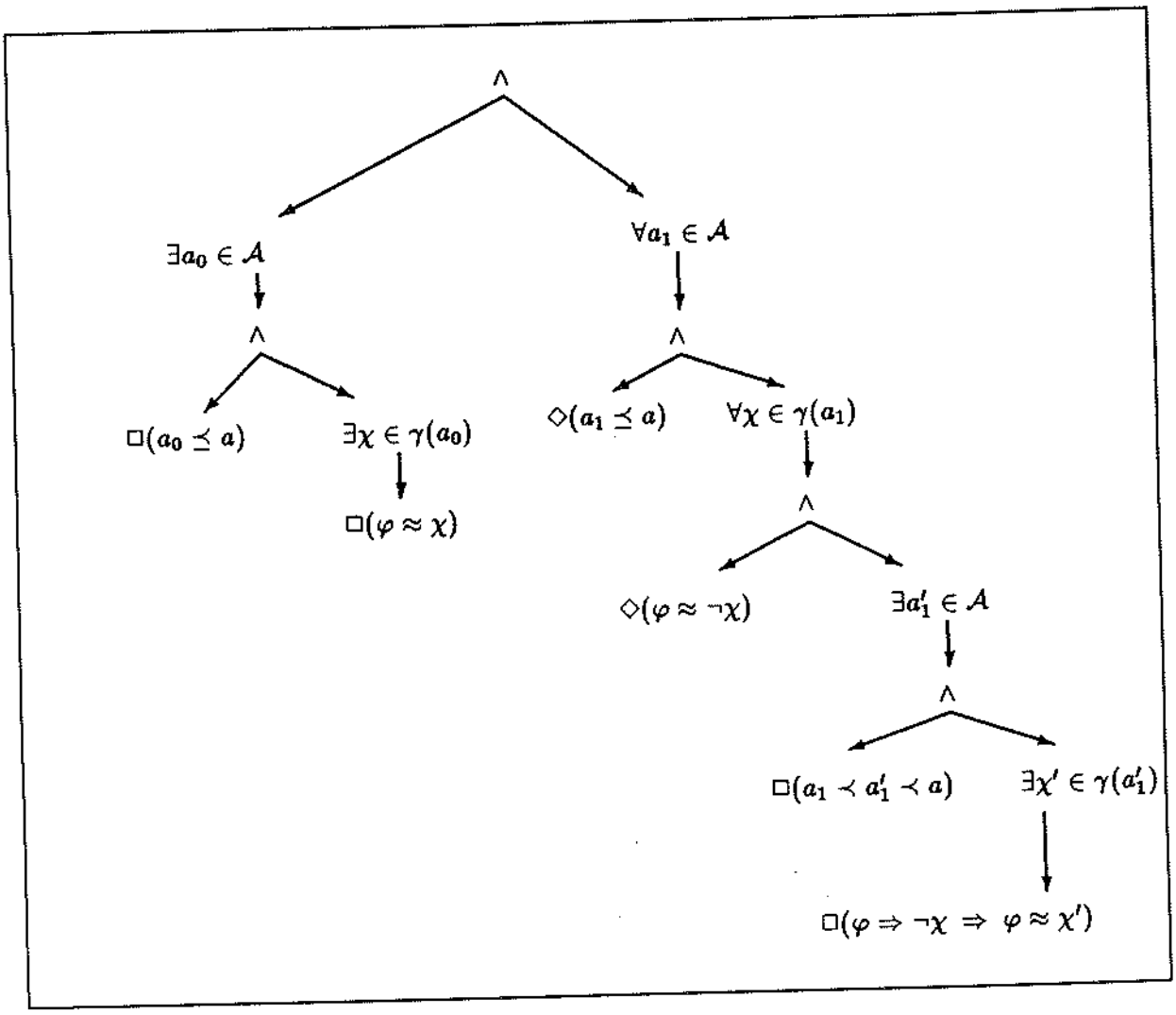


Figure 6.1: Interprétation graphique du critère de vérité d'ordre 1

2. Choisir l'un de ces schémas d'actions peut être basé sur des critères syntaxiques. Par analogie avec les moteurs d'inférences, ce pourrait être : la règle qui a le moins de prémisses (pour tenter d'éviter une explosion des sous-buts), la règle qui a le plus de conclusions (la règle la plus informante), ...Le nombre de critères effectivement utilisés pour ces moteurs d'inférences montre bien l'embarras du tout-syntaxique ; la planification non-linéaire permet d'introduire des notions sémantiques minimales par l'ajout d'un champ *déclencheur* (*trigger*) à un schéma d'actions, qui serait constitué de prémisses relatives non pas à l'exécution de l'action (une fois sa présence décidée) mais relatives à sa présence dans le plan courant [Wilkins 86].

Les contraintes d'unification posées sur cette nouvelle action se réduisent à l'unification nécessaire entre la conclusion tant désirée et φ. Les contraintes de précédence posées sur cette nouvelle action a_0 sont minimales : en appelant a_{initiale} l'action virtuelle créant la situation initiale et a_{finale} l'action virtuelle créant la situation finale, on a : $a_{\text{initiale}} \prec a_0 \prec a_{\text{finale}}$. Ceci conserve simplement a_{initial} comme *source* et a_{final} comme *puits* (a_0 est placée en parallèle à tout le plan).

Le danger principal résultant de cet ajout d'actions est l'explosion (combinatoire) du nombre d'actions présentes dans le plan, par exemple en partant sur une branche infinie.

Considérons par exemple Tarzan essayant de s'adapter à sa jungle pour y être heureux. Le schéma d'actions qu'il a à sa disposition est celui dans lequel un homme et une femme peuvent concevoir un enfant (une fille, en l'occurrence[1]) :

```
(defactschema (concevoir-fille x y z)
    (femme x) (homme y)
   ->
    (femme z) (mere x z) (pere y z) (heureux x) (heureux y)
)
```

Pour que le but *heureux*(*Tarzan*) soit réalisé, le planificateur ne peut guère qu'instancier `concevoir-fille` en *concevoir-fille*(x_1, *Tarzan*, z_1) (voir figure 6.2).

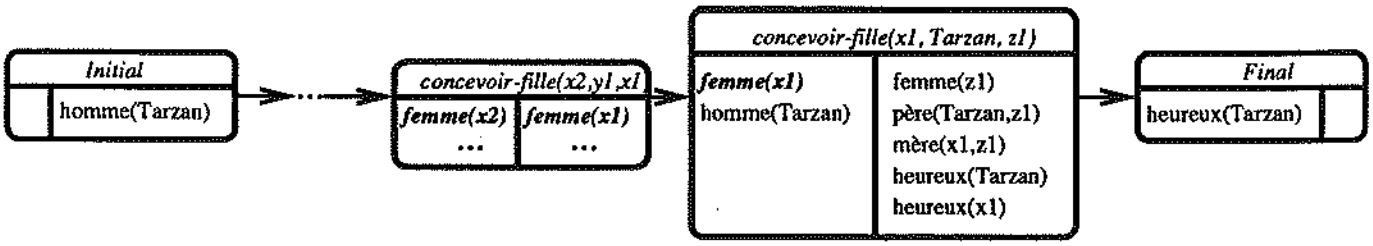


Figure 6.2: Emergence d'une branche infinie

Sur les deux buts de l'action *concevoir-fille*(x_1, *Tarzan*, z_1), *homme*(*Tarzan*) est déjà réalisé, *femme*(x_1) ne l'est pas : comme le plan n'a pas d'établisseur s'y rattachant, le planificateur instancie à nouveau le schéma d'actions `concevoir-fille` en l'action *concevoir-fille*(x_2, y_1, x_1) (littéralement, il cherche une mère à la femme qui rendra Tarzan heureux ...). Le sous-but *homme*(y_1) peut se réaliser par la contrainte d'adéquation $\Box(y_1 \approx Tarzan)$ (*adéquation de la conclusion établissante*, cf. deux paragraphes ci-dessous). Mais le sous-but *femme*(x_2) nécessite comme précédemment de créer un schéma d'actions, ce qui re-génère le même sous-but *femme*(x_3), ... et ainsi de suite en une série de *concevoir*(x_{n+1}, *Tarzan*, x_n). En se contentant de notre critère, le planificateur n'a *aucun moyen* d'échapper à cette branche infinie.

Pallier cette explosion potentielle peut se faire (classiquement) de deux manières extrêmes. D'abord, prouver que le problème nécessite effectivement l'ajout d'une action (i-e que cet ajout est valide), ce qui peut s'effectuer en parcourant *en largeur* les branches de *méta-planification* ; on est alors certain qu'il n'existe aucune solution à moins de n coups de la racine (en appelant n le nombre de décisions prises par le méta-planificateur depuis le début du processus de planification sur un problème donné). Ensuite, trouver une heuristique donnant une borne supérieure du nombre de coups que doit jouer le méta-planificateur (exemple typique de ce genre d'évaluation au début de 7.1), solution brutale mais qui a l'avantage de couper les branches infinies tout en s'adaptant à un grand nombre de contrôles.

[1] Un schéma d'actions `concevoir-garcon` n'aurait changé que la première itération, puisqu'il faudrait de toutes façons trouver une `mere(x,z)` au-dit garçon.

Après l'ajout de ce nouveau nœud, on se ramène au deuxième cas initial, dans lequel il existe déjà un établisseur potentiel. Cet établisseur potentiel deviendra établisseur effectif par l'ajout des contraintes réalisant le reste de la condition d'ajout, ce qui est discuté dans les deux paragraphes suivants.

Précédence de l'établisseur ($\Box(a_0 \preceq a)$) Un établisseur devant se trouver avant a, tout établisseur potentiel situé en parallèle de a peut convenir, s'il lui est ajouté cette contrainte de précédence.

Cet ajout de contrainte de précédence, simple ajout d'un lien dans le formalisme que nous avons développé (cf. 5.1.3), est caractéristique des planificateurs non-linéaires : ni les planificateurs linéaires, qui tentent de simuler cet ajout de contrainte, (cf. Waldinger en 3.1.2), ni a *fortiori* les exécuteurs (tels les moteurs d'inférences ou les résolveurs de problèmes parcourant des graphes d'états), ne peuvent représenter cet ajout de contrainte de précédence autrement qu'en l'exécutant.

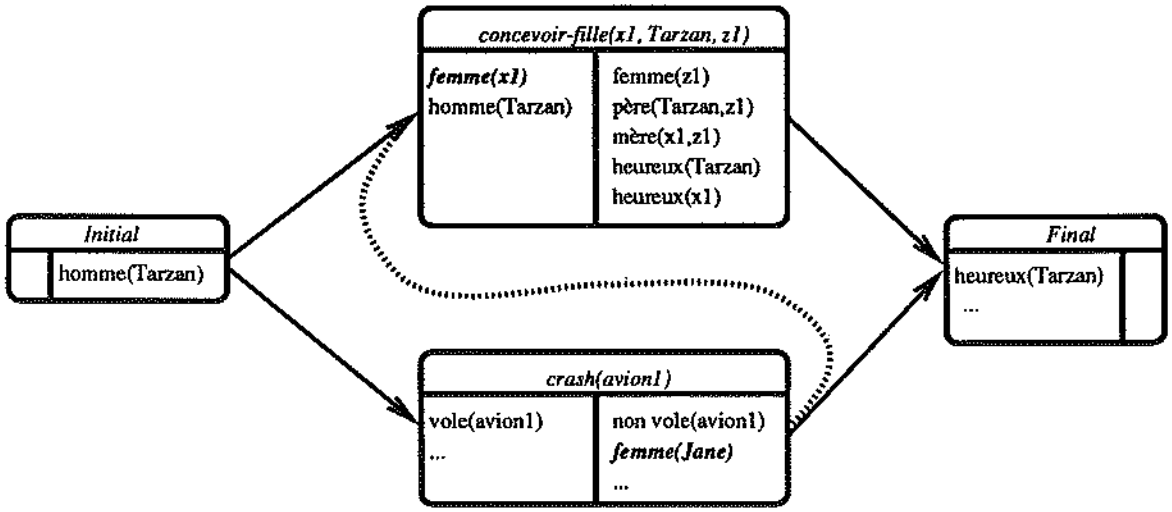


Figure 6.3: Etablissement par ajout d'une contrainte de précédence

Dans l'exemple de la figure 6.3 (poursuivant celui de la figure 6.2), le planificateur a instancié comme précédemment le schéma d'actions *concevoir-fille*(x, y, z) en *concevoir-fille*$(x_1, Tarzan, z_1)$ pour établir le but *heureux*$(Tarzan)$, et il restait le sous-but *femme*(x_1) à satisfaire. Pour satisfaire un autre but de la pseudo-action finale, le planificateur a maintenant du instancier un schéma d'actions *crash*(x) en *crash*$(avion_1)$, dont une des conclusions est *femme*$(Jane)$ (Jane était passager de $avion_1$). Le planificateur va repérer que *femme*$(Jane)$ est un établisseur potentiel de *femme*(x_1). Comme c'est le seul, le planificateur ajoute la contrainte de précédence situant *crash*$(avion_1)$ nécessairement avant *concevoir-fille*$(x_1, Tarzan, z_1)$.

Le terme *femme*(x_1) n'est pas encore tout à fait satisfait : il manque la contrainte d'unification, que nous considérerons au paragraphe suivant.

Adéquation de la conclusion établissante ($\exists \chi \in \gamma(a_0), \Box(\varphi \approx \chi)$) Un établisseur potentiel a_0 établit φ si une de ses conclusions s'unifie nécessairement à φ ; comme on l'a vu en 5.3.2,

l'unification *possible* n'est pas suffisante puiqu'elle laisserait échapper des instanciations de φ qui ne seraient pas établies. Etablir φ en a en rendant effectif un établisseur potentiel passe par l'ajout d'une contrainte d'unification sur l'une de ses conclusions (*contrainte d'adéquation*).

Une utilisation courante de cette contrainte correspond au cas où l'établisseur potentiel χ est plus "précis" que φ : χ est sans variable alors que φ est sans constante, par exemple. La contrainte d'adéquation consiste alors à instancier les variables de φ sur les constantes correspondantes de χ.

Cette utilisation de la contrainte d'adéquation en tant qu'*instanciation* est systématique pour les nouvelles actions qui viennent juste d'être dérivées d'un schéma d'actions ; dans le vocabulaire des moteurs d'inférences, cela correspondrait à l'appariement de patrons (*pattern-matching*) lors du déclenchement d'une règle d'ordre 1 en chaînage avant.

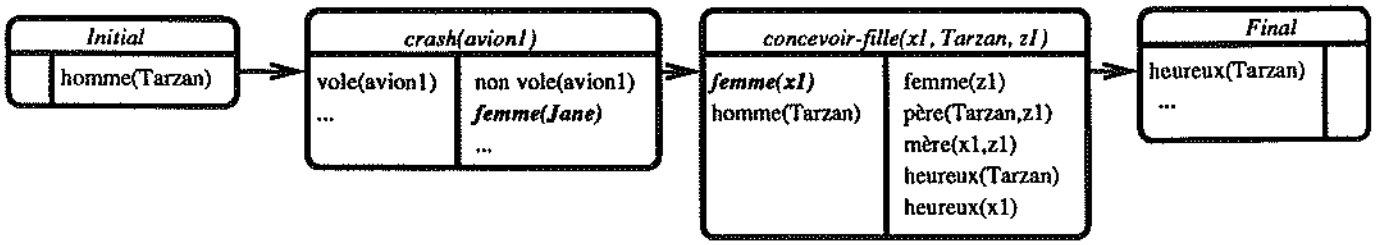


Figure 6.4: Pseudo-chaînage avant par instanciation d'une variable

Dans l'exemple de la figure 6.4 (poursuivant celui de la figure 6.3), l'établissement de *heureux*(*Tarzan*) a déjà imposé l'instanciation du schéma d'actions *concevoir-fille*(x, y, z)[2] en l'action *concevoir-fille*$(x_1, Tarzan, z_1)$, le lien de précédence entre *crash*$(avion_1)$ et *concevoir-fille*$(x_1, Tarzan, z_1)$; Le planificateur cherche ensuite à satisfaire le sous-but *femme*(x_1) de l'action *concevoir-fille*$(x_1, Tarzan, z_1)$ à partir de l'unique établisseur potentiel *femme*(*Jane*) de la pseudo-action initiale. Ceci est réalisé par la contrainte d'unification $\Box(x_1 \approx Jane)$, i-e l'instanciation de x_1 en *Jane*. Tous les buts sont réalisés, aucun conflit ne subsiste : le plan est terminé (en ce qui concerne les termes exprimés). Cette dernière instanciation aurait pu être obtenue de façon analogue par un moteur d'inférences d'ordre 1 fonctionnant en chaînage avant.

6.1.2 Non-destruction intermédiaire

La clause de non-destruction intermédiaire, deuxième terme du critère, s'assure qu'une fois que le terme φ a été établi (par le premier terme), il n'y a pas d'action qui détruise même partiellement la valeur de φ et que si le terme φ est quand même détruit (conclusion ψ d'un masqueur a_1), il y a toujours une action salvatrice qui vient reconstruire la valeur détruite (conclusion ψ' du démasqueur a'_1). Ou, dit autrement, depuis la situation à laquelle appartient φ, on ne doit pas pouvoir "voir" de termes négatifs (relativement à φ), la portée de cette vision arrière s'arrêtant à la dernière action relative à φ sur chaque branche.

[2]Cette fois, utiliser le *concevoir-fille*(x, y, z) ou *concevoir-garçon*(x, y, z) n'a aucune importance, puisqu'il ne figure aucun but *femme*(...) en prémisse de la pseudo-action finale.

Promotion d'un masqueur ($\Diamond(a_1 \preceq a)$) Pour être un masqueur effectif, un masqueur potentiel doit (au moins) être avant a ou en parallèle de a. Une première façon de se débarrasser d'un masqueur a_1 est d'ajouter la contrainte $a_1 \succ a$ (correspondant à $\neg(\Diamond(a_1 \preceq a))$, i-e à $\Box(a_1 \succ a)$). Cette *promotion* du masqueur [Chapman 85] n'est bien sûr applicable que pour les masqueurs situés initialement en parallèle de a.

Dans l'exemple de la figure 6.5, le but est de peindre le plafond *et* l'escabeau [Sacerdoti 77] : la pseudo-action finale a en prémisse les deux buts *peint-escabeau* et *peint-plafond*.

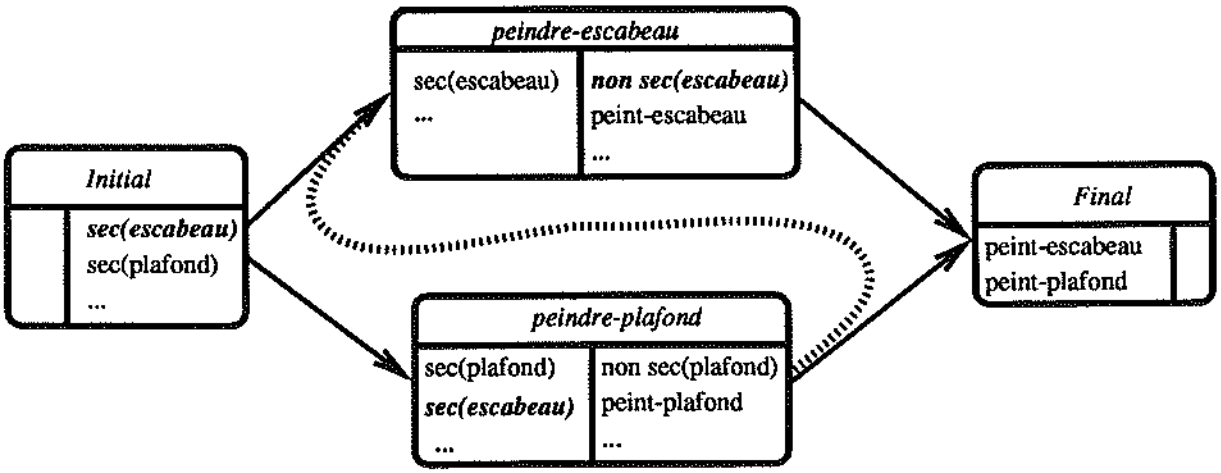


Figure 6.5: Promotion d'un masqueur

Initialement, l'escabeau et le plafond sont secs (les conclusions de la pseudo-action initiale sont *sec*(*escabeau*) et *sec*(*plafond*)). Pour peindre l'escabeau, il faut, dans l'absolu, (1) avoir de la peinture, (2) avoir un escabeau, ...(cf. le problème de la qualification en 2.1.3) ; la seule chose qui importe ici est que l'action *"peindre-escabeau"* possède la conclusion $\neg$*sec*(*escabeau*) en plus de la conclusions normale *peint-escabeau* satisfaisant le but final. Pour peindre le plafond, il faut pouvoir monter sur un escabeau sec (pour ne pas se tacher), *sec*(*escabeau*), en plus d'avoir un plafond sec, *sec*(*plafond*) ; le plafond sera peint, *peint-plafond*, et ne sera plus sec $\neg$*sec*(*plafond*).

Le problème du peintre est qu'il est impossible de peindre l'échelle sur laquelle il est monté pour peindre le plafond. Dans le langage de notre planificateur, ce problème se traduit par un conflit qui provient de la prémisse *sec*(*escabeau*) en prémisse de l'action *"peindre-plafond"* : l'algorithme VALEUR0 de 5.3.1 (on est évidemment en ordre 0) renvoie $\Diamond$*sec*(*escabeau*), à cause de la situation initiale, et $\Diamond\neg$*sec*(*escabeau*), à cause de l'action *"peindre-escabeau"* située en parallèle.

Pour éviter ce masquage en se servant de la contrainte de précédence $\Diamond(a_1 \preceq a)$, le planificateur pose la contrainte inverse $\Box(a_1 \succ a)$, qui repousse le masqueur *"peindre-escabeau"* (a_1) après l'action demandeuse *"peindre-plafond"* (a). On retrouve (heureusement) l'heuristique de linéarisation de [Sacerdoti 77], qui est conforme à l'intuition la plus immédiate. Cette solution, minimale en nombre d'actions, n'est pourtant pas la seule, comme nous le verrons ci-après.

Inadéquation de la conclusion masqueuse (ou séparation) ($\Diamond(\varphi \approx \neg\psi)$) Un masqueur potentiel devient effectif si (au moins) l'une de ses conclusions ψ peut nier le but φ (unification possible avec la négation). Une autre façon de libérer φ d'un terme masqueur ψ du masqueur

a_1 est donc de poser $\square(\varphi \not\approx \neg\psi)$, i-e d'ajouter une contrainte d'unification entre φ et toutes conclusions potentiellements masqueurs ψ.

Cette contrainte d'inadéquation est l'exact pendant pour un masqueur de la contrainte d'adéquation d'un établisseur (cf. 6.1.1) : une utilisation courante de cette contrainte d'inadéquation correspond au cas où ψ est sans variable alors que φ est sans constante ; repousser un masqueur par cette contrainte revient à enlever les constantes de ψ des instances possibles des variables de φ.

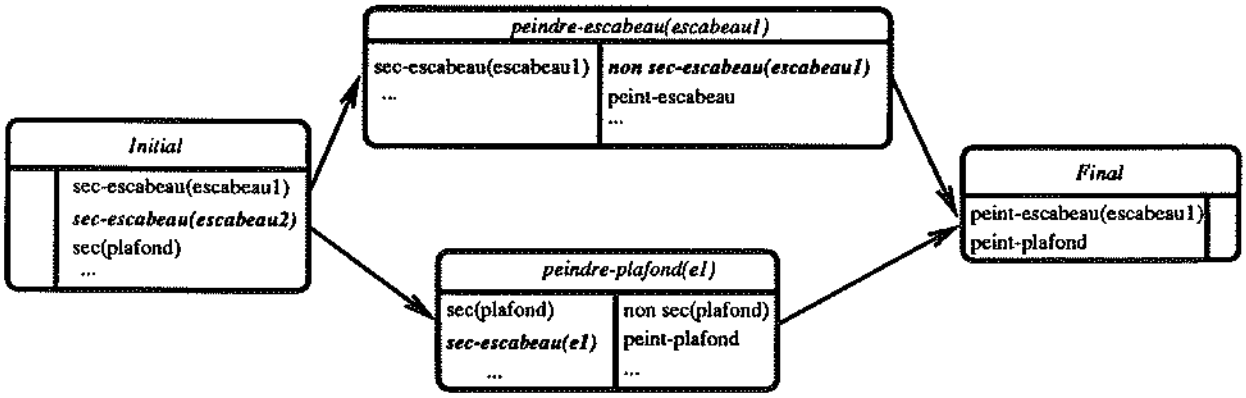


Figure 6.6: Inadéquation de la conclusion masqueuse

Nous reprenons l'exemple du plafond et de l'escabeau de la figure 6.5, mais en ajoutant un deuxième escabeau (voir figure 6.6). Comme précédemment, le planificateur a instancié un schéma d'actions *peindre-escabeau*(e) en *peindre-escabeau*$(escabeau_1)$ pour réaliser le but *peint-escabeau*$(escabeau_1)$, et a instancié le schéma d'actions *peindre-plafond*(e) en l'action *peindre-plafond*(e_1). Comme précédemment, la prémisse *sec-escabeau*$(escabeau_1)$ de l'action *peindre-escabeau*$(escabeau_1)$, ainsi que *sec*$(plafond)$ de l'action *peindre-plafond*(e_1), sont directement satisfaites par la pseudo-action initiale. Le planificateur cherche maintenant à satisfaire la prémisse *sec-escabeau*(e_1) de l'action *peindre-plafond*(e_1) à partir des établisseur potentiel *sec-escabeau*$(escabeau_1)$ ou *sec-escabeau*$(escabeau_2)$: si le planificateur instancie la variable e_1 en la constante $escabeau_1$, la pseudo-action initiale est masquée par la conclusion $\neg$*sec-escabeau*$(escabeau_1)$ de l'action *peindre-escabeau*$(escabeau_1)$, ce qui se résout par la promotion du masqueur (cf. paragraphe précédent) ; si par contre le planificateur instancie e_1 en $escabeau_2$, cette conclusion $\neg$*sec-escabeau*$(escabeau_1)$ de l'action *peindre-escabeau*$(escabeau_1)$ ne masque plus *sec-escabeau*$(escabeau_2)$. Dans ce dernier cas, il n'y a plus de conflits et tous les buts et sous-buts sont satisfaits : le plan est donc terminé. On peut effectivement peindre un plafond et un escabeau en allant chercher un deuxième escabeau pour peindre le plafond, ce qui découple les deux sous-problèmes ; il n'y a alors effectivement plus aucune de raison de peindre l'un avant l'autre.

Le choix d'instancier e_1 en $escabeau_1$ ou en $escabeau_2$ dépend de la structure de contrôle (cf. 6.2).

Existence d'un rétablisseur ($\exists a'_1 \in \mathcal{A}$) La présence d'un démasqueur n'est nécessaire que lorsqu'il y a un masqueur ne pouvant ni être *promu* (cf. 2 paragraphes ci-dessus), ni être rendu *inadéquat* (cf. ci-dessus) ; l'action négative du masqueur est effacée soit par un démasqueur existant soit en créant un démasqueur.

Indépendamment des (futures) contraintes de précédence et d'unification que ce démasqueur devra entretenir (cf. les 2 paragraphes suivants), se pose le problème du *choix* du schéma d'actions qui va s'instancier en un démasqueur. Cette question a déjà été discutée en 6.1.1 à propos de l'existence d'un établisseur. La seule différence est la suivante : la détermination des schémas d'actions candidats s'effectue sur l'existence d'une conclusion ψ' vérifiant l'implication $\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi'$ (cf. ci-dessous). Il ne s'agit donc plus à proprement parler d'un authentique chaînage arrière.

Comme exemple, nous reprenons le cas de la figure 6.5, dans lequel un quidam devait peindre sans se tacher un escabeau et un plafond avec un seul escabeau, pour en extraire maintenant l'autre solution. Si l'on se passe de la contrainte d'alors $a_1 \succ a$ (promotion), le plan résultant est celui de la figure 6.7, dans lequel il faut pouvoir monter sur un escabeau non encore sec.

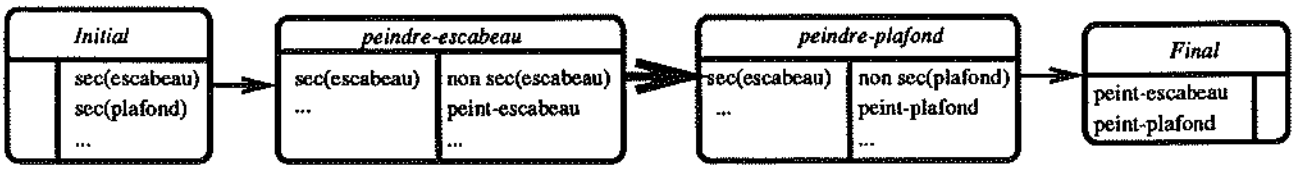


Figure 6.7: L'état initial avant ajout de démasqueur

Le sous-but *sec(escabeau)* est établi par la pseudo-action initiale, est masqué par l'action *"peindre-escabeau"* et il n'y a pas de démasqueur possible (le plan est déja linéaire et il n'y a aucune action entre *"peindre-escabeau"* et *"monter-escabeau"*). Le seul espoir est de trouver un schéma d'actions pouvant conclure sur *sec(escabeau)* ; c'est le schéma d'actions *"sèche-tout-seul"*, caractérisant un comportement implicite du micro-monde modélisé[3] :

```
;;; Les chaussettes de l'archiduchesse sont-elles sèches ?

(defactschema (seche x)
    (non (sec x))
   ->
    (sec x)
)
```

Finalement, une action `(seche x1)` est dérivée de ce schéma d'actions. Les contraintes sur un démasqueur (cf. les deux paragraphes suivants) imposeront que $\Box x_1 \approx$ *escabeau* (c'est bien l'escabeau qui sèche) et que l'action `(seche escabeau)` se trouve entre `peindre-escabeau` et `monter-escabeau` (voir figure 6.8).

La solution alternative ainsi construite suggérerait donc de peindre d'abord l'échelle, d'attendre qu'elle sèche, puis de s'en servir pour peindre le plafond ... Le planificateur n'a, en l'état, aucune raison de choisir l'une plutôt que l'autre ; c'est précisément le rôle de la structure de contrôle de s'informer suffisamment sur le micro-monde pour trancher :

- en l'absence d'informations supplémentaires, l'heuristique implicite *"minimiser le nombre*

[3] C'est action devrait durer "un certain temps" (dans le cas d'un fût de canon par exemple), dépendant de l'état de surface de l'objet x, de la qualité de la peinture, ... Ceci ne peut pas être spécifié dans notre planificateur puisqu'aucune notion de durée numérique n'a été intégrée (voir 5.1.3).

Figure 6.8: Résolution d'un masquage par ajout d'un démasqueur

d'actions" (sous-entendant une notion de *coût* par action effectuée) vote pour la solution "promotion".

- si notre mini-plan fait partie d'un plan plus général, dans lequel on doit déjà peindre l'échelle avant le plafond (un sous-traitant y passe les gaines électriques, par exemple), la solution "démasqueur" profite de ce temps de latence.

Rétablisseur intermédiaire ($\Box(a_1 \prec a'_1 \prec a)$) Un rétablisseur potentiel devient effectif si (au moins) il se situe nécessairement entre le masqueur et l'action qu'il rétablit. Ce démasquage ne peut se faire que par l'ajout explicite des deux contraintes $a_1 \prec a'_1$ et $a'_1 \prec a$.

Une conséquence est que $\Box(a_1 \prec a)$: tout masqueur situé en parallèle de a n'a aucune chance d'être démasqué puisqu'il ne peut pas vérifier la condition du rétablisseur intermédiaire.

Dans la figure 6.8 par exemple, le rétablisseur *sèche(escabeau)* a été justement intercalé entre le masqueur *peindre-escabeau* et l'action demandeuse *peindre-plafond.*

L'implication d'unification ($\exists \psi' \in \gamma(a'_1), \Box(\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi')$) L'implication d'unification entre la conclusion masqueuse ψ, la conclusion démasqueuse ψ' et le terme demandeur φ, signifie que φ et ψ' doivent s'unifier chaque fois que φ et $\neg\psi$ s'unifient. En terme d'*instanciations* ι (cf. 5.2.2), cette implication $\Box(\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi')$ s'écrit : $\forall \iota \in \mathcal{I}, \iota(\varphi) = \iota(\neg\psi) \Rightarrow \iota(\varphi) = \iota(\psi'))$, c'est-à-dire : $\forall \iota \in \mathcal{I},\ \iota(\varphi) \neq \iota(\neg\psi) \vee \iota(\varphi) = \iota(\psi'))$.

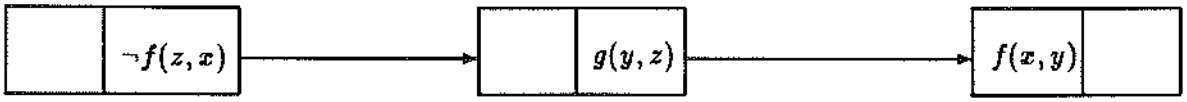


Figure 6.9: Implication sur les termes masqueurs, démasqueurs et demandeurs

Par exemple (voir figure 6.9), dans le cas où $\neg f(z,x)$ est le masqueur, $g(y,z)$ est le rétablisseur et $f(x,y)$ le demandeur, l'implication précédente se récrit en :

$$\begin{aligned} & \neg(\neg f(z,x)) \approx f(x,y) \Rightarrow g(y,z) \approx f(x,y) \\ \text{soit :} \quad & (z \approx x \wedge x \approx y) \Rightarrow (g = f \wedge y \approx x \wedge z \approx y) \\ \text{soit encore :} \quad & x \approx y \approx z \Rightarrow g = f \wedge x \approx y \approx z \\ \text{en simplifiant par } x \approx y \approx z : \quad & g = f \end{aligned}$$

Ce genre de manipulation formelle nécessite la présence d'un système de réécriture. Considérant que ce cycle supplémentaire ralentirait énormément le processus de planification pour

un gain qualitatif somme toute ridicule, nous avons implémenté cette implication :

$$\forall \iota \in \mathcal{I},\ \iota(\varphi) \neq \iota(\neg\psi) \vee \iota(\varphi) = \iota(\psi'))$$

par une condition suffisante un peu plus forte :

$$(\forall \iota \in \mathcal{I},\ \iota(\varphi) \neq \iota(\neg\psi)) \vee (\forall \iota' \in \mathcal{I},\ \iota'(\varphi) = \iota'(\psi'))$$

c'est-à-dire : $\Box(\varphi \not\approx \neg\psi) \vee \Box(\varphi \approx \psi')$.

Le premier terme du "$\vee$" ne dépend pas de a'_1 : on retrouve exactement le cas de l'*inadéquation des conclusions masqueuses*, développé plus haut. Si cette inadéquation ne nous a pas alors débarrassé du masqueur a_1, il n'y aucune raison pour laquelle elle le ferait maintenant. Aussi la vérification de cette implication est ramenée au deuxième terme du "$\vee$" : $\exists\psi' \in \gamma(a'_1), \Box(\varphi \approx \psi')$.

Moyennant la simplification (suffisante) présentée, la contrainte d'implication se ramène à une simple contrainte d'unification $\Box(\varphi \approx \psi')$ entre la conclusion démasqueuse ψ' et le terme demandeur φ, ce qui revient finalement à créer un établisseur supplémentaire, mais bien placé.

6.2 Stratégie de contrôle

Après avoir présenté (tous) les moyens d'actions dont dispose le planificateur, nous présentons maintenant la stratégie déterminant l'emploi de l'un plutôt que l'autre.

6.2.1 Contrôle global

Cycle déduction / induction L'algorithme le plus général est adapté de PLANIFIER1 de 3.2.1. En notant $\mathcal{A}$ l'ensemble des actions du plan, $\prec$ l'ordre partiel entre ces actions, $\mathcal{U}$ la base d'unification, $\models_a \varphi$ le fait que le terme φ est vrai dans la situation juste avant l'action a, cet algorithme est le suivant :

Procédure PLANIFIER2($\mathcal{A}, \prec, \mathcal{U}$)
  si $\exists a \in \mathcal{A}, \exists\varphi \in \pi(a), \models_a \neg\Box\varphi$, **alors**
    - **choisir** un tel couple (φ, a)
    - **si il existe une** modification m du plan, telle que $\models_a \Box\varphi$, **alors**
      - **choisir** une telle modification m
      - $(\mathcal{A}', \prec', \mathcal{U}') \leftarrow$ application de m au plan $(\mathcal{A}, \prec, \mathcal{U})$
      - appeler PLANIFIER2($\mathcal{A}', \prec', \mathcal{U}'$)
    sinon ECHEC
  sinon SUCCES

La sous-boucle de PLANIFIER1 de 3.2.1 qui résorbait les conflits générés pourrait se coder par :

; Enlever tous les conflits résultants de la satisfaction de (φ, a)
- **tant que** $\exists a' \in \mathcal{A}, \exists \varphi' \in \pi(a'), \models_{a'} \Diamond\neg\varphi'$, **faire**
  - **choisir un tel couple** (φ', a')
  - **si il existe une contrainte** c' **telle que** $\models_{a'} \neg\Diamond\neg\varphi'$, **alors**
    - **choisir une telle contrainte** c'
    - **appliquer** c' **au plan** $(\mathcal{A}, \prec, \mathcal{U})$

  sinon ECHEC

Mais la dualité des modalités $\Diamond$ et $\Box$ ($\forall \varphi \in T_\Sigma[X]$, $\neg\Box\varphi \Leftrightarrow \Diamond\neg\varphi$, voir 2.4) rend le test d'existence d'un conflit ($\exists a' \in \mathcal{A}, \exists \varphi' \in \pi(a'), \models_{a'} \Diamond\neg\varphi'$) équivalent à celui d'existence d'une prémisse non satisfaite ($\exists a \in \mathcal{A}, \exists \varphi \in \pi(a), \models_a \neg\Box\varphi$). **Résoudre un conflit et satisfaire une prémisse sont une seule et même opération**, que l'on choisit de considérer sous l'angle (optimiste) de la *satisfaction de prémisse.*

La résolution du problème (SUCCES) de planification initial par le plan $(\mathcal{A}, \prec, \mathcal{U})$ est prononcée lorsque toutes les prémisses sont satisfaites ($\neg(\exists a \in \mathcal{A}, \exists \varphi \in \pi(a), \models_a \neg\Box\varphi)$). Sinon, PLANIFIER2 choisit une prémisse insatisfaite, choisit une modification la satisfaisant, l'applique au plan courant, ... et recommence. La modification m du plan $(\mathcal{A}, \prec, \mathcal{U})$ peut concerner soit l'ajout d'une action ($\mathcal{A}$), soit l'ajout d'une contrainte de précédence ($\prec$), soit l'ajout d'une contrainte d'unification ($\mathcal{U}$), d'après 6.1. L'ajout de contraintes réduit l'espace des solutions et assurerait seul une convergence à terme. Mais l'ajout d'action l'augmente, aussi on ne peut déterminer la convergence du planificateur à la seule vue de l'algorithme PLANIFIER2.

Remarquons que le calcul de la satisfaction de chaque prémisse n'a pas à être toujours effectué à chaque cycle : la valeur de vérité d'une prémisse est déterminée par un critère *nécessaire*, i-e cette valeur est valable pour toute complétion en ordre total ou pour toute instanciation des variables, donc n'a a fortiori pas de raison de changer lors de l'ajout d'une contrainte d'unification ou de précédence. Par contre, l'ajout d'une action peut changer la valeur nécessaire des prémisses (comme le montre n'importe quel exemple). Des trois méta-actions possibles, l'ajout d'action est donc la seule qui nécessite un recalcul de la valeur de ces prémisses.

Type de parcours Des cas passeront toujours au travers de ces heuristiques et conduiront le planificateur à une impasse, i-e un plan trop contraint (ECHEC). Nous utilisons contre cela un mécanisme de *retour-arrière*, permettant de rétablir exactement le plan dans un état antérieur. YAPS peut ainsi faire marche arrière lorsqu'il se trouve dans une impasse, ou chercher d'*autres* solutions lorsqu'il en a trouvé une.

L'espace des états dont les transitions sont les modifications successives apportées au plan est parcouru en *profondeur d'abord* avec retour-arrière *chronologique.*

Le *profondeur d'abord* a l'avantage de nécessiter peu de structures supplémentaires (la *pile de méta-actions*[4]). Par contre, son principal défaut connu est de se perdre systématiquement

[4]Nous avons vu au chapitre 5 qu'une implémentation efficace de ces "méta-actions" requéraient beaucoup d'effets de bord. Leur réversibilité implique de constituer une deuxième pile, relative à la précédence (quel lien a été ajouté ou détruit par NFT-EFFACE ?), et une troisième, pile relative à l'unification (quelles modifications

sur toute branche infinie (cf. l'exemple de la figure 6.2). L'ajout de contraintes de précédence et d'unification restreignant toujours l'espace des solutions (même si cet espace est grand), la seule branche infinie constructible concerne l'ajout d'actions. Cette explosion nous a conduit à adopter l'heuristique suivante :

Heuristique 1 *Définir un diamètre maximal du plan.*

Nous appelons *distance horizontale* entre deux actions vérifiant $a \prec a'$ le maximum de l'ensemble des longueurs des chemins menant de a à a' (un lien ayant une valeur de 1). Le diamètre induit par cette distance est la distance horizontale maximale entre deux actions du graphe, i-e la distance horizontale entre la pseudo-action initiale et la pseudo-action finale.

Une implémentation non naïve de cette heuristique s'effectue par l'ajout d'un champ "*profondeur-horizontale*" à chaque action, qui représente la distance entre l'action propriétaire et la pseudo-action initiale. Le calcul de cette distance s'effectue par l'algorithme PROPAGER-DÉBUT de 2.2, où , pour tout arc (α, β, γ), $\alpha = \gamma = 0$ et $\beta = 1$. Les *profondeurs horizontales* des actions étant toujours partiellement initialisées, la complexité de PROPAGER-DÉBUT est améliorée en ne le lançant non pas systématiquement depuis la pseudo-action initiale, mais depuis les actions a dont une contrainte de précédence $a \prec \cdots$ ou $\cdots \prec a$ a été modifiée. Cette profondeur horizontale est aussi parfois appelée *tri topologique*.

Le diamètre du plan est alors la profondeur horizontale de la pseudo-action finale.

Disposant de **méta-actions réversibles**, presque tous les algorithmes connus de parcours d'arbre peuvent être envisagés à ce stade (largeur d'abord, définir une fonction d'évaluation pour un A^*-like, trouver des heuristiques fines, ...). Il serait d'ailleurs extrêmement simple de les implémenter dans le code actuel de notre planificateur : l'algorithme de contrôle est clairement séparé du reste du code (15 lignes à changer ...).

Nous n'argumenterons donc pas exagérément sur l'intérêt en soi d'un "profondeur d'abord retour-arrière chronologique", et nous bornerons à signaler que :

1. le principe du moindre engagement limite son aspect borné ;
2. trouver quelques heuristiques est plus important que sophistiquer le contrôle global ;
3. Wilkins (SIPE, cf. 3.2) et Tate (NONLIN, id.) se sont eux aussi contentés de cet algorithme ; Descottes (GARI, cf. 3.2.2) explique même pourquoi le retour-arrière "dirigé par les dépendances", fût-il à base de TMS, est bien trop lourd en général ;
4. substituer un autre algorithme au nôtre est, encore une fois, évident au vu du code (15 lignes à changer).

FORCE-UNIF vient-il d'effectuer sur la base de contraintes d'unification ?). En pratique, une méta-action ne consomme que peu de niveaux de ces deux autres piles : un niveau de la pile des méta-actions contient les deux morceaux de pile de précédence et d'unification (*"spaghetti stack"*).

Plutôt que de chercher un contrôle en terme d'*exécution* d'un méta-planificateur, nous esquisserons en 6.2.2 une voie qui nous semble bien plus prometteuse (et, aussi, difficile ...).

Quel sous-but satisfaire

Le critère de vérité a identifié les leviers sur lesquels le planificateur peut agir. Cependant, l'ordre dans lequel les prémisses non-établies sont considérées ne relève plus de ce critère, mais de la stratégie propre du planificateur. Un module d'inférences complet peut être envisagé à ce niveau [Dean 86], mais se paierait par une lourdeur et une inefficacité. Nous avons préféré câbler certaines heuristiques, que nous présentons maintenant selon leur priorité, pour combler les trous du critère en choisissant "mieux que ne le ferait le hasard" [Laurière 86].

Ce câblage (vs. module d'inférences) a été toléré dans la mesure où le code relatif à chaque heuristique peut être facilement retrouvé. Un effort particulier a été effectué pour comparimenter le code représentant chaque heuristique et identifier clairement son appel depuis la fonction de contrôle (cf. ci-dessus). Le code de ces heuristiques peut, alors seulement, être considéré comme le résultat de la compilation de la base de règles virtuelle les représentant proprement.

Heuristique 2 *Satisfaire les prémisses de la pseudo-action finale.*

Les prémisses de la pseudo-action finale font partie de la définition du problème de planification posé au planificateur. Elles ont à ce titre une priorité absolue, indépendamment de toute heuristique postérieure.

Cette heuristique assure que le planificateur tentera toujours de satisfaire le problème posé.

Heuristique 3 *N'ajouter de nouvelle action qu'en dernier recourt.*

La crainte *a posteriori* du risque d'explosion (en terme de nombre d'actions) du plan a déjà induit l'heuristique 1. La version *a priori* de cette explosion conduit à n'adopter l'instanciation d'un schéma d'actions que lorsqu'il n'y a pas d'autre moyen de progresser.

Cette heuristique est à sa priorité maximale : si elle était placée avant l'heuristique 2, le planificateur resterait pétrifié sur la ligne de départ, préférant ne rien résoudre du tout par peur de se perdre dans une branche infinie[5] ...

Les heuristiques suivantes traduisent l'implémentation de ce principe.

[5] Ce bug de réglage, digne d'un robot d'Asimov, s'est produit sur une ancienne version du présent planificateur, et fut l'un des arguments pratiques nous ayant conduit à isoler proprement chacune des heuristiques.

Heuristique 4 *Satisfaire d'abord les prémisses des nouvelles actions (par instanciation).*

Une nouvelle action a, qui vient d'être instanciée à partir d'un schéma d'actions et intégrée dans le plan, ne possède que deux contraintes de précédence ($a_{\text{initial}} \prec a \prec a_{\text{final}}$) et la contrainte d'unification sur la conclusion qui est à l'origine de son ajout.

Ses *variables libres* (ou plutôt, quantifiées existentiellement, cf. 5.2.2) sont de remarquables générateurs de branches infinies (cf. l'exemple de la figure 6.2) inutiles et coûteuses. Cette heuristique évalue d'abord le nombre d'instanciations possibles de ces variables libres (les constantes disponibles sont fixées par la situation initiale). Lorsqu'une seule instanciation est possible, l'instanciation est réalisée (il n'y a pas de choix ...).

Vis-à-vis de l'implémentation, la méthode d'instanciation d'un schéma d'actions stocke systématiquement l'action générée dans un endroit convenu. L'heuristique 4 consiste à : regarder si le niveau juste inférieur de la pile des méta-actions est une instanciation de schéma d'actions ; si c'est le cas, chercher l'action au point de rendez-vous ; parcourir ses prémisses en comptant le nombre d'établissement en cas de variable libre.

Ce comportement serait celui adopté par les heuristiques suivantes, mais de façon plus lourde, après une analyse coûteuse. L'intérêt de l'isolement de cette heuristique est ici sa *priorité* plus que sa nature.

Heuristique 5 *Réaliser les démasquages avant les établissements.*

Cette heuristique résulte de l'heuristique 3 et de deux constatations :

- un masquage est souvent résolu par ajout de contrainte, i-e restriction de l'ensemble des solutions ;
- un établissement est souvent résolu par ajout d'action, i-e par extension de l'ensemble des solutions ;

Ces arguments généralisent a posteriori la seule heuristique qu'implémente l'algorithme PLANIFIER1 de 3.2.1, basée sur l'idée qu'on ne peut ajouter une action que dans un plan sans conflits. Ce principe est faux dans le cas général : l'ajout d'une action peut justement être un moyen de résoudre un conflit (cf. 6.1). Sur un critère de coût, c'est l'explosion du plan qu'il faut craindre, non l'incapacité du planificateur à résoudre les conflits ...

Heuristique 6 *Satisfaire les prémisses dont le chemin de dépendance causale avec la pseudo-action finale est de longueur minimale.*

Si une action a est introduite pour satisfaire une prémisse d'une action a', on dit que a et a' sont en *relation de dépendance causale*, notée $cause(a, a')$ (cf. 3.1, notion que l'on retrouve également chez McDermott avec le lien *ECAUSE*, cf 2.6.2).

Cette relation ne doit pas être confondue avec la relation de précédence $\prec$. D'après le critère de vérité, une action ne peut influer sur une autre que si elle se trouve avant elle : $cause(a, a') \Rightarrow a \prec a'$. Mais on peut avoir $a \prec a'$ par un ajout de contrainte pour une raison quelconque, ces deux actions étant initialement en parallèle et n'ayant aucune dépendance causale *cause*. La réciproque est donc fausse.

Ensuite, si $cause(a, a')$, rien n'indique que a et a' doivent être consécutives $a \rightsquigarrow a'$ dans le plan (relation de minimalité du graphe représentant l'ordre partiel $\prec$, cf. 5.1.3) ; d'autres actions peuvent s'intercaler entre a et a' sans violer $a \prec a'$ ni $cause(a, a')$. On a en fait : $cause(a, a') \Rightarrow a \prec a' \Rightarrow a \rightsquigarrow a'$.

Ce lien de dépendance causale devrait alors être conservé, générant un graphe extrait du plan [Tate 77]. Mais l'heuristique 6 concerne la *longueur* du chemin menant de l'action à la pseudo-action finale, dans le graphe de dépendance causale.

On appelle *distance causale* entre deux actions a et a' le minimum de l'ensemble des longueurs des chemins (causaux) menant de a à a' *dans ce graphe de dépendance*. Par construction (i-e d'après l'heuristique 3), une action est *toujours* introduite pour satisfaire une prémisse, et est donc toujours reliée à la pseudo-action finale. La distance causale d'une action a avec la pseudo-action finale est appelée *profondeur verticale*[6] (et est notée $p_v(a)$).

L'heuristique 6 suggère que plus une action contribue à la résolution d'un sous-but proche du but global (plus sa profondeur verticale est faible), plus cette action est importante (plus il faut satisfaire rapidement ses prémisses). Elle définit une *fonction d'évaluation* de l'importance de la satisfaction des prémisses.

Cette heuristique est implémentée, en ajoutant un champ *"profondeur verticale"* à chaque action, de la façon suivante :

- la profondeur verticale de la pseudo-action finale est fixée à 1 ;
- si une prémisse d'une action a est satisfaite par l'ajout d'une action a', alors on a : $p_v(a') = p_v(a) + 1$.

[6] Cette notion de profondeur verticale ne doit pas être confondue avec la notion de hiérarchie (erreur commise par Sacerdoti, NOAH [Sacerdoti 77]) : deux niveaux hiérarchiques différents raisonnent sur des objets différents ; à *la hiérarchie des actions doit correspondre une hiérarchie isomorphe des objets* [Stefik 81], ce qui n'est absolument pas le cas pour la profondeur verticale : deux actions de profondeur verticale différente peuvent tout à fait manipuler les mêmes objets (cf. n'importe quel exemple de 6.1).

La profondeur verticale étant définie par un *minimum*, l'implémentation ci-dessus vérifie la définition.

Pour un ensemble de prémisses équivalentes selon les autres heuristiques, cette heuristique choisit la plus "importante" par un tri selon la profondeur verticale des actions qui les possèdent.

Comment satisfaire le sous-but choisi

Les heuristiques ci-dessus indiquent comment trier les sous-buts non satisfaits en catégories reflétant leur importance. Nous présentons maintenant la (les) façon(s) dont ces sous-buts peuvent être satisfaits, ce qui nous conduit à présenter d'autres sous-heuristiques liées, cette fois, à des facilités d'implémentation.

Établissement Nous avons vu en 6.1 quelles contraintes il fallait ajouter à un établisseur potentiel pour le transformer en établisseur nécessaire. La table 6.1, les résumant, indique, selon que l'établisseur potentiel a' soit nécessairement avant l'action a contenant le sous-but ψ à établir ou en parallèle, et selon que la conclusion établisseuse ψ' de a' s'unifie nécessairement ou possiblement avec ψ, s'il faut ajouter une contrainte d'unification ($\approx$) et/ou de précédence ($\prec$).

	Unification	
	□	◇
$a' \prec a$	$\emptyset$ (cas 0)	$\approx$ (cas 1)
$a' // a$	$\prec$ (cas 2)	$\prec, \approx$ (cas 3)

Table 6.1: Contraintes $\prec$ et $\approx$ pour l'établissement

Pour établir une prémisse, si les heuristiques du paragraphe précédent ne suffisent pas, les établisseurs potentiels sont répartis selon ces trois catégories (le cas 0 n'est pas une catégorie intéressante ici, puisqu'une action lui appartenant est un établisseur nécessaire, ce qui signifie que la prémisse est déjà satisfaite). Les sous-heuristiques suivantes permettent de trier ces catégories :

Sous-Heuristique 1 *Minimiser le nombre de contraintes à appliquer.*

Malgré notre algorithme de contrôle en profondeur (cf. ci-dessus), les solutions les plus proches (la distance étant le nombre de méta-actions) du problème posé sont tout de même recherchées.

Le cas 3 est donc le moins prioritaire.

Sous-Heuristique 2 *Contraindre la précédence avant l'unification.*

La contrainte d'unification établissant un sous-but est l'instanciation directe d'une variable, et non la pose d'une contrainte d'unification négative. L'espace des solutions est réduit brutalement. La pose d'une contrainte de précédence entre deux actions permettra quand même d'insérer d'autres actions entre ces deux actions : l'espace des solutions est comparativement moins réduit par une contrainte de ce dernier type.

Cette sous-heuristique ne fait que mettre en pratique le *principe du moindre engagement*, traduit ici par diminution la plus régulière possible (i-e la plus réfléchie) de l'espace des solutions.

Remarque : on pourrait croire que l'heuristique 3, qui essaie d'éviter les branches infinies, nous pousserait au résultat inverse, pour se débarrasser des variables libres. Mais ces établisseurs potentiels ont été introduits pour d'autres raisons que la satisfaction de la prémisse choisie : ils ont souvent déjà des contraintes de non-unification et ne sont pas d'authentiques variables libres.

Le classement final des cas de la table 6.1 est le suivant : cas 2 > cas 1 > cas 3.

Sous-Heuristique 3 *Maximiser le nombre de nouvelles prémisses déjà établies.*

Enfin, si, malgré l'heuristique 3, l'établissement requiert l'instanciation d'un schéma d'actions, cette sous-heuristique suggère un tri des schémas d'actions (dont une conclusion peut déja s'unifier avec la prémisse à établir). Cette sous-heuristique minimise le nombre de sous-buts non établis, et suggère de choisir d'instancier le schéma d'actions qui s'intègre le mieux au plan courant (minimum local).

Un argument contre cette heuristique consiste à remarquer que le plan courant n'est pas le plan final : rien ne garantit qu'un autre schéma d'actions s'intégrant mal à court terme, ne s'intégrera pas mieux à long terme lorsque le plan sera plus détaillé (hiérarchie [Wilkins 86]). On atteint là en effet les limites de ce qu'il est possible de déduire dans notre formalisme.

A l'implémentation, le planificateur parcourt les schémas d'actions candidat, crée l'environnement de liaison des variables (unification de la conclusion, déterminant la candidature, avec le sous-but à satisfaire), et utilise le "nombre de prémisses déjà établies" comme une fonction d'évaluation de ces candidats. Le premier minimum est effectivement instancié.

Cette liste de candidats, triée par fonction d'évaluation croissante, est conservée en vue des futurs retours-arrière (elle est stockée dans le niveau de pile correspondant à la méta-action).

Démasquage Pour décider de la façon de satisfaire une prémisse φ d'une action a masquée par la conclusion ψ d'une action a', nous distinguons les 4 cas suivants (voir table 6.2) :

	Unification	
	□	◇
$a' \prec a$	cas 1	cas 2
$a' // a$	cas 3	cas 4

Table 6.2: Contraintes $\prec$ et $\approx$ pour le démasquage

Comme précédemment, ces cas correspondent au croisement de la position du masqueur a' relativement à a ($a' \prec a$ ou $a' // a$) et la modalité d'unification des termes en conflits ($\Box(\varphi \approx \neg\psi)$ ou $\Diamond(\varphi \approx \neg\psi)$. Contrairement à l'établissement, le cas 1 (cas 0 de la table 6.1) ne peut pas cette fois être évité. L'ordre d'observation de ces cas est le suivant : cas 1 > cas 3 > cas 2 > cas 4, exactement pour les mêmes raisons que pour la table 6.1.

Sous-Heuristique 4 *Repousser un masqueur plutôt que le démasquer.*

La résolution d'un démasquage peut s'effectuer de deux façons (cf. 6.1) :

- le masqueur ne devient plus une action masqueuse (repoussement) ;
- le masqueur reste masqueur, mais son effet est annihilé par une autre action intermédiaire (démasquage).

La présente sous-heuristique se justifie ainsi : le deuxième cas peut nécessiter l'ajout d'une nouvelle action (s'il n'existe aucun démasqueur potentiel), ce qui est justement déconseillé par l'heuristique 3.

En conséquence, seul le premier cas de la table 6.2 ne peut pas être traité par cette sous-heuristique. Dans ce cas, la satisfaction de la prémisse φ de l'action a, masquée par la conclusion ψ ($\Box(\varphi \approx \neg\psi)$) de l'action a_1 ($\Box(a_1 \prec a_0)$), nécessite un démasquage complet, selon la procédure esquissée ci-dessous :

Procédure DÉMASQUAGE$(\varphi, a_0, \psi, a_1)$
; Détecter les démasqueurs potentiels existants
Soit $\mathcal{D} = \{a'_1 \in \mathcal{A},\ \Diamond(a'_1 \prec a_0) \wedge \Diamond(a_1 \prec a'_1), \exists \psi' \in \gamma(a'_1),\ \Diamond(\varphi \approx \neg\psi \Rightarrow \varphi \approx \psi')\}$
si $\mathcal{D} = \emptyset$, **alors**
  instancier un schéma d'actions
; Trier les démasqueurs $\mathcal{D}$
soit $\mathcal{D}_\Box = \{a'_1 \in \mathcal{D} \,/\, \Box(a'_1 \prec a_0) \wedge \Box(a_1 \prec a'_1), \exists \psi' \in \gamma(a'_1)\}$
soit $\mathcal{D}_\Diamond = \mathcal{D} \setminus \mathcal{D}_\Box$
soit $\mathcal{D}_{\Box,1} = \{a'_1 \in \mathcal{D}_\Box \,/\, \Box(a'_1 \prec a_0) \vee \Box(a_1 \prec a'_1)\}$
soit $\mathcal{D}_{\Box,2} = \mathcal{D}_\Box \setminus \mathcal{D}_{\Box,1}$
soit $\mathcal{D}_{\Diamond,1} = \{a'_1 \in \mathcal{D}_\Diamond \,/\, \Box(a'_1 \prec a_0) \vee \Box(a_1 \prec a'_1)\}$
soit $\mathcal{D}_{\Diamond,2} = \mathcal{D}_\Diamond \setminus \mathcal{D}_{\Diamond,1}$
Traiter dans l'ordre : $\mathcal{D}_{\Box,1}$, $\mathcal{D}_{\Box,2}$, $\mathcal{D}_{\Diamond,1}$, $\mathcal{D}_{\Diamond,2}$.

Cette procédure n'instancie de nouveau démasqueur que lorsque $\mathcal{D}$ est vide, respectant l'heuristique 3. Le choix du schéma d'actions géniteur s'effectue comme pour l'établissement.

L'ensemble des démasqueurs potentiels $\mathcal{D}$ est scindé en $\mathcal{D}_\Box$ et $\mathcal{D}_\Diamond$, selon que le démasqueur vérifie nécessairement ou possiblement l'implication de démasquage. Chacun est re-scindé en deux sous-ensembles, $\mathcal{D}_{x,1}$ et $\mathcal{D}_{x,2}$ ($x \in \{\Box, \Diamond\}$), selon qu'il faille une ou deux contraintes de précédence pour vérifier $a_1 \prec a'_1 \prec a_0$.

La sous-heuristique 1 suggère d'utiliser les démasqueurs de $\mathcal{D}_{\Box,1}$ et $\mathcal{D}_{\Diamond,1}$ avant ceux de $\mathcal{D}_{\Box,2}$ et $\mathcal{D}_{\Diamond,2}$ (une contrainte de précédence au lieu de deux). Elle suggère également d'utiliser les démasqueurs de $\mathcal{D}_{\Box,x}$ avant ceux de $\mathcal{D}_{\Diamond,x}$. L'ordre d'observation des ensembles $\mathcal{D}_{x,y}$ est alors celui de la procédure.

Ces quatre ensembles sont conservés dans le niveau de pile correspondant à la méta-action courante, pour les futurs retours-arrière.

Les trois autres cas de la table 6.2 se ramènent à ce premier cas. Ils sont considérés dans l'ordre établi ci-dessus, en essayant à chaque fois de repousser d'abord le masqueur, selon la sous-heuristique 4 :

- **pour le cas 3 :**
  1. **contraindre $a_1 \succ a_0$ (promotion),**
  2. **sinon (retour-arrière), voir le cas 1.**
- **pour le cas 2 :**
  1. **contraindre $\varphi \not\approx \neg\psi$ (inadéquation),**
  2. **sinon (retour-arrière), voir le cas 1.**
- **pour le cas 4 :**
  1. **contraindre $a_1 \succ a_0$ (promotion),**
  2. **sinon (retour-arrière), contraindre $\varphi \not\approx \neg\psi$ (inadéquation),**
  3. **sinon (retour-arrière), voir le cas 1.**

L'ordre des essais du dernier cas (4) provient de la sous-heuristique 2.

6.2.2 Méta-Planification

Si l'on distingue "programme" et "données" dans le processus de planification, on doit distinguer le plan (l'organisation des actions) et la structure de contrôle (quels traitements vont subir les données pour atteindre le but ?), qui peut, comme toute structure de contrôle, interroger ses données (cf. chapitre 5) et leur faire subir des modifications déterminées (cf. 6.1). Ce contrôle fait se déplacer à son tour le planificateur dans un graphe d'états, un état étant un graphe d'actions avec sa base de contraintes d'unification. Un planificateur se déplace "verticalement" (voir figure 6.10) dans ces états, alors qu'un moteur se déplace "horizontalement", de gauche à droite pour ceux qui fonctionnent en chaînage avant et de droite à gauche pour ceux qui fonctionnent en chaînage arrière.

Vu sous cet angle, beaucoup de types de contrôles connus peuvent s'appliquer (profondeur ou largeur d'abord, A^*, tous les types de retour-arrière, ...), et il est tentant de gérer ce contrôle ...par un système de planification du second ordre ! On appelle *méta-planification* la planification d'une activité de planification [Stefik 81].

Intervention des agents Nous nous sommes intéressés jusqu'à présent à la *simulation* de l'exécution d'actions. Le rôle moteur de celui qui effectue l'action (i-e l'*agent* qui réalise le calcul de la transformation des objets dans le temps, selon 1.1) n'intervient pas : tout se passe comme si c'était le planificateur personnifié qui exécutait lui-même les actions sur lesquelles il raisonne. La seule façon de représenter un agent dans le formalisme actuel est de l'intégrer en tant que prémisse de l'action qu'il exécute[7]. Cette distinction entre représentation d'une action et agent exécutant qui la réalise effectivement est l'analogue en planification de distinctions bien connues : données/programme, S-exp/toplevel en lisp, ...

[7] SIPE pourrait faire un peu mieux : un agent serait une *ressource* (dans le sens particulier de Wilkins) de l'action qu'il exécute. Dans les logiciels de PERT, un agent se représente également comme une ressource de l'action, mais une *ressource* se réduit uniquement à un nombre. Un agent est, de toutes manières, un *attribut* de l'action.

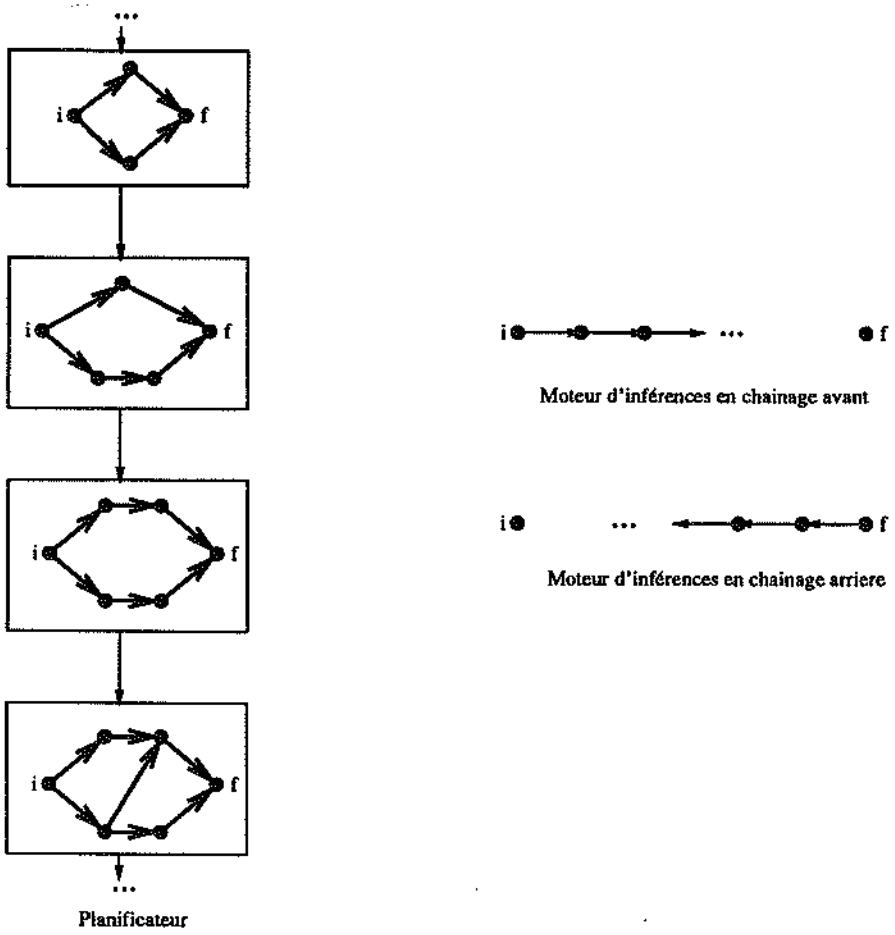


Figure 6.10: Comparaison des graphes des états d'un planificateur et d'un moteur d'inférences

Empiler plusieurs niveaux emboîtés d'action / agent semble séduisant : le comportement d'un agent serait le résultat de l'action de micro-agents le composant (voir figure 6.11), ... etc ...

Pour entrer dans ce cycle (le niveau 0 de l'empilement), nous considérons d'abord plusieurs agents, réalisant chacun leur action propre (une *société d'agents*).

Si l'on considère non plus le comportement propre de chacun de ces agents mais l'évolution globale dans le temps des objets qu'ils manipulent, émerge a *posteriori* un comportement cohérent de cette société (*Gestalt*, cf. 2.1, ou [Minsky 88]) : l'agrégation des effets de bords, représentés par l'influence des agents sur leur environnement commun, peut s'interpréter comme un plan d'actions (l'exemple des figures 6.5, 6.6, 6.7, 6.8 est résolu dans le langage d'acteurs PALADIN [Ferber 89][8]).

Mais cette vision n'a de sens que pour un observateur situé *hors de cette société*, sinon cet observateur serait un agent de plus contribuant à la construction (inconsciente, à son niveau) du

[8]Ferber, à la recherche de la réflexivité, a déjà indiqué le rapport entre méta-planification et langage d'acteurs : *"Les approches les plus voisines [de la mienne, qui consiste à introduire la réflexivité dans les langages d'acteurs], bien que situées dans un cadre plus restreint d'application, sont les systèmes de planification construits autour de la notion de méta-planification, c'est-à-dire des systèmes qui peuvent planifier leur raisonnement [Stefik 81], [Hayes-Roth 85]."*, [Ferber 89, p. 212].

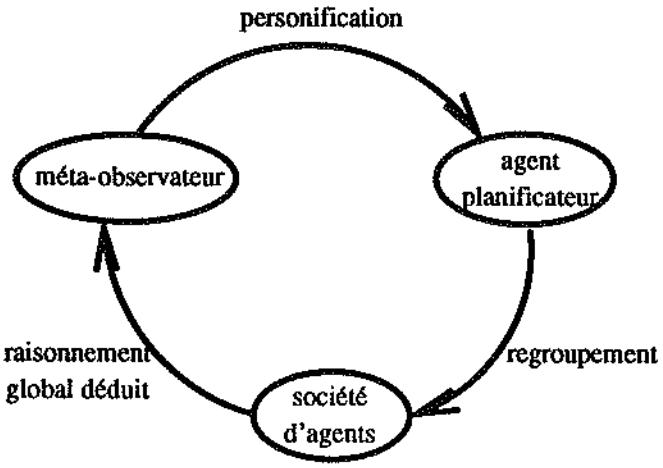


Figure 6.11: La réflexivité de la planification

plan (le *méta-observateur*). Dans son monde, l'observateur ne considère que des plans, observés à partir des effets de bord issus d'exécution d'actions par des agents (déduits a posteriori) ou construits par ses propres moyens (induits a priori). Notre planificateur-programme est un exemple de cet observateur, qui déduit en simulant une exécution et qui induit en modifiant la façon dont cette exécution se déroulera.

Si l'on dissocie les actions exécutées du planificateur déduisant et induisant des informations, ce planificateur peut se modéliser à son tour par un agent (l'*agent planificateur*), qui exécute son action personnelle : créer un plan résolvant un problème de planification.

Nous avons vu qu'il existait plusieurs critères déterminant la vérité d'un terme dans une situation ; on peut représenter ces approches planificatrices particulières en distingant plusieurs agents planificateurs, chacun ayant une compétence donnée (un critère donné). Ce regroupement d'agents planificateurs définit alors une nouvelle société d'agents d'ordre 2 (considérer le cycle de la figure 6.11 comme une spirale vue du dessus).

Cette présentation suggère de continuer l'analyse : l'agrégation des effets de bords des actions des agents planificateurs est observée comme un plan d'ordre 2, et ainsi de suite ...Il devient malheureusement difficile de trouver une signification pratique à ces concepts après deux ou trois tours.

Réflexivité En appelant p la fonction de *planification* de l'ensemble des agents dans lui-même (ou de l'ensemble des parties de l'ensemble des actions dans lui-même, le comportement d'un agent étant une succession d'actions), qui indique pour chaque agent l'agent qui planifie son comportement, la relation entre les niveaux précédents a_i, successivement agent du niveau n et action organisée par le niveau $n+1$, peut s'exprimer par $a_{i+1} = p(a_{i+1})$. L'exécution des actions des agents a_i composant le niveau i est planifiée par les agents a_{i+1} du niveau $n+1$, dont le comportement résulte de l'exécution de ses actions propres, qui sont à leur tour planifiées par ... etc ...

Chaque niveau s'exprime souvent dans son formalisme propre (1 niveau de données structurées, contrôlé par un niveau d'algorithme), le maximum d'uniformité de formalisme ayant été

réalisé dans MOLGEN (3 niveaux, le dernier étant exécuté par un interprète, cf. 3.3.2). La chaîne des niveaux a_i, même sous formalisme unique, s'arrête toujours sur un niveau hors formalisme, qui est en fait un interprète.

Une façon classique de représenter et générer cette chaîne passe par la recherche d'un *point fixe* a, un niveau qui vérifie l'équation $a = p(a)$. Une solution non triviale de cette équation serait le *planificateur idéal*, un méta-planificateur *réflexif*[9], qui saurait, dans un même formalisme, décrire et planifier son propre fonctionnement. Il serait l'interprète à partir duquel on pourrait générer autant de couches planificatrices que désirées.

Les avantages d'un tel planificateur idéal seraient les suivants :

- résolution du boot d'un planificateur : l'amorce d'un langage quelconque est toujours délicate, et particulièrement lourde dans le cas d'un planificateur ;
- preuve d'applicabilité : planifier son propre fonctionnement démontre son efficacité. On pourrait générer autant de niveaux de méta-planification que désirés ;
- lisibilité : une structuration claire facilite le codage d'applications.

Les schémas d'actions du niveau 1, décrivant notre planificateur (ou plutôt son critère de vérité), seraient :

- pose d'une contrainte de précédence ;
- pose d'une contrainte d'unification ;
- pose d'une nouvelle action (du niveau 0).

Ces trois méta-schémas d'actions s'appliquent à une précondition (du niveau 0) qui n'est pas satisfaite : il faut donc coder le critère en précondition de ces méta-schémas d'actions ! Ensuite, pour la pose d'une contrainte de précédence, les post-conditions ne se réduisent pas au simple fait de donner la valeur vraie à un terme du type *précédence*$(action_1, action_2)$: il faudrait aussi respecter la *contrainte de minimalité* de 5.1.3, qui imposerait de coder l'algorithme NFT-EFFACE dans notre formalisme ! De même, pour le deuxième méta-schéma d'actions, il faudrait coder en post-condition l'algorithme FORCE-UNIF de 5.2.3 dans notre formalisme !

Notre formalisme est évidemment incapable de faire tout cela, aussi :

> YAPS n'est pas réflexif.

[9] Ce bouclage de la chaîne des méta par une solution d'une équation de point fixe (qui est à peine esquissé ici) n'est pas nouveau et a donné lieu à d'importantes clarifications sur : la notion de méta-classes des Langages Orientés Objets (réflexivité du lien d'instanciation [Cointe 87]), la notion d'attribut et facette dans les LOOs (*les facettes sont les attributs d'eux-mêmes* [Ferber 89]), la notion de récursivité dans les lambda-langages (problème des *FUNctionnal ARGuments* [Perrot 86]), la notion de toplevel en lisp ("méta-circularité" du toplevel de `lelisp` [Chailloux et coll. 86]), ...

Ce pessimisme n'est pas particulier à notre planificateur [Chapman 85] :

- Si on s'en contente en codant platement les actions du niveaux 1, alors les effets de bords imposés par ces algorithmes rendent caduques notre variante de l'hypothèse STRIPS (Hypothèse 2 de 5.3) : une situation peut changer entre deux actions et le critère n'est plus valable. La solution locale au *problème du cadre* adoptée jusqu'à présent ne tient plus ;
- Si on augmente le formalisme, alors notre critère de vérité n'est plus valable et il faut en trouver un plus complexe, les actions se complexifient, les méta-actions aussi et la mise en évidence de la réflexivité est reculée d'autant ...

Mais, si on interprète pratiquement la NP-complétude d'un problème comme la preuve que ce problème est *intéressant*, nous pensons que l'affinage successif des principes de planification non-linéaire conduit (ou conduira) vers un modèle de méta-planification réflexive.

Chapitre 7

Exemples et perspectives

Notre planificateur dispose maintenant à la fois d'outils d'observation du plan et d'une stratégie suggérant à chaque pas une "bonne" modification du plan pour résoudre le problème de planification posé.

Nous donnons d'abord un exemple de formalisation de l'archétype du domaine-gadget de démonstration en planification. Ceci nous permettra de fournir quelques exemples de résolution effectuée par notre planificateur.

Enfin, étant donné l'ouverture pratique de notre planificateur, nous suggérerons quelques modifications complémentaires de fond.

7.1 Le monde des blocs

Historiquement, les recherches sur la planification ou génération de plans dérivent d'applications visant à automatiser, si possible intelligemment, des travaux d'ouvriers dans une chaîne de montage. Des questions simples comme *"où poser ce que l'on tient ?"* ou *"comment attraper cette pièce sous cette pile d'objets sans tout répandre n'importe où dans l'atelier ?"* arrivent naturellement. Dans un premier temps, la forme réelle des objets manipulés peut être ignorée : ils sont modélisés sous forme de volumes géométriques simples (cônes, sphères, pyramides, parallélépipèdes, ...) et particulièrement sous formes de *blocs* (de différentes tailles ou de taille unique, avec ou sans encoche, multi- ou mono-colores, ...).

C'est le célèbre *monde des blocs* ou *monde des cubes*, monde-jouet utilisé par quasiment tous les auteurs s'intéressant à la planification pour démontrer la capacité de leur planificateur à éviter la combinatoire d'un problème[1]. Nous formalisons ici ce monde une bonne fois pour toutes. Un *problème de blocs* est un problème de planification posé dans ce monde-jouet[2].

[1] Combinatoire contre laquelle ne luttait pas le programme SHDRLU de Winograd (cf. [Winograd 72]) (centré sur le domaine du langage naturel et utilisant pour la première fois un monde très proche de celui des cubes).

[2] Beaucoup de problèmes de cubes sont dus à un certain Allan Brown (rapporté dans [Waldinger 75] et [Chapman 85]). Comme dit Waldinger, *"Beaucoup d'enfants y avaient sans doute pensé avant, mais n'avaient pas réalisé que le problème était difficile"* ....

7.1.1 Formalisation

Remarquons que tout problème de blocs admet la solution évidente : poser *tous* les blocs sur le sol, et construire l'état demandé ex *nihilo*. Cette solution brutale, ne satisfaisant bien sûr pas certains critères légitimes d'optimalité (nombre minimal d'actions, nombre minimal de mouvements, ...) prouve l'existence d'une solution et fournit une borne supérieure du nombre de mouvements (deux fois le nombre de blocs présents), ce qui constitue un bon test de bouclage d'un système de planification.

Axiomes Formellement, on pose le symbole fonctionnel binaire *sur* d'une signature Σ quelconque (les constantes représentant les blocs) et on interprète $sur(x, y)$ par *"le bloc représenté par la variable x se trouve sur le bloc représenté par la variable y"*. On définit une sémantique par les axiomes suivants :

$$\begin{aligned}
\forall x : \quad & libre(x) && \Leftrightarrow \quad \forall y \neq x, \neg\, sur(y, x) && (1)\\
\forall x, y, z : \quad & \left.\begin{array}{l} sur(y, x) \\ sur(z, x) \end{array}\right\} && \Rightarrow \quad y = z && (2)\\
\forall x, y, z : \quad & \left.\begin{array}{l} sur(x, y) \\ sur(x, z) \end{array}\right\} && \Rightarrow \quad y = z && (3)\\
\forall x : \quad & \neg\, sur(x, x) && && (4)
\end{aligned}$$

L'axiome (1) définit la relation unaire *libre* : un bloc est dit *libre* si aucun bloc ne se trouve sur lui. L'axiome (2) indique qu'un bloc ne peut supporter qu'un seul autre bloc ; elle permet également définir la fonction *est-sous*, de l'ensemble des blocs T_Σ (les termes fermés, ou constantes, de la Σ-algèbre) dans $T_\Sigma \cup \bot$ (le bloc inexistant), qui, pour tout bloc, donne l'unique bloc qui se trouve sur lui. L'axiome (3) indique qu'un bloc ne peut se trouver que sur un seul autre bloc (pas de bloc à cheval sur deux blocs), ce qui, allié à l'axiome (2), fixe une taille unique pour tous les blocs ; comme précédemment, on peut définir alors la fonction *est-sous*, de T_Σ dans $T_\Sigma \cup \bot$, qui, pour tout bloc, donne l'unique bloc qui se trouve sous lui. L'axiome (4) indique qu'un bloc ne peut pas se trouver sur lui-même ; dans la suite, on supposera par défaut que des variables de noms différents sont implicitement reliées par une contrainte d'inégalité.

Ces axiomes doivent être vrais à tout instant de l'exécution (en appelant *instant i* l'ensemble des couples variable-valeur du programme après application des i premières (futures) transformations) : en toute rigueur, la modalité $\Box$, au sens de 2.4 devrait qualifier chaque axiome.

Transformation On cherche à définir des règles de réécriture à partir des deux transformations élémentaires :

$$\left\{\begin{array}{rcl} sur(x, y) & \rightarrow & \neg\, sur(x, y) \\ \neg\, sur(x, y) & \rightarrow & sur(x, y) \end{array}\right.$$

La première transformation devra enlever le bloc x du bloc y ; inversement, la deuxième devra poser le bloc x sur le bloc y.

Notre problème est maintenant de déterminer les prédicats qu'il faut ajouter à ces transformations élémentaires pour qu'elles transforment syntaxiquement un monde du modèle en un

autre monde du modèle (i-e pour que le monde d'avant et le monde d'après vérifient les axiomes (1), (2), (3) et (4) ci-dessus). Comme la seconde transformation est l'inverse de la première, on ne considérera dans la suite que la première.

Déduction 1 *$libre(y)$ est vrai après.*

Preuve On veut prouver $\forall z \neq y, \neg\, sur(z,y)$ (d'après (1)). Soit $z \neq y$ un bloc quelconque. Si $z = x$, la transformation nous donne directement $\neg\, sur(z,y)$. Supposons maintenant $z \neq x$; $sur(z,y)$ était vrai après, il le serait également avant (seul $sur(x,y)$ a changé) et on aurait $z = x$ (d'après l'axiome (2)), ce qui serait en contradiction avec l'hypothèse ; $\neg\, sur(z,y)$ est donc vrai après (par l'absurde). Finalement, $\forall z \neq y, \neg\, sur(z,y)$ après. ■

$libre(y)$, qui était faux avant (d'après l'axiome (2)), a donc changé de valeur dans la transformation, qui doit alors s'écrire ($enleve(x,y)$) :

$$\left.\begin{array}{r} sur(x,y) \\ \neg\, libre(y) \end{array}\right\} \rightarrow \left\{\begin{array}{r} \neg\, sur(x,y) \\ libre(y) \end{array}\right.$$

On note dans la suite une transformation sous la forme d'une fraction :

$$enleve(x,y) = \frac{sur(x,y) \wedge \neg\, libre(y)}{\neg\, sur(x,y) \wedge libre(y)}$$

Inversement, la transformation d'*empilement* $pose(x,y)$ doit s'écrire :

$$pose(x,y) = \frac{\neg\, sur(x,y) \wedge libre(y)}{sur(x,y) \wedge \neg\, libre(y)}$$

Composition de transformations On voudrait maintenant pouvoir définir la transformation $deplace(x,z,y)$ qui serait composée des transformations $enleve(x,z)$ et $pose(x,y)$. On définit donc la composée de deux transformations par la loi de composition interne $\times$: soit t_1, t_2 deux transformations, $\mathcal{N}_1, \mathcal{N}_2$ leur ensemble de prédicats en prémisse, $\mathcal{D}_1, \mathcal{D}_2$ leur ensemble de prédicats en conclusion, soit $\mathcal{S}$ l'ensemble des prédicats p qui se simplifieront, c'est-à-dire tels que $\forall p \in \mathcal{S}, (p \in \mathcal{D}_1 \wedge \neg\, p \in \mathcal{N}_2) \vee (\neg\, p \in \mathcal{D}_1 \wedge p \in \mathcal{N}_2)$. On note $\mathcal{S}_{/\mathcal{X}}$ l'ensemble des prédicats p de $\mathcal{X}$ tel que $p \in \mathcal{S}$ ou $\neg\, p \in \mathcal{S}$. On définit la transformation composée $t_1 \times t_2$, d'ensemble $\mathcal{N}_{t_1 \times t_2}$ de prémisses, d'ensemble $\mathcal{D}_{t_1 \times t_2}$ de conclusions, par :

$$\left\{\begin{array}{l} \mathcal{N}_{t_1 \times t_2} = \mathcal{N}_1 \cup (\mathcal{N}_2 \setminus \mathcal{S}_{/\mathcal{N}_2}) \\ \mathcal{D}_{t_1 \times t_2} = \mathcal{D}_2 \cup (\mathcal{D}_1 \setminus \mathcal{S}_{/\mathcal{D}_1}) \end{array}\right.$$

L'ensemble des prémisses $\mathcal{N}_{t_1 \times t_2}$ de la composée $t_1 \times t_2$ est formé des prémisses de t_1 et des prémisses de t_2 qui n'ont pas été simplifiées par des conclusions de t_1. L'ensemble des conclusions $\mathcal{D}_{t_1 \times t_2}$ de la composée $t_1 \times t_2$ est formé des conclusions de t_2 et des conclusions de t_1 qui n'ont pas été simplifiées par des prémisses de t_2.

Graphiquement (avec un abus de langage sur $\wedge$) :

$$t_1 \times t_2 = \frac{\mathcal{N}_1}{\mathcal{D}_1} \times \frac{\mathcal{N}_2}{\mathcal{D}_2} = \frac{\mathcal{N}_1}{(\mathcal{D}_1 \setminus \mathcal{S}_{/\mathcal{D}_1}) \wedge \mathcal{S}} \times \frac{\mathcal{S} \wedge (\mathcal{N}_2 \setminus \mathcal{S}_{/\mathcal{N}_2})}{\mathcal{D}_2} = \frac{\mathcal{N}_1 \wedge (\mathcal{N}_2 \setminus \mathcal{S}_{/\mathcal{N}_2})}{(\mathcal{D}_1 \setminus \mathcal{S}_{/\mathcal{D}_1}) \wedge \mathcal{D}_2}$$

On peut alors calculer la transformation $deplace(x, z, y)$:

$$\begin{aligned} deplace(x,z,y) &= enleve(x,z) \times pose(x,y) \\ &= \frac{sur(x,z) \wedge \neg\, libre(z)}{\neg\, sur(x,z) \wedge libre(z)} \times \frac{\neg\, sur(x,y) \wedge libre(y)}{sur(x,y) \wedge \neg\, libre(y)} \\ &= \frac{sur(x,z) \wedge \neg\, libre(z) \wedge \neg\, sur(x,y) \wedge libre(y)}{\neg\, sur(x,z) \wedge libre(z) \wedge sur(x,y) \wedge \neg\, libre(y)} \end{aligned}$$

Il n'y a donc pas de simplification directe ($\mathcal{S} = \emptyset$).

Remarquons qu'il n'a pas été fait l'hypothèse $libre(x)$ En déplaçant un bloc, on déplace du même coup la pile de blocs qui se trouve sur lui. Si cette interprétation est plausible dans un monde de blocs supposés légers, elle l'est moins dans un monde réel (cf. les chaînes de montages évoquées en introduction). S'interdire de déplacer une pile revient à pouvoir poser l'hypothèse $libre(x)$ dans la transformation. Dans le formalisme d'une transformation, cette seconde interprétation s'obtient par l'ajout de la tautologie $libre(x) \rightarrow libre(x)$. Moyennant l'acceptation de transformations élémentaires du type $p \rightarrow p$ (en plus de $p \rightarrow \neg\, p$), l'expression finale de la transformation $deplace(x, z, y)$ est la suivante :

$$deplace(x,z,y) = \frac{libre(x) \wedge sur(x,z) \wedge \neg\, libre(z) \wedge \neg\, sur(x,y) \wedge libre(y)}{libre(x) \wedge \neg\, sur(x,z) \wedge libre(z) \wedge sur(x,y) \wedge \neg\, libre(y)}$$

De transformation à action L'approche utilisée pour la construction de la transformation précédente est de lister l'intégralité des prédicats qui changent de valeur (ou même qui ne changent pas de valeur mais qui doivent être vrais) de façon à obtenir une règle de réécriture. L'application de cette transformation suppose que l'on dispose d'une situation où tous les prédicats sont explicitement fournis, y compris ceux qui sont redondants, i-e immédiatement dérivables d'un des axiomes initiaux. Par exemple, $\neg\, libre(z)$ est une prémisse de $deplace(x, z, y)$ qui se dérive directement de $sur(x, z)$ via l'axiome (1).

En pratique, la liste exhaustive des prédicats vrais et faux n'est pas constructible (cf. le *frame problem* en 2.1.3). Les prédicats trivialement faux ou trivialement vrais (i-e obtensibles en une dérivation, par exemple, à partir des axiomes initiaux) ne sont pas listés en partie numérateur ou dénominateur. Dans l'écriture de la transformation $deplace(x, z, y)$ réelle, le numérateur $\neg\, libre(z)$ (dérivable de $sur(x, z)$ et de l'axiome (1)) est omis. Même remarque pour le numérateur $\neg\, sur(x, y)$ (dérivable de $sur(x, z)$ par la convention de nommage des variables). En partie dénominateur, $libre(x)$ peut seul être omis (il était déjà vrai avant). Les autres dénominateurs correspondent à des changements de valeurs de prédicats et ne peuvent pas être omis de l'écriture de la transformation (même s'ils peuvent se dériver les uns des autres : $\neg\, sur(x, z)$ de $libre(z)$, et $\neg\, libre(y)$ de $sur(x, y)$).

Après reformulation, l'écriture de l'action est la suivante :

$$deplace(x,z,y) = \frac{libre(x) \wedge libre(y) \wedge sur(x,z)}{\neg\, sur(x,z) \wedge libre(z) \wedge sur(x,y) \wedge \neg\, libre(y)}$$

7.1.2 Schéma d'Actions

L'action primitive Le schéma d'actions correspondant à la transformation $deplace(x, z, y)$ est le suivant :

```
;;; Déplace le cube ?x du bloc ?z au bloc ?y.

(defactschema (deplace ?x ?z ?y)
   (non (= ?z ?y)) (non (= ?z sol)) (non (= ?y sol)) (non (= ?x ?y))
   (sur?x ?z) (libre ?x) (libre ?y)
  ->
   (sur?x ?y) (non (sur?x ?z)) (non (libre ?y)) (libre ?z))
```

Si ?x est sur ?y et si ?x et ?y sont libres, alors on peut effectivement prendre ce bloc ?x et le mettre sur ?y, ce qui rend ?y non libre et ce qui libère ?z (sur lequel se trouvait initialement ?x) puisque ?x n'est plus dessus.

La deuxième ligne contient des contraintes sur les instanciations possibles des variables. Le planificateur ne pouvant pas appliquer la convention sur les variables (*deux variables de noms différents ont a priori des valeurs différentes*), on lui indique d'abord que ?x, ?y et ?z doivent avoir des valeurs différentes ((non (= ?z ?y)) et (non (= ?x ?y))). Les deux autres contraintes ((non (= ?z sol)) et (non (= ?y sol))) servent à éviter le problème du sol (cf. paragraphe suivant).

On donne une consistance matérielle au planificateur en le représentant sous forme d'un robot manipulateur de type SHRDLU [Winograd 72], c'est-à-dire avec un seul bras (ce qui lui interdit de déplacer plusieurs cubes en même temps et, entre autres d'intervertir deux cubes) de faible puissance (ce qui lui interdit de faire de l'équilibrisme en déplaçant une pile de cubes à la fois — cf. le prédicat *libre*(x) en partie numérateur).

Un bloc très particulier : le sol Que devient un bloc lorsqu'il n'est pas sur un autre bloc ? Introduisons pour cela le **sol** dans notre modélisation, c'est-à-dire un objet sur lequel (resp. duquel) le bras manipulateur peut empiler (resp. dépiler) un bloc et sur lequel plusieurs blocs peuvent tenir (le **sol** est supposé avoir des dimensions infinies, et être toujours libre). Axiomatiquement :

$$\begin{array}{rll} & \mathit{libre}(\mathit{sol}) & (1') \\ \forall x : & \neg\, \mathit{sur}(\mathit{sol}, x) & (3') \\ & \neg\, \mathit{sur}(\mathit{sol}, \mathit{sol}) & (4') \end{array}$$

L'axiome (1'), révision de l'axiome (1) précédent pour le sol, indique que le sol doit toujours être *libre* ; ceci interdit à la variable ?y de s'instancier en ce sol ((non (libre y)) est en conclusion) et justifie la contrainte (non (= ?y sol)) ; ceci interdit également à la variable ?z de s'instancier en ce sol, puisqu'il n'y plus besoin de répéter (libre sol) ((libre ?z) est en conclusion), ce qui justifie la contrainte (non (= ?z sol)). L'axiome (2'), révision de l'axiome (2), indiquerait que plusieurs blocs peuvent se trouver sur le *sol* ; ceci se traduit précisément par une absence d'axiome. L'axiome (3'), révision de l'axiome (3), empêche toute tentative (syntaxiquement concevable) de déplacer le sol sur un bloc. L'axiome (4') est l'axiome (4) pour lequel $x = \mathit{sol}$.

Comme précédemment, ces axiomes doivent être vrais à tout instant de l'exécution : en toute rigueur, la modalité $\Box$, au sens de 2.4 devrait qualifier chaque axiome.

Le schéma d'actions (`deplace ?x ?z ?y`) n'est pas correct vis-à-vis de ces axiomes, aussi les contraintes `?x, ?y, ?z <> sol` sont explicitement contenues dans l'opérateur. Pour intégrer les axiomes précédents, on définit les schémas d'actions `sol>(?x,?y)`, qui prend le bloc `?x` au sol et le pose sur `?y`, et `>sol(?x,?z)`, qui prend le bloc `?x` de `?z` et le pose sur le sol :

```
;;; Met le bloc ?x, initialement au sol, sur le bloc ?y.

(defactschema (sol> ?x ?y)
   (non (= ?y sol)) (non (= ?x ?y))
   (libre ?x) (libre ?y)
  ->
   (sur ?x ?y) (non (libre ?y)))

;;; Pose par terre le bloc ?x, initialement sur ?z.

(defactschema (>sol ?x ?z)
   (non (= ?z sol))
   (sur ?x ?z) (libre ?x)
  ->
   (non (sur ?x ?z)) (libre ?z))
```

Diverses résolutions Aussi simple qu'il puisse paraître, la prise en compte du sol dans un problème de blocs est un bon révélateur du pouvoir expressif d'un planificateur. Nous donnons ici quelques exemples de résolution de ce problème dans des planificateurs connus :

- *Décrire les places licites*

  Cette approche consiste à nommer des endroits licites de cet ailleurs anonyme qu'est le sol ("pas japonais"), i-e ajouter des blocs factices au modèle. L'objet `sol` a disparu, ou plutôt s'est instancié en plusieurs blocs[3]. Cette démarche reporte le problème en croyant l'évacuer, puisque ces cubes factices, n'ayant pas de raison de ne pas être déplacés, augmentent la combinatoire.

- *Définir des actions conditionnelles*

  Wilkins propose dans SIPE (cf. 3.2.1) de coder ce sol sous forme d'un opérateur déductif. Le schéma d'actions est le suivant (version simplifiée de puton, [Wilkins 84]) :

```
(defactschema (sur ?x ?y)
   (non (= ?x ?y))
   (libre ?x) (?libre ?y)
  ->
   (sur ?x ?y))
```

  Il n'y a a priori aucune contrainte sur `?x` et `?y`, et certains effets dits annexes ne sont pas indiqués (`(non (libre ?y))`, par exemple). Ce schéma d'actions est toujours valable,

[3] Même solution que pour les partisans du moteur 0 contre le moteur 1 pour les problèmes ne requérant que des classes avec peu d'instances, où l'on instancie virtuellement des règles d'ordre 1 sur toutes les combinaisons de faits de l'ordre 0.

que `?x` ou `?y` vale `sol` ou non. L'action spécifique du sol, ou tout autre effet de bord d'un schéma d'actions, est prise en compte par des *opérateurs déductifs*, qui réalisent des inférences à l'intérieur d'une situation. En fait, ils implémentent exactement les axiomes (1), (2), (3) et (4).

La déduction manquante `(libre ?z)`, dans le cas où le bloc `?z` (et non le sol) se trouve initialement sous `?x`, se formule de la façon suivante :

```
(defdeducop (libre ?z)
   (sur ?x ?z)
  >
   (sur ?x ?y) (non (= ?y ?z))
  ->
   (libre ?z))
```

Ce qui se traduit par : s'il existe deux situations successives s_1 et s_2 ($s_1 \preceq s_2$) telles que `(sur ?x ?z)` soit vrai dans s_1 et `(sur ?x ?y)` soit vrai dans s_2, et que `(non (= ?y ?z))`, alors `(libre ?z)` est vrai dans s_2. L'autre déduction manquante `((non (libre ?y)))` s'obtient de façon tout-à-fait analogue[4]. Bien que souffrant de problèmes théoriques aigus (cf. chapitre 4), ce formalisme représente une des rares tentatives pour améliorer le pouvoir expressif de ces actions directement héritées de STRIPS, en évitant la lourdeur d'un système déductif complet en logique des prédicats d'ordre 1 travaillant sur les axiomes carcatérisant la sémantique du monde observé.

7.2 Exemples

Bien que YAPS, planificateur pur, ne prétende pas rivaliser dans leur domaine avec les résolveurs de problèmes, nous testons quand même son contrôle sur quelques casse-têtes combinatoires.

7.2.1 Le singe et les bananes

Le *"problème du singe et des bananes"* peut se formuler ainsi : *Dans une pièce se trouvent un singe, une caisse et des bananes accrochées au plafond. Sachant que le singe peut sous certaines conditions marcher, grimper sur la caisse, en descendre, la pousser, attraper un objet et le lâcher, le problème qui lui est posé est d'atteindre les bananes et les attraper* [Vialatte 85, p. 160] (par exemple[5]). La seule difficulté (pour un système informatique) est que, pour que le singe puisse

[4] Le véritable opérateur déductif `(libre ?z)` est un peu plus élaboré car le monde modélisé par Wilkins ne contient pas l'axiome (2) (plusieurs blocs peuvent se trouver sur un bloc).

[5] On trouve une variante inquiétante dans le roman *"La planète des singes"* de P. Boulle (Le Livre de Poche, Paris, 1977, p 73) : *"Au lieu de déposer nos aliments dans nos cages, comme ils le faisaient d'ordinaire, (...) les deux gorilles (...) les hissèrent au plafond dans des paniers, au moyen d'un système de poulie (...). En même temps, ils placèrent quatre cubes en bois, d'asses gros volume, dans chaque cellule. Puis, s'étant reculés, ils nous observèrent.*

C'était pitié de voir la mine déconfite de mes compagnons. Ils essayèrent de sauter, mais aucun ne put atteindre le panier. Certains grimpèrent le long des grilles, mais, parvenus au sommet, ils avaient beau étendre les bras, ils ne pouvaient saisir les aliments qui se trouvaient trop loin des parois. J'étais honteux de la sottise de ces hommes. Moi, est-il besoin de le mentionner, j'avais trouvé du premier coup la solution du problème."

attraper les bananes, il doit *d'abord* placer une caisse sous le régime de bananes pour pouvoir l'atteindre.

Ce problème peut se représenter en YAPS de la façon suivante :

```
;; Le singe place l'objet <?a> sous les bananes.

(defactschema (mets-dessous ?a ?b)
  (non (en-dessous ?a ?b))
  (non (au-niveau-de Tchita ?a))    ;;; Ne pas être dessus ...
  ->
  (en-dessous ?a ?b)
  )

;; Le singe passe de la caisse <?a> (y compris le sol) à la caisse <?b>

(defactschema (monte-sur ?a ?b)
  (non (= ?a ?b))
  (au-niveau-de Tchita ?a)
  ->
  (non (au-niveau-de Tchita ?a))
  (au-niveau-de Tchita ?b)
  )

;; Si le singe, qui est sur <?c>, est à la même hauteur que les bananes,
;; il a gagné (et les mange).

(defactschema (attrape-bananes ?c)
  (au-niveau-de bananes ?c)
  (en-dessous ?c bananes)
  (au-niveau-de Tchita ?c)    ;;; Le singe est à la hauteur des bananes.
  ->
  (possede Tchita bananes)
  )
```

Dans la trace d'exécution suivante (cf. annexe C) sont indiqués les méta-buts successifs (`>> ...`) et leur résolution (`<< ...`). Le premier mot d'un méta-but est `etablir` ou `demasque` ; le premier mote de la résolution d'un méta-but est soit `=` (instanciation de variable), `<>` (contrainte de non-unification) ou `<` (contrainte de précédence). Le préfixe `#A` référence une action dont le titre est la S-expression qui suit : au retour du pas 1, le terme `#A(attrape-bananes ?c3)` référence une action instanciation du 3e schéma d'actions ci-dessus. De même, le préfixe `#S` référence les *schémas* d'actions. Ces référence sont généralement suivies de la prémisse ou de la conclusion étudiée.

Par exemple, la résolution du 3e méta-but indique que, pour satisfaire la prémisse `(en-dessous ?c3 bananes)` de l'action `#A(attrape-bananes ?c3)`, YAPS a instancié le schéma d'actions `#S(mets-dessous ?a ?b)`, dont une conclusion est `(en-dessous ?a ?b)`, en l'action `#A(mets-dessous ?c3 bananes)`.

```
? (control)
```

```
;;;  YAPS (by PhM) version 1.03 (23 Mars 91)
================================================================================
(defplan
 (au-niveau-de tchita sol) (non (au-niveau-de tchita caisse)) (non (en-dessous
caisse bananes)) (au-niveau-de bananes caisse)
 ->
 (possede tchita bananes))

1>> (etablir (#Afinal possede tchita bananes))
1<< (instancier (#Afinal possede tchita bananes) (#S(attrape-bananes ?c)
possede tchita bananes) #A(attrape-bananes ?c3))
  2>> (etablir (#A(attrape-bananes ?c3) au-niveau-de bananes ?c3))
  2<< (= (#A(attrape-bananes ?c3) au-niveau-de bananes ?c3) (#Ainitial
  au-niveau-de bananes caisse))
    3>> (etablir (#A(attrape-bananes ?c3) en-dessous ?c3 bananes))
    3<< (instancier (#A(attrape-bananes ?c3) en-dessous ?c3 bananes) (#S(
    mets-dessous ?a ?b) en-dessous ?a ?b) #A(mets-dessous ?c3 bananes))
      4>> (etablir (#A(attrape-bananes ?c3) au-niveau-de tchita ?c3))
      4<< (instancier (#A(attrape-bananes ?c3) au-niveau-de tchita ?c3) (#S(
      monte-sur ?a ?b) au-niveau-de tchita ?b) #A(monte-sur ?a5 ?c3))
        5>> (etablir (#A(monte-sur ?a5 ?c3) au-niveau-de tchita ?a5))
        5<< (= (#A(monte-sur ?a5 ?c3) au-niveau-de tchita ?a5) (#Ainitial
        au-niveau-de tchita sol))
          6>> (demasque (#A(mets-dessous ?c3 bananes) non (au-niveau-de tchita
          ?c3)))
          6<< (< (#A(mets-dessous ?c3 bananes) non (au-niveau-de tchita ?c3))
          (#A(monte-sur ?a5 ?c3) au-niveau-de tchita ?c3))
            7>> Succes

** Temps CPU  : 2.880005
```

Ces 2.9 secondes CPU[6] sont obtenues sur une machine Sun 3/60 sous Lelisp15.22 en mode *interprété* (ce qui explique ce temps élevé).

Le problème est suffisamment simple pour que les heuristiques de YAPS suffisent (aucun retour-arrière n'est nécessaire). Les actions sont d'abord instanciées à la façon d'un chaînage arière d'un moteur d'inférences-exécuteur. Mais, ici, à des sous-buts de même niveau correspondent des actions de même niveau, i-e non ordonnées.

Le singe ne s'aperçoit qu'au pas 6 (heuristique 5 de 6.2.1) qu'il ne peut déplacer une caisse tout en étant dessus (voir figure 7.1).

Le critère de vérité détecte un masquage possible de la prémisse
`(non (au-niveau-de tchita ?c3))` (la variable `?c3` est lié à la constante `caisse`) de l'action `#A(mets-dessous ?c3 bananes)` par la conclusion `(au-niveau-de tchita ?c3)` de l'action `#A(monte-sur ?a5 ?c3)` (la variable `?a5` est liée à la constante `sol`). La sous-heuristique 4 suggère de repousser le masqueur après le demandeur (promotion de 6.1.2, "demasque" à la résolution 6), ce qui incite le singe à déplacer effectivement la caisse avant de monter dessus.

[6] Mesure avec la fonction `time` de Lelisp.

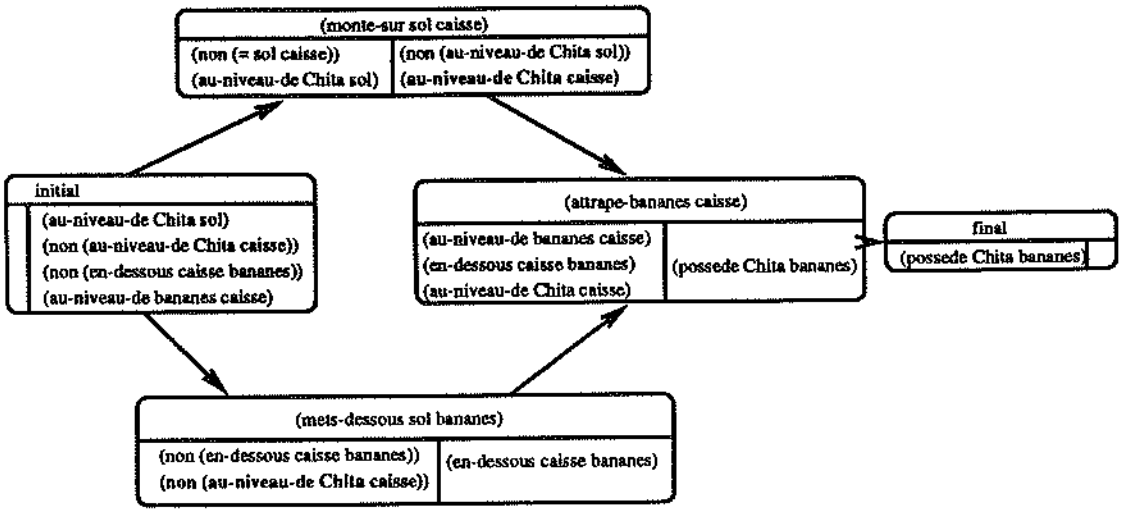


Figure 7.1: Le singe et les bananes (contrainte de précédence)

Les 3 actions[7] du graphe-solution (linéaire et intuitivement évident) sont ensuite affichées par YAPS de la façon suivante :

```
> Actions (5)

 ** Nom : (monte-sur sol [?a5] caisse [?c3])  **
 Pre  : (au-niveau-de tchita sol [?a5])
 Post : (non (au-niveau-de tchita sol [?a5])) (au-niveau-de tchita
caisse [?c3])
 -->  : #A(attrape-bananes ?c3)

 ** Nom : (mets-dessous caisse [?c3] bananes)  **
 Pre  : (non (en-dessous caisse [?c3] bananes)) (non (au-niveau-de tchita
caisse [?c3]))
 Post : (en-dessous caisse [?c3] bananes)
 -->  : #A(monte-sur ?a5 ?c3)

 ** Nom : (attrape-bananes caisse [?c3])  **
 Pre  : (au-niveau-de bananes caisse [?c3]) (en-dessous caisse [?c3] bananes)
(au-niveau-de tchita caisse [?c3])
 Post : (possede tchita bananes)
 -->  : #Afinal

 ** Nom : final  **
 Pre  : (possede tchita bananes)
 Post :
 -->  :

 ** Nom : initial  **
 Pre  :
 Post : (au-niveau-de tchita sol) (non (au-niveau-de tchita caisse)) (non (
en-dessous caisse bananes)) (au-niveau-de bananes caisse)
 -->  : #A(mets-dessous ?c3 bananes)
```

[7] YAPS en trouve 5 car il y inclut les pseudo-actions initiale et finale.

YAPS affiche les valeurs des variables, lorsque c'est possible ; après chaque valeur figure la variable source entre crochets (dans le titre de la première action, sol est la valeur de la variable ?a5, par exemple).

La base d'unification précise l'état de ces variables (voir la syntaxe d'affichage en 5.2.3 :

```
> Base de contraintes d'unification
> ?a5 -> sol
   <> ?c3
> ?c3 -> caisse
   <> ?a ?a5
```

7.2.2 Anomalie de Sussman

Nous utilisons les 3 schémas d'actions construits en 7.1 sur l'exemple de l'*anomalie de Sussman* (HACKER [Sussman 75]).

Le problème est le suivant :

```
(defplan
    (libre c) (sur c a) (libre b)
   ->
    (sur a b) (sur b c))
```

Résolution directe Le planificateur est lancé sous le toplevel `lelisp` :

```
? (control)
;;;  YAPS (by PhM) version 1.03 (23 Mars 91)
==========================================================================
(defplan
 (libre c) (sur c a) (libre b)
 ->
 (sur a b) (sur b c))
1>> (etablir (#Afinal sur a b))
1<< (instancier (#Afinal sur a b) (#S(sol> ?x ?y) sur ?x ?y) #A(sol> a b))
  2>> (etablir (#Afinal sur b c))
  2<< (instancier (#Afinal sur b c) (#S(sol> ?x ?y) sur ?x ?y) #A(sol> b c))
    3>> (demasque (#A(sol> b c) libre b))
    3<< (< (#A(sol> b c) libre b) (#A(sol> a b) non (libre b)))
      4>> (etablir (#A(sol> a b) libre a))
      4<< (instancier (#A(sol> a b) libre a) (#S(>sol ?x) libre ?z) #A(>sol
      ?x7))
```

```
5>> (etablir (#A(>sol ?x7) sur ?x7 a))
5<< (= (#A(>sol ?x7) sur ?x7 a) (#Ainitial sur c a))
  6>> (demasque (#A(>sol ?x7) libre ?x7))
  6<< (< (#A(>sol ?x7) libre ?x7) (#A(sol> b c) non (libre c)))
    7>> Succes

** Temps CPU  : 2.6
```

Ce problème est résolu en 2.6 secondes CPU (Sun 3/60, Lelisp interprété, mesure par `time`). Détaillons cet exemple :

Pas 1 : sur les deux prémisses `(sur a b)` et `(sur b c)` de la pseudo-action finale `#Afinale`, aucune des heuristiques et sous-heuristiques de 6.2 ne se déclenche, le planificateur décide d'établir la première qui lui est fournie `(sur a b)`.

Les deux schémas d'actions `#S(sol> ?x ?y)` et `#S(deplace ?x ?z ?y)` peuvent conclure tous les deux sur `(sur ?x ?y)`, qui peut s'unifier à `(sur a b)` ; le schéma d'actions `#S(>sol ?x)` n'a pas de conclusion unifiable à `(sur a b)`, aussi `#S(>sol ?x)` n'est pas un schéma d'actions candidat.

Aucune des heuristiques de 6.2 n'a de raison de se déclencher, aussi c'est le premier schéma d'actions candidat qui est choisi : les termes `(sur ?x ?y)` et `(sur a b)` sont contraints à s'unifier, aussi le schéma d'actions `#S(sol> ?x ?y)` est instancié en l'action `#A(sol> a b)`.

Pas 2 : l'ajout de `#A(sol> a b)` a généré de nouveaux sous-buts non satisfaits, mais l'heuristique 2 de 6.2 stipule que la pseudo-action finale `#Afinale` est prioritaire : le planificateur décide d'établir la deuxième prémisse `(sur b c)` de `#Afinale`.

Comme précédemment, les schémas d'actions `#S(sol> ?x ?y)` et `#S(deplace ?x ?z ?y)` sont candidats. Comme précédemment, `#S(sol> ?x ?y)` est choisi et est instancié en l'action `#A(sol> b c)` (avec unification nécessaires des variables `?x` et `?y` renommées).

Pas 3 : deux prémisses ne sont pas satisfaites : `(libre a)` de `#A(sol> a b)` n'est pas établie et `(libre b)` de `#A(sol> b c)` est établie mais est nécessairement masquée par la conclusion `(non (libre b))` de `#A(sol> a b)` (voir figure 7.2).

L'heuristique 5 de 6.2 suggère de résoudre un démasquage avant un établissement : le planificateur décide de démasquer d'abord `(libre b)`.

La sous-heuristique 4 de 6.2 suggère d'essayer de repousser les masqueurs et on se trouve ici dans le cas 3 de la table 6.2 de 6.2 : la contrainte de précédence "`#A(sol> b c)` $\prec$ `#A(sol> a b)`" repousse la conclusion masqueuse `(non (libre b))` après la prémisse `(libre b)`. Cette conclusion `(non (libre b))` n'est donc plus un masqueur de `(libre b)`.

Pas 4 : il reste la deuxième prémisse issue du pas 2, `(libre a)` de `#A(sol> a b)`, qui n'est toujours pas établie. Les deux schémas d'actions `#S(>sol ?x ?z)` et `#S(deplace ?x ?z ?y)` sont candidats, et le planificateur choisit pour l'instant d'instancier le premier, `#S(>sol ?x)`, en l'action `#A(>sol ?x1 a)`.

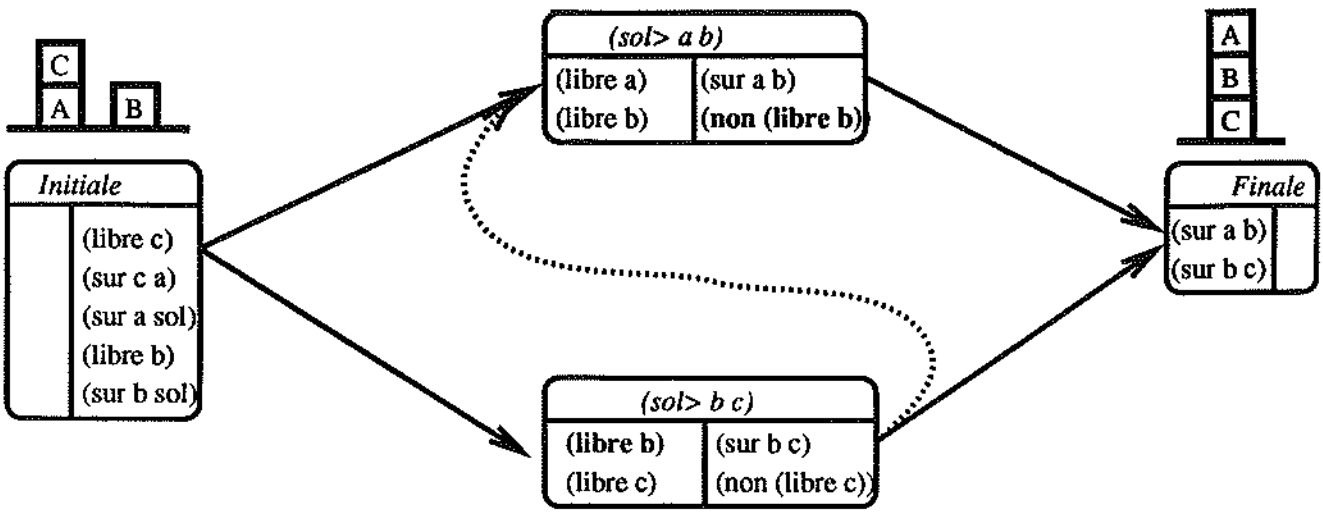


Figure 7.2: Contrainte de précédence (pas 3, anomalie de Sussman)

Pas 5 : les deux prémisses `(sur ?x1 a)` et `(libre ?x1)` de l'action `#A(>sol ?x1 a)` sont potentiellent établies par la pseudo-action initiale et sont potentiellement masquées par `#A(sol> b c)`. Sans l'heuristique 4, il faudrait trier ces établissements et masquages potentiels, alors que cette heuristique suggère de considérer la première prémisse concernant la nouvelle variable libre `?x1` `(sur ?x1 a)` et d'essayer d'établir `(sur ?x1 a)` sans se préoccuper des multiples masqueurs potentiels.

La réponse est alors évidente : l'établisseur potentiel `(sur c a)` de `#Ainitiale` devient établisseur nécessaire en contraignant $\Box(x1 \approx c)$.

Pas 6 : il ne reste plus qu'une seule prémisse non satisfaite, `(libre c)` de l'action `#A(>sol c a)`, qui se résout en repoussant le masqueur (cf. figure 7.3) par une contrainte de précédence comme pour le pas 3.

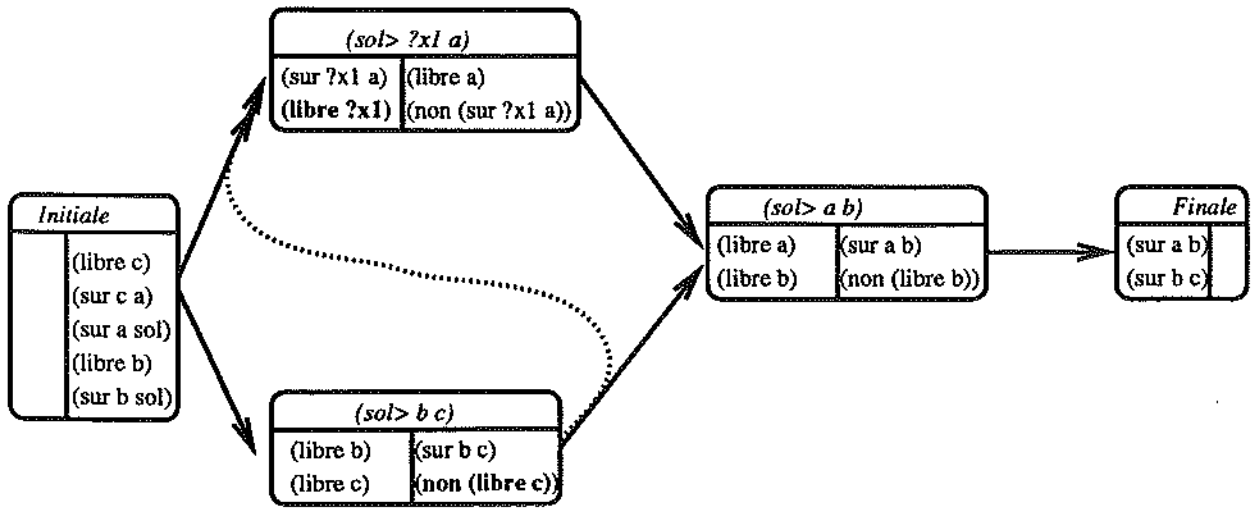


Figure 7.3: Autre contrainte de précédence (pas 6, anomalie de Sussman)

Pas 7 : le planificateur ne trouve plus aucune prémisse non satisfaite : il a donc trouvé une

solution (d'après PLANIFIER1). Les actions du graphe courant sont affichées succinctement : `#A(>sol ?x1 a)` (la valeur de `?x1` est donnée entre crochets), `#A(>sol b c)`, `#A(sol> a b)` et les deux pseudo-actions initiales et finales.

```
> Actions (5)

 ** Nom : (>sol c [?x7])  **
 Pre  : (sur c [?x7] a) (libre c [?x7])
 Post : (non (sur c [?x7] a)) (libre a)
 -->  : #A(sol> b c)

 ** Nom : (sol> b c)  **
 Pre  : (libre b) (libre c)
 Post : (sur b c) (non (libre c))
 -->  : #A(sol> a b)

 ** Nom : (sol> a b)  **
 Pre  : (libre a) (libre b)
 Post : (sur a b) (non (libre b))
 -->  : #Afinal

 ** Nom : final  **
 Pre  : (sur a b) (sur b c)
 Post :
 -->  :

 ** Nom : initial  **
 Pre  :
 Post : (libre c) (sur c a) (sur a sol) (libre b) (sur b sol)
 -->  : #A(>sol ?x7)
```

La base de contraintes d'unification affiche l'entrée de la seule variable présente, comme en 5.2.3.

```
> Base de contraintes d'unification
> ?x7 -> c
```

Autres solutions Les autres solutions sont recherchées en relançant le planificateur sur les points de choix laissés en suspens :

```
? (recontrol)
          << DETRUIRE (< (#A(>sol ?x7) libre ?x7) (#A(sol> b c) non (libre c
          )))
         [>>] (demasque (#A(>sol ?x7) libre ?x7))
         ...
 62[>>] (etablir (#Afinal sur b c))
 62<< (instancier (#Afinal sur b c) (#S(deplace ?x ?z ?y) sur ?x ?y) #A(
```

```
  deplace b ?z111 c))
    63>> (etablir (#A(deplace b ?z111 c) sur b ?z111))
    63<< (instancier (#A(deplace b ?z111 c) sur b ?z111) (#S(sol> ?x ?y) sur
    ?x ?y) #A(sol> b ?z111))
      64>> (etablir (#A(sol> b ?z111) libre ?z111))
      64<< (instancier (#A(sol> b ?z111) libre ?z111) (#S(>sol ?x) libre ?z)
      #A(>sol ?x116))
        65>> (etablir (#A(>sol ?x116) sur ?x116 ?z111))
        65<< (= (#A(>sol ?x116) sur ?x116 ?z111) (#Ainitial sur c a))
          66>> (demasque (#A(deplace b ?z111 c) libre b))
          66<< (< (#A(deplace b ?z111 c) libre b) (#A(sol> a b) non (libre b))
          )
            67>> Succes

** Temps CPU  : 69.12
```

Le planificateur détruit le dernier choix qu'il a effectué (la contrainte de précédence entre `#A(>sol ?x7 a)` et `#A(sol> b c)`), rappelle le problème qu'il avait alors sélectionné et suggère une autre solution. Toutes les autres branches de recherche sont alors explorées comme précédemment jusqu'au pas 62 (ce que nous n'avons pas jugé utile d'infliger au lecteur). Au pas 62, le planificateur prouve que, en posant b sur c par le schéma d'actions `sol>`, dans le diamètre auorisé, la seule solution était la première, aussi il tente le second schéma possible qui implique que b ne vienne pas du sol. Une autre solution est alors trouvée en 4 pas, moyennant l'instanciation d'une nouvelle action `#A(sol> b ?z111)` posant b sur a.

```
> Actions (6)

 ** Nom : (>sol c [?x116]) (1) **
 Pre  : (sur c [?x116] a [?z111]) (libre c [?x116])
 Post : (non (sur c [?x116] a [?z111])) (libre a [?z111])
 -->  : #A(sol> b ?z111)

 ** Nom : (sol> b a [?z111]) (1) **
 Pre  : (libre b) (libre a [?z111])
 Post : (sur b a [?z111]) (non (libre a [?z111]))
 -->  : #A(deplace b ?z111 c)

 ** Nom : (deplace b a [?z111] c)  **
 Pre  : (sur b a [?z111]) (libre b) (libre c)
 Post : (sur b c) (non (sur b a [?z111])) (non (libre c)) (libre a [?z111])
 -->  : #A(sol> a b)

 ** Nom : (sol> a b)  **
 Pre  : (libre a) (libre b)
 Post : (sur a b) (non (libre b))
 -->  : #Afinal

 ** Nom : final  **
 Pre  : (sur a b) (sur b c)
 Post :
 -->  :
```

```
** Nom : initial  **
Pre  :
Post : (libre c) (sur c a) (sur a sol) (libre b) (sur b sol)
-->  : #A(>sol ?x116)

> Base de contraintes d'unification
> ?x116 -> c
> ?z111 -> a
```

Il ne faudrait certainement pas croire que la résolution se passe toujours aussi bien que pour l'anomalie de Sussman ou pour des variations "évidentes" (pour un humain) : si la solution la plus naturelle (i-e celle contenant le moins d'actions) ne peut pas échapper au planificateur (étant donné le parcours effectué), elle n'est pas toujours trouvée sans retour-arrière, même dans ce monde-gadget.

Les heuristiques présentées ne font pas appel à la sémantique du domaine d'application (aucune heuristique n'indique que les piles de blocs sont de type LIFO ...) et fonctionnent pourtant correctement sur ce monde-gadget. Mais on peut quand même construire des exemples les mettant en défaut, le planificateur s'embarquant dans une branche qu'une heuristique "**indépendante** du domaine" pourrait difficilement éviter.

De même, des techniques de maintenance de la vérité, bien que nettement plus lourdes, pallieraient la faiblesse du retour-arrière chronologique, qui doit, par exemple, consommer 62 pas pour prouver qu'il n'existe qu'une solution sur la branche dans laquelle le méta-but `(etablir (#Afinal sur b c))` est réalisé par l'instanciation de `(#S(sol> ?x ?y) sur ?x ?y)`.

7.2.3 Construction d'une maison

L'exemple que nous présentons maintenant concerne un domaine particulier, la construction d'une maison (tiré de [Tate 77]). Par rapport à l'original, seule la tâche *"mise en place de la climatisation"* a été omise, et nous avons rajouté des étages.

Schémas d'actions Les schémas d'actions définissant les tâches du gros-œuvre sont les suivants (le nombre, figurant après le titre, représente la durée de réalisation du schéma d'actions, cf. la syntaxe en annexe C) :

```
;; — Gros œuvre

(defactschema (faire gros-oeuvre) ;;; Principale
  (allees damees) -> (gros-oeuvre termine))

(defactschema (faire excavations) 4
  (terrain achete) -> (excavation finie))

(defactschema (couler fondations-beton) 2
```

```
  (excavation finie) -> (plancher 0 termine))

(defactschema (eriger-etage ?dessous ?dessus)
  (au-dessus ?dessus ?dessous)
  (plancher ?dessous termine)
  ->
  (plancher ?dessus termine)
  (murs ?dessus eriges))

 (defactschema (poser briques)
  (murs 4 eriges)
  ->
  (briques posees))

(defactschema (couvrir toit) 2
  (briques posees) -> (toit couvert))

(defactschema (fixer gouttieres) 1
  (toit couvert) -> (gouttieres fixees))

(defactschema (poser escalier) 2
  (gouttieres fixees) (conduites-verticales installees) -> (escalier pose))

(defactschema (damer allees-jardin) 5
  (escalier pose) -> (allees damees))
```

Les schémas d'actions définissant les tâches des corps de métiers sont les suivants :

```
;; — Les corps de métiers

(defactschema (corps metier) ;;; Principal
  (conduites-verticales installees)
  (plomberie installee)
  (cuisine installee)
  (travaux-electriques finis)
  ->
  (corps-metier termines))

(defactschema (installer canalisations) 1
  (plancher 0 termine) -> (canalisations installees))

(defactschema (installer conduites-verticales) 1
  (plancher 0 termine) -> (conduites-verticales installees))

(defactschema (poser plomberie-principale) 3
  (canalisations installees) -> (plomberie-principale installee))

(defactschema (finir plomberie) 2
  (plomberie-principale installee)
  (carrelage-parquet poses)
  ->
  (plomberie installee))
```

```
(defactschema (cabler edf) 2
  (terrain achete) -> (cablage-edf installe))

(defactschema (passer fils-electriques) 1
  (cablage-edf installe) -> (travaux-electriques finis))

(defactschema (installer cuisine) 1
  (plomberie-principale installee)
  (cablage-edf installe)
  (carrelage-parquet poses)
  ->
  (cuisine installee))
```

Les schémas d'actions définissant les tâches d'aménagement intérieur sont les suivants :

```
;; - décoration

(defactschema (decorations-interieures) ;;; Principale
  (sol sable-poli) -> (decorations-interieures))

(defactschema (faire platres) 10
  (plomberie-principale installee) (cablage-edf installe)
  ->
  (platres essuyes))

(defactschema (poser sol-RdC) 2
  (canalisations installees) -> (sol-RdC pose))

(defactschema (poser carrelage-parquet) 3
  (sol-RdC pose) (platres essuyes) -> (carrelage-parquet poses))

(defactschema (finir charpente) 3
  (carrelage-parquet poses) -> (charpente finie))

(defactschema (sabler-polir sol) 2
  (charpente finie) (peintures terminees) -> (sol sable-poli))

(defactschema (peindre) 3
  (plomberie installee) (cuisine installee) -> (peintures terminees))
```

Résolution directe Comme précédemment, la trace brute issue de notre planificateur est fournie[8] :

```
? (setq #:plan:step:margin-step 0)
= 0
? (control)
```

[8] La première évaluation annule l'indentation.

```
1>> (etablir (#Afinal gros-oeuvre termine))
1<< (instancier (#Afinal gros-oeuvre termine) (#S(faire gros-oeuvre)
gros-oeuvre termine) #A(faire gros-oeuvre))
2>> (etablir (#Afinal corps-metier termines))
2<< (instancier (#Afinal corps-metier termines) (#S(corps metier) corps-metier
termines) #A(corps metier))
3>> (etablir (#Afinal decorations-interieures))
3<< (instancier (#Afinal decorations-interieures) (#S(decorations-interieures)
decorations-interieures) #A(decorations-interieures))
4>> (etablir (#A(decorations-interieures) sol sable-poli))
4<< (instancier (#A(decorations-interieures) sol sable-poli) (#S(sabler-polir
sol) sol sable-poli) #A(sabler-polir sol))
5>> (etablir (#A(corps metier) conduites-verticales installees))
5<< (instancier (#A(corps metier) conduites-verticales installees) (#S(
installer conduites-verticales) conduites-verticales installees) #A(installer
conduites-verticales))
6>> (etablir (#A(corps metier) plomberie installee))
6<< (instancier (#A(corps metier) plomberie installee) (#S(finir plomberie)
plomberie installee) #A(finir plomberie))
7>> (etablir (#A(corps metier) cuisine installee))
7<< (instancier (#A(corps metier) cuisine installee) (#S(installer cuisine)
cuisine installee) #A(installer cuisine))
8>> (etablir (#A(corps metier) travaux-electriques finis))
8<< (instancier (#A(corps metier) travaux-electriques finis) (#S(passer
fils-electriques) travaux-electriques finis) #A(passer fils-electriques))
9>> (etablir (#A(faire gros-oeuvre) allees damees))
9<< (instancier (#A(faire gros-oeuvre) allees damees) (#S(damer allees-jardin)
allees damees) #A(damer allees-jardin))
10>> (etablir (#A(sabler-polir sol) charpente finie))
10<< (instancier (#A(sabler-polir sol) charpente finie) (#S(finir charpente)
charpente finie) #A(finir charpente))
11>> (etablir (#A(sabler-polir sol) peintures terminees))
11<< (instancier (#A(sabler-polir sol) peintures terminees) (#S(peindre)
peintures terminees) #A(peindre))
12>> (etablir (#A(passer fils-electriques) cablage-edf installe))
12<< (instancier (#A(passer fils-electriques) cablage-edf installe) (#S(cabler
edf) cablage-edf installe) #A(cabler edf))
13>> (etablir (#A(installer cuisine) plomberie-principale installee))
13<< (instancier (#A(installer cuisine) plomberie-principale installee) (#S(
poser plomberie-principale) plomberie-principale installee) #A(poser
plomberie-principale))
14>> (etablir (#A(installer cuisine) cablage-edf installe))
14<< (< (#A(cabler edf) cablage-edf installe) (#A(installer cuisine)
cablage-edf installe))
15>> (etablir (#A(installer cuisine) carrelage-parquet poses))
15<< (instancier (#A(installer cuisine) carrelage-parquet poses) (#S(poser
carrelage-parquet) carrelage-parquet poses) #A(poser carrelage-parquet))
16>> (etablir (#A(finir plomberie) plomberie-principale installee))
16<< (< (#A(poser plomberie-principale) plomberie-principale installee) (#A(
finir plomberie) plomberie-principale installee))
17>> (etablir (#A(finir plomberie) carrelage-parquet poses))
17<< (< (#A(poser carrelage-parquet) carrelage-parquet poses) (#A(finir
plomberie) carrelage-parquet poses))
18>> (etablir (#A(installer conduites-verticales) plancher 0 termine))
```

```
18<< (instancier (#A(installer conduites-verticales) plancher 0 termine) (#S(
couler fondations-beton) plancher 0 termine) #A(couler fondations-beton))
19>> (etablir (#A(damer allees-jardin) escalier pose))
19<< (instancier (#A(damer allees-jardin) escalier pose) (#S(poser escalier)
escalier pose) #A(poser escalier))
20>> (etablir (#A(peindre) plomberie installee))
20<< (< (#A(finir plomberie) plomberie installee) (#A(peindre) plomberie
installee))
21>> (etablir (#A(peindre) cuisine installee))
21<< (< (#A(installer cuisine) cuisine installee) (#A(peindre) cuisine
installee))
22>> (etablir (#A(poser carrelage-parquet) sol-rdc pose))
22<< (instancier (#A(poser carrelage-parquet) sol-rdc pose) (#S(poser sol-rdc)
sol-rdc pose) #A(poser sol-rdc))
23>> (etablir (#A(poser carrelage-parquet) platres essuyes))
23<< (instancier (#A(poser carrelage-parquet) platres essuyes) (#S(faire
platres) platres essuyes) #A(faire platres))
24>> (etablir (#A(poser plomberie-principale) canalisations installees))
24<< (instancier (#A(poser plomberie-principale) canalisations installees) (#S
(installer canalisations) canalisations installees) #A(installer canalisations
))
25>> (etablir (#A(finir charpente) carrelage-parquet poses))
25<< (< (#A(poser carrelage-parquet) carrelage-parquet poses) (#A(finir
charpente) carrelage-parquet poses))
26>> (etablir (#A(couler fondations-beton) excavation finie))
26<< (instancier (#A(couler fondations-beton) excavation finie) (#S(faire
excavations) excavation finie) #A(faire excavations))
27>> (etablir (#A(poser escalier) gouttieres fixees))
27<< (instancier (#A(poser escalier) gouttieres fixees) (#S(fixer gouttieres)
gouttieres fixees) #A(fixer gouttieres))
28>> (etablir (#A(poser escalier) conduites-verticales installees))
28<< (< (#A(installer conduites-verticales) conduites-verticales installees) (
#A(poser escalier) conduites-verticales installees))
29>> (etablir (#A(faire platres) plomberie-principale installee))
29<< (< (#A(poser plomberie-principale) plomberie-principale installee) (#A(
faire platres) plomberie-principale installee))
30>> (etablir (#A(faire platres) cablage-edf installe))
30<< (< (#A(cabler edf) cablage-edf installe) (#A(faire platres) cablage-edf
installe))
31>> (etablir (#A(installer canalisations) plancher 0 termine))
31<< (< (#A(couler fondations-beton) plancher 0 termine) (#A(installer
canalisations) plancher 0 termine))
32>> (etablir (#A(poser sol-rdc) canalisations installees))
32<< (< (#A(installer canalisations) canalisations installees) (#A(poser
sol-rdc) canalisations installees))
33>> (etablir (#A(fixer gouttieres) toit couvert))
33<< (instancier (#A(fixer gouttieres) toit couvert) (#S(couvrir toit) toit
couvert) #A(couvrir toit))
34>> (etablir (#A(couvrir toit) briques posees))
34<< (instancier (#A(couvrir toit) briques posees) (#S(poser briques) briques
posees) #A(poser briques))
35>> (etablir (#A(poser briques) murs 4 eriges))
35<< (instancier (#A(poser briques) murs 4 eriges) (#S(eriger-etage ?dessous
?dessus) murs ?dessus eriges) #A(eriger-etage ?dessous4 4))
```

```
36>> (etablir (#A(eriger-etage ?dessous4 4) au-dessus 4 ?dessous4))
36<< (= (#A(eriger-etage ?dessous4 4) au-dessus 4 ?dessous4) (#Ainitial
au-dessus 4 3))
37>> (etablir (#A(eriger-etage ?dessous4 4) plancher ?dessous4 termine))
37<< (instancier (#A(eriger-etage ?dessous4 4) plancher ?dessous4 termine) (#S
(eriger-etage ?dessous ?dessus) plancher ?dessus termine) #A(eriger-etage
?dessous6 ?dessous4))
38>> (etablir (#A(eriger-etage ?dessous6 ?dessous4) au-dessus ?dessous4
?dessous6))
38<< (= (#A(eriger-etage ?dessous6 ?dessous4) au-dessus ?dessous4 ?dessous6) (
#Ainitial au-dessus 3 2))
39>> (etablir (#A(eriger-etage ?dessous6 ?dessous4) plancher ?dessous6 termine
))
39<< (instancier (#A(eriger-etage ?dessous6 ?dessous4) plancher ?dessous6
termine) (#S(eriger-etage ?dessous ?dessus) plancher ?dessus termine) #A(
eriger-etage ?dessous8 ?dessous6))
40>> (etablir (#A(eriger-etage ?dessous8 ?dessous6) au-dessus ?dessous6
?dessous8))
40<< (= (#A(eriger-etage ?dessous8 ?dessous6) au-dessus ?dessous6 ?dessous8) (
#Ainitial au-dessus 2 1))
41>> (etablir (#A(eriger-etage ?dessous8 ?dessous6) plancher ?dessous8 termine
))
41<< (instancier (#A(eriger-etage ?dessous8 ?dessous6) plancher ?dessous8
termine) (#S(eriger-etage ?dessous ?dessus) plancher ?dessus termine) #A(
eriger-etage ?dessous10 ?dessous8))
42>> (etablir (#A(eriger-etage ?dessous10 ?dessous8) au-dessus ?dessous8
?dessous10))
42<< (= (#A(eriger-etage ?dessous10 ?dessous8) au-dessus ?dessous8 ?dessous10)
(#Ainitial au-dessus 1 0))
43>> (etablir (#A(eriger-etage ?dessous10 ?dessous8) plancher ?dessous10
termine))
43<< (< (#A(couler fondations-beton) plancher 0 termine) (#A(eriger-etage
?dessous10 ?dessous8) plancher ?dessous10 termine))
44>> Succes

** Temps CPU   : 188.44
** Delai       : 30
```

La solution est trouvée en 189 secondes CPU (Sun 3/60, Lelisp15.22, mode interprété, mesure par `time`) et, les actions étant valuées en une unité de temps arbitraire (le jour, par exemple), le délai global est de 30 unités de temps.

Sur les 36 décisions prises, 24 créent des actions en chaînage arrière et 12 (aux pas 14, 16, 17, 20, 21, 25, 28, 29, 30, 31, 32 et 36) posent des contraintes de précédence réalisant des établissements sans instancier de schémas d'actions (heuristique 3).

L'ajout d'une contrainte de précédence oblige YAPS à recalculer les valeurs de vérités de tous les termes se trouvant après ou en parallèle des deux actions relatives à la contrainte, ce qui justifie ce temps CPU important.

Pour les 24 actions suivantes, après le titre de chaque action figure un intervalle `[t1 (d) t1]` où `t1` est la date de début au plus tôt de l'action, `t2` sa date de fin au plus tard et `d` sa durée

(issue du schéma d'actions). Le caractère "C" repère les actions critiques.

```
> Actions (29)

 ** Nom : (eriger-etage 0 [?dessous10] 1 [?dessous8])  **
 Pre  : (au-dessus 1 [?dessous8] 0 [?dessous10]) (plancher 0 [?dessous10]
termine)
 Post : (plancher 1 [?dessous8] termine) (murs 1 [?dessous8] eriges)
 -->  : #A(eriger-etage ?dessous8 ?dessous6)

 ** Nom : (eriger-etage 1 [?dessous8] 2 [?dessous6])  **
 Pre  : (au-dessus 2 [?dessous6] 1 [?dessous8]) (plancher 1 [?dessous8]
termine)
 Post : (plancher 2 [?dessous6] termine) (murs 2 [?dessous6] eriges)
 -->  : #A(eriger-etage ?dessous6 ?dessous4)

 ** Nom : (eriger-etage 2 [?dessous6] 3 [?dessous4])  **
 Pre  : (au-dessus 3 [?dessous4] 2 [?dessous6]) (plancher 2 [?dessous6]
termine)
 Post : (plancher 3 [?dessous4] termine) (murs 3 [?dessous4] eriges)
 -->  : #A(eriger-etage ?dessous4 4)

 ** Nom : (eriger-etage 3 [?dessous4] 4)  **
 Pre  : (au-dessus 4 3 [?dessous4]) (plancher 3 [?dessous4] termine)
 Post : (plancher 4 termine) (murs 4 eriges)
 -->  : #A(poser briques)

 ** Nom : (poser briques)  **
 Pre  : (murs 4 eriges)
 Post : (briques posees)
 -->  : #A(couvrir toit)

 ** Nom : (couvrir toit)  [6 (2) 22] **
 Pre  : (briques posees)
 Post : (toit couvert)
 -->  : #A(fixer gouttieres)

 ** Nom : (fixer gouttieres)  [8 (1) 23] **
 Pre  : (toit couvert)
 Post : (gouttieres fixees)
 -->  : #A(poser escalier)

 ** Nom : (faire excavations)  [0 (4) 4] C **
 Pre  : (terrain achete)
 Post : (excavation finie)
 -->  : #A(couler fondations-beton)

 ** Nom : (installer canalisations)  [6 (1) 7] C **
 Pre  : (plancher 0 termine)
 Post : (canalisations installees)
 -->  : #A(poser sol-rdc) #A(poser plomberie-principale)

 ** Nom : (faire platres)  [10 (10) 20] C **
 Pre  : (plomberie-principale installee) (cablage-edf installe)
```

```
Post : (platres essuyes)
-->  : #A(poser carrelage-parquet)

** Nom : (poser sol-rdc)  [7 (2) 20] **
Pre  : (canalisations installees)
Post : (sol-rdc pose)
-->  : #A(poser carrelage-parquet)

** Nom : (poser escalier)  [9 (2) 25] **
Pre  : (gouttieres fixees) (conduites-verticales installees)
Post : (escalier pose)
-->  : #A(damer allees-jardin)

** Nom : (couler fondations-beton)  [4 (2) 6] C **
Pre  : (excavation finie)
Post : (plancher 0 termine)
-->  : #A(eriger-etage ?dessous10 ?dessous8) #A(installer canalisations) #A(
installer conduites-verticales)

** Nom : (poser carrelage-parquet)  [20 (3) 23] C **
Pre  : (sol-rdc pose) (platres essuyes)
Post : (carrelage-parquet poses)
-->  : #A(finir charpente) #A(finir plomberie) #A(installer cuisine)

** Nom : (poser plomberie-principale)  [7 (3) 10] C **
Pre  : (canalisations installees)
Post : (plomberie-principale installee)
-->  : #A(faire platres)

** Nom : (cabler edf)  [0 (2) 10] **
Pre  : (terrain achete)
Post : (cablage-edf installe)
-->  : #A(faire platres) #A(passer fils-electriques)

** Nom : (peindre)  [25 (3) 28] C **
Pre  : (plomberie installee) (cuisine installee)
Post : (peintures terminees)
-->  : #A(sabler-polir sol)

** Nom : (finir charpente)  [23 (3) 28] **
Pre  : (carrelage-parquet poses)
Post : (charpente finie)
-->  : #A(sabler-polir sol)

** Nom : (damer allees-jardin)  [11 (5) 30] **
Pre  : (escalier pose)
Post : (allees damees)
-->  : #A(faire gros-oeuvre)

** Nom : (passer fils-electriques)  [2 (1) 30] **
Pre  : (cablage-edf installe)
Post : (travaux-electriques finis)
-->  : #A(corps metier)
```

```
** Nom : (installer cuisine)  [23 (1) 25] **
Pre  : (plomberie-principale installee) (cablage-edf installe) (
carrelage-parquet poses)
Post : (cuisine installee)
-->  : #A(peindre) #A(corps metier)

** Nom : (finir plomberie)  [23 (2) 25] C **
Pre  : (plomberie-principale installee) (carrelage-parquet poses)
Post : (plomberie installee)
-->  : #A(peindre) #A(corps metier)

** Nom : (installer conduites-verticales)  [6 (1) 23] **
Pre  : (plancher 0 termine)
Post : (conduites-verticales installees)
-->  : #A(poser escalier) #A(corps metier)

** Nom : (sabler-polir sol)  [28 (2) 30] C **
Pre  : (charpente finie) (peintures terminees)
Post : (sol sable-poli)
-->  : #A(decorations-interieures)

** Nom : (decorations-interieures)  **
Pre  : (sol sable-poli)
Post : (decorations-interieures)
-->  : #Afinal

** Nom : (corps metier)  **
Pre  : (conduites-verticales installees) (plomberie installee) (cuisine
installee) (travaux-electriques finis)
Post : (corps-metier termines)
-->  : #Afinal

** Nom : (faire gros-oeuvre)  **
Pre  : (allees damees)
Post : (gros-oeuvre termine)
-->  : #Afinal

** Nom : final  **
Pre  : (gros-oeuvre termine) (corps-metier termines) (decorations-interieures
)
Post :
-->  :

** Nom : initial  **
Pre  :
Post : (terrain achete) (au-dessus 4 3) (au-dessus 3 2) (au-dessus 2 1) (
au-dessus 1 0)
-->  : #A(faire excavations) #A(cabler edf)

> 9 action(s) critique(s)
```

La base d'unification est la suivante :

```
> Base de contraintes d'unification
> ?dessous10 -> 0
> ?dessous8 -> 1
> ?dessous6 -> 2
> ?dessous4 -> 3
```

Ces actions sont représentées sur la figure 7.4.

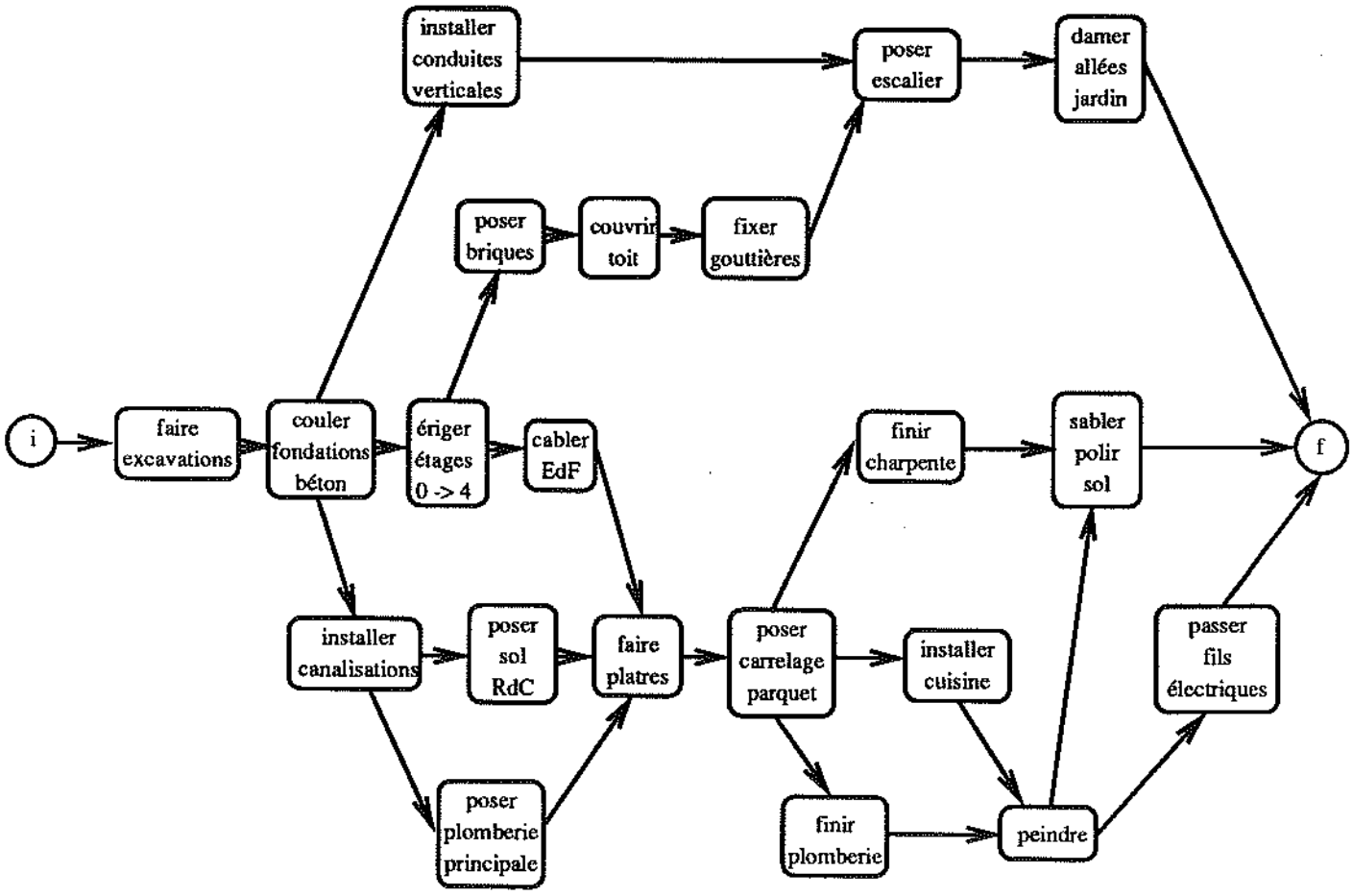


Figure 7.4: Graphe d'actions après planification

Si l'on relance YAPS (fonction `recontrol`), il effectue des retours-arrières sur ces 12 contraintes de précédence et les inverse une à une, ce qui revient, dans ce cas particulier, à dupliquer partiellement le graphe d'actions pour fournir les $(2^{12} - 1)$ autres solutions. Nous avons arrêté YAPS au bout de 200 solutions (sur les 4096) obtenues en un peu moins d'une heure (temps réel, toujours sur Sun 3/60) avec 10 secondes CPU par solution en moyenne (les allocations de structures représentant les actions ayant été faites en grande partie à la première solution).

Cependant ce temps, comme les précédents, ne reflète pas totalement les performances du programme, puisque s'y greffe le temps de Glanage des Cellules (le ramasse-miettes GC) inhérent à lisp.

Remarque Contrairement à ce qu'une observation hâtive pourrait laisser croire, des domaines à expertise, tels que celui de la construction d'une maison, sont *plus simples* que les autres problèmes combinatoires, pour plusieurs raisons :

1. les tâches-actions n'utilisent peu de variables car elles commencent à introduire une expertise qui ne contient que peu de motifs répétitifs (les étages, dans notre cas) ; l'élimination des contraintes d'unification rend possible l'utilisation de techniques d'indexation de ces propositions, améliorant ainsi notablement les performances du systèmes (cf. les TMM en 3.3.1, ou [Dean 85] [Agre & Chapman 87] [Ghallab & Mounir Alaoui 89]) ;
2. l'expertise (même très faible, comme ici) impose déjà un ordre entre tâches ;
3. il y a peu d'interactions entre branches en parallèle, chaque tâche correspond à une technique propre de construction.

7.3 Critique et perspectives

Différences entre TWEAK et YAPS Nous pouvons résumer les apports de YAPS par rapport à TWEAK par les points suivants (d'importance décroissante) :

- Sans heuristiques, un planificateur exact reste remarquablement inefficace. Les heuristiques utilisées (et plus généralement le contrôle) sont clairement décrites (cf. 6.2), justifiées autant que possible, et surtout sont physiquement séparées du reste du code implémentant la mécanique, maintenant plus classique, du planificateur.
- Les rétablisseurs (*"White Knights"* selon l'imagerie de Chapman) sont considérés comme utiles et sont effectivement implémentés. Chapman n'a curieusement pas implémenté cette technique [Chapman 85, p. 32], qui soutient pourtant une part importante de sa théorie[9].
- La gestion des contraintes d'unification et de non-unification est précisément décrite (cf. 5.2).
- Plusieurs points d'importance moindre sont précisés : le critère est exprimé sans passer par les situations (puisque les situations ne doivent jamais être explicitées 4.1), la modalité est étendue aux contraintes d'unification pour homogénéiser l'expression du critère (cf. 5.2.1), la quantification existentielle implicite des variables libres est mise en évidence (cf. 5.2.2), ...
- Du point de vue de l'implémentation pure, un pisteur pas-à-pas et un visualisateur de plans (cf. C.3) permettent de mettre facilement au point de nouvelles heuristiques.

Ce que notre planificateur est Nous rappelons les contraintes que nous nous sommes fixées dans la conception et le codage de notre planificateur :

- *indépendance du domaine* : les heuristiques ne tirent aucun *parti* des lois de comportement d'un monde particulier (*aucune expertise* liée au domaine d'application).

[9] Remarque due à Eric Jacopin (LAFORIA).

- *explicitation du critère de vérité* : la façon de calculer ce qui est vrai ou faux à un "instant" donné n'est pas noyée sous quelques heuristiques, mais est exposée en premier ; son implémentation est présentée et sa complexité évaluée.
- *ordre 1 et non-linéarité* : les variables (*objets formels, partiellement contraints*) ne sont pas instanciées au plus vite, mais pleinement intégrées au formalisme ; de même, l'interprétation de la non-linéarité en terme d'absence d'ordre sur les actions en parallèle permet aussi de retarder les décisions tant que l'information n'est pas suffisante.
- *souplesse d'implémentation d'heuristiques (soin du codage)* : le soin apporté à l'implémentation[10] assure la facilité d'implémentation de nouvelles heuristiques à partir de la base formée par le critère de vérité, de la gestion des variables et de l'algorithme (très général) de contrôle. De ce point de vue, le portage du code de Lelisp vers C (par exemple) nous semble nuisible, bien qu'induisant un gain immédiat des peformances. Nous pensons que le présent planificateur (contrairement à ses prédécesseurs, cf. l'introduction de la partie II) peut facilement être étendu[11].

Ce que notre planificateur n'est pas (ce qu'il pourrait être) Nous indiquons ici les principales limitations constatées de notre planificateur, et suggérons à chaque fois quelques idées déblayant une voie potentielle de recherche :

- *notre planificateur n'est pas hiérarchique* : Notre critère de vérité ne gère pas les interactions entre des niveaux hiérarchiques, mais peut néanmoins s'appliquer à l'intérieur d'un même niveau, une fois les éventuels points de contrôle hiérarchiques posés (fonctionnement top-down sur les niveaux hiérarchiques). Un critère gérant cette propagation serait de toutes façons requis.
- *notre planificateur manque encore d'heuristiques générales* : Sans prétendre rivaliser avec les résolveurs de problèmes, l'implémentation de quelques heuristiques, telles celles que l'on peut trouver dans ALICE [Laurière 76], étendrait le domaine d'application de YAPS vers les problèmes combinatoires ;
- *notre planificateur n'est pas intelligemment réactif* : L'intervention de la "Mère Nature" se traduirait par l'ajout d'une action sans prémisses, et le processus de planification se poursuivrait normalement. Un ersatz de la typologie requise des actions est actuellement constitué par la notion de profondeur verticale : la profondeur verticale d'une action de type "Mère Nature" est fixée à une valeur arbitrairement grande, signifiant une dépendance causale la plus faible possible avec le but global.
- *le formalisme des objets de notre planificateur est trop faible* : Un objet est actuellement une constante, ses contraintes, des égalités ou différences, limitations draconiennes liées au critère de vérité. Les objets, même actuellement, sont sous-exploités : on pourrait tirer parti de la finitude de l'ensemble des constantes[12], ou de leur aspect numérique par

[10] Application du principe du moindre engagement au développement du planificateur lui-même. Cette approche n'est pas dogmatique mais bien pragmatique (cf. les deux planificateurs "malheureux" précédents).

[11] La recherche d'optimalité succédant à la preuve d'existence.

[12] Si $T_\Sigma = \{a_1, \ldots, a_n\}$, on pourrait implémenter des heuristiques comme :

$$\forall x \in X, (\exists j \in [1,n], \forall i \neq j, x \neq a_i) \rightarrow x = a_j$$

l'intégration de contraintes d'inégalités dans le critère, ou plus tirer parti des contraintes existantes[13].

Utiliser les constantes comme des index d'objets à part entière (tels que ceux de `leloo`, cf. annexe D) permettrait de représenter les relations entres objets non comme des termes de la logique des prédicats, mais des relations attribut/valeur des Langages Orientés Objets.

[13]Evaluer un plan par le nombre de variables libres, ou la moyenne du nombre d'instanciations possibles pour chaque variable, ... tout ceci tentant de définir le *degré d'instanciation* d'une variable sans avoir à lister toutes les instanciations (entropie).

Partie III

Planification avec expertise

Introduction

La partie précédente traitait des techniques de propagation de la vérité dans un graphe d'actions. Elles identifient certaines définitions permettant de donner un sens à la conjonction de deux mondes possibles, ce qui permet de considérer le réseau d'actions comme une base de données temporelle, puisque des requêtes sur la valeur d'un fait à un instant donné peuvent lui être adressées. Certaines heuristiques de stabilisation du réseau permettent l'induction d'information (liens de précédence, instanciation ou présence d'actions), transformant la base de données temporelle en planificateur vrai, puisqu'il crée un plan d'actions dont l'exécution réalise un but fixé.

Le point de vue adopté jusqu'alors est la découverte rigoureuse de l'existence des liens des actions, à travers un outil théorique portant sur des casse-têtes combinatoires, c'est-à-dire des domaines jouets. L'approche de la présente partie est diamétralement opposée : COPLANER[14] est une application opérationnelle et basée sur une expertise, dans le domaine de la planification de chantiers de bâtiments (COPLANER est donc un planificateur *dépendant du domaine*). Nous avons bénéficié de l'aide d'un ingénieur de Dumez, spécialiste de la planification de chantiers, déclaré *expert*, qui nous a fourni les données pratiques nécessaires (*recueil d'expertise*). L'approche sous-jacente (un critère de vérité particulier) a été discutée avec des experts d'autres domaines.

Ensuite, COPLANER est opérationnel car l'aspect I.A. pur n'a pas été seulement abordé : ce système devant être remis à terme en des mains non-informaticiennes, la convivialité de l'interface homme-machine a nécessité des développements conséquents (long mais facile).

Cette partie est scindée en deux chapitres. Le premier pose le problème et identifie les questions de base qu'il soulève. Le second chapitre décrit l'architecture adoptée et le fonctionnement du système. Nous verrons l'influence de l'introduction de l'expertise sur le critère de vérité développé en partie II ; nous verrons en particulier comment l'obligation de mener de front le problème de la planification avec celui de la représentation nous a amené à sacrifier des pans entiers du planificateur de la partie précédente sur l'autel du pragmatisme.

[14] COPLANER est le nom du système de planification développé conjointement avec Dumez pour le compte du MELATT (Ministère de l'Equipement, du Logement et de l'Aménagement du Territoire). Ce système est actuellement la propriété de la société Cognitech.

Chapitre 8

Analyse du problème

Nous présentons maintenant les connaissances nécessaires au traitement du problème de la planification de chantier de bâtiments. Après avoir rapidement survolé les connaissances techniques pures de ce domaine particulier (voir [Assémat et coll. 89] pour de plus amples détails), nous étudierons le point de vue adopté par un expert dans sa résolution quotidienne, et replacerons ce point de vue dans celui adopté par un groupe d'experts généraux en gestion de projet.

8.1 Description du domaine

8.1.1 Pourquoi et comment planifier un chantier de bâtiments

Description succincte Dans une entreprise de construction arrivent constamment des cartes techniques représentant un ensemble d'immeubles à réaliser. Ces cartes décrivent à la fois les mesures détaillées de chacun des immeubles à construire (vue du dessus, vue verticale en coupe, ...) et celles de l'environnement (immeubles mitoyens, rues et avenues, terrain et sous-sol. ...).

Un "expert en construction" doit alors, en quelques jours, trouver une technique de construction de ces immeubles, déterminer l'enchaînement des opérations (gros-œuvre, corps de métier, sous-traitants, ...), résumer son organisation en un planning des tâches des intervenants, estimer un coût et un délai (global et par poste), et prouver ainsi la faisabilité du projet au coût le plus bas.

Eléments d'expertise Nous donnons quelques éléments permettant de mesurer le problème, dans le cas de la planification du gros œuvre de logements sociaux ou de bureaux, que nous considérerons dans COPLANER.

Un chantier débute par d'éventuelles destructions de ruines ou baraques préexistantes (terrassements généraux) et par la protection des abords directs du chantier (immeubles mitoyens ou rues frontalières : voiles par avenue, tranchées blindées, parois moulées, ...).

La qualité du terrain initial est déterminante : une forte pente du terrain suggère de construire les bâtiments les plus bas d'abord. Avec un faible taux de pression du sol, le bâtiment flotterait et coulerait, comme un navire trop chargé, ce qui implique de prévoir des fondations profondes. Inversement, la présence de nappes d'eau souterraines est un danger pour les sous-sols en période de crues, ce qui implique de prévoir des fondations *peu* profondes.

L'architecture du bâtiment, pour être tenue dans le sol, induit un type de fondations (semelles, puits, radier, pieux). Ces fondations débouchent sur la construction des niveaux inférieurs au rez-de-chaussée (infra-structure), montés en séquence (super-structure) jusqu'au toit (édicule). Chaque niveau (étage) possède sa propre géométrie (superficie, hauteur sous plafond — la disposition des cloisons non porteuses, définissant appartements et leurs pièces, ne relève pas du gros œuvre).

Le placement des bâtiments sur le chantier, la hauteur des immeubles mitoyens, induit le nombre, la place et la hauteur des grues (rayon d'action fixé).

Les corps de métiers (peintre, plombier, électricien, ...) peuvent intervenir alors que le gros œuvre n'est pas totalement terminé.

8.1.2 Caractéristiques du problème

Aspect Expertise L'expert doit manipuler deux types de connaissances (voir table 8.1) :

- *connaissances techniques (connaissances du domaine, ou* structurelle*)* : techniques de construction d'un bâtiment (quel type de fondations fera tenir le batiment dans ce terrain ?), contraintes que poseront chacun des corps de métiers, connaissances légales, ...
- *connaissances d'organisation (connaissances de planification, ou* opérationnelle*)* : placement des ressources (limiter le nombre de grues), appel à la sous-traitance (camion à béton, ...), circulation des ressources sur le chantier même (rampe d'accès au trou de fondation), ...

Connaissances Structurelles	*Connaissances Opérationnelles*
Structure générale d'un bâtiment Connaissances techniques (type et conditions de mise en œuvre, modes constructifs fondations ...) Données géographiques Environnement du chantier (pente, état de surface, taux de pression du sol, présence d'eau, ...)	Descriptifs des tâches Sous-plans partiels Descriptif des liens Table des ressources (cadences, ...)

Table 8.1: Types de connaissances manipulés par un expert en construction

Aspect Génération de plan Chaque tâche dispose de conditions de démarrage : il est impossible de construire le 5^e étage avant le 3^e, par exemple. Un formalisme de type prémisse / conclusion peut sembler appplicable à ce niveau, décrivant les dépendances logiques entre tâches

en termes de buts et sous-buts. Mais on verra que ces dépendances préexistent sous forme compilée (apprentissage) chez l'expert.

Aspect Allocation de ressources La lourdeur d'une entreprise comme la construction de bâtiments se traduit par une quantité importante de ressources fixes (matières premières) : le volume global de chaque ressource peut être évalué au vu des cartes, indépendamment de leur future consommation au jour le jour.

Pour dégager un gain, l'expert joue plus volontiers sur l'*agent*, ressource motrice d'une tâche (équipe d'ouvriers, ...). De la répartition des équipes sur chaque tâche dépend la durée de réalisation. Des ajustements en cours d'exécution sont constamment effectuées par le chef de chantier, afin de gagner du temps sur le délai global (ou, plus souvent, de les tenir ...).

Mais la permutation et la répartition des ressources n'influent que sur les *délais* et consommation globaux du planning, non sur la *logique du planning*, qui résulte de considérations techniques à propos de l'architecture du bâtiment et de son environnement. Le problème de l'allocation des ressources apparaît comme *postérieur* à celui de la mise en place du planning nécessaire : des techniques de Recherche Opérationnelle sont alors appropriées pour exploiter les degrés de liberté laissés par la génération de ce planning nécessaire.

Aspect Incertain Dans le cas d'un projet déjà accepté par le client, l'expert dispose de tous les renseignements qu'il peut souhaiter sur les bâtiments. Mais dans le cas de réponse à des appels d'offre, il doit se faire une opinion sur une carte encore imprécise : tous les choix techniques n'ont pas été faits (fondations, ...). Notre expert doit pouvoir évaluer la carte et construire un planning *à divers degrés de détails* (hiérarchie).

Outils de l'expert La planification effectuée par l'expert constitue un exercice difficile (grande quantité de critères intervenant dans le processus, complexité de ces critères, ldots). L'expert est déjà libéré de l'*évaluation* de son planning, grâce à des logiciels de type PERT / méthodes des potentiels (tous basés sans exception sur l'algorithme PROPAGER-DÉBUT de 2.2).

L'intérêt de ces "progiciels de gestion de projet" pour un expert tient en trois points :

- évaluation en quelques minutes (pour un réseau de l'ordre d'un millier de tâches) du coût et du délai d'un planning existant ;
- qualité de la sortie graphique : diagrammes en S, diagramme de Gantt, réseau des tâches, histogrammes ...qu'il peut facilement insérer en annexe de sa réponse à l'appel d'offre.

Ces outils *évaluent* un planning, mais ne le *créeront* pas, n'ayant aucun moyen d'intégrer la sémantique du projet qu'ils représentent. Ils disposent cependant de stratégie de décalage de tâche réagissant à la sur-consommation de ressources (appellations quasi-standardisées : "lissage" et "nivellement"). L'effet en est une augmentation du délai ou du coût global, mais jamais la remise en cause de la logique du planning (l'aspect de causalité n'a aucun moyen d'être représenté).

Ces logiciels d'évaluation sont également utilisés à l'exécution comme calculette lors de modifications du planning.

8.2 Approche experte

8.2.1 Cognitique

Constats Le recueil d'expertise auprès de notre expert et d'experts d'autres domaines[1] a mis en lumière les poins suivants :

- **Les réseaux PERT manquent de sémantique ;**

  Ni la durée d'une tâche, ni ses jalons, ni les liens algébriques (entre dates de début au plus tôt / fin au plus tard) ne représentent ce que fait réellement une tâche. La polysémie des liens, due à la faiblesse du langage d'expression, a été remarquée (quelle est la raison de la présence d'un lien ? dépendance causale ? consommation d'une ressource ? ... ?). L'absence d'aspect microscopique a également été remarquée[2] (cf. discussion en 1.2.1).

- **Il existe un "flux d'informations" circulant entre les tâches du réseau ;**

  Une tâche est perçue comme une modification d'un environnement : il y a ce qui est "avant", ce qui est effectué, et ce qui est "après". Cette remarque constitue précisément la base de la formalisation d'action en prémisse / conclusion. Il suggère d'appliquer cette description des modifications, qu'apportent une tâche, au *"contrôle qualité"* (vérification de l'adéquation entre spécification et réalisation).

  Ces experts distinguent ensuite deux sous-concepts : un "environnement instantané", qu'il soit avant ou après une tâche (une photographie d'un chantier), et les "modifications" qu'apportent une tâche (ce qui change entre deux "environnements instantanés" successifs).

  On aura reconnu là la définition intuitive d'une *situation* et d'une *action* du chapitre 1.

- **Un planning est toujours construit à partir de morceaux d'anciens plannings ;**

  Un morceau (*composant*) de bâtiment est réalisé par un mini-planning, variation autour d'un mini-planning typique, immuable. Ces experts adaptent des sous-solutions partielles existantes et savent les agrèger de façon cohérente.

  Le principe de décomposition de buts en sous-buts est ainsi redécouvert. Mais ces sous-buts, qu'un mini-planning-solution réalise, ne peuvent pas être archivés par ces logiciels de PERT (qui réduisent le sous-but qu'une tâche satisfait à son *nom*). Le comportement des experts montre pourtant qu'il est (souvent) moins coûteux (en temps d'expert) d'adapter une solution ancienne que d'en redécouvrir une nouvelle. C'est le dilemne de la *replanification* (cf. 3.3.4), ici tranché en faveur de la replanification *minimale*. Cette approche est possible pour des domaines où les différentes techniques de construction sont fortement

[1] Pour COPLANER, le recueil d'expertise auprès de notre expert en planification de chantiers a duré plusieurs mois (Cognitech). L'approche ainsi construite a été testée lors d'interviews avec d'autres personnes, expertes plus généralement en "gestion de projet" (Syseca).

[2] Un chef de projet, en entretien : *"un PERT, c'est ce qu'on peut faire quand on ne connaît rien au projet"*.

disjointes (faibles interactions entre activités concurrentes, chaque branche ne pouvant que suivre sa technique de construction personnelle).

- **L'objet à construire est toujours décomposé en sous-objets ;**

  Décomposer un chantier selon des bâtiments, un bâtiment en étages, un étage en cloison et plancher permet à ces expert d'isoler progressivement les compétences nécessaires (sous-problèmes faiblement couplés).

Approche humaine Les phases successives qui constituent le mécanisme commun de construction de planning chez les experts consultés sont les suivantes :

1. **Mise en place des phases immuables intervenant dans un projet** : pour le gros œuvre d'un chantier de bâtiment : préparation, fondation, infra-structure, super-structure, toit. Pour un projet informatique, ce serait : spécifications, réalisation, intégration et validation.
2. **Visualisation de l'"objet complexe" dont le futur plan devra assurer la construction** : cette phase vise à évaluer rapidement les caractéristiques très globales du chantier.
3. **Décomposition de cet objet en morceaux élémentaires** : un composant final a le niveau de détail le plus élevé.
4. **Recherche de mini-plannings pouvant réaliser ces morceaux élémentaires** : ces mini-plannings sont adaptés d'anciens, issus des archives, ou au pire (re-)construit *ex nihilo*.
5. **Intégration de ces mini-plans en le plan global** : l'expert utilise ses *connaissances d'organisation* pour gérer les interactions entres tâches issus de mini-plannings différents.

8.2.2 Différences avec la planification "indépendante du domaine"

Planification en conception Deux représentations sont manipulées par les experts :

- *représentation structurelle* (*quoi*) Elle est constituée de la description la plus fine possible des bâtiments à construire, et plus généralement de l'*objet complexe* à construire. Les connaissances techniques (ou *connaissances structurelles*) de l'expert assure la cohérence de cette représentation.
- *représentation opérationnelle* (*comment*) Elle est constituée par la description des phases, macro-tâches et tâches, des ressources et acteurs de tout type intervenant dans la réalisation de l'*objet complexe*.

Les données constitutives du plan que manipulent les planificateurs indépendants du domaine (cf. parties I et II) peuvent être considérées sous le prisme structurel / opérationnel de la façon suivante :

- les prémisses des actions ou schémas d'actions décrivent la situation désirée. Notre objet complexe serait descriptible (même mal) dans ce formalisme. Dans COPLANER, ces prémisses, buts ou sous-buts, sont regroupés dans la *représentation structurelle*.

- dans un planificateur "indépendant du domaine", une conclusion a toujours un lien de dépendance causale avec une prémisse (cf. la *profondeur verticale* de 6.2.1). Un conflit entre une prémisse et une conclusion traduit alors un conflit implicite entre cette prémisse et la prémisse dont la conclusion dépend. Mais le planificateur n'a aucun moyen de détecter un conflit entre prémisses : sans description suffisamment fine du but, le planificateur se soupçonnera toujours en cas de conflit (cohérence d'une prémisse et d'une conclusion que lui-même a ajoutée) avant de soupçonner l'utilisateur (cohérence des buts eux-mêmes).

Hypothèses de l'approche experte Dans COPLANER, cette résolution de conflit n'aura pas lieu d'être pour plusieurs raisons.

Si l'objet complexe (ou but global) est représenté de façon suffisamment détaillée, la détection des conflits des planificateurs indépendants du domaine se traduit par une vérification de cohérence sur cet objet complexe. COPLANER dispose d'une représentation des connaissances techniques de l'expert pour cela, ce qui est difficilement implémentable dans un planificateur indépendant du domaine.

Distinguer représentation structurelle et opérationnelle permet d'utiliser des outils d'inférences plus classiques pour cette détection. On évite ainsi de reporter au niveau opérationnel des conflits d'origine structurelle, puisque ces conflits potentiels sont détectés avant toute génération de plan.

Fondements de COPLANER :

- *Exprimer le plus complètement l'objet complexe à construire dans la représentation structurelle* ;
- *Détecter le maximum d'incohérences à partir de cette représentation.*

L'ouverture de certains planificateurs indépendants du domaine rendrait possible la représentation de cette expertise : Sacerdoti indique que l'utilisateur peut intégrer ses propres *critiques* dans NOAH. De même, notre planificateur de la partie II peut être couplé à un système d'inférences plus classique. Mais tous les auteurs regrettent la pauvreté sémantique de leur planificateur, la représentation structurelle ayant disparu ou étant diluée dans la représentation opérationnelle[3].

C'est cette structure d'accueil de l'expertise structurelle, avec son mécanisme de rattachement aux nœuds opérationnels, qui est le cœur de COPLANER.

[3] *"Les nœuds du plan peuvent être considérés comme des instanciations des 'frames' en action. Ce serait cependant des 'frames' d'un genre restreint. Tous les attributs des entités associées à un nœud dans le plan peuvent être déduits soit de l'opérateur, soit de la structure du plan lui-même. Mais il y a beaucoup d'autres sortes d'informations utiles qui pourraient être associées à un nœud individuel. Aucunes n'ont jamais été incluses dans le système* NOAH *parce qu'il n'y existe aucun mécanisme pour spécifier leur rattachement à des nœuds individuels."* [Sacerdoti 77], p 93.

Chapitre 9

Architecture et fonctionnement

Dans ce chapitre, nous présentons le planificateur à expertise COPLANER. Après avoir indiqué son organisation générale, nous nous attarderons sur ses capacités planificatrices, et sur la façon dont leur implémentation tire parti de la structuration de l'expertise suggérée par le chapitre 8.

Enfin, nous profiterons de cette double expérience (COPLANER, planificateur avec expertise, et YAPS, planificateur indépendant du domaine) pour suggérer une articulation de leurs techniques respectives.

9.1 Présentation du système COPLANER

9.1.1 Brève description

Le but du système COPLANER[1] est de proposer un planning d'exécution à partir de la description d'un bâtiment (HLM ou bureau) et de son environnement [Assémat et coll. 88], [Assémat et coll. 89]. Le bâtiment et son environnement sont représentés par des objets du langage orienté objets TG2 [Assémat & Morignot 87]. L'expertise complétant ce bâtiment et testant sa cohérence est représentée par un ensemble des structures d'inférences d'ordre 1. Chaque objet dispose physiquement de l'expertise qui le concerne.

Une session se déroule ainsi :

1. Décrire un bâtiment à réaliser ainsi que son environnement ;
2. Planifier sa réalisation, i-e simuler le comportement de notre expert en construction face à sa carte.

Des données globales permettent à l'utilisateur d'évaluer la qualité du planning (nombre de mètre-cubes de béton consommé, ...). Couramment, il relance la planification en ayant

[1] Le système COPLANER a été développé par la société Cognitech dans le cadre du programme IN.PRO.BAT (INformatique PROductique et BATiment) du MELATT (Ministère de l'Equipement, du Logement et de l'Aménagement du territoire et des Transports).

légèrement modifié les données (combien de jours gagnerai-je si j'enlève un étage ? si j'enlève les 50 cm de plafond pour la galerie ?).

9.1.2 Organisation

La tâche de COPLANER est décomposée en trois étapes lors de chaque itération (voir figure 9.1) :

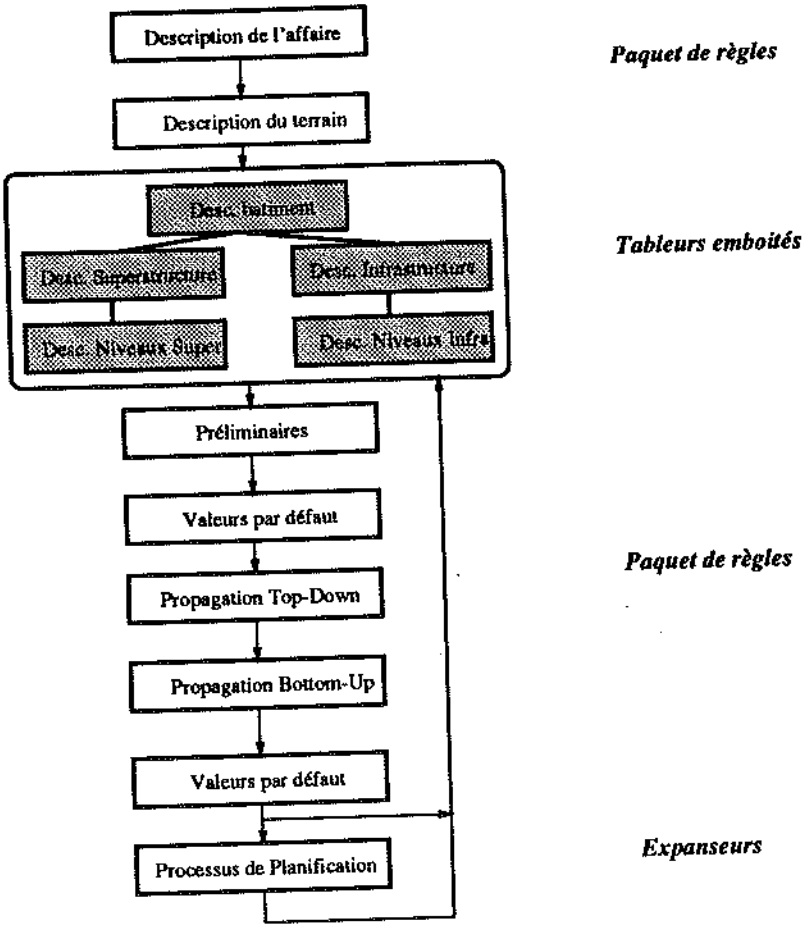


Figure 9.1: Les phases de traitement dans COPLANER

Acquisition Elle consiste à créer les instances des classes représentant le bâtiment, ses composants et son environnement, à renseigner les attributs propres de chaque objet et à mettre en place les liens entre instances (composition, géométrie, ...).

Estimation des valeurs manquantes / détection d'incohérences Les objets sont classés par *hiérarchie* : affaire, terrain, bâtiment, infrastructure / superstructure, niveaux / fondations. Les attributs géométriques inconnus recoivent une valeur issue de la représentation de l'expertise attachée à leur classe. Si besoin est, un objet d'un niveau instancie les objets du niveau hiérarchique inférieur, dont il aura le contrôle (propagation top-down).

Lorsque les objets du dernier niveau hiérarchique ont été complétés et testés, chaque niveau transmet ses résultats aux objets hiérarchiquement supérieurs (propagation bottom-up). L'ultime objet "affaire" possède ainsi les données globales caractérisant le futur chantier.

COPLANER peut à ce niveau retourner en phase d'acquisition lorsqu'il a épuisé tous ses modes d'inférences sans avoir pu évaluer certains attributs.

Génération du plan Les objets les plus bas hiérarchiquement (les composants de bâtiment les plus détaillés) doivent être traduits en un mini-plan partiel chacun, dont l'exécution assurera la réalisation de l'objet. Cette expertise de planification est répartie sur chaque classe au moyen d'*expanseurs*, structure d'inférences se chargeant de cette traduction de la représentation structurelle en représentation opérationnelle. Chaque mini-plan partiel est détaillé en fonction des caractéristiques de l'objet dont il provient. Puis ces mini-plans complétés sont agrégés en un plan global : les *liens* entre mini-plans sont inférés.

Une fois le planning stabilisé, COPLANER fournit un planning ingérable par un logiciel de PERT du commerce.

9.1.3 Implémentation

L'implémentation de *Coplaner* est complètement articulée autour du langage orienté objet TG2 pour deux raisons :

- *un bâtiment se représente bien sous forme d'objets* :

  Etant donné le nombre d'objets (physiques) différents et leur taille (et surtout les financements colossaux engagés), les experts adoptent d'eux-mêmes une démarche très méthodique basée sur une décomposition hiérarchique (cf. 8.2.1). Le plus haut niveau est l'"affaire", par exemple, qui peut regrouper plusieurs chantiers géographiquement éloignés. Il nous a suffi de comprendre et recopier cette hiérarchie, sans grande modification de fond.

- *un code volumineux comme* COPLANER *est facilement géré sous forme d'objets* :

  Plusieurs modes d'inférences se sont avérés nécessaires : règles d'inférences (ordre 1, chaînage avant), mais aussi réflexes (démons), contraintes algébriques, ... *Coplaner* comprend des sous-logiciels importants et différents, tableur (acquisition) et système de gestion de base de données (tâche des expanseurs au format des experts).

  Tout représenter systématiquement sous forme d'objet uniformise la communication entre ces entités de nature très diverses, ce qui facilite l'écriture et la mise au point du contrôle. Par exemple, une règle d'inférence est une instance de la classe *"règle"*, le fonctionnement instantané du moteur est décrit au moyen d'une instance de la classe *"moteur"*, le sgbd est une instance de l'objet "SQL-INFORMIX", une requête à ce sgbd est un envoi de message à cet objet et stocke ses résultats dans une instance de *"requête"*, ...

  D'autre part, l'uniformité de *structure* permet de répartir ces entités différentes sur les composants de bâtiment qui les concerne (contrôle réparti) :

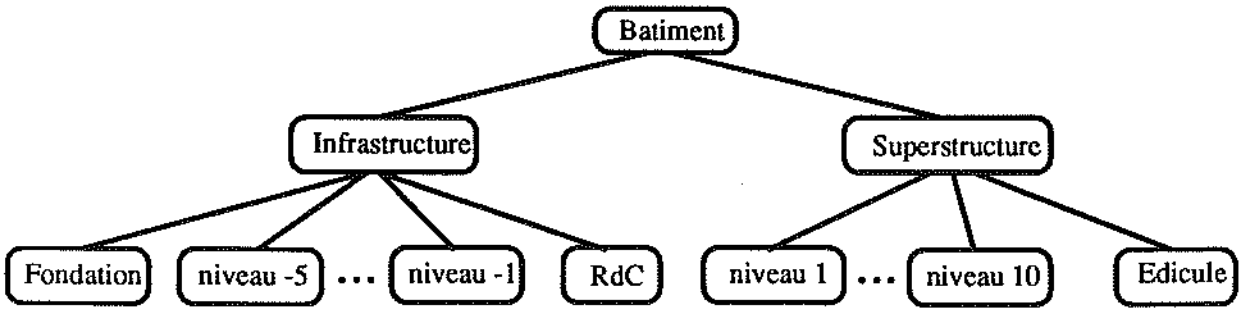
9.2 Planification

9.2.1 Représentations structurelle et opérationnelle

Représentation structurelle Comme indiqué en 8.2.2, la représentation *structurelle* est la description du bâtiment à construire et de son environnement. C'est elle qui est renseignée par la phase d'acquisition, et qui est complétée et testée dans la deuxième phase.

Le groupe d'objets représentant un bâtiment particulier est composé d'instances des classes *"bâtiment"*, *"infrastructure"*, *"superstructure"*, *"niveau"* (en infra- ou en super-structure), *"fondation"* et *"édicule"*. Chaque classe contient l'expertise la concernant (règles d'inférences, contraintes arithmétiques, réflexes, mais aussi expanseurs — voir ci-après).

Représentation structurelle



Représentation opérationnelle

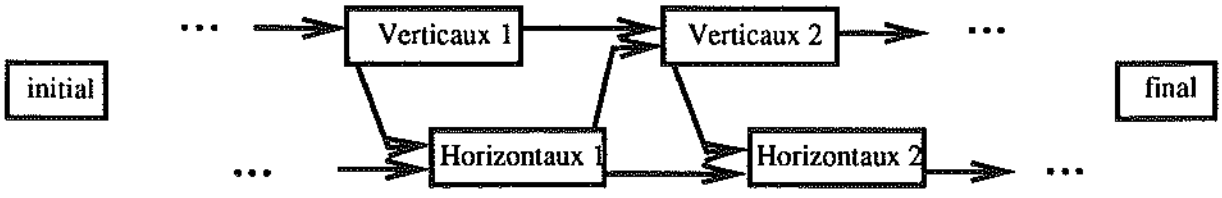


Figure 9.2: Représentations structurelle et opérationnelle

Les instances particulières sont créées en parcourant ces classes en top-down gauche-droite (voir figure 9.2) ; pour chaque classe vue, l'utilisateur spécifie, au moyen d'interfaces récursivement emboitées (éditeurs d'objets sous forme de tableur), le nombre d'instances et ses caractéristiques propres. Le système complète alors les liens de composition (correspondant au niveau de détail) et de juxtaposition (lien bas-haut).

Cette représentation correspond à celle qui serait utilisée dans un système-expert classique (la

base de fait), la distinction classe / instance correspondant à la distinction expertise / données en utilisation.

Représentation opérationnelle La représentation opérationnelle contient le planning, qui sera modifié jusqu'à obtention d'une solution. Il est généré (en deux fois, voir 9.2.2) lorsque la représsentation structurelle est complètement validée par l'expertise.

Les objets de cette représentation sont donc simplement des tâches (nom, durée, date de début, date de fin, marge) et des liens temporels (coefficients α, β, γ de 2.2).

9.2.2 Expanseurs

Nous appellons *expanseur* la structure d'inférences qui permet de déduire l'opérationnel du structurel. Il répond à un double but : d'abord, indiquer comment *réaliser* chaque composant de bâtiment (mini-plan partiel) ; ensuite, indiquer comment *intégrer* ce mini-plan dans le plan global.

A partir d'un composant atomique, un expanseur exprime :

- à l'intérieur d'un mini-plan :
  - l'existence des tâches,
  - les caractéristiques en situation de chacun de ces tâches,
  - leurs liens temporels (internes) ;
- entre le mini-plan et le reste du plan global : les liens temporels (externes) entre des tâches du mini-plan et d'autres tâches du plan global.

Fonctionnement interne (élection / instanciation) Pour ce qui concerne la construction même du mini-plan (premier point), le comportement d'un expanseur se rapproche de celui d'une règle d'inférences dont la partie gauche se situerait dans le monde structurel (objets) alors que la partie droite se situerait dans le monde opérationnel (tâches).

Un mini-plan est généré en trois phases (voir figure 9.3) :

- *Choix de l'ossature :* L'expertise montre que, pour un composant donné de bâtiment, seules quelques techniques de construction (donc quelques mini-plans typiques) sont envisageables (moins de 5). La *méthode*[2] *de choix* de l'expanseur (règles d'inférences) se charge d'élire l'un de ces plans partiels.
- *Complétion de l'ossature :* L'expertise montre également qu'il existe des *motifs (répétitifs) de tâches* dans un plan partiel, le nombre de motif, et leur raccordement en terme de liens temporels, dépendant d'une caractéristique du composant. La gestion de la création et de l'enchaînement des motifs est assurée par une *tâche fictive itérative*, dont le comportement

[2] *"méthode"* au sens des Langages Orientés Objets.

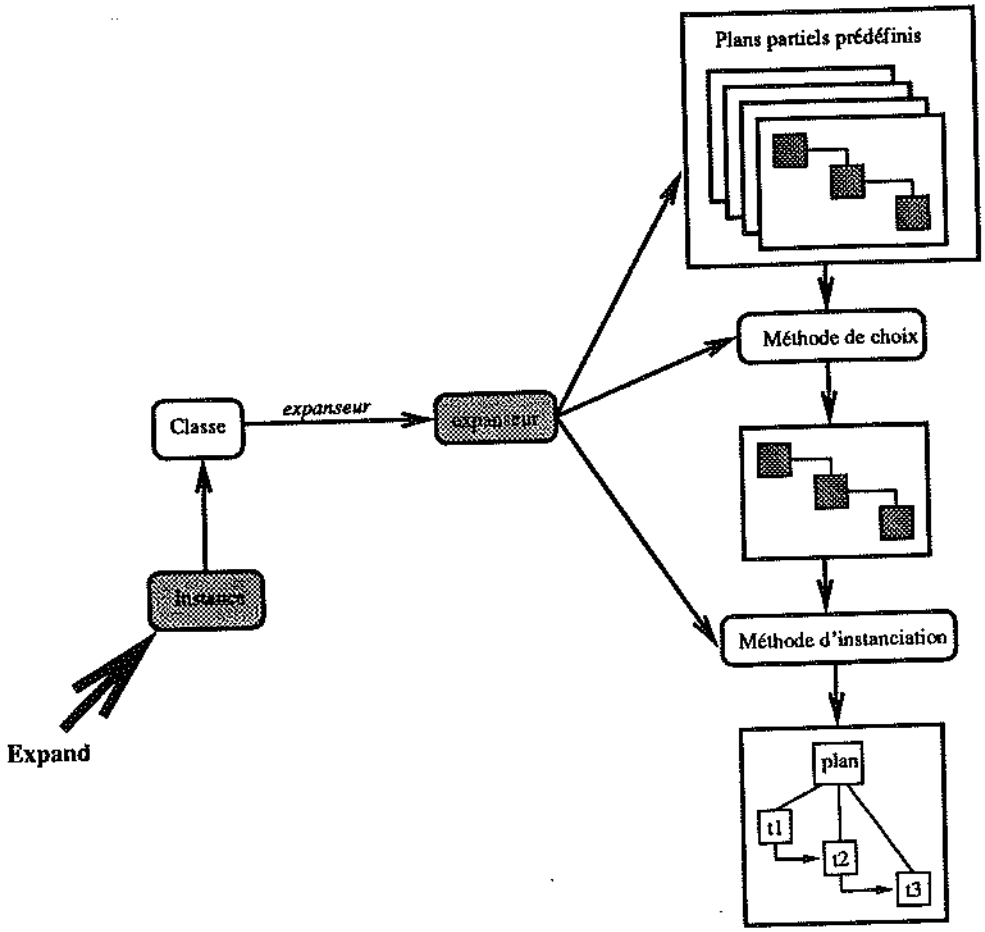


Figure 9.3: Développement d'un expanseur

est décrit par la procédure TÂCHE-FICTIVE-ITÉRATIVE (pour une tâche fictive itérative t, une condition de répétition c et un motif m) :

Procédure TÂCHE-FICTIVE-ITÉRATIVE(t, c, m)
Tant que la condition c est vraie, **faire**
Créer une nouvelle instance du motif (tâches et liens)
Intercaler ce motif juste avant la tâche fictive.
Effacer cette tâche fictive (devenue inutile)

Cette tâche fictive itérative indique l'endroit où le motif doit être successivement introduit dans le plan partiel (la variable courante de la boucle "tant-que").

- *Complétion de chaque tâche* Les attributs des tâches et liens sont précisés (règle d'inférences) : durée, cadence, ressources, ...(pour une tâche), durée, type, ...(pour un lien).

Fonctionnement externe (intégration) La phase d'intégration du mini-plan est ramenée à la recherche d'une des extrémités des liens temporels externes (l'autre extrémité du lien étant

connectée à une tache interne au mini-plan). Ces extrémités libres, appelées *tâches fictives de recherche*, sont des patterns sur les tâches du plan global, qui devront être remplacés par une autre tâche issue de l'expansion d'un autre composant de bâtiment. Comme dans un problème de références croisées, ces tâches fictives de recherche sont résolues *après toute génération de mini-plan par tous les expanseurs*, par une deuxième passe sur la classe des tâches. En pratique, un pattern d'une tâche fictive de recherche est une partie gauche d'une règle d'ordre 1 du moteur.

Finalement, un expanseur est donc :

- attaché à un type de composant (une classe d'objets de la représentation structurelle)
- qui contient :
  - l'ensemble des sous-plans généralement applicables à ce type de composant :
    - mini-plans bruts ;
    - mini-plans calculés par motifs itératifs.
  - une méthode de choix d'un mini-plan,
  - une méthode attribuant les caractéristiques en situation aux tâches de ce mini-plan,
  - les descriptions des extrémités des liens externes.
- qui est activable selon les deux étapes Election et Instanciation.

Le cas dual, dans lequel ce ne serait plus un *nœud* de la représentation structurelle (composant de bâtiment), mais un *lien* (géométrique, hiérarchique), qui serait traduit en son image opérationnelle, devrait être envisagé de façon analogue. Un lien de la représentation structurelle se traduirait en un lien (temporel) de la représentation opérationnelle entre mini-plans : c'est exactement le rôle joué par les liens temporels externes implémentés par les *tâches fictives de recherche*, aussi ces expanseurs de *liens*, déjà virtuellement implémentés, ne nécessitent pas de structure de données supplémentaires.

9.2.3 Contraintes algébriques

Les contraintes algébriques sont un des modes d'inférences de TG2. Une telle contrainte relie un nombre quelconque d'attributs d'objets par une formule mathématique simple [Steele & Sussman 79]. Par exemple, une instance de la contrainte :

$$\textit{altitude-etage}\,(n+1) = \textit{altitude-etage}\,(n) + \textit{hauteur-etage}\,(n)$$

relie systématiquement l'attribut `altitude` de l'objet représentant un niveau aux attributs `altitude` et `hauteur` de l'objet représentant le niveau inférieur.

L'intérêt d'une contrainte générique est d'abord provient d'abord d'un problème de place-mémoire : une contrainte "$x = y + z$", par exemple, évite d'avoir à représenter trois règles d'inférences : "si x et y sont connus et si z est inconnu, alors $z = x - y$", et de même pour déduire x et y. Ensuite, une contrainte algébrique réagit :

- en *induction*, ssi toutes les variables de la formule sont connues sauf une ;
- en *déduction*, ssi toutes les variables sont connues et qu'une change de valeur.

Dans ce dernier cas, si la contrainte n'est plus vérifiée, un réflexe de contrôle de la contrainte peut se charger de maintenir la valeur, ou propager la modification sur une autre variable.

Autre exemple, le mécanisme des contraintes algébriques implémente extrèmement simplement une propagation de dates (au plus tôt, au plus tard, cf. l'algorithme PROPAGER-DÉBUT de 2.2) pour une évaluation finale (type PERT).

9.3 Réflexion sur la planification et l'expertise

Pour générer un plan dont l'exécution réalisera un objectif fixé, deux approches sont possibles, que l'on peut grossièrement résumer ainsi :

- si **on sait déjà** comment réaliser cet objectif (approche experte, macroscopique), alors la difficulté est de distinguer le motif de connaissance adéquat dans l'ensemble des connaissances disparates dont on a la chance de disposer. C'est exactement à ce problème de structuration des connaissances préexistantes que répondait la structure des *frames* de Minsky [Minsky 75], et auquel répond la structure des *expanseurs* dans COPLANER.
- si **on ne sait pas déjà** comment réaliser cet objectif (approche calculatoire, microscopique), il faut commencer par découvrir une façon de le réaliser, en raisonnant sur l'observation des détails des actions dont on dispose par exemple (*schémas d'actions* dans YAPS), avant de chercher à optimiser quoi que soit.

Sous cet angle, ces deux approches sont au moins complémentaires, en un outil générant un plan puis l'optimisant [Levitt & Kunz 87], voire même déductibles l'une de l'autre, l'expertise apparaissant comme le souvenir des hypothèses et de la conclusion d'un calcul dont le détail a été oublié (apprentissage) et qu'il serait d'ailleurs inutile et coûteux de refaire à chaque fois.

L'intégration des deux approches pourrait s'envisager de deux façons, selon le point de départ choisi :

- *Intégrer de l'expertise dans* YAPS : les liens entre actions, issus de considérations expertes, ne seraient pas justifiés en termes calculatoires (autres que les modes d'inférences experts échappant au raisonnement du planificateur). A ce titre, ils seraient *inamovibles*, et nécessiteraient un critère de vérité particulier dont le rôle ne serait qu'entériner les choix effectués par le module expert, oracle parfait.
- *Intégrer du calcul dans* COPLANER : la description formelle des tâches permettrait soit d'inférer a priori des liens temporels entre tâches, lorsque l'expertise laisse un degré de liberté, soit de contrôler a posteriori (*"contrôle qualité"*) la bonne réalisation de chacunes des tâches. Ces deux applications nécessiteraient d'expliciter chaque tâche selon un formalisme de description d'actions (comme le nôtre), qui aboutirait à une méthodologie de recueil d'expertise.

Il appartiendra à des travaux ultérieurs de confirmer ou d'infirmer ces suggestions d'intégration. Nous prétendons seulement brosser une esquisse de recherche, basée sur notre double expérience.

Conclusion

Caractéristiques

Après avoir suivi la tradition classique en planification indépendante du domaine, comprenant la définition d'une sémantique à partir de quelques exemples et sa généralisation ex abrupto, nous avons bien dû constater l'impossibilité pratique d'évolution de telles approches empiriques, volumineuses, disparates et certainement asymptotiques.

Ce souci de clarification nous a poussé à expliciter de prime abord le *critère de vérité* adopté par un planificateur, comme unique moyen de donner une signification à des activités telles que la "détection" ou la "résolution" de "conflits" entre actions, qu'elles soient en parallèle ou en *série*.

Nous exploitons les contraintes de non-linéarité et d'ordre 1, adoptées sur le graphe d'actions, pour retarder la construction des choix possibles, en trouvant un moyen de les exprimer implicitement (principe déjà appliqué dans des domaines allant de la manipulation formelle d'équations au traitement du langage naturel). Une représentation facilement calculable, et quand même suffisamment informante, de telles *absences de choix*, ambiguïtés combinatoires, respecte le formalisme initial et s'y s'intègre, tant pour la non-linéarité vis-à-vis de la succession temporelle, que pour la variable vis-à-vis de l'objet.

Réalisations

Nous avons dû restreindre notre analyse à un formalisme simple de description des actions et à un critere logique simple, mais suffisant pour clarifier le processus de génération de plan, qu'il vise à l'indépendance d'un domaine d'application ou même qu'il se base sur une expertise.

Par rapport aux planificateurs classiques, notre approche présente les avantages suivants :

- uniformisation des modalités de déduction d'information (traitement surfacique, pouvant dériver en *base de données temporelle*) ;
- taxinomie exhaustive et rationnelle (vs. empirique) des amendements possibles du graphe d'actions ;
- isolement des heuristiques de contrôle (séparation physique du planificateur et de son méta-planificateur).

La réalisation d'un planificateur complet, basé sur l'implémentation de ce critère particulier, permettant de retrouver de façon unifiée les résultats classiques (issus d'approches empiriques *nettement plus lourdes*), nous conforte dans cette approche.

Un deuxième planificateur, expert en chantiers de bâtiments, dont la structuration de l'expertise peut aussi s'interpréter selon un critère de vérité (symbolique, cette fois), a donné des résultats conformes à ceux d'un expert humain.

L'hybridation reste à réaliser, si elle n'est pas utopique : le planificateur pur n'est appliqué pour l'instant qu'à des micro-mondes, et la description des tâches du planificateur expert en terme d'actions a été jugée inutile par l'expert dans un premier temps. De même, nous avons dû renoncer pour l'instant à décrire un grand nombre de phénomènes perceptibles tels que la continuité de transformation, le partage ou l'exclusion de moyens, ou la hiérarchie de détail, qu'il serait nécessaire de maîtriser pour prétendre comprendre *pourquoi nous effectuons une chose avant une autre*. Il nous a bien fallu assumer ces deux types de restrictions pour satisfaire d'abord notre double objectif relatif à l'ordre 1 et à la non-linéarité d'une part, et relatif à l'expertise d'autre part.

Perspectives

Outre l'aspect relatif à l'expertise (cf. 9.3), l'extension du planificateur pur s'effectuerait naturellement vers un formalisme et des critères moins rigides (dont STRIPS est coupable), et vers une souplesse toujours plus grande du contrôle (voire dissocié du planificateur, devenu *boîte à outils* de planification).

D'abord, il reste à découvrir et à faire coexister des critères de vérité autres que notre critère logique (minimal), le critère *numérique* (définissant une propagation simple de type PERT) ou le critère logique à un *coup* (introduisant les ressources), pour se rapprocher des diverses visions imaginables qu'aurait un observateur projeté dans notre graphe d'actions.

Ensuite, des trois axes d'abstraction possibles (objet, temps et calcul), celui sur le calcul, qui correspondrait à des *actions à contexte*[3], peut difficilement être représenté dans notre planificateur (une simulation n'étant qu'un moyen d'écrire une exécution, sans plus) ; le défi réside ici dans la découverte d'un tel nouveau formalisme permettant l'expression des critères de vérité déjà élucidés. De même semble maintenant s'imposer la perversion du modèle présenté, par l'ajout de primitives à effets de bord, échappant au champ de vision du critère de vérité maître et sous la responsabilité de l'utilisateur.

Cette structuration progressive d'un planificateur, amorce manuelle de l'organisation d'un système visant à l'autonomie, tend à laisser le champ libre et clair au *méta-planificateur*, qu'il soit heuristique, algorithmique ou autre, et dont *nous ne pouvons douter* qu'il sera un jour réflexif.

[3]Le mécanisme des *opérateurs déductifs* constitue à ce jour la seule et unique tentative, même totalement empirique, en ce sens [Wilkins 84].

Annexes

Annexe A

Logiques des intervalles Allen

A.1 Représentation planaire des relations entre intervalles

La représentation planaire [Rit 88] des relations de Allen considère un intervalle comme un point du plan, de coordonnées (d, f), où d est la date début de l'intervalle et f sa date de fin. La contrainte d'intégrité $d \leq f$ interdit à un intervalle de se trouver sous la première bissectrice des axes. L'intérêt de cette notation est qu'il est possible de représenter les 13 relations temporelles de Allen par des relations *géométriques* simples, que l'on pouvait en fait poser (voir figure A.1) une fois pour toutes, l'interprétation en termes de dates ne servant qu'à la construction du modèle.

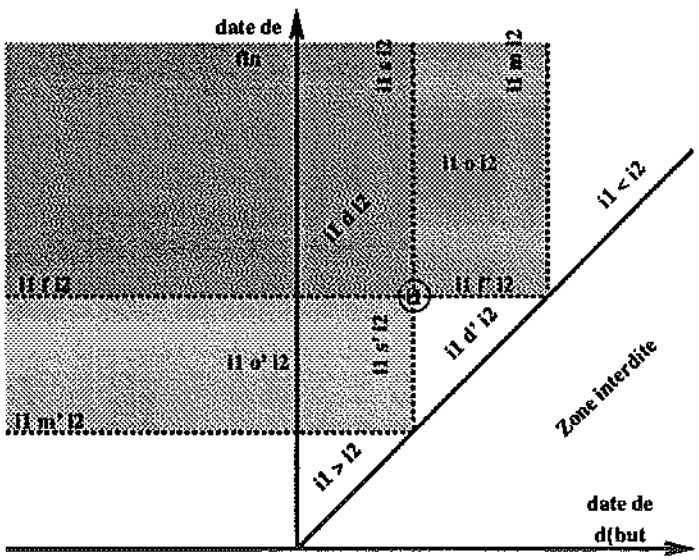


Figure A.1: Représentation planaire des treize relations de Allen

A.2 Composition de relations élémentaires

En mettant de côté l'identité (élément neutre pour $\circ$), Allen réduit les $13 \times 13 = 169$ cas à examiner a priori à $12 \times 12 = 144$. Bestougeff [Bestougeff & Ligozat 90] utilise la transposition et la symétrie pour réduire ces 144 cas aux 43 de la table ci-dessous.

La *transposition* est l'opération (notée r') définie par $i_1 \, R \, i_2 \overset{\text{def}}{\Longleftrightarrow} i_2 \, R' \, i_1$ et qui possède la propriété (générale) suivante $(r_1 \circ r_2)' = r_2' \circ r_1'$. La *symétrie*, (notée r^s) est la transformation obtenue lorsque le sens du temps est inversé ($>$ remplace $<$ sur les dates). Les douze relations ($=$ est mis à part) se transforment en leurs transposées, sauf d et d' qui ne changent pas ($d^s = d$, $d'^s = d'$), f et s qui s'échangent ($f^s = s$, $s^s = f$), ainsi que leur transposées f' et e' ($f'^s = s'$, $s'^s = f'$).

$\circ$	$<$	m	o	e'	s	d	d'	e	s'	o'	m'	$>$
$<$	$<$	$<$	$<$	$<$	$<$	m, o, s, d	$<$	$<, m, o, s, d$	$<$	$<, m, o, s, d$	$<, m, o, s, d$	Σ
m		$<$	$<$	$<$	m	o, s, d	$<$	o, s, d	m	o, s, d	$e, e', =$	
o			$<, m, o$	$<, m, o$	o	o, s, d	$<, m, o, e', d'$	o, s, d	o, e', d'	$o, e', s', d, d', e, s', o', =$		
s				$<, m, o$	s	d	$<, m, o, e', d'$	d	$s, s', =$			
e'					o	o, s, d	d'	$e, e', =$				
d						d	Σ					
d'						$o, e', s, d, d', e, s', o'$						

Table A.1: Composition des relations de Allen

Les compositions ne se trouvant pas dans la table s'y ramènent par transposition et/ou symétrie. Pour calculer $e' \circ >$, par exemple, le chemin menant à une composition du tableau est la suivante :

$$e' \circ > \xrightarrow{s} s' \circ < \xrightarrow{'} > \circ \, s \xrightarrow{s} < \circ \, e = (< mosd)$$

Annexe B

Définitions Usuelles

Ces définitions sont tirées du *Petit Robert, 1987.*

- *Planification*
  Organisation selon un plan. *La planification consiste à déterminer des objectifs précis et à mettre en œuvre les moyens propres à les atteindre dans les délais prévus (par une organisation administrative, technique, etc ...)*
- *Organiser*
  - Soumettre à une méthode, à une façon de vivre ou de penser.
  - Préparer (une action) pour qu'elle se déroule dans les conditions les meilleures, les plus efficaces
- *Plan*
  Tout projet élaboré, comportant une suite ordonnée d'opérations, destinée à atteindre un but.
- *Action*
  - Exercice de la faculté d'agir.
  - Ce que fait quelqu'un et par quoi il réalise une intention ou une impulsion.
  - Fait de produire un effet, manière d'agir sur quelqu'un ou quelque chose.
- *Temps*
  Milieu indéfini où paraissent se dérouler irréversiblement les existences dans leur changement, les événements et les phénomènes dans leur succession.

Annexe C

Manuel Utilisateur

L'approche directrice du codage de cet outil de planification est d'atteindre le plus grand niveau de clarté possible. La réécriture constante d'un code dans un but de clarification converge souvent vers l'élucidation de structures informatiques signifiantes, pour peu que son auteur croie en un sens esthétique de la programmation[1].

L'indépendance vis-à-vis des langages informatiques d'I.A. trop particuliers a été recherchée : cet outil ne dépend ni d'un moteur particulier, ni d'un module d'interface particulier. Dans ce même souci de transparence, il est écrit en Le_lisp 15.2x [Chailloux et coll. 86] (voir [Wertz 85] par exemple pour une introduction à Lisp en général), en faisant appel aux fonctions les plus communes. Un portage sur un autre dialecte nécessiterait l'écriture de quelques macro-fonctions simples (remaquillage syntaxique uniquement). Nous nous sommes seulement autorisés l'utilisation du package `defstruct`, simple définition de `RECORD`s pascaliens (se retrouvant dans tout dialecte lisp) à légère tendance objets, pour améliorer le partitionnement du code. Le style de programmation ne tient pas explicitement compte du fait que Le_lisp soit un interprète à liaison dynamique. Seules quelques variables semi-globales tirent parti de la liaison dynamique ; ces cas sont cependant clairement signalés en commentaire et peuvent facilement être réécrits pour un lisp à liaison lexicale (via la consommation d'une variable globale).

C.1 Chargement

Les fichiers de la table C.1, d'extension ".ll", définissent le planificateur. Le fichier `LLMakefile`, évidemment inspiré du `Makefile` de la fonction `make` d'Unix, définit les liens de dépendances entre ces fichiers venant de l'expansion des macro-fonctions écrasantes.

Pour charger le planificateur, charger le fichier `Makefile.ll` et évaluer `(makefile T)` sous le toplevel de Le_lisp. Les drapeaux de configuration ont la signification suivante :

Debug Traces intermédiaires et chargement des packages de test ;

Coherency Teste la cohérence de l'implémentation du graphe à chaque modification ;

[1] cf. la définition des *métaclasses* et leur codage en Le_lisp [Cointe 87], [Cointe 88].

Nodepth Annule les heuristiques sur les distances horizontales et verticales ;

No-Heuristic Annule le test de profondeur maximale du plan.

Par défaut, tous les drapeaux sont à `nil`, i-e les heuristiques sont chargées en mode exécution.

Fichier	Descriptif	Drapeaux utilisés
batch	Trace sur fichier	
bit-field	Manipulation de champs de bits	
declobbering	Recherche des masqueurs potentiels	
deductif	Critère de vérité (déduction)	
ensembliste	Manipulation d'ensembles	
environnement	Les environnements de liaison de variable	
establishing	Etablir un événement	
example	Des problèmes de blocs	No-Heuristic
macros	Fonctions-utilisateur de définition de plans	
meth-noeud	Méthodes `defstruct` des actions	Nodepth, Debug
meth-plan	Méthodes `defstruct` des plans	
meth-template	Méthodes `defstruct` des schémas d'actions	
objets	Définition des objets `defstruct`	No-Heuristic, Nodepth
setf	Le `setf` de Common-Lisp	
step	Définition du mode pas-à-pas	
toplevel	L'algorithme de contrôle général	Debug, Coherency, No-Heuristic, Nodepth
unification	Définition de la base d'unification (ajout, accès nécessaire et possible)	Debug
variable	Définition des variables	
video-edit	Définition de l'éditeur de plan	
yf	Le `if` de Ch. Queinnec	

Table C.1: Fichiers définissant le planificateur en Le_lisp 15.2x

C.2 Déroulement d'une Session

Une session sous Le_lisp15.2x se déroule de la façon suivante :

1. chargement des actions :

```
(defactschema (deplace ?x ?z ?y)
  (non (= ?z ?y)) (non (= ?z sol)) (non (= ?y sol)) (non (= ?x ?y))
  (sur ?x ?z) (libre ?x) (libre ?y)
  ->
  (sur ?x ?y) (non (sur ?x ?z)) (non (libre ?y)) (libre ?z)
)
```

La syntaxe générale de la fonction **defactschema** est la suivante (en BNF[2]) :

```
<Sexp>      ::= (defactschema <terme> {<nombre>} <terme>* -> <terme>*)
<terme>     ::= (<relation> <argument>*)
<relation>  ::= <constante>
<argument>  ::= <variable> | <constante>
<variable>  ::= ?<constante>
<constante> ::= <symbole_Lelisp> | <nombre>
<nombre>    ::= <entier_Lelisp> | <flottant_Lelisp>
```

Pour le mode-pas-à-pas, il est préférable de regrouper toutes les variables utilisées dans le titre (premier argument de **defactschema**). Le deuxième argument de **defactschema** est la durée d'exécution des actions qu'instanciera ce schéma d'actions (0 par défaut) ; si une action du plan final a une durée non nulle, un calcul de PERT (algorithmes PROPAGER-DÉBUT et PROPAGER-FIN de 2.2) sera effectué après la génération. Pour un terme, les relations "=" (binaire) et "non" (unaire) sont prédéfinies.

Le rôle de ces actions consiste à définir les transitions d'états dans le micro-monde modélisé. Une fois choisie une modélisation du micro-monde (i-e les noms et arités des symboles fonctionnels et les noms des constantes), ces actions sont chargées de définir les "règles du jeu" sur ce micro-monde : leur définition constitue le pendant dynamique du choix des prédicats (modélisation statique), et à ce titre doit être effectuée en premier dans le déroulement d'une session.

2. chargement d'un plan initial :

```
(defplan (a b c sol)
   (libre c) (sur c a) (sur a table) (libre b) (sur b table)
  ->
   (sur a b) (sur b c))
```

La syntaxe de la fonction **defplan** est la suivante :

```
<Sexp>      ::= (defplan (<constante>*) <terme>* -> <terme>*)
<terme>     ::= (<relation> <argument>*)
<relation>  ::= <constante>
<argument>  ::= <variable> | <constante>
<variable>  ::= ?<constante>
<constante> ::= <symbole_Lelisp> | <nombre_Lelisp>
```

[2]En Le_lisp, il est d'usage d'adopter les conventions suivantes pour la définition d'une syntaxe :

- un nom entre un "<" et ">" est un motif de la syntaxe ;
- "<xxx> | <yyy>" signifie "le motif <xxx> ou le motif <yyy>" ;
- {<xxx>} signifie "0 ou 1 motif <xxx>" ;
- <xxx>* signifie "0 motif <xxx> ou plus" ;
- <xxx>+ signifie "1 motif <xxx> ou plus" ;

Le premier argument de `defplan` est la liste des constantes du problème (non nécessaire). Un terme peut contenir les deux relations prédéfinies = et non.

La définition du plan initial a pour but de définir la situation initiale (ou état initial) et la situation finale (ou état final) du micro-monde, toujours avec les prédicats retenus.

Le fichier `example.ll` charge des schémas d'actions modélisant le célèbre micro-monde des cubes, et permet un chargement simple de problèmes dans ce monde.

3. mise en place du mode de contrôle : comme tout moteur, le planificateur peut fonctionner selon différents modes, allant de l'automatique complet au pas-à-pas le plus fin. Ceci consiste à définir le mode de pas-à-pas (si désiré) ainsi que le type d'informations affichées, grâce à la fonction `control-mode`.

4. déclenchement de la phase de planification : la fonction `control` lâche le planificateur sur le dernier plan fourni. Le planificateur s'arrête dans deux cas :

   (a) lorsqu'une solution a été trouvée.

   (b) lorsque toutes les branches ont été explorées et qu'aucune n'a fourni de solution : le planificateur a ainsi démontré l'infaisabilité du plan initial fourni.

   Une fois une solution trouvée, il est possible de relancer le planificateur à la recherche de la solution suivante par la fonction `recontrol`, et ce, jusqu'à l'exploration de toutes les branches (de méta-planification).

C.3 Les Outils de Mise au Point

C.3.1 L'éditeur de Plan

Suivre patiemment l'évolution d'un plan, en se basant uniquement sur la *liste des prédécesseurs* et la *liste des successeurs* de chaque action, devient insupportable lors d'une utilisation intensive du planificateur sur des plans de plusieurs dizaines d'actions : on passe presque plus de temps à chercher à *visualiser* le plan qu'à en comprendre la logique.

L'éditeur de plan présente graphiquement cette même information pour une action, ce qui correspond mieux à l'intuition ; il permet de se déplacer de proche en proche dans le plan et fournit des informations annexes telles que le contenu de l'action courante ou la base des contraintes d'unification. Cet éditeur est défini en tant que *méthode*, ce qui est une forme d'appel particulièrement pratique, le rendant facilement utilisable en mode pas-à-pas (cf. C.3.2).

L'éditeur de plan permet d'observer (inspecter) un plan en se déplaçant d'action en action, tout en ayant constamment sous les yeux les prédécesseurs et successeurs de l'action en cours d'inspection (philosophie "*tel écran, tel écrit*"[3]). Il est ainsi possible de se déplacer d'action en action par le lien "est-successeur-de" ou "est-prédécesseur-de" en déplaçant le curseur, indiquant l'action courante, à l'aide des touches du clavier.

[3]Francisation standardisée du WYSIWYG (*What You See Is What You Get*).

e
a

été initia-
uel tous les
e aucun rôle
r leur valeur
nte).

pect de la classe

direction
uche (Backward)
bas (Next)
haut (Previous)
droite (Forward)

tions et postconditions

es variables par leur valeur.

sponibles).

C.3.2 Le mode *pas-à-pas*

Par analogie avec le stepper de Le_lisp, un stepper pas-à-pas a été défini pour le planificateur, permettant d'observer au plus près le fonctionnement interne du planificateur. La syntaxe et la présentation est d'inspiration lelispienne évidente.

Le déroulement d'une session en mode pas-à-pas est indiqué au planificateur grâce à la fonction `control-mode` (cf. partie II), permettant de paramétrer le fonctionnement du planificateur. Pour un mode pas-à-pas complet (arrêt à chaque pas de planification), appeler la fonction précédente (`control-mode`) avec un argument à `T`, i-e sous toplevel, faire : `(control-mode T)`.

Aspect et Commandes Le planificateur s'arrêtera à chaque étape de son fonctionnement, c'est-à-dire principalement aux phases suivantes :

- après la détection de chaque problème ;
- après avoir trouvé une solution au problème courant.

Un *pas* de planification est constitué par l'exécution d'un cycle complet *détection / résolution* d'un problème. Le stepper a donc une résolution d'un demi-pas de planification : l'utilisateur peut observer les raisons qui ont amené le planificateur à identifier un problème donné (observation dite *entrante*), ainsi que les raisons qui ont amené le planificateur à le résoudre d'une certaine manière (observation dite *sortante*).

Pratiquement, à chaque pas de fonctionnement, le stepper affiche une ligne de format : `n direction problem/action step> ?` où :

- `n` est le numéro du "tic" d'horloge, qui correspond à la profondeur de la pile des buts non résolus.
- `direction` est l'un des deux symboles `>>` (*"observation entrante"*) ou `<<` (*"observation sortante"*).
- `problem/action` est le libellé du problème (resp. action) dans l'observation entrante (resp. sortante) qui vient d'être détectée (resp. effectuée).

Le stepper attend alors un caractère, qui peut être :

RETURN Va jusqu'à l'arrêt suivant.

< Passe n cycles problèmes/actions, où n est le contenu de la variable `#:plan:step:pass-count`.

b (En observation sortante uniquement) Déclare que la solution trouvée par le planificateur au problème courant n'est pas correcte. Le planificateur effectue un *retour-arrière* sur la solution suivante.

. Réaffiche la dernière impression.

= Entre dans un sous-toplevel d'inspection.

h Réaffiche la succession des problèmes / actions depuis le début du mode pas-à-pas.

q Quitte le mode pas-à-pas.

? Affiche un texte d'aide sur les présentes commandes.

Toplevel d'Inspection La méthode d'observation la plus fine consiste à passer à chaque pas dans le **sous-toplevel d'observation** (commande =). Ce sous-toplevel se différencie du toplevel Le_lisp par son prompt >>? (resp. <<?) pour un sous-toplevel correspondant à une observation entrante (resp. sortante).

Le comportement de ce sous-toplevel de pas-à-pas est quasiment identique à un toplevel normal (boucle `READ-EVAL-PRINT`) à la différence près que le premier caractère peut être interprété comme une commande.

Les commandes-caractères sont les suivantes :

i Inspecte le plan courant : l'éditeur de plan est lancé sur le plan courant. Ceci est assurément le mode le plus agréable pour observer toutes les caractéristiques du plan en cours de construction. A la sortie de ce mode, l'utilisateur est toujours sous le même sous-toplevel d'observation.

v Imprime les noms et valeurs des variables internes du planificateur, jugées intéressantes (i-e significatives). Ce mode est surtout intéressant lorsque l'utilsateur veut modifier les heuristiques de résolution ou l'ordre de leur observation par le moteur.

g Affiche la pile des buts en cours de résolution.

s Affiche la pile des actions correspondant aux buts en cours de résolution.

h Affiche la pile complète du planificateur. Cette pile est une liste de couples (`<probleme>` `<action>`) où chaque `<action>` est la solution actuelle (i-e sans préjuger des phases de retour-arrière postérieures) du `<probleme>` correspondant.

q Sort de la boucle d'inspection et retourne sous le stepper.

? Affiche un texte d'aide sur les présentes commandes.

Si la S-expression lue par ce sous-toplevel d'observation n'est pas une des commandes-caractères décrites ci-dessus, le sous-toplevel retrouve son comportement standard (`READ-EVAL-PRINT`).

Annexe D

Code de leloo

Le langage orienté objet `leloo` est une version expurgée et remaquillée du cœur de T-G2 [Assémat & Morignot 87], d'inspiration `ObjVlisp`-ienne [Cointe 87], dans un but pédagogique. `leloo` n'en reprend pas la notion de *variable de classe* (les noms des variables d'instances d'une classe ne sont pas *directement* visibles par le transmetteur), implémente un *héritage multiple* "largeur d'abord, gauche-droite", le tout en 300 lignes utiles en `lelisp15.x`.

En `leloo` comme en `ObjVlisp`, le monde des instances est séparé en instances *génératrices* et *terminales*, selon que l'instance peut ou non voir la méthode `Cree` de la classe `Metaclasse`. Le monde des instances génératrices est séparé en *méta-classes* et *classes*, selon que l'instance génératrice crée ou non d'autres instances *génératrices*. Ces définitions sont rendues cohérentes grâce aux aberrations[1] que sont la (méta-)classe `Classe`, réflexive par le lien d'instanciation (voir figure D.1) et porteuse de la méthode `Cree`, et la classe `Objet`, réflexive par le lien d'héritage, et porteuse des méthodes les plus communes (`Affiche`, ...).

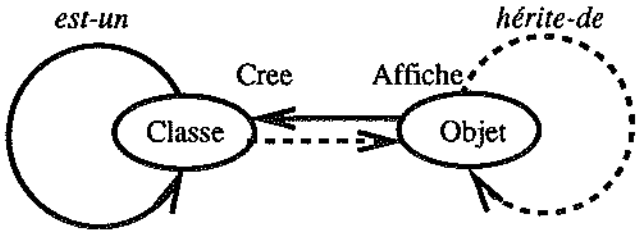


Figure D.1: Graphes initiaux d'instanciation et d'héritage en leloo

Une définition propre des *attributs* nécessiterait une troisième classe *abberrante*, `Attribut`, réflexive par le lien de composition (voir figure D.2) :

L'attachement d'attributs à une classe s'effectue par le lien d'*héritage* ; le lien entre une classe et une instance met en évidence le lien d'*instanciation* ; enfin, une instance est une collection de valeur d'attributs, inaugurant le lien de *composition* comme troisième lien constitutif d'un Langage Orienté Objet. L'aspect fondamental de cette classe d'attributs se manifeste par une troisième réflexivité, concernant ce lien de composition : les instances de cette classe sont com-

[1] i-e il est difficile dans le monde réel de donner un sens aux classes `Classe` et `Objet`.

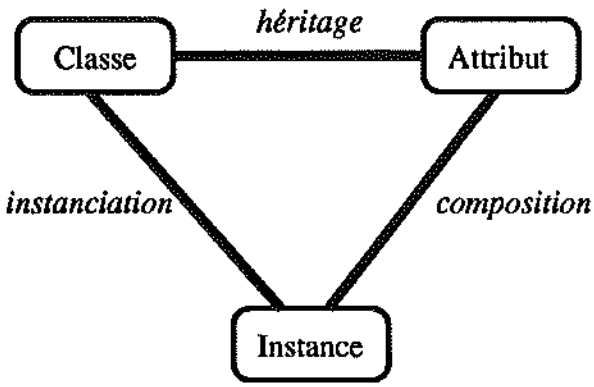


Figure D.2: Concepts fondamentaux d'un LOO et leurs relations

posés d'eux-mêmes (les *facettes* sont les attributs des instances de la classe *attribut* [Ferber 89], les attributs sont attributs d'eux-mêmes).

Toutes les fonctions définies ci-dessous se trouvent dans le package `loo` :

```
(setq #:sys-package:colon 'loo)
```

D.1 Toplevel

Lors de l'affichage des objets `leloo`, l'imprimeur de `lelisp` ne voit d'un objet que la structure de donnée l'implémentant et qu'il affiche (ou essaie d'afficher) effectivement. Cette impression de l'intérieur d'un objet est inesthétique et souvent même catastrophique (une avalanche de pages d'écrans) lorsque la structure de l'objet contient des références croisées. Bien que `lelisp` fournisse des outils minimaux (variable limitant l'impression des listes infinies, bibliothèque `cirlib` de gestion des circularités), nous avons préféré gérer l'impression des objets au niveau du toplevel `leloo` par l'ajout d'un `if` à la boucle standard `read-eval-print` qui substitue le `print` standard par un envoi direct de la méthode `Affiche` à l'objet. Nous profitons de ce sur-toplevel `leloo` pour gérer correctement les erreurs d'envoi de message.

```
;;; Les prompts d'entrée et de sortie.

(defvar :prompt-in  "loo")
(defvar :prompt-out "loo= ")

(de leloo ()
  (tag :*out-leloo*
    (with ((prompt (catenate :prompt-in (prompt))))
      (while T
        (lock (lambda (tag val)
                (ifn tag val
                  (selectq tag
                    (:error)                    ;;; Si erreur, on boucle ...
                    (:fin (exit :*out-leloo*))
                    (T))))
              (while T
                (let ((:resultat (eval (read))))
                  (prin :prompt-out)
                  (if (:ObjetP :resultat)       ;;; Tout cela pour ce test ...
                      (<- 'Affiche :resultat)
                    (print :resultat)))))))))
```

```
(de :error (:fct :mess :arg)
  (print "**(loo) " :fct " : " :mess " : " :arg)
  (exit :error))

(de fin () (exit :fin))
```

D.2 Choix de codage

Les objets Un objet `leloo` est codé par un vecteur, dont :

- le type contient la valeur de l'attribut `est-un` (le nom de sa classe) ;
- le premier champ contient les *composantes* de l'objet (liste des instances locales des superclasses) ;
- les champs suivants contiennent les valeurs des attributs locaux de la classe de l'objet.

```
;;; Crée un objet leloo qui est un est-un et dont les valeurs des attributs sont fournies
;;; dans la liste args.

(dmd :Cree-Objet (:est-un :args)
  '(let ((:i (apply 'vector ,:args)))
     (:est-un :i ,:est-un)
     :i))
```

Le prédicat associé `ObjetP` sépare les objets `leloo` des autres objets `lelisp`.

```
;;; Filtre les structures lelisp : si l'argument est un objet leloo, le renvoie.
;;; Renvoie nil sinon.

(de :ObjetP (:o)
  (and (vectorP :o)
       (let ((:nom-ig (:est-un :o)))
         (and :nom-ig
              (symbolP :nom-ig)))
       :o))
```

Lors de l'amorce, il faut pouvoir accéder physiquement aux valeurs d'attributs d'une instance génératrice `leloo`.

```
;;; Accès interne à l'attribut est-un (le nom de l'instance génératrice
;;; dont obj est instance).

(dmd :est-un (:obj . :val) '(typevector ,:obj . ,:val))

;;; Variable-fonction accédant aux autres champs de l'objet obj (via leur numéro n).

(dmd :Attr-Objet (:obj :n . :val)
  (if :val
      '(vset ,:obj ,:n ,(car :val))
    '(vref ,:obj ,:n)))
```

L'implémentation des objets ne pose problème que pour les valeurs des attributs hérités.

Dans une première approche, on considère que l'algorithme de résolution de conflits de noms par héritage définit implicitement un *tri* des attributs visibles par un objet, dont le calcul pourrait être effectué une fois pour toutes pour chaque instance génératrice (à leur création, par exemple). Cette implémentation, optimale vis-à-vis de la complexité en place des objets, nécessite de redéfinir les méthodes d'accès pour toute sous-classe, car, en héritage multiple[2], l'index de l'endroit physique où est stockée la valeur d'un attribut dans la structure représentant un objet varie avec la sous-classe. C'est l'héritage *statique* des méthodes d'accès (CLOS [Bobrow et coll. 87]).

Préférant l'héritage *dynamique* des méthodes (pour un langage pédagogique), nous implémentons un objet sous forme d'un graphe : une instance *i* d'une instance génératrice *ig* est (récursivement) composée de toutes les instances *super-i* des super-classes *super-ig* de *ig*. Au graphe d'héritage de *ig* correspond le graphe des composantes de *i* : chaque nœud du graphe des composantes est formé des valeurs des attributs locaux de l'instance génératrice homologue, issue du graphe d'héritage. L'index de l'endroit physique où est stocké la valeur d'un attribut local ne change alors, par définition, pas en fonction de la sous-classe. Seule entorse à ce duplicata, le graphe des composantes est arborisé : les branches redondantes (selon l'algorithme de résolution de conflit de noms) du graphe des composantes sont coupées.

[2] En héritage simple, cet index ne varie justement pas, si l'on décide d'ajouter les valeurs d'attributs locaux en bout de chaîne : les objets peuvent alors être implémentés par de simples structures linéaires (package `defstruct` de `lelisp` [Chailloux et coll. 86]).

```
;;; Crée une instance de l'instance génératrice ig de nom nom-ig.
;;; Utilise la variable (semi-)globale lnom-ig (liste des instances génératrices
;;; dont une instance locale a déjà été créée).

(de :creer (:ig :nom-ig)
  (cond
   ((memq :nom-ig :lnom-ig)                    ;;; Arborisation d'un DAG
    ())
   ((eq :ig (:ref-ig 'Objet))
    ())
   (T                                          ;;; Cas standard : on récurre ...
    (setq :lnom-ig (cons :nom-ig :lnom-ig))    ;;; Petit caillou
    (let ((:i (<- 'Cree-Local :ig)))
      (:Attr-Objet :i 0
                   (let ((i (mapcar (lambda (:snig)
                                      (:creer (:ref-ig :snig) :snig))
                                    (<- 'supers :ig))))
                     (if (and (null (car i))
                              (null (cdr i)))
                         ()
                       i)))
      :i))))
```

Références Le toplevel affiche une instance génératrice de nom `nig` sous la forme `#Rnig` (`R` comme référence, voir la méthode `Affiche` de l'instance génératrice `Classe` ci-dessous) ; la compatibilité lecture / écriture nous oblige à définir la dièse-macrofonction `#R` qui effectue la correspondance entre un symbole `lelisp` et la structure `lelisp` implémentant l'instance génératrice qui a pour nom ce symbole[3].

```
(defvar :*ig-marque* ':*ig-objet*)

;;; Renvoie l'instance génératrice de nom nig.

(defsharp |R| :unused '(:ref-ig ',(read)))

;;; Accède à une instance génératrice à partir de son nom.

(dmd :ref-ig (:nom-ig . :arg)
  (if :arg
      '(putprop ,:nom-ig ,(car :arg) ',:*ig-marque*)
    '(getprop ,:nom-ig ',:*ig-marque*)))
```

L'attribut `methodes` d'une instance génératrice contient la liste des *noms* des méthodes disponibles pour les instances, mais ne contient pas les *codes* de ces méthodes. Pour éviter de surcharger la liste des attributs, le code de ces méthodes est "caché" sur la P-liste du symbole représentant le nom de cette instance génératrice.

[3] L'hypothèse implicite de cette implémentation est que la correspondance *nom-de-classe* → *structure-de-classe* est non seulement surjective (toute classe possède un nom) mais injective : deux instances génératrices structurellement distinctes ont des noms distinctes. Ceci n'est pas évident a priori : on peut tout à fait imaginer que deux instances génératrices structurellement distinctes aient un même nom, mais qu'elles diffèrent par leur *position* dans le graphe d'héritage ; une instance génératrice ne serait pas (seulement) repérée par son nom, mais par un *chemin* y menant depuis la classe `objet`, par exemple.

```
(defvar :*ig-methode-marque* ':*ig-meth*)

;;; Accès global à la liste des corps des méthodes de l'instance génératrice de nom nom-ig.

(dmd :ref-methodes (:nom-ig . :arg)
  (if :arg
      '(putprop ,:nom-ig ,(car :arg) ',:*ig-methode-marque*)
    '(getprop ,:nom-ig ',:*ig-methode-marque*)))

;;; Ajoute le code code en tête de liste des codes de l'instance génératrice
;;; de nom nom-ig.

(dmd :ref-methode-add (:nom-ig :code)
  '(:ref-methodes ,:nom-ig (cons ,:code (:ref-methodes ,:nom-ig))))

;;; Renvoie, si elle le trouve, le corps de la méthode <nom-meth> de l'instance génératrice
;;; de nom nom-ig, moyennant la founiture de la liste des noms des-dites
;;; méthodes lnom-meth (pour éviter d'appeler le transmetteur).

(dmd :ref-methode? (:nom-meth :lnom-meth :nom-ig)
  '(any (lambda (:var :val)
          (and (eq :var ,:nom-meth)
               :val))
        ,:lnom-meth
        (:ref-methodes ,:nom-ig)))
```

Le transmetteur Le transmetteur `<-` envoie un message (`mess`) à un objet (`obj`) avec éventuellement des arguments (`arg1 ... argN`) ; la syntaxe d'une transmission est la suivante (pour les francophones) : `(<- <mess> <obj> <arg1> ... <argN>)`. Une transmission est réalisée de la façon suivante :

1. résolution de conflits de noms de méthodes : une méthode de nom `mess` est recherchée dans le graphe d'héritage à partir de la classe de `obj` par un parcours en *largeur d'abord*. Cet algorithme simple tend à privilégier les classes les plus spécialisées, qui sont les plus informantes[4].

2. évaluation de la méthode : La lambda-expression représentant la méthode ainsi trouvée est évaluée par un `apply` avec `arg1 ... argN` comme arguments. Dans le corps d'une méthode, la variable `moi` pointe sur l'objet-récepteur de départ `obj`, ce qui permet à un objet de s'envoyer des messages. Pour que les méthodes d'accès soient aussi applicables sur les instances des sous-classes, la variable `:moi` pointe sur le composant de l'objet `moi` correspondant à l'instance génératrice où a été trouvée la méthode.

[4] Un parcours en "largeur d'abord" n'est certainement pas le meilleur de ce point de vue, bien qu'il soit déjà préférable au trop simple "profondeur d'abord". Respecter complètement cette stratégie du "plus informant d'abord" nécessite de coder un *tri* qui soit une extension en ordre total de la relation d'héritage considérée comme un ordre partiel [Ducourneau & Habib 89].

```
;;; Envoie le message mess à l'instance MOI avec les arguments arg1 ... argN.

(de <- (:mess MOI . :args)
  (unless (:objetP MOI)
          (:error '<- "L'argument n'est pas un objet" MOI))
  (:yf (let ((:nom-ig (:nom-ig MOI)))
         (:lmethode (list MOI) :mess (list :nom-ig) (list :nom-ig)))
       (lambda (:meth)
         (apply (cdr :meth) (car :meth) :args))
       (:error '<- "Attribut ou methode inconnu" :mess)))

;;; Recherche en largeur d'abord de la methode de nom mess depuis les
;;; composants li, pseudo-instances locales des instances génératrices de
;;; noms lnig. La variable interne lnig- est la liste des noms des instances
;;; génératrices déjà' vues.
;;; Appel : (:lmethode (list :i) :mess (list :nom-ig) (list :nom-ig))
;;; Renvoie un cons (<instance-interne> . <lambda-expr>), ou nil.

(de :lmethode (:li :mess :lnig :lnig-)
  (or (any (lambda (:i :nig) (:lmethode-locale :i :nig (:ref-ig :nig) :mess))
           :li :lnig)
      (let (((:sli . :slnig) (:next-lmethode :li :lnig)))
        (when :slnig
          (:lmethode :sli :mess :slnig :lnig-)))))

(dmd :yf (:test :fct . :reste) ;; Le IF de Ch.Queinnec
  '(tag :*out-yf*
     (,:fct (or ,:test
                (exit :*out-yf* ,@:reste)))))

(de :nom-ig (:i)
  (cond
   ((:ObjetP :i) (:est-un :i))
   ((consP   :i) 'Cellule)
   ((null    :i) 'Nul)
   ((symbolP :i) 'Symbole)
   ((stringP :i) 'Chaine)
   ((vectorP :i) 'Vecteur)
   (T
    (:error '<- "OVNI" :i)))) ;;; Objet Volant Informatique Non Identifié ...
```

Le transmetteur <-+ diffère de <- en ce qu'il recherche la *deuxième* plus proche méthode correspondant au message et à l'objet. Ceci permet de définir incrémentalement une méthode à partir d'une autre plus ancienne (i-e située plus haut dans le graphe d'héritage).

```
(de <-+ (:mess MOI . :args)
  (:yf (let ((:nom-ig (:nom-ig MOI)))
         (:lmethode+ (list MOI) :mess (list :nom-ig) (list :nom-ig) T))
       (lambda ((:i . :lambda-exp)) (apply :lambda-exp :i :args))
       (:error '<- "Attribut ou methode inconnu" :mess)))

(de :lmethode+ (:li :mess :lnig :lnig- :firstP)
  (or (any (lambda (:i :nig)
             (:yf (:lmethode-locale :i :nig (:ref-ig :nig) :mess)
                  (lambda (:out)
                    (if :firstP
                        (setq :firstP ())
                      :out))))
           :li :lnig)
      (let (((:sli . :slnig) (:next-lmethode :li :lnig)))
        (:lmethode+ :sli :mess :slnig :lnig- :firstP))))
```

L'algorithme de parcours en *largeur d'abord* pour les transmetteurs est défini par les fonctions suivantes. Remarquons que le transmetteur <- est récursivement utilisé (pour supers et methodes), mais le risque de bouclage est évité (le diamètre du graphe d'instanciation est rarement plus de 4).

```
;;; Renvoie la liste des instances génératrices qui sont super-classes des
;;; instances génératrices lig de noms lnig.
;;; Les doublons sont éliminés (l'instance la plus à gauche reste) et les
;;; instances déjà rencontrées dans le parcours (qui sont stockées dans la variable
;;; semi-globale lnig-) aussi.

(de :next-lmethode (:lig :lnig)
  (let (:sli :slnig)
    (mapc (lambda (:i :nig)
            (unless (eq :nig 'Objet)          ;;; Pour éviter de boucler
              (mapc (lambda (:si :snig)
                      (unless (memq :snig :lnig-) ;;; Si pas déjà vu ...
                        (newl :sli :si)
                        (newl :slnig :snig)
                        (newl :lnig- :snig)))  ;;; Petit caillou
                    (or (:Attr-Objet :i 0)     ;;; Cas d'"objet"
                        (cirlist '*))
                    (<- 'supers (:ref-ig :nig)))))
          :lig :lnig)
    (cons (nreverse :sli) (nreverse :slnig))))

;;; Teste si la méthode de nom mess est locale à l'instance génératrice ig de nom nom-ig.
;;; Si c'est le cas, elle renvoie un doublet (obj . lambda), où obj est la composante
;;; de l'objet global qui est instance de ig et où lambda est la
;;; lambda-expression représentant le corps de la méthode. Sinon, elle
;;; renvoie nil.

(de :lmethode-locale (:i :nom-ig :ig :mess)
  (let ((:lmeth (if (eq :ig (:ref-ig 'Classe)) ;;; Pour éviter de boucler
                    (:Attr-Objet :ig 4)
                  (<- 'methodes :ig))))
    (and (memq :mess :lmeth) ;;; Recherche locale
         (cons :i (:ref-methode? :mess :lmeth :nom-ig)))))
```

D.3 Notations

La macro-fonction definst permet de créer et renseigner une instance (généralement terminale). L'appel (definst ref [symb] (symb1 Sexp1) ... (symbN SexpN)) crée une instance de l'instance génératrice ref et donne à l'attribut symb1 (resp. symb2, ... symbN) la valeur Sexp1 (resp. Sexp2, ... SexpN). Les noms d'attribut symb1 ... symbN ne sont pas évalués et l'instance créée est renvoyée. Si un argument intermédiaire symb est fourni, il s'agit d'un symbole lelisp dont la C-val recevra l'instance créée : (setq symb baby-inst).

```
(dmd definst (:ref-ig . :reste)
  `(let ((:i (<- 'cree ,:ref-ig)))                          ;;; Création
     ,@(when (and :reste (symbolP (car :reste)))            ;;; Cval possible
         `((setq ,(nextl :reste) :i)))
     ,@(mapcar (lambda ((:attr :val)) `(<- ',:attr :i ,:val)) ;;; Attributs
               :reste)
     :i))
```

L'expansion de cette macro-fonction peut se définir en termes d'envoi de messages de la façon suivante :

```
(definst #Rrectangle
  (Longueur 10)
  (Largeur  (+ 2 3)))
```

s'expanse en :

```
(let ((i (<- 'cree #Rrectangle)))
  (<- 'Longueur i 10)
  ;; Rq: (+ 2 3) sera évalué ...
  (<- 'Largeur  i (+ 2 3)))
```

La macro-fonction **defig**, cas particulier de **definst**, permet de créer et renseigner une instance génératrice. L'appel (**defig symb ref (symb1 Sexp1) ... (symbN SexpN)**) crée une instance (génératrice), de nom **symb**, de l'instance génératrice (et même métaclasse) **ref** et donne à l'attribut **symb1** (resp. **symb2 ... symbN**) la valeur **Sexp1** (resp. **Sexp2 ... SexpN**). Comme pour **definst**, les noms d'attribut **symb1 ... symbN** ne sont pas évalués et l'instance génératrice créée est renvoyée.

```
(dmd defig (:nom-ig :ref-ig . :reste)
  `(let ((:ig (definst ,:ref-ig . ,:reste)))
     (<- 'nom :ig ',:nom-ig)                    ;;; le "nom" nécessaire
     (:ref-ig ',:nom-ig :ig)                    ;;; structure "cachée" des igs
     (:ref-methodes ',:nom-ig ())               ;;; initialisation du dico de l'ig
     (let ((: n 0)                              ;;; les méthodes d'accès
           (:vis ,(cadr (assq 'Vi :reste))))
       (mapc (lambda (:vi)
               (incr :n)
               (:ref-methode-add ',:nom-ig
                                 `(lambda (MOI . :args)
                                    (if :args
                                        (:Attr-Objet MOI ,:n (car :args))
                                      (:Attr-Objet MOI ,:n)))))
             (reverse :vis))
       (<- 'methodes :ig (append (<- 'methodes :ig) :vis)))
     :ig))
```

L'expansion de cette macro-fonction peut se définir en termes de messages **leloo** de la façon suivante :

```
(defig Rectangle #RmetaRect
  (supers '(Surface))
  (vi     '(Longueur Largeur)))
```

s'expanse en :

```
(let ((ig (definst #RmetaRect
            (supers '(Surface))
            (vi     '(Longueur Largeur)))))
  (<- 'nom ig 'Rectangle)
  ...      ; Le "cache" de l'objet physique
  (...)    ; La méthode d'accès "Longueur"
  (...)    ; La méthode d'accès "Largueur"
  ig)      ; Pour être propre ...
```

La macro-fonction **defmeth** permet de définir le code d'une méthode et de l'attacher à une instance génératrice. Sa syntaxe s'inspire de façon évidente de la (méta-)fonction **DE** de **lelisp**. L'appel (**defmeth symb1 symb2 args Sexp1 ... SexpN**) crée une méthode de nom **symb1** pour l'instance génératrice de nom **symb2**, de code (**lambda args Sexp1 ... SexpN**). Le nom de la méthode (**symb1**) est renvoyé.

```
(dmd defmeth (:nom-meth :nom-ig :args . :code)
  `(progn
     (let ((:ig (:ref-ig ',:nom-ig)))
       (when (memq ',:nom-meth (<- 'methodes :ig));; Si elle existe, la virer.
         (:val-meth- ',:nom-ig :ig ',:nom-meth);; valeurs
         (<- 'methodes :ig (remq ',:nom-meth (<- 'methodes :ig))));; noms
       (<- 'methodes :ig (cons ',:nom-meth (<- 'methodes :ig)))
       (:ref-methode-add ',:nom-ig
                         ;; La variable MOI
                        '(lambda (:MOI . ,',:args) . ,',:code))
       ',:nom-meth)))

;;; enlève le code de la méthode de nom nom-meth de la liste des codes
;;; de méthodes de l'instance génératrice ig de nom nom-ig.

(de :val-meth- (:nom-ig :ig :nom-meth)
  (let ((:vars (<- 'methodes :ig))
        (:vals (:ref-methodes :nom-ig)))
    (if (eq :nom-meth (car :vars))
        (progn                          ;;; Premier élément
          (<- 'methodes :ig (cdr :vars))
          (:ref-methodes :nom-ig (cdr :vals)))
      (map (lambda (:lvar :lval)        ;;; Récursion sur les cadr permise
             (if (cddr :lvar)
                 (when (eq (cadr :lvar) :nom-meth)
                   (rplac (cdr :lvar) (caddr :lvar) (cdddr :lvar))
                   (rplac (cdr :lval) (caddr :lval) (cdddr :lval)))
               ;; La var cherchée se trouvait forcément dans le cdr.
               (rplacd (cdr :lvar) ())
               (rplacd (cdr :lval) ())))
           :vars :vals))))
```

L'expansion de cette macro-fonction peut se définir en termes de messages `leloo` de la façon suivante :

```
;;; Définition de la somme vectorielle de deux rectangles
;;; Le rectangle-récepteur se voit "augmenté" du rectangle fourni en argument.

(defmeth + Rectangle (r)
  (<- 'Longeur MOI          ;;; La nouvelle "longeur" de MOI ...
      (+ (<- 'Longeur MOI)  ;;; ... est la somme ...
         (<- 'Longeur r)))  ;;; ... des deux anciennes "longeur"s.
  (<- 'Largeur MOI          ;;; Id. ("dualité")
      (+ (<- 'Largeur MOI)
         (<- 'Largeur r)))
  r)
```

s'expanse en :

```
(progn
  (when (memq '+ (<- 'methodes #Rrectangle))
    (...))                      ;;; Destruction d'une méthode préexistante
  (<- 'methodes #Rrectangle ;;; Le nouveau nom
      (cons '+ (<- 'methodes #Rrectangle)))
  ...                           ;;; Stockage du corps de la méthode (lambda)
  '+)                           ;;; Le nom est renvoyé (propreté)
```

D.4 Amorce

Pose des objets physiques

```
(:ref-ig 'Objet
         (:Cree-Objet 'Classe
                      '(()                          ;;; Cache
                        Objet
                        (est-un)
                        (Objet)
                        (Est-un Affiche))))

(:ref-ig 'Classe
         (:Cree-Objet 'Classe
                      '(()                          ;;; Cache
                        Classe
                        (nom vi supers methodes)
                        (Objet)
                        (Nom Vi Supers Methodes Cree Cree-Local))))
```

Cablage des méthodes d'objet et de classe L'attribut `methodes` de la classe `objet` est physiquement initialisé : les corps des méthodes `Est-un` et `Affiche` y sont placés (en ordre inverse, cf. `ref-methodes`).

```
(:ref-methodes 'Objet ())
(:ref-methode-add 'Objet '(lambda (:MOI) (print MOI)))
(:ref-methode-add 'Objet
                  '(lambda (:MOI . :arg)
                     (if :arg
                         (:est-un  MOI (car :arg))
                       (:est-un MOI))))
```

Même opération pour la classe `classe` avec les méthodes `Nom`, `Vi`, `Supers`, `Methodes`, `Cree` et `Cree-local`.

```
(:ref-methodes 'Classe ()) ;;; On efface tout et on recomence ...
(:ref-methode-add 'Classe
                  '(lambda (:MOI)
                     (:Cree-Objet (<- 'nom MOI)
                                  (makelist (1+ (length (<- 'vi MOI))) ()))))

(:ref-methode-add 'Classe
                  '(lambda (:MOI)
                     (let (:lnom-ig)
                       (:Creer MOI (<- 'nom MOI)))))
(:ref-methode-add 'Classe
                  '(lambda (:MOI . :arg)
                     (if :arg
                         (:Attr-Objet :MOI 4 (car :arg))
                       (:Attr-Objet :MOI 4))))

(:ref-methode-add 'Classe
                  '(lambda (:MOI . :arg)
                     (if :arg
                         (:Attr-Objet :MOI 3 (car :arg))
                       (:Attr-Objet :MOI 3))))

(:ref-methode-add 'Classe
                  '(lambda (:MOI . :arg)
                     (if :arg
                         (:Attr-Objet :MOI 2 (car :arg))
                       (:Attr-Objet :MOI 2))))

(:ref-methode-add 'Classe
                  '(lambda (:MOI . :arg)
                     (if :arg
                         (:Attr-Objet :MOI 1 (car :arg))
                       (:Attr-Objet :MOI 1))))
```

Méthodes La méthode `Affiche` fournit une impression propre d'une instance génératrice sous la forme `#Rnig`, où `nig` est le nom de cette instance génératrice.

```
(defmeth Affiche Classe ()
  (print "#R" (<- 'nom MOI)) ;;; Utilisation de la fameuse variable MOI.
  MOI)
```

La méthode `Bo` (le caractère "ô" n'existe pas en ASCII standard) fournit une description d'une instance sous forme `(definst ...)` ou `(defig ...)`. Elle est posée sur la classe `objet` pour afficher `(definst ...)`.

```
(defmeth Bo Objet ()
  (let* ((:nom-ig (:est-un MOI))
         (:ig     (:ref-ig :nom-ig)))
    (prin  "(definst ") (<- 'Affiche :ig)
    (mapc (lambda (:vi) (print "   (" :vi " '" (<- :vi MOI) ")"))
          (<- 'vi :ig))
    (print ")")
    MOI))
```

Elle est aussi posée sur la classe `classe` pour afficher `(defig ...)`.

```
(defmeth Bo Classe ()
  (let* ((:nom-ig (:est-un MOI))
         (:ig     (:ref-ig :nom-ig)))
    (prin  "(defig " (<- 'nom MOI) " ") (<- 'Affiche :ig)
    (mapc (lambda (:vi)
            (unless (eq :vi 'nom)
              (print "   (" :vi " '" (<- :vi MOI) ")")))
          (<- 'vi :ig))
    (print ")")
    MOI))
```

Les types `lelisp` Quelle est la signification d'une classe *qui n'hérite de rien* ? Par définition, le message `cree` peut lui être envoyé ; mais l'instance ainsi créée ne peut pas voir par héritage l'attribut `est-un` (contenu dans l'attribut `vi` de la classe `objet`) et ne peut donc pas savoir, selon `leloo`, *de qui* elle est instance ! La seule façon de connaître la classe de ces instances, amnésiques en `leloo`, est de s'en remettre au langage d'implémentation, qui, lui, connaît au moins le *type* de la structure de données implémentant ces instances. Les types de données `lelisp`, ou tout changement de codage des objets `leloo`, sont donc intégrés dans `leloo` au moyen d'instances génératrices qui n'héritent pas de la classe `objet`. Les attributs de ces classes particulières sont les champs usuels des types `lelisp` correspondants.

```
(defig Type-Lelisp #RClasse  (supers '(Classe)))

(defig Cellule #RType-Lelisp  (supers '())  (vi '(car cdr)))
(defig Nul     #RType-Lelisp  (supers '())  (vi '()))
(defig Symbole #RType-Lelisp  (supers '())  (vi '(pname objval cval plist)))
(defig Chaine  #RType-Lelisp  (supers '())  (vi '(type longueur valeurs)))
(defig Vecteur #RType-Lelisp  (supers '())  (vi '(type longueur valeurs)))
```

D.5 Exemple

Cet exemple est tiré de [Cointe 87].

```
? (defig MetaPoint #RClasse  ; Création d'une instance génératrice ...
?    (supers '(Classe))      ; ... qui est une métaclasse ...
?    (vi     '(etoile)))     ; ... avec une place pour une étoile.
= metapoint

? (defig Point #RMetaPoint   ; Création d'une instance (génératrice) de MétaPoint ...
?   (supers '(Objet))
?   (etoile '*))             ; ... avec '*' pour valeur d'étoile.
= point

? (defmeth Affiche Point ()      ; Pour qu'un point puisse voir des étoiles
?   (print (<- 'etoile (#:loo:ref-ig (#:loo:est-un MOI)))))
= affiche

? (<- 'affiche (<- 'cree #RPoint)) ; Roulement de tambours ...
*                                  ; ... coup de cymbale ...

= *                                ; ... applaudissements !
```

Bibliographie

Cette bibliographie est classée par ordre alphabétique, avec les mots-clés suivants :

ALGO Algorithmique,

IA Intelligence Artificielle (ouvrage général),

LANG... Langage :

- **LOO** Langage Orienté Objet,
- **LISP** Langage Lisp,
- **LANGNAT** Langage naturel,

LOG... Logique mathématique :

- **LOGCLA** Logiques classiques,
- **LOGMOD** Logiques modales,
- **LOGTEMP** Logiques temporelles,
- **LOGREIF** Logiques réifiées,

ORDO Ordonnancement,

PHI Philosophie,

PLAN Planification,

PSY Psychologie,

SE Systèmes experts.

[Agre 87] Ph. Agre, D. Chapman, *Pengi : an Implementation of a Theory of Activity*, AAAI 1987, pp. 268-272. *PLAN*

[Agre & Chapman 87] Ph. Agre, D. Chapman, *Indexicality and the Binding Problem*, Unpublished paper, AI Lab, MIT, Cambridge, 1987. *PLAN*

[Allen 81] J. F. Allen, *An Interval-Based Representation of Temporal Knowledge*, IJCAI 81, pp. 221-226. *LOGREIF*

[Allen 83] J. F. Allen, *Maintaining knowledge about temporal intervals*, CACM n. 26, vol. 11, pp. 832-843, 1983. *LOGREIF*

[Allen 84] J. F. Allen, *Towards a general theory of action and time*, Artificial Intelligence, vol. 23, pp. 123-154, 1984. *LOGREIF*

[Assémat & Morignot 87] Ch. Assémat, Ph. Morignot *TG2, Manuel de référence*, Rapport Interne, Cognitech, 1987. *LOO*

[Assémat et coll. 88] Ch. Assémat, Ph. Morignot, Ph. Vernet, *COPLANER : un système-expert en planification de bâtiment*, EuropIA 88, Actes des Journées européennes sur les applications de l'intelligence artificielle en architecture, bâtiment et génie civil, pp. 33-47. *SE, PLAN*

[Assémat et coll. 89] Ch. Assémat, Ph. Morignot, Ph. Vernet, *COPLANER : un système-expert en planification de bâtiment. Vers une expertise fonctionnelle en planification*, Actes des Neuvièmes Journées Internationales, Les systèmes-experts et leurs applications, Avignon 1989, vol. 2, pp. 741-753. *SE, PLAN*

[Audureau et coll. 90] E. Audureau, P. Enjalbert, L. Fariñas del Cerro, *Logique temporelle : sémantique et validation de programmes parallèles*, coll. Etudes et Recherches en Informatique, Masson, Paris, 1990. *LOGCLA, LOGMOD, LOGTEMP*

[VanBeek 89] P. Van Beek, *Approximation algorithms for temporal reasoning*, IJCAI 89, pp. 1292-1296. *LOGREIF, ALGO*

[Bel et coll. 86] G. Bel, E. Bensana, D. Dubois, *Un système d'ordonnancement prévisionnel d'atelier utilisant des connaissances théoriques et pratiques*, 6es Journées Internationnales. Les Systèmes Experts et leurs Applications, Avignon, 1986. *ORDO*

[Bel et coll. 88] G. Bel, E. Bensana, D. Dubois, *Construction d'ordonnancements prévisionnels : un compromis entre approches classiques et système-experts*, RAIRO APII (1988), vol. 22, n. 5, pp. 509-534. *ORDO*

[Bensana et coll. 88] E. Bensana, G. Bel, D. Dubois, *OPAL : A multi-knowledge-based system for industrial job-shop scheduling*, Internationnal Journal of Production Research, 1988, vol. 26, n. 5, p. 795-819. *ORDO*

[Bergadaã 89] M. Bergadaã, *Le temps et le comportement de l'individu*, Recherche et Applications en Marketing, vol. 3, n. 4 (partie 1) et vol. 4, n. 4 (partie 2), 1989. *PHI, PSY*

[Bestougeff & Ligozat 90] H. Bestougeff, G. Ligozat, *Outils logiques pour le traitement du temps : de la linguistique à l'intelligence artificielle*, Masson, coll. Etudes et Recherches en Informatiques, 1990, Paris. *LOGREIF, LOGTEMP, LOGMOD, LOGCLA*

[Bobrow et coll. 87] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, D. Moon, *Common Lisp Object System Specifications*, ANSI Document X3J13, March - July - November 1987. *LOO*

[Brown 84] D. R. Brown, *Planning System Survey : a Review of the State of Practice*, Task report n. 7306, SRI International, dec. 1984. *PLAN*

[Carlier & Chrétienne 82] J. Carlier, P. Chrétienne, *Un domaine très ouvert : les problèmes d'ordonnancement*, RAIRO (août 1982), vol. 16, n. 3, pp. 175-217. *ORDO*

[Chailloux et coll. 86] J. Chailloux, M. Devin, J. M. Hullot, B. Serpette, J. Vuillemin, *Le_LISP, Version 15.2, Manuel de Référence*, INRIA, Rocquencourt, 1986. *LISP*

[Chapman 85] D. Chapman, *Planning for Conjunctive Goals*, Technical Report 802, Computer Science Department, MIT, 1985. Ou également dans sa re-publication ultérieure, sous le même titre et sans les annexes : Artificial Intelligence (1987), vol. 32, pp. 333-377. *PLAN*

[Clément 75] M. Clément, *Categorical Axiomatics of Dynamic Programming* Journal of Mathematical Analysis and Applications (1975), vol. 51, n. 1, pp. 47-55. *ALGO*

[Clocksin & Mellish 81] W. Clocksin, C. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981. *LANG*

[Cointe 87] P. Cointe, *Metaclasses are First Class : the ObjVlisp Model*, OOPSLA-87. *LOO*

[Cointe 88] P. Cointe, *A Tutorial Introduction to Metaclass Architecture as provided by Class Oriented Languages*, Internationnal Conference on Fifth Generation Computer Systems, Tokyo 1988. *LOO*

[Colmerauer 84] A. Colmerauer, *Prolog langage de l'intelligence artificielle*, La Recherche, n. 158, sept. 84. *LANG*

[Constant 91] P. Constant, *Analyse syntaxique par couche*, Thèse de Doctorat, ENST, Département Informatique, Paris, 1991 (à paraître).

[Davis 77] E. Davis, *Production Rules as a Representation for a Knowledge-Based Consultation Program*, Artificial Intelligence (1977), vol. 8, pp. 15-45. *SE*

[Davis 87] E. Davis, *Constraint Propagation with Interval Labels*, Artificial Intelligence (1987), vol. 32, pp. 281-331. *ALGO*

[Dean 85] Th. L. Dean, *Temporal Imagery : An Approach to Reasoning about Time for Planning and Problem Solving*, PhD thesis, Technical report 433, Yale University, Computer Science Department, oct. 1985. *PLAN, ALGO*

[Dean 86] Th. L. Dean, *Intractability and Time-Dependent Planning*, in Reasoning about Actions and Plans, Ed. Georgeff & Lansky, SRI Internationnal, 1986. *PLAN*

[Dean & McDermott 87] Th. L. Dean, D. V. McDermott, *Temporal Data Base Management*, Artificial Intelligence (1987), vol. 32, pp. 1-55. *PLAN*

[Dean & Boddy 88] Th. L. Dean, M. Boddy, *Reasoning about Partially Ordered Events*, Artificial Intelligence (1988), vol. 36, n. 3, pp. 375-399. *PLAN, ALGO*

[Descottes & Latombe 85] Y. Descottes, J.C. Latombe, *Making Compromises among antagonist Constraints in a Planner*, Artificial Intelligence (1985), vol. 27, pp. 183-217. *PLAN, SE*

[Ducourneau & Habib 89] R. Ducourneau, M. Habib, *La multiplicité de l'héritage dans les langages orientés objets*, TSI, vol. 8, n. 1, pp. 41-62, 1989. *LOO*

[Dubois & Prade 85] D. Dubois, H. Prade, *Théorie des Possibilités : Applications à la Représentation des Connaissances en Informatique*, Masson, 1985. *LOG*

[Dubois et coll. 86] D. Dubois, G. Bel, E. Bensana, *Un système d'ordonnancement prévisionnel d'atelier utilisant des connaissances théoriques et pratiques*, Sixièmes Journées Internationnales (Avignon), avril 1986, p 757. *ORDO, SE*

[Escalada-Imaz & Ghallab 88] G. Escalada-Imaz, M. Ghallab, *A Practilly Efficient and Almost Linear Unification Algorithm*, Research Note, Artificial Intelligence (1988), vol. 36, n.2, pp. 249-263. *ALGO*

[Esquirol 87] P. Esquirol, *Règles et Processus d'inférence pour l'aide à l'ordonnancement de tâches en présence de contraintes*, Thèse de 3ème cycle, Laboratoire d'Automatique et d'Analyse des Systèmes du CNRS, Toulouse, décembre 1987. *ORDO, SE*

[Ferber 89] J. Ferber, *Objets et agents : une étude des structures de représentations et de communications en Intelligence Artificielle*, Thèse d'Etat de l'Université Pierre et Marie Curie (Paris VI), juin 1989. *LOO*

[Fikes & Nilsson 71] R. E. Fikes, N. J. Nilsson, *STRIPS : a new approach to the application of theorem proving to problem solving*, Artificial Intelligence (1971), vol. 2, pp. 198-208. *PLAN*

[Fikes et coll. 72] R. E. Fikes, P. E. Hart, N. J. Nilsson, *Learning and Executing Generalized Robot Plans*, Artificial Intelligence (1972), vol. 3, n. 4, pp. 251-288. *PLAN*

[Fox 83] M. S. Fox, *Constraint-Directed Search : A case study of Job-Shop Scheduling*, PhD Thesis, Technical Report 161, Research Institute, Carnegie Mellon University, 1983. Republié dans la collection *Research Note*. *IA, ORDO*

[Fox et coll. 83] M. S. Fox, B. P. Allen, S. F. Smith, G. A. Strohm, *ISIS : A constraint-directed reasoning approach to Job-Shop Scheduling*, Technical Report 8, Research Institute, Carnegie Mellon University, 1983. *ORDO*

[Fox et coll. 85] M. S. Fox, J. M. Wright, D. Adam, *Experiences with SRL : An Analysis of a Frame-based Knowledge Representation*, Expert Database Systems, 1986, pp. 161-172. *LOO*

[Georgeff 87] M. P. Georgeff, *Planning*, Annual Reviews in Computer Science, vol. 2, 1987, pp. 359-400. *PLAN*

[Ghallab 86] M. Ghallab, *Coping with Complexity in Inference and Planning Systems*, Robotics Research 3, pp. 101-107, Ed. Faugeras & Giralt, MIT Press, 1986. *LOGREIF, ALGO*

[Ghallab et coll. 88] M. Ghallab, R. Alami, R. Chatila, *Dealing with Time in Planning and Execution Monitoring*, Robotics Research 4, Ed. R. Bolles, MIT Press, 1988. *LOGREIF, ALGO*

[Ghallab & Mounir Alaoui 89] M. Ghallab, A. Mounir Alaoui, *The Indexed Time Table Approach for planning and Acting*, Proc. NASA Conference on Space, Tele. Robotics, Pasadena, Feb. 1989. *ALGO*

[Ginsberg 86] M. L. Ginsberg, *Possible World Planning*, in Reasoning about Actions and Plans, Ed. Georgeff & Lansky, SRI Internationnal, 1986. *PLAN*

[Ginsberg & Smith 88a] M. L. Ginsberg, D. E. Smith, *Reasoning about Action : A possible Worlds Approach*, Artificial Intelligence (1988), vol. 35, pp. 165-195. *PLAN*

[Ginsberg & Smith 88b] M. L. Ginsberg, D. E. Smith, *Reasoning about Action : The Qualification Problem*, Artificial Intelligence (1988), vol. 35, pp. 311-342. *PLAN*

[Ginsberg 89] M. L. Ginsberg, *Universal Planning : An (Almost) Universally Bad Idea*, A.I. Magazine, vol. 10, n. 4 (hiver 89), pp. 40-44. *PLAN*

[Gondran & Minoux 79] M. Gondran, M. Minoux, *Graphes et Algorithmes*, Eyrolles, Paris, 1979. *ALGO*

[Haren & Neveu 84] P. Haren, B. Neveu, *SMECI : An Expert System for Civil Engineering Design*, SIGART Newsletter, n. 90, octobre 1984, pp. 19-32. *SE, PLAN*

[Hayes 71] P. J. Hayes, *The Frame Problem and Related Problems on Artificial Intelligence*, Stanford AI Project, Memo AIM-153, n. CS-242, nov. 1971. *IA*

[Hayes-Roth et coll. 83] F. Hayes-Roth, D. Waterman, D. Lenat, *Building Expert Systems*, Addison Wesley, Reading, Mass., 1983. *SE, IA*

[Hayes-Roth 85] B. Hayes-Roth, *A Blackboard Architecture for Control*, Artificial Intelligence (1985), vol. 26, pp. 251-321. *IA*

[Hendler et coll. 90] J. Hendler, A. Tate, M. Drummond, *AI Planning : Systems and Techniques*, A.I. Magazine, vol. 11, n. 2, été 1990. *PLAN*

[Hendler 91] J. Hendler (`hendler@dormouse.cs.umd.edu`), *Nonlin in commonlisp - code available*, transaction `news`, réseau USENET, groupe `comp.archives`, id. `<1991Feb7.061539.18668@ox.com>`, 7 février 1991. *PLAN*

[Hofstädter 86] D. Hofstädter, *Gödel, Escher, Bach : les Brins d'une Guirlande Eternelle*, InterEditions, Paris, 1986. *IA, PSY, LOGCLA, PHI*

[Joslin & Roach 89] D. Joslin, J. Roach, *A Theoretical Analysis of Conjunctive-Goal Problems*, Artificiel Intelligence, Research Note, vol. 41, pp. 97-106. *PLAN*

[Kauffmann 68] A. Kauffmann, *Introduction à la combinatorique en vue des Applications*, Dunod, 1968. *ALGO*

[Lalement & Saint 86] R. Lalement, J.B. Saint, *Logique et Informatique*, Cours du DEA IARFAG, Ecole Nationale des Ponts et Chaussées / Paris VI, 1986. *LOGCLA*

[Laurière 76] J. L. Laurière, *Un langage et un programme pour énoncer et résoudre des problèmes combinatoires*, Thèse d'Etat, Université Pierre et Marie Curie, mai 1976. *IA*

[Laurière 86] J. L. Laurière, *Intelligence Artificielle, Résolution de problèmes par l'homme et la machine*, Eyrolles, Paris, 1986. *IA*

[LePape 85] C. Le Pape, B. Sauve, *SOJA : un système d'ordonnancement journalier d'atelier*, 5es Journées Internationnales, Les Systèmes Experts et leurs Applications, Avignon, 1985, p 849-867. *ORDO*

[LePape & Smith 87] C. Le Pape, S. F. Smith, *Management of temporal constraints for factory scheduling*, Temporal Aspects in Information Systems (mai 1987), pp. 165-176. *ORDO*

[Levitt & Kunz 87] R. E. Levitt, J. Kunz, *Using Artificial Intelligence to support project management*, AI EDAM (1977), vol. 1, n. 1, pp. 3-24. *SE, PLAN, ORDO*

[Lifschitz 86] V. Lifschitz, *On the Semantics of STRIPS*, in Reasoning about Actions and Plans, Ed. Georgeff & Lansky, SRI Internationnal, 1986. *PLAN*

[McCarthy & Hayes 69] J. Mc Carthy, P. Hayes, *Some philosophical problems from the standpoint of artificial intelligence*, Machine Intelligence 4, eds. B. Meltzer & D. Michie, pp. 463-502, American Elsevier, New York, 1969. *IA, PHI*

[McDermott 78] D. Mc Dermott, *Planning and Acting*, Cognitive Science, n. 2, vol. 2, pp. 71-109, 1978. *PLAN*

[McDermott 82] D. Mc Dermott, *A temporal logic for reasoning about Processes and Plans*, Cognitive Science, n. 6, pp. 101-155, 1982. *LOGREIF*

[Malik & Bindford 83] J. Malik, T.O. Bindford, *Reasoning in time and space*, IJCAI 83, pp 343-346. *LOGREIF, ALGO*

[Minsky 75] M. Minsky, *A Framework for Representing Knowledge*, in *The Psychology of Computer Vision*, P. Winston ed., New York, McGraw-Hill, 1975. *IA*

[Minsky 88] M. Minsky, *La société de l'esprit*, InterEditions, Paris, 1988. *IA, PSY, PHI*

[Mosnier 90] I. Mosnier, *Le Temps en Philosophie*, Communication personnelle, 1990. *PHI*

[Moon & Weinreb 87] D. Moon, Weinreb, *Common Lisp Reference Manual*, AI Lab., MIT, 545 Technology Square, Cambridge, Mass., 1980. *LISP*

[Morignot 86] Ph. Morignot, *Etude de représentation des graphes en Planification Hiérarchique*, Rapport de stage de fin d'études, Groupe Système Expert, C. G. E., Laboratoires de Marcoussis, 1986. *PLAN*

[Morignot 87] Ph. Morignot, *Réalisation d'un système-expert d'aide à la planification de chantiers*, Rapport de stage de DEA, Cognitech, juillet 1987. *SE, PLAN*

[Newell & Simon 63] A. Newell, H. Simon, *GPS : A Programs that Simulates Human Thoughts*, in Computers and Thought, E. Feigenbaum & J. Feldman eds., Mc Graw-Hill, 1963, pp. 279-293. *SE, PLAN*

[Nilsson 80] N. J. Nilsson, *Basic plan-generating systems* (§7) & *Advanced plan-generating systems* (§8), in *Principles of Artificial Intelligence*, SRI International, Tioga Publishing Company, Palo Alto, California. *IA*

[Parello et coll. 86] B. D. Parello, W. C. Kabat, *Job-Shop Scheduling using Automated Reasoning : A Case Study of the Car Sequencing Problem*, Journal of Automated Reasoning 2 (1986), pp. 1-42. *ORDO*

[Pelavin & Allen 86] R. Pelavin, J. F. Allen, *A Formal Logic of Plans in Temporally Rich Domains*, Proceedings of the IEEE (octobre 1986), vol. 74, n. 10, pp. 1364-1382. *LOGREIF*

[Perrot 86] J.F. Perrot, *Programmation avancée*, Cours du DEA IARFAG, Ecole Nationale des Ponts et Chaussées / Paris VI, 1986. *LISP*

[Picardat 87] J. F. Picardat, *Contrôle d'exécution, Compréhension et Apprentissage de plans d'actions : Développement de la méthode de la Table Triangulaire*, Thèse de 3ème cycle, Université Paul Sabatier, Toulouse, juin 1987. *PLAN*

[Prior 57] A. Prior, *Time and Modality*, Clarendon Press, 1957. *LOGTEMP*

[Pnuelli 71] Pnuelli, *The temporal logic of programs*, Proc. of the 18th Ann. Symp. Foundations of Computer Science, IEEE, pp. 46-57, 1977. *LOGTEMP*

[RAP 86] *Reasoning about Actions and Plans*, Proceedings of the 1986 Workshop, ed. M.P. Georgeff & A.L. Lansky, Timberline, Oregon, juin-juillet 1986. *PLAN, PHI*

[Rescher 71] N. Rescher, A. Urquhart, *Temporal Logic*, Springer-Verlag, Berlin, 1971. *LOGTEMP*

[Rit 88] J. F. Rit, *Modélisation et Propagation de Contraintes Temporelles pour la Planification*, Thèse de 3ème cycle, Institut National Polytechnique de Grenoble, mars 88. *LOGREIF, ALGO*

[RIP 90] *Readings in planning*, J. Allen J. Hendler A. tate eds., Morgan Kaufmann, 1990. *LOGREIF, ALGO*

[Sacerdoti 75] E. D. Sacerdoti, *The Nonlinear Nature of Plans*, IJCAI-75, Tbilisi, pp. 206-214. *PLAN*

[Sacerdoti 77] E. D. Sacerdoti, *A Structure for Plan and Behavior*, Elsevier, North-Holland, New-York, 1977. *PLAN*

[Schoppers 87] M. J. Schoppers, *Universal Plans for Reactive Robots in Unpredictable Environments*, IJCAI 87, pp. 1039-1046. *PLAN*

[Shoham & McDermott 88] Y. Shoham, D. McDermott, *Problems in Formal Temporal Reasoning*, Artificial Intelligence (1988), vol. 36, pp. 49-61. *IA*

[Siklòssy & Roach 75] L. Siklòssy, J. Roach, *Model Verification and Improvement using DISPROVER*, Artificial Intelligence (1975), vol. 6, pp. 41-52. *PLAN*

[Si Ow 86] P. Si Ow, *Experiments in Knowledge-based Scheduling*, CMU, Pittsburgh, PA 15213, Working Paper 50-85-86. *SE, ORDO*

[Smith 83] S. F. Smith, *Exploiting Temporal Knowledge to Organize Constraints*, Technical Report 12, Carnegie Mellon University, Research Institute, 1983. *SE, ORDO*

[Smith et coll. 86] S. F. Smith, P. Si Ow, C. Le Pape, N. Muscettola, *Reactive Management of Factory Schedules*, AAAI 1986. *ORDO*

[Smith 87] J.U.M. Smith, *Expert systems in Project management*, Internationnal Journal of Project Management, vol. 5, n. 1, february 1987. *SE, PLAN*

[Steele & Sussman 79] G. L. Steele Jr., G. J. Sussman, *CONSTRAINTS*, AI Lab., MIT, 545 Technology Square, Cambridge Massachusetts 02139, 1979. *ALGO*

[Stefik 81] M. Stefik, *Planning with Constraints : MOLGEN (Part 1 & 2)*, Artificial Intelligence (1981), vol. 16, pp. 111-170. *PLAN*

[Sussman 75] G. J. Sussman, *A Computer Model of Skill Acquisition*, AI Series, Elsevier, New York, 1975. *PLAN*

[Swartout 88] W. Swartout, *DARPA Santa Cruz Workshop on Planning*, A. I. Magazine (été 1988), pp. 115-130. *PLAN*

[Tate 77] A. Tate, *Generating Project Networks*, IJCAI 77, pp. 888-893. *PLAN*

[Tate 86] A. Tate, *Goal Structure, Holding Periods and 'Clouds'* , in Reasoning about Actions and Plans, Ed. Georgeff & Lansky, SRI Internationnal, 1986. *PLAN*

[Tate et coll. 90] A. Tate, J. Hendler, M. Drummond, *A Review of AI Planning Techniques*, in Readings in Planning, Ed. Allen, Hendler, Tate, Morgan Kauffman, 1990, pp. 26-49. *PLAN*

[Tsang 87] J. P. Tsang, *Planification par Combinaison de Plans. Application à la Génération de Gamme d'Usinage*, Thèse de 3ème cycle, INPG, Grenoble, juillet 1987. *PLAN, SE*

[Turner 86] R. Turner, *Logiques pour l'Intelligence Artificielle*, Masson, Paris, 1986. *LOGMOD, LOGTEMP, LOGREIF*

[Vere 83] S. A. Vere, *Planning in time : windows and durations for activities and goals*, IEEE Transactions, mai 1983, vol. PAMI-5, n. 3, pp. 246-267. *PLAN*

[Vialatte 85] M. Vialatte, *Description et applications du moteur d'inférences Snark*, Thèse de Doctorat, Université Pierre et Marie Curie, Paris VI, 1985. *SE*

[Vilain & Kautz 86] M. Vilain, H. Kautz, *Constraint Propagation Algorithms for Temporal Reasoning*, AAAI 86, pp. 377-382. *ALGO, LOGREIF*

[Waldinger 75] R. Waldinger, *Achieving Several Goals Simultaneously*, Technical Note 107, SRI AI Center, Menlo Park, Calif. *PLAN*

[Wertz 85] H. Wertz, *LISP, une introduction à la programmation*, Masson, Paris 1985. *LISP*

[Wilkins 84] D. Wilkins, *Domain-Independent Planning : Representation and Plan Generation*, Artificial Intelligence (1984), vol. 22, pp. 269-301. *PLAN, SE*

[Wilkins 85] D. Wilkins, *Recovering from errors execution in SIPE*, Computer Intelligence (1985), vol. 1, pp. 33-45. *PLAN, SE*

[Wilkins 86] D. Wilkins, *Hierarchical Planning : Definition and Implementation*, ECAI-86, pp. 466. *PLAN*

[Wilkins 88] D. Wilkins, *Practical Planning : Extending the Classical AI Planning Paradigm*, ed. Morgan Kaufmann, San Mateo, 1988. *PLAN*

[Winograd 72] T. Winograd, *Understanding Natural Language*, Academic Press, New York, 1972. *LANGNAT*

Index

Dépôt légal : 4e trimestre 1991
Imprimé à l'Ecole Nationale Supérieure des Télécommunications - Paris
ISSN : 0751-1353