Planification d'actions hiérarchique pour la simulation tactique

Alexandre Menif

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à l'Université Paris-Dauphine

Planification d'actions hiérarchique pour la simulation tactique

École Doctorale de Dauphine — ED 543

Spécialité **Informatique**

COMPOSITION DU JURY :

M. Tristan Cazenave
Université Paris-Dauphine
Directeur de thèse

M. Éric Jacopin
École Spéciale Militaire de Saint-Cyr
Directeur de thèse

M. Frédérick Garcia
INRA
Président du jury

M. Bruno Bouzy
Université Paris Descartes
Rapporteur

Mme. Amal El Fallah Seghrouchni
Université Pierre et Marie Curie
Rapporteur

M. Christophe Guettier
Safran Electronics & Defense
Membre du jury

M. Vincent Corruble
Université Pierre et Marie Curie
Membre du jury

Soutenue le 11/01/2017
par Alexandre MENIF

Dirigée par **Tristan CAZENAVE**

DAUPHINE UNIVERSITÉ PARIS | PSL RESEARCH UNIVERSITY PARIS

Planification d'actions hiérarchique pour la simulation tactique

Alexandre Menif

18 janvier 2017

À mes grand-pères

Remerciements

Ces travaux ont fait l'objet d'une convention CIFRE entre l'entreprise Sagem Défense Sécurité (devenue Safran Electronics & Defense), le centre de recherche des écoles de Coëtquidan, ainsi que le Laboratoire d'analyse et modélisation de systèmes pour l'aide à la décision (LAMSADE) de l'université Paris Dauphine. Je remercie donc toutes les personnes qui ont œuvré afin de mettre en place cette convention, ainsi que l'Association Nationale de la Recherche et de la Technologie (ANRT).

Je souhaite remercier mes directeurs de thèse, qui ont accepté de me confier ces travaux et m'ont guidé tout au long de ces années. Je remercie Tristan Cazenave, professeur à l'université Paris Dauphine, qui a accepté d'assumer l'encadrement de cette thèse, et m'a permis à plusieurs reprises de présenter mes travaux au sein de son séminaire. Je remercie spécialement Éric Jacopin, enseignant-chercheur à l'École Spéciale Militaire de Saint-Cyr, qui m'a encadré activement à toutes les étapes de cette thèse, et a patiemment accepté de suivre la direction que j'ai donné à ces travaux. Je remercie aussi Christophe Guettier, directeur Architecture des Systèmes à la division R & T de Sagem Défense Sécurité, qui m'a co-encadré durant ces années et a partagé avec moi de nombreuses idées.

Je remercie les membres de mon jury de soutenance : Amal El Fallah Seghrouchni, Bruno Bouzy, Frédérick Garcia et Vincent Corruble. Je l'ai remercie de l'intérêt qu'ils ont porté à mes travaux, et qui les a amené à accepter de participer à mon jury de soutenance et à me permettre de soutenir devant eux ma thèse de doctorat.

Je remercie David Renaudet, responsable du pôle Interface Homme-Machine (IHM) chez Sagem Défense Sécurité, qui m'a accueilli dans ses effectifs. Je remercie Jordane Grenier, docteur en biomécanique et ingénieur chez Sagem Défense Sécurité, qui a partagé son expérience d'ancien doctorant, et qui s'est assuré du bon déroulement de mes travaux au quotidien. Je remercie de plus tous mes collègues du pôle IHM, et en particulier Thomas Dazenière et Pascal Gaden, dont la compagnie a été particulièrement enrichissante et agréable.

Je remercie amicalement Florent Taralle, mon camarade doctorant, avec qui j'ai partagé pendant trois années les joies et les difficultés inhérentes à un travail de thèse de doctorat.

Je remercie chaleureusement tous les membres de ma famille, pour leur amour, leur soutien et leur réconfort, et parce qu'ils m'ont permis à plusieurs reprises de m'évader de la capitale et d'oublier ainsi pendant quelques jours les difficultés de la thèse. Je remercie particulièrement Aurélien, Cécile, Florent, Sarah, Vincent ainsi que mes parents, qui ont pris le temps de relire ce manuscrit.

Enfin, je remercie du fond de mon cœur ma compagne Sylvia, qui grâce à l'amour qu'elle me porte, m'a poussé à aller jusqu'au bout de ces travaux, et je lui souhaite à elle aussi d'achever la thèse de doctorat dans laquelle elle s'est engagée. Tu vas finir ta thèse !

Table des matières

1 Introduction **13**
- 1.1 Contributions . . . 13
- 1.2 Contexte industriel . . . 13
- 1.3 Problématique . . . 15
- 1.4 Organisation du manuscrit . . . 18

2 État de l'art **19**
- 2.1 Planification classique . . . 19
  - 2.1.1 Cadre général de la planification classique . . . 20
  - 2.1.2 ADL : un langage de description d'actions . . . 21
  - 2.1.3 Planification dans l'espace des états . . . 25
  - 2.1.4 Planification dans l'espace des plans partiels . . . 30
  - 2.1.5 Planification classique pour les jeux vidéo . . . 33
- 2.2 Planification HTN . . . 35
  - 2.2.1 Formalisme général . . . 35
  - 2.2.2 Planification HTN dans l'espace des décompositions . . . 39
  - 2.2.3 Planification HTN dans l'espace de progression d'état . . . 40
  - 2.2.4 Planification hiérarchique angélique . . . 42
  - 2.2.5 Propriété des espaces de décompositions . . . 45
  - 2.2.6 Application de la planification HTN à la simulation militaire . . . 47
- 2.3 Planification par abstraction hiérarchique . . . 49
  - 2.3.1 Abstraction d'un problème de planification . . . 50
  - 2.3.2 Décrire l'abstraction d'un problème de planification d'actions . . . 52
  - 2.3.3 Planification et hiérarchie d'abstractions . . . 54
  - 2.3.4 Propriétés des hiérarchies d'abstraction . . . 57
- 2.4 Conclusion . . . 61

3 Système de planification en ligne pour les jeux vidéo **63**
- 3.1 Architecture de planification réactive . . . 63
  - 3.1.1 Gestionnaire de mission . . . 64
  - 3.1.2 Contrôleur d'exécution . . . 65
  - 3.1.3 Perception de l'environnement . . . 65
- 3.2 SHPE : un moteur de planification HTN . . . 66
  - 3.2.1 Description de l'implémentation . . . 66
  - 3.2.2 Précompilation du domaine . . . 68
  - 3.2.3 HTDL : un langage de haut niveau pour la modélisation de domaines HTN . . . 69
- 3.3 Évaluation des performances de SHPE . . . 74
  - 3.3.1 *SimpleFPS* : un domaine de planification pour l'animation d'un PNJ 75
  - 3.3.2 Protocole expérimental . . . 76
  - 3.3.3 Résultats . . . 76

3.4 Conclusion 77

4 Planification HTN avec sémantique d'exécution pour les tâches composées 79
4.1 Planification HTN avec hiérarchie d'abstractions 80
4.1.1 Sémantiques de transitions abstraites pour les tâches composées . . 80
4.1.2 Description des fonctions d'abstraction et des effets abstraits 83
4.1.3 Algorithmes 86
4.2 Modélisation et expériences 114
4.2.1 Domaine de planification pour la reconnaissance de bâtiment 114
4.2.2 Évaluation du système pour un domaine de répartition des cibles . 116
4.2.3 Évaluation du système pour un domaine de navigation hiérarchique 120
4.3 Seconde approche : planification HTN avec abstraction 124
4.3.1 Espace d'états d'abstraits 125
4.3.2 Actions abstraites 135
4.3.3 Planification 144
4.4 Conclusion 149

5 Planification HTN optimale 151
5.1 Recherche heuristique pour planification HTN optimale 154
5.1.1 Estimation du coût des tâches composées 154
5.1.2 Algorithme de planification 164
5.1.3 Heuristiques 167
5.2 Expériences 170
5.2.1 Robotbox 170
5.2.2 Logistics 176
5.2.3 SimpleFPS 181
5.3 Conclusions 186

6 Conclusions 189
6.1 Contributions 189
6.2 Perspectives 191

A Définition formelle de HTDL 193

B SimpleFPS pour JSHOP2 197

C Domaines de planification HTDL avec hiérarchie d'abstractions 207
C.1 Reconnaisance de bâtiment 207
C.2 Affectation de cibles 213
C.3 Navigation dans un graphe hiérarchique 215

D Domaine de planification HTN avec abstraction est coûts de haut niveau 219
D.1 *Robotbox* 219
D.2 logistics 223
D.3 SimpleFPS 231

Table des figures

1.1 Organisation d'une section d'infanterie mécanisée française 14
1.2 Modèle 3D d'un soldat de l'armé de terre dans Virtual Battle Space 2 . . . 15
1.3 Machine à état pour le contrôle d'une équipe de fantassins 16
1.4 Planification automatique d'un plan d'actions pour une équipe de fantassins 16

2.1 Représentation graphique d'un problème de planification 21
2.2 Description d'un état 23
2.3 Exemple d'opérateurs décrits avec la syntaxe du langage ADL 26
2.4 Représentation d'un réseau d'actions 32
2.5 Méthodes de décomposition 37
2.6 Planification HTN par progression des états 41
2.7 Méthodes de décomposition avec préconditions 42
2.8 Tâche récursive 46
2.9 Limitation de la profondeur d'appel d'une tâche récursive 47
2.10 Planification par abstraction 50
2.11 Fonction d'abstraction 52
2.12 Concrétisation d'un plan abstrait 55

3.1 Architecture de planification globale pour jeu sérieux 64
3.2 Exemple d'action encodée en Python pour Pyhop 67
3.3 Interface C++ pour l'implémentation des tâches dans SHPE 68
3.4 Précompilation de domaine 71
3.5 Une représentation d'une situation pour un problème du domaine *SimpleFPS*. 75
3.6 Comparaison des performances de SHPE et JSHOP2 77

4.1 Décomposition d'un plan abstrait 81
4.2 Domain de planification HTN avec hiérarchie d'abstraction 85
4.3 Prédicat abstrait dérivé 87
4.4 Domaine de planification HTN pour accomplir un voyage entre plusieurs lieux 90
4.5 Exemple d'incomplétude pour les effets abstraits 91
4.6 Plan multi-niveaux 102
4.7 Problème de reconnaissance de bâtiment 115
4.8 Problème d'affectation de cible 117
4.9 Temps d'exécution pour les problèmes d'affectation de cibles 119
4.10 Répartition des temps de calculs pour les problèmes d'affectation de cible . 120
4.11 Graphe hiérarchique 121
4.12 Temps d'exécution avec abstraction de mauvaise qualité 123
4.13 Temps d'exécution avec abstraction de bonne qualité 124
4.14 Treillis d'abstractions 125
4.15 Double treillis des valeurs de vérité 127
4.16 État abstrait en logique trivaluée 128
4.17 Action abstraite 137

4.18 Progression d'état par une action abstraite 140
4.19 Rang d'un nœud composé dans un réseau de tâches 146

5.1 Modélisation du déplacement d'une équipe de soldats 153
5.2 Coûts optimiste et pessimiste 155
5.3 Coût pessimiste bornée 156
5.4 Exemple de situation pour le domaine *Robotbox*. 171
5.5 Résultat de Robotbox pour la première solution retournée 175
5.6 Résultat de Robotbox pour la meilleure solution retournée 176
5.7 Exemple de situation pour le domaine *Logistics*. 177
5.8 Résultat de Logistics pour la première solution retournée 180
5.9 Résultat de Logistics pour la meilleure solution retournée 181
5.10 Résultat de SimpleFPS pour la première solution retournée 186
5.11 Résultat de SimpleFPS pour la meilleure solution retournée 187

Liste des tableaux

2.1 Clauses associées aux effets 25

3.1 Temps de calculs pour retourner une solution sous-optimale 78

5.1 Résumé des différentes tâches pour le domaine *Robotbox*. 172
5.2 Tableau des résultats pour *Robotbox* 174
5.3 Résumé des différentes tâches pour le domaine *Logistics*. 178
5.4 Tableau des résultats pour *Logistics* 179
5.5 Résumé des différentes tâches pour le domaine *SimpleFPS*. 183
5.6 Tableau des résultats pour *SimpleFPS*. 184

Chapitre 1

Introduction

1.1 Contributions

Cette thèse étudie la planification hiérarchique appliquée à la simulation informatique en temps réel. Elle s'intéresse à la planification HTN (*Hierarchical Task Network*), un modèle de décision fondé sur la décomposition récursive de tâches en sous-tâches, et propose d'associer ces tâches à des actions abstraites, c'est-à-dire des transitions dans un espace d'états abstraits. Les contributions apportées par ces travaux sont les suivantes :

- Une sémantique pour des actions abstraites dans le cadre de la planification HTN, où ces actions appartiennent à une hiérarchie d'abstractions de l'espace concret.
- Une brève étude empirique des gains de performance obtenus en exploitant les actions abstraites pour évaluer l'exécutabilité des solutions partielles.
- Une sémantique alternative pour les actions abstraites, interprétées comme des transitions dans un espace d'états abstraits, décrit par un langage logique à trois valeurs.
- Un langage de modélisation HTN, nommé HTDL (*Hierarchical Task Description Language*), permettant de modéliser un domaine de planification HTN avec des actions abstraites.
- Un cadre théorique pour la planification HTN optimale, fondé sur la modélisation de coûts au niveau des actions abstraites et incluant un algorithme de planification *Anytime*.
- La définition d'heuristiques pondérées à partir des coûts des actions abstraites.
- Une évaluation empirique des performances de l'algorithme de planification optimal, pour différentes heuristiques et différentes stratégies de décomposition des actions abstraites.

1.2 Contexte industriel

Les travaux présentés dans ce manuscrit ont été effectués dans le cadre d'une thèse CIFRE au sein de la société Sagem Défense Sécurité, et ont été accomplis en collaboration avec le centre de recherche des écoles de Saint-Cyr Coëtquidan et le LAMSADE de l'université Paris-Dauphine. Ils s'inscrivent dans le cadre du développement d'outils de simulation tactique, c'est-à-dire de simulateurs informatiques adaptés aux besoins d'unités militaires à l'échelle de la section d'infanterie. En France, une section d'infanterie est une unité constituée d'un effectif de trente à quarante fantassins. Ces fantassins sont organisés à travers plusieurs échelons de hiérarchie, et obéissent à une chaine de commandement. Ainsi chaque fantassin fait partie d'une équipe de deux ou trois individus appelée respectivement « binôme » ou « trinôme », au moins deux équipes peuvent être regroupées pour former un groupe de combat, et plusieurs groupes de combat forment une section d'infanterie.

Chacun de ces sous-ensembles est commandé par un « chef » (d'équipe, de groupe ou de section) qui exécute les ordres émis depuis l'échelon supérieur et contrôle l'exécution des ordres transmis à ses subordonnés. Cette organisation de la section d'infanterie et de sa chaine de commandement est illustrée en figure 1.1.

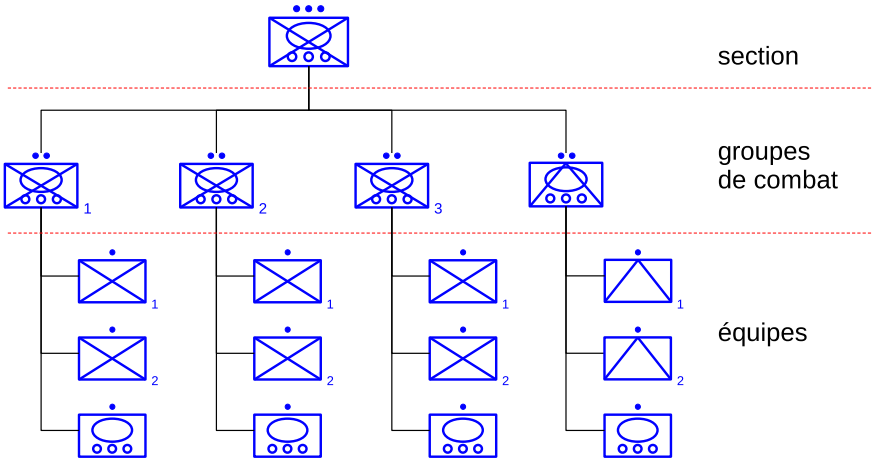


FIGURE 1.1 – Organisation d'une section d'infanterie mécanisée française
Cette section d'infanterie contient trois groupes de voltige et un groupe antichar.
Chaque unité est représentée par son symbole OTAN, d'après le standard *APP-6C*.

L'échelle de temps qui cadence les manœuvres d'une section d'infanterie est courte, se comptant généralement en heures, et les décisions prises se fondent en grande partie sur l'observation du terrain au contact duquel se trouve les fantassins. En conséquence, le niveau de représentation spatial et temporel d'un simulateur doit être au plus proche de l'environnement réel. Concrètement, cela signifie que le simulateur doit représenter un terrain de manœuvre virtuel en trois dimensions, et qu'il doit animer cet environnement en temps réel. Afin de répondre à un tel besoin, les *serious games*, où *jeux sérieux*, constituent une plateforme idéale. Derrière cette expression au allure d'oxymore, les jeux sérieux sont des logiciels conçus sur les mêmes bases que les jeux vidéo, mais qui répondent à un objectif utilitaire. Étant techniquement équivalent au jeux vidéo, ils dépendent donc des mêmes technologies : moteurs graphiques, moteurs physiques, intelligence artificielle, etc. Comme exemple de jeu sérieux développé pour la simulation d'infanterie, on peut citer la franchise *Virtual Battle Space* (VBS), un produit de *Bohémia Interactive Simulations*, et sur lequel Sagem s'appuie pour ces projets de simulation. La figure 1.2 représente par exemple un modèle de fantassin équipé du système FÉLIN (Fantassin à Équipement et Liaisons Intégrés), développé par Sagem, dans l'environnement virtuel de VBS2.

Ces outils de simulation tactique ont de nombreux usages, à la fois dans le milieu militaire et dans le milieu industriel. Ainsi, la simulation est d'abord utilisée comme outil de formation et d'entrainement des troupes. Le fantassin est alors directement projeté dans l'environnement virtuel en vue subjective, et doit accomplir un ensemble d'actions dans différents scénarios afin d'acquérir ou de renforcer ses savoir-faire. La qualité de cette phase de formation ou d'entrainement est directement liée à la capacité du simulateur à maintenir un bon niveau d'immersion de l'opérateur. Le problème de l'immersion se pose en partie au niveau du rendu graphique de l'environnement virtuel, ou de l'interface entre l'opérateur et la machine, mais pas uniquement. En effet, l'opérateur est amené à interagir avec des personnages virtuels, à la fois en tant que forces alliées et ennemies. Ces personnages non joueur (PNJ) doivent exprimer des comportements crédibles et conformes aux doctrines enseignées, sous peine de nuire à la qualité de la formation. Parmi les autres usages de la simulation, on peut aussi mentionner la préparation de missions, qui consiste à étudier l'efficacité d'un plan d'action en l'exécutant dans un environnement virtuel, ou

FIGURE 1.2 – Modèle 3D d'un soldat de l'armé de terre dans Virtual Battle Space 2
Le soldat est équipé du système FÉLIN, développé par Sagem Défense Sécurité.

encore l'utilisation de simulateur à des fins d'expérimentation, pour évaluer de nouvelles doctrines ou étudier l'impact de nouveaux équipements. Pour tous ces usages, la qualité du comportement des unités simulées est à nouveau essentielle, car elle détermine la cohérence des conditions expérimentales, et donc la qualité des résultats obtenus.

1.3 Problématique

Le point sensible d'un outil de simulation d'infanterie est donc l'intelligence artificielle « comportementale » des unités simulées. Dans l'industrie du jeux vidéo, ces comportements sont généralement modélisés à partir de machines à états finis. Dans ces machines à états, chaque état est associé à une action. Ainsi, une entité exécute la même action tant que son état n'est pas modifié. Le passage d'un état à un autre est modélisé par des transitions entre les états, et ces transitions sont associées à des conditions. Quand le moteur du simulateur évalue que la condition d'une transition est satisfaite, cette transition est exécutée et le comportement de l'unité concernée est modifié. Un exemple de machine à état permettant d'animer une équipe de fantassins dans un simulateur est représentée dans la figure 1.3. Ces machines à états finis sont bien adaptées à la modélisation de comportements simples. Pour des comportements plus complexes, comprenant de nombreuses actions de bas niveau et devant s'adapter à un grand nombre de situations différentes, la modélisation d'une machine à états devient une tâche très complexe. De plus, une machine à états complexe est un système difficile à maintenir et à faire évoluer.

C'est pour résoudre ce problème que la planification d'actions a été introduite dans l'industrie du jeu vidéo, dans le cadre du développement du jeu *F.E.A.R.* en 2005 [81]. *F.E.A.R.* est un jeu de tir en vue subjective (désigné sous l'acronyme FPS en anglais, pour *First-Person Shooter*), un type de jeu techniquement comparable à un simulateur d'infanterie. En planification d'actions, l'enchainement des actions n'est pas modélisé explicitement, mais est automatiquement calculé à partir d'un modèle logique des actions spécifiée dans un « domaine » de planification. L'état du monde est représenté sous la forme d'un ensemble de variables et chaque action est modélisée par un ensemble

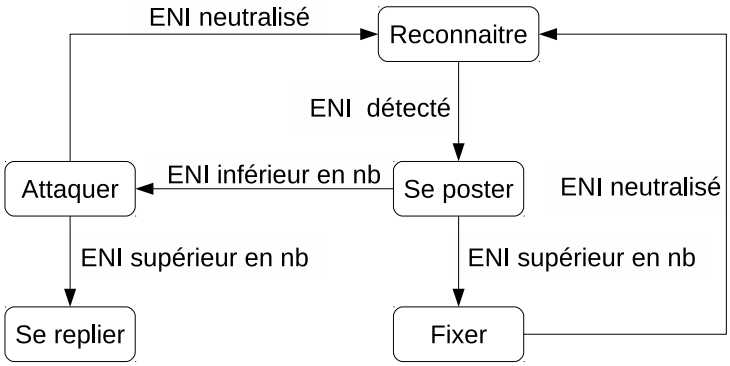


FIGURE 1.3 – Machine à état pour le contrôle d'une équipe de fantassins
La forme abrégée « ENI » désigne l'ennemi.

de préconditions et un ensemble d'effets. Les préconditions sont des formules logiques spécifiant dans quels états du monde l'action peut être exécutée, et les effets modélisent comment l'action modifie l'état du monde. Pour permettre à une entité de prendre une décision, on lui attribue alors un « but » à accomplir, sous la forme de conditions à satisfaire. Le planificateur prend en entrée la description de l'état du monde (les valeurs de chaque variable), ainsi que ce but et la description des actions, et retourne en sortie un « plan », qui est une séquence d'actions permettant d'accomplir le but donné à partir de l'état du monde courant. La figure 1.4 illustre ce processus de planification.

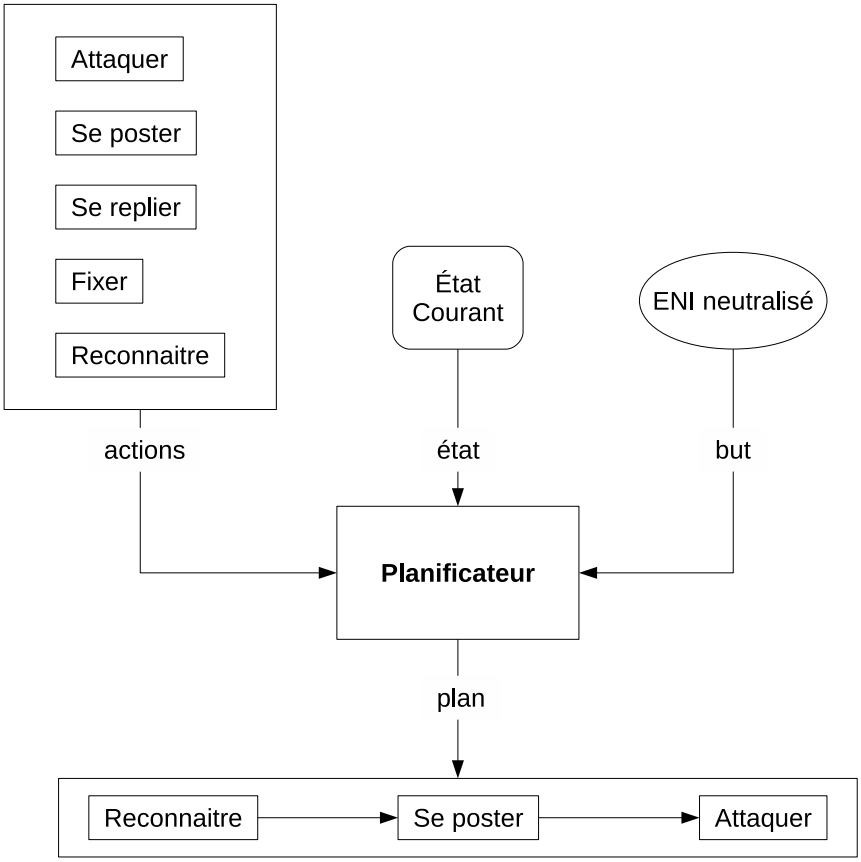


FIGURE 1.4 – Planification automatique d'un plan d'actions pour une équipe de fantassins

Cependant, cette simplicité au niveau de la modélisation se paye en temps de calcul. En effet les algorithmes de planification souffrent de l'explosion combinatoire : lors de la recherche d'un plan, par exemple en explorant l'espace des états du monde, le nombre d'états parcourus est exponentiel par rapport à la taille du plan retourné. Or, dans un

jeu vidéo ou un simulateur, le temps de calcul disponible est partagé entre l'intelligence artificielle et d'autres composants qui, comme le moteur graphique, consomment déjà une fraction importante du temps de calcul disponible. En conséquence, il est généralement admis que moins de 10 % du temps de calcul est disponible pour l'intelligence artificielle. Ainsi, il est possible de se livrer à une estimation de l'échelle du temps disponible pour planifier les actions des membres d'une section d'infanterie. Tout d'abord, on peut faire l'hypothèse que dans un environnement temps réel, une entité doit être capable de réagir à un évènement dans un délais de l'ordre de la seconde. Ramené à 10 %, et en considérant qu'une section d'infanterie contient de 30 à 40 fantassins, on arrive alors à un temps de calcul estimé à 2-3 millisecondes par fantassin. Le délai disponible pour produire un plan est donc très restreint.

Afin de satisfaire cette contrainte temporelle, les FPS récents implémentent généralement un type de planification hiérarchique appelée planification HTN (pour *Hierarchical Task Network*, soit « réseau de tâches hiérarchique » en français). En planification HTN, les actions sont associées à des tâches dites « primitives », auxquelles on ajoute des tâches non primitives décomposables sous la forme d'un réseau de sous-tâches (primitives et non primitives) ordonnées entre-elles. Les règles spécifiant la décomposition des tâches sont appelées « méthodes », et doivent être modélisées dans le domaine de planification. Ce type de planification requiert donc davantage de connaissances utilisateurs, mais s'avère aussi plus efficace en pratique que la planification d'actions classique. En effet la structure hiérarchique définit par les méthodes contraint les actions pouvant apparaitre dans le plan, et réduit donc l'explosion combinatoire. Pour obtenir un aperçu des performances de ces systèmes appliqué à la simulation, on peut se pencher sur une étude [49] qui a été menée sur deux jeux vidéo commerciaux : *Killzone 3* (2011) et *Transformers 3 : Fall of Cybertron* (2012). L'étude montre que les planificateurs HTN de ces deux jeux sont capables de générer en ligne des plans de 5 à 12 actions pour une douzaine de personnages non joueur simultanément. Ce nombre demeure toutefois deux à trois fois inférieur à l'effectif d'une section d'infanterie.

Mais la planification HTN dispose d'un avantage supplémentaire lorsqu'elle est appliquée pour la prise de décision par un ensemble d'agents organisés hiérarchiquement. En effet, sa propre structure hiérarchique permet de modéliser le principe de transmission des ordres dans la chaine de commandement via les méthodes de décomposition, et les contraintes d'ordre associées à ces méthodes permettent de modéliser les mesures de coordination entre les unités subordonnées. C'est par exemple l'approche suivie pour le planificateur *PlannedAssault* [108], dont les plans servent à générer automatiquement des scénarios de missions coordonnant des unités militaires dans le cadre de la simulation tactique. Par opposition, les jeux vidéo font généralement un usage décentralisé de la planification HTN : chaque entité dispose de son propre planificateur, et la transmission des ordres ainsi que la coordination sont gérées par un système supplémentaire.

Dans ces travaux, c'est l'approche centralisée qui a été retenue, du fait des avantages qu'elle apporte pour la modélisation de la chaine de commandement et de la coordination d'agents organisés hiérarchiqeuement. Toutefois, il y a un aspect de la prise de décision hiérarchisée que la planification HTN ne couvre pas : celui de pouvoir raisonner sur l'exécutabilité d'un plan à plusieurs niveaux du processus de décision. En effet, lorsqu'un chef de groupe ou de section planifie, il est capable de raisonner sur l'enchainement logique des actions de son plan sans avoir à considérer le détail des actions de chaque fantassin placé sous son commandement. Cette capacité d'abstraction est absente du cadre théorique général de la planification HTN, où seules les actions associées aux tâches primitives sont définies comme des transitions d'états. Ces travaux proposent donc d'étendre la planification HTN en considérant une sémantique de transition pouvant être associée aux tâches non primitives, qui seront alors considérées comme étant des actions

abstraites.

Pour rappel, l'objectif pratique de ces travaux est de pouvoir planifier en ligne pour une section d'infanterie complète, un objectif qui n'a pas encore été atteint, malgré l'emploi de la planification HTN . La problématique scientifique abordée est donc la suivante : la modélisation d'actions abstraites associées aux tâches non primitives permet-elle d'améliorer l'efficacité de la recherche de solution en planification HTN, pour des problèmes fortement hiérarchisés ? Cette problématique est explorée dans ces travaux à travers deux hypothèses. La première de ces hypothèses est directement liée à la sémantique de transition des actions abstraites, et suppose que la modélisation d'actions abstraites permet de réduire le nombre de solutions partielles à explorer en déterminant si un plan ne sera pas exécutable sans avoir à le décomposer au niveau primitif. La seconde hypothèse concerne la planification dans la perspective de l'optimalité, c'est à dire pour trouver un plan qui minimise le coût de ses actions. L'optimalité est un critère significatif pour l'application de la planification à la simulation. En effet, le comportement d'un personnage virtuel ne paraitra pas cohérent si celui-ci apparait ostensiblement sous-optimal. Cette hypothèse suppose qu'en associant les actions abstraites avec des coûts, on peut en plus obtenir des solutions partielles permettant d'estimer précisément le coût d'un plan primitif, et donc de fournir de bonnes heuristiques pour guider un planificateur HTN vers une solution optimale.

1.4 Organisation du manuscrit

La suite de ce manuscrit est organisée de la façon suivante :

— Le chapitre 1 présente un état de l'art de la planification d'actions orienté sur les technologies qui ont été appliquées dans les jeux vidéo. Cette état de l'art introduit aussi le cadre théorique et les notations qui seront exploités dans les chapitres suivants.

— Le chapitre 2 présente l'architecture de planification en ligne dans lequel s'inscrit le planificateur HTN multi-niveaux étudié. Il décrit l'implémentation et les performances de SHPE , le système de planification ayant servi de base à ce planificateur, et décrit le cœur de HTDL, le langage utilisé dans ces travaux pour modéliser les domaines de planification HTN.

— Le chapitre 3 présente deux approches pour la planification HTN combinée à un système de transition abstrait. La première approche combine la planification HTN avec la planification par hiérarchie d'abstractions, et contient une brève étude empirique des performances de ce système. La seconde approche permet de lever certaines restrictions de la première, et s'appuie sur un unique système de transitions abstrait fondé sur une logique à trois valeurs de vérité.

— Le chapitre 4 étudie la planification HTN combinée avec un système de transition abstrait dans le cadre de la planification optimale. Il définit un cadre théorique pour la planification HTN optimale, décrit et étudie un algorithme de planification optimal, et définit des heuristiques pondérées. Il se termine par une étude empirique des performances du système par rapport à trois domaines de planification.

— Enfin, le dernier chapitre conclut ces travaux et propose plusieurs pistes en vue de les approfondir dans le futur.

Chapitre 2

État de l'art

Ce chapitre poursuit deux objectifs. Le premier est de présenter un état de l'art des techniques de planification applicables dans le contexte de la simulation. Le second objectif est d'introduire les notations, définitions, propositions et algorithmes sur lesquels viendront s'appuyer les travaux présentés dans les chapitres suivants.

Au cours de cette étude de la littérature, nous étudierons successivement la planification d'actions classique, la planification HTN, ainsi que la planification par abstraction. Nous serons ainsi amenés à aborder les questions suivantes :

1. Pourquoi s'appuyer sur le langage ADL plutôt qu'un autre langage de planification plus simple (comme STRIPS) ?
2. En quoi la planification HTN est-elle adéquate pour la modélisation de problèmes appliqués aux unités militaires ?
3. En quoi la planification par abstraction dispose-t-elle d'un potentiel pour résoudre efficacement les problèmes de planification d'actions appliqués aux unités militaires ?

2.1 Planification classique

Le cadre de la planification abordé dans ces travaux est celui de la planification classique. Ce cadre suppose les hypothèses suivantes :

1. L'environnement est totalement observable.
2. L'environnement est statique.
3. Les actions s'exécutent instantanément.
4. Les actions sont déterministes.

L'hypothèse 1 signifie que toute information liée à l'environnement est disponible, et donc que l'état du monde peut être modélisé sans aucune incertitude. L'hypothèse 2 implique que seules les actions contrôlées par le système peuvent modifier l'état du monde, ce qui exclut donc la présence d'événements extérieurs. L'hypothèse 3 considère que les actions n'ont pas de durée, et enfin l'hypothèse 4 considère que le résultat de l'application d'une action est toujours prévisible. Dans le contexte de la simulation, la pertinence de chacune de ces hypothèses est discutable. Toutefois, elles demeurent raisonnables si l'objectif de la planification est l'exécution d'actions à un horizon temporel court. De plus, les techniques de planification utilisées actuellement par l'industrie du jeu vidéo s'inscrivent toutes dans le cadre posé par ces hypothèses.

Les techniques de planification étudiées dans cette section s'inscrivent donc dans ce cadre. La première sous-section décrit le modèle général de la planification classique, tandis que la sous-section suivante décrit la sémantique particulière des états et des actions pour le langage ADL. Les troisième et quatrième sous-sections décrivent deux catégories

distinctes de méthodes de planification classique : respectivement la planification dans l'espace des états, et la planification dans l'espace des plans. Enfin, la dernière sous-section traite de l'application de la planification classique aux jeux-vidéo.

2.1.1 Cadre général de la planification classique

Un problème de planification classique se modélise sous la forme d'un système de transition d'états déterministe. Dans ce système de transition d'états, chaque état correspond à une représentation abstraite d'une situation à un instant donné dans l'environnement réel. De même, les transitions entre ces états correspondent aux actions applicables dans chacune de ces situations, et chaque action peut être associée à un coût, c'est-à-dire à une pénalité liée à son exécution, et qui est modélisée numériquement. Ces différents éléments constituent une sous-partie du problème de planification appelé « domaine de planification », et qui est formellement définie de la façon suivante :

Définition 1 (Domaine de planification classique)**.** *Un domaine de planification classique est un quadruplet $D = (S, A, \mathcal{T}, \mathcal{C})$ tel que :*

- *S est un ensemble fini d'états du monde.*
- *A est un ensemble fini d'actions.*
- *$\mathcal{T} \in S \times A \to S$ est une fonction de transition. Il s'agit d'une fonction partiellement définie sur $S \times A$. Pour tout état $s \in S$ et action $a \in A$, on dira que a est applicable à s si et seulement si $\mathcal{T}(s, a)$ est défini.*
- *$\mathcal{C} \in S \times A \to \mathbb{R}^+$ est une fonction de coût. Le domaine de définition de cette fonction est identique à celui de la fonction $\mathcal{T}$.*

Pour compléter le modèle d'un problème de planification, on y adjoint un état initial, qui correspond à la situation dans laquelle se trouve l'agent, et on définit un but (aussi appelé objectif), correspondant à un ensemble d'états que l'on souhaite atteindre :

Définition 2 (Problème de planification classique)**.** *Un problème de planification classique est un triplet $P = (D, s_0, G)$ tel que :*

- *D est un domaine de planification classique.*
- *$s_0 \in S$ est l'état initial.*
- *$G \subseteq S$ est l'ensemble des états correspondant au but à atteindre.*

Une solution à ce problème est un plan, c'est-à-dire une séquence d'actions. On considérera dans ce manuscrit qu'une séquence d'action est un élément du monoïde libre A^* sur l'ensemble des actions A, en considérant la concaténation (noté $\circ$) comme loi de composition interne, ainsi que le plan vide (noté ϵ) comme élément neutre. Une séquence d'actions est dite exécutable en un état si toutes ses actions sont successivement applicables :

Définition 3 (Séquence d'actions exécutable)**.** *Soient $\sigma \in A^*$ une séquence d'actions et un état $s \in S$. σ est exécutable en s si et seulement si :*

1. *$\sigma = \epsilon$.*
2. *Il existe $a \in A$ applicable à s et $\sigma' \in A^*$ exécutable en $\mathcal{T}(s, a)$ tels que $\sigma = a \circ \sigma'$.*

Note : dans la suite de ce manuscrit, on étendra la définition de la fonction $\mathcal{T}$ aux séquences d'actions, et on en fera de même pour la fonction $\mathcal{C}$. Ainsi, on considérera que pour tout état $s \in S$, $\mathcal{T}(s, \epsilon) = s$ et $\mathcal{C}(s, \epsilon) = 0$. De plus, pour toute action $a \in A$ applicable à s et toute séquence d'actions $\sigma \in A^*$ exécutable dans $\mathcal{T}(s, a)$, on définit l'état final atteint par $a \circ \sigma$ de la façon suivante :

$$\mathcal{T}(s, a \circ \sigma) = \mathcal{T}(\mathcal{T}(s, a), \sigma).$$

Et de même pour le coût de $a \circ \sigma$:

$$\mathcal{C}(s, a \circ \sigma) = \mathcal{C}(s, a) + \mathcal{C}(\mathcal{T}(s, a), \sigma).$$

Une séquence d'actions est alors qualifiée de plan solution à un problème de planification, si et seulement si elle est exécutable dans l'état initial, et l'état final qu'elle atteint correspond au but spécifié :

Définition 4 (Solution à un problème de planification classique)**.** *Soient $P = (D, s_0, G)$ un problème de planification classique, et $\sigma \in A^*$ une séquence d'actions. σ est un plan solution de P si et seulement si :*

— *σ est exécutable dans s_0.*

— *$\mathcal{T}(s_0, \sigma) \in G$.*

Parmi l'ensemble des plans solutions à un problème de planification, certains de ces plans sont moins coûteux que tous les autres plans solutions. On considère que ces plans sont des solutions *optimales* au problème de planification posé :

Définition 5 (Solution optimale à un problème de planification classique)**.** *Soient $P = (D, s_0, G)$ un problème de planification classique, et $\sigma^* \in A^*$ un plan solution de P. σ^* est une solution optimale de P si et seulement si :*

$$\mathcal{C}(s_0, \sigma^*) = min\{\mathcal{C}(s_0, \sigma) \mid \sigma \text{ est une solution de } P\}$$

Pour terminer cette sous-section, la figure 2.1 propose un exemple simple de problème de planification sous la forme d'un système de transition d'états, et permet d'illustrer les différents éléments introduits.

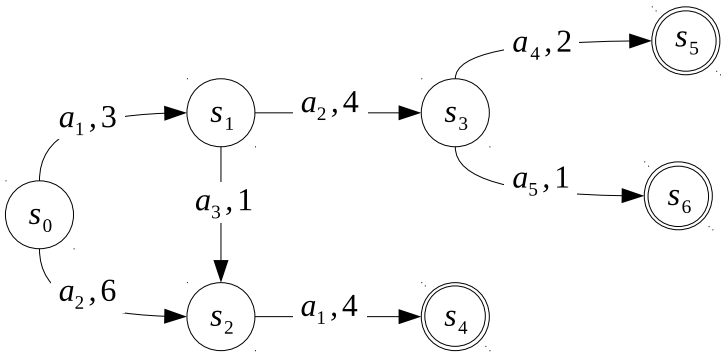


FIGURE 2.1 – Représentation graphique d'un problème de planification
s_0 est l'état initial, tandis que s_4, s_5 et s_6 sont des états correspondant au but à atteindre. Les séquences d'actions $a_1 \circ a_2 \circ a_4$, $a_1 \circ a_2 \circ a_5$, $a_1 \circ a_3 \circ a_1$ et $a_2 \circ a_1$ sont toutes des plans solutions, mais seules $a_1 \circ a_2 \circ a_5$ et $a_1 \circ a_3 \circ a_1$ sont des solutions optimales avec un coût total égal à 8.

2.1.2 ADL : un langage de description d'actions

La sous-section précédente a présenté une définition générale du modèle de transition d'états de la planification classique. Cette nouvelle sous-section s'intéresse au langage utilisé pour décrire les actions, et se focalise sur le langage ADL (*Action Description Language*) fondé par E. Pednault [83, 82]. D'autres langages de description d'actions existent, par exemple STRIPS [29] ou le calcul situationnel [62]. Dans STRIPS, un état est représenté par un ensemble de formules correspondant aux propositions satisfaites par cet état, et les effets d'une action sont représentés par deux ensembles de formules : une liste de

formules à ajouter à l'état (les nouvelles formules satisfaites après l'application de l'action), et une liste de formules à supprimer de l'état (les formules n'étant plus satisfaites après l'application de l'action). L'expressivité de STRIPS est limitée par la nécessité de figer les formules ajoutées ou supprimées par chaque action, sans pouvoir les définir en fonction de l'état sur lequel ces transformations sont appliquées. Cette simplicité permet toutefois de raisonner efficacement sur ces transformations. À l'opposé, le calcul situationnel propose une axiomatisation de la sémantique de la transition des actions en logique du premier ordre, et permet de résoudre ce problème d'expressivité, mais au détriment de l'efficacité computationnelle. ADL se situe juste entre ces deux langages. Il repose lui aussi sur la logique du premier ordre, et apporte ainsi un niveau d'expressivité supérieur à celui de STRIPS, mais impose des restrictions par rapport au calcul situationnel.

Pourquoi choisir ADL ? Ou, en formulant cette question autrement, pourquoi effectuer un choix en faveur de l'expressivité plutôt qu'en faveur de la complexité ? Ce choix a été motivé par la nature des problèmes abordés dans cette thèse, qui correspondent à un domaine de planification d'actions dédié au comportement de fantassins dans une section d'infanterie. Afin de décrire les règles du comportement de fantassins en fonction d'une chaîne de commandement, il est essentiel de pouvoir formuler des phrases quantifiées telles que « Il existe un soldat de l'équipe 1 équipé d'une arme antichar. » ou encore « Tout fantassin du groupe de combat n° 2 se trouve dans la zone 3. », etc. Or, de telles phrases ne peuvent être formulées que par un langage de planification s'appuyant sur la logique du premier ordre, ce que permet ADL.

Afin de décrire ADL, il faut tout d'abord considérer un langage logique du premier ordre, que l'on désignera par le symbole $\mathcal{L}$. Ce langage est constitué des éléments suivants :

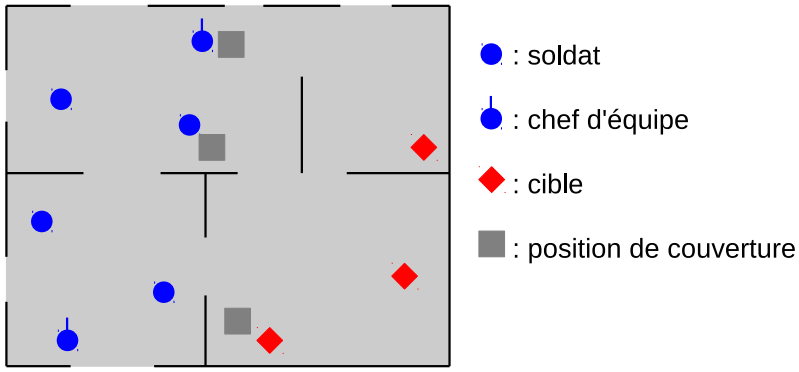
- les symboles vrai ($\top$) et faux ($\bot$) ;
- les connecteurs logiques usuels ($\neg$, $\wedge$, $\vee$, $\Longrightarrow$, $\Longleftrightarrow$) ;
- les quantificateurs existentiels et universels (respectivement $\exists$ et $\forall$) ;
- un ensemble infini de symboles de variables noté $VAR(\mathcal{L})$ (les variables seront représentées par des lettres minuscules, éventuellement indicées ou suivies du signe prime : x, y', z_1, etc.) ;
- un ensemble fini de symboles de relation (R_1, R_2, etc.), aussi appelés symboles de prédicats, et noté $PRED(\mathcal{L})$;
- le symbole d'égalité ($=$) ;
- un ensemble fini de symboles de fonctions (F_1, F_2, etc.) noté $FUNC(\mathcal{L})$.

On considérera ici que les constantes sont simplement des fonctions d'arité nulle. L'ensemble de tous ces éléments permet d'assembler des formules logiques (en suivant les règles de la logique des prédicats [107]). On note alors $PHRASE(\mathcal{L})$ l'ensemble des phrases de $\mathcal{L}$, c'est-à-dire l'ensemble des formules logiques sans variable libre, et $TERME_c(\mathcal{L})$ l'ensemble des termes clos (sans variable libre) de $\mathcal{L}$.

Ainsi dans ADL, les états du monde sont des interprétations du langage $\mathcal{L}$. Pour un état $s \in S$, on désignera alors par $dom(s)$ le domaine du discours associé à s, et par $s(R)$ et $s(F)$ les interprétations respectives dans s du symbole d'une relation R ou d'une fonction F. Un exemple d'état est proposé en figure 2.2. Cet état représente une situation où deux trinômes de fantassins sont placés dans un bâtiment occupé par trois unités ennemies. Cet état peut correspondre par exemple à un problème de capture de bâtiment en combat urbain.

Les éléments de l'univers du discours sont désignés par les termes clos du langage $\mathcal{L}$. Ces termes sont exprimés par des constantes, c'est-à-dire des symboles de fonction d'arité 0, et par des symboles de fonction suivis de listes de termes. L'interprétation dans un état s d'un terme t est alors noté $s(t)$. On considère que chaque objet du domaine du discourt correspond à une constante. Pour tout objet $o \in dom(s)$, on considère donc que :

$$s(o) = o$$



$$
\begin{aligned}
s(Coverpoint) &= \{position4,\ position6,\ position8\}\\
s(InRoom) &= \{(position1,\ room1),\ (position2,\ room1),\\
&\quad (position3,\ room1),\ (position4,\ room2),\\
&\quad (position5,\ room2),\ (position6,\ room2),\\
&\quad (position7,\ room3),\ (position8,\ room4),\\
&\quad (position9,\ room4)\}\\
s(Adjacent) &= \{(room1,\ room2),\ (room1,\ room4),\ (room2,\ room1),\\
&\quad (room2,\ room3),\ (room2,\ room4),\ (room3,\ room2),\\
&\quad (room3,\ room4),\ (room4,\ room1),\ (room4,\ room2),\\
&\quad (room4,\ room3)\}\\
s(MemberOfFireteam) &= \{(soldier1,\ fireteam1),\ (soldier2,\ fireteam1),\\
&\quad (soldier3,\ fireteam1),\ (soldier4,\ fireteam2),\\
&\quad (soldier5,\ fireteam2),\ (soldier6,\ fireteam2)\}\\
s(FireteamLeader) &= \{(fireteam1,\ soldier1),\ (fireteam2,\ soldier4)\}\\
s(SoldierAt) &= \{(soldier1,\ position1),\ (soldier2,\ position2),\\
&\quad (soldier3,\ position3),\ (soldier4,\ position4),\\
&\quad (soldier5,\ position5),\ (soldier6,\ position6)\}\\
s(TargetAt) &= \{(target1,\ position7),\ (target2,\ position8),\\
&\quad (target3,\ position9)\}
\end{aligned}
$$

FIGURE 2.2 – Description d'un état
Description partielle d'un état correspondant à une structure d'interprétation pour un langage logique du premier ordre. Les symboles de prédicat et de fonction de ce langage permettent de décrire une situation pour la modélisation d'un problème de capture de bâtiment.

D'autre part, pour tout symbole de fonction F d'arité n, liste de termes $t_1, ..., t_n$ et objet $o \in dom(s)$, on a :

$$s(F(t_1, ..., t_n)) = o \text{ si et seulement si } (s(t_1), ..., s(t_n), o) \in s(F)$$

Une interprétation des phrases de $\mathcal{L}$ dans s peut alors être défini via un opérateur d'évaluation $[\![\cdot]\!]_s \in PHRASE(\mathcal{L}) \to \{0, 1\}$. Cet opérateur associe chaque phrase de $\mathcal{L}$ avec la valeur 1 si cette phrase est satisfaite dans s ou 0 si cette phrase n'est pas satisfaite. Il est défini de la façon suivante :

1. $[\![\bot]\!]_s = 0$
2. $[\![\top]\!]_s = 1$
3. $[\![t_1 = t_2]\!]_s = 1$ si $s(t_1) = s(t_2)$ sinon 0
4. $[\![R(t_1, ..., t_n)]\!]_s = 1$ si $(s(t_1), ..., s(t_n)) \in s(R)$ sinon 0
5. $[\![\neg\varphi]\!]_s = 1 - [\![\varphi]\!]_s$

6. $[\![\varphi \wedge \psi]\!]_s = \min([\![\varphi]\!]_s, [\![\psi]\!]_s)$
7. $[\![\varphi \vee \psi]\!]_s = \max([\![\varphi]\!]_s, [\![\psi]\!]_s)$
8. $[\![\varphi \implies \psi]\!]_s = \max(1 - [\![\varphi]\!]_s, [\![\psi]\!]_s)$
9. $[\![\varphi \iff \psi]\!]_s = 1 - |[\![\varphi]\!]_s - [\![\psi]\!]_s|$
10. $[\![\forall x : \varphi]\!]_s = \min\{[\![\varphi[o/x]]\!]_s \mid o \in dom(s)\}$
11. $[\![\exists x : \varphi]\!]_s = \max\{[\![\varphi[o/x]]\!]_s \mid o \in dom(s)\}$

où R représente un symbole de relation, les t_1,t_2,...,t_n représentent des termes clos de $\mathcal{L}$, φ et ψ représentent des phrases de $\mathcal{L}$, et $\varphi[t/x]$ est la formule logique obtenue en substituant les occurrences de la variable x dans φ par le terme t. On utilisera par la suite le symbole $\models$ pour exprimer qu'un état s satisfait une phrase $\varphi \in PHRASE(\mathcal{L})$: $s \models \varphi \iff [\![\varphi]\!]_s = 1$.

Pour toute phrase $\varphi \in PHRASE(\mathcal{L})$, on notera aussi $S_\varphi = \{s \in S \mid s \models \varphi\}$ l'ensemble des états satisfaisant φ. Ainsi en ADL, l'ensemble G des états à atteindre est décrit par une phrase de $\mathcal{L}$, notée g. On a donc $G = S_g$. De même, les préconditions des actions ainsi que les transformations d'états sont décrites sous la forme de formules logiques du langage $\mathcal{L}$. Il existe trois types de formules permettant de décrire les transformations d'état : les conditions d'ajout et de suppression (*add conditions* et *delete conditions* en anglais), qui s'appliquent aux symboles de relation, et les conditions d'affectation (*update conditions* en anglais), qui s'appliquent aux symboles de fonctions. À ces formules sera ajouté un élément supplémentaire, et qui ne fait pas partie de la définition ordinaire d'ADL : un terme dont l'interprétation correspond à un nombre dans n'importe quel état, et qui permet de calculer le coût de l'action (on considérera pour cela que le domaine du discours inclut les nombres réels).

Définition 6 (Description d'une action)**.** *En ADL, une action $a \in A$ est décrite par l'ensemble des expressions de $\mathcal{L}$ suivantes :*

- *Une condition d'exécution (aussi appelée préconditions) π^a qui définit l'ensemble des états auxquels a est applicable : $\forall s \in S, s \models \pi^a$ si et seulement si a est applicable à s (S_{π_a} est donc l'ensemble des états auxquels a est applicable).*
- *Des conditions d'ajout $\alpha^a_R(x_1, ..., x_n)$ et de suppression $\delta^a_R(x_1, ..., x_n)$ pour chaque symbole de relation R d'arité $n \in \mathbb{N}$ où $x_1, ..., x_n$ sont des variables libres de $\mathcal{L}$.*
- *Une condition d'affectation $\mu^a_F(x_1, ..., x_n, y)$ pour chaque symbole de fonction F d'arité $n \in \mathbb{N}$ où $x_1, ..., x_n, y$ sont des variables libres de $\mathcal{L}$.*
- *Un terme de coût κ^a tel que $\forall s \in S, s(\kappa^a) \in \mathbb{R}^+$.*

Les conditions d'ajout, de suppression et d'affectation sont les éléments qui permettent de modéliser la fonction de transition $\mathcal{T}$ du domaine de planification. Elles se traduisent par des transformations d'ensembles, où ces ensembles sont les interprétations des symboles de relations et de fonctions des états. On utilisera désormais l'expression de « progression d'état » pour désigner cette transformation d'un état en un nouvel état après l'exécution d'une action.

Définition 7 (Progression d'un état)**.** *Soient une action $a \in A$ et deux états $s, s' \in S$ tels que $s \models \pi^a$ et $\mathcal{T}(s, a) = s'$. Alors on peut calculer s' de la façon suivante :*

- *$dom(s') = dom(s)$.*
- *Pour tout symbole de relation R d'arité $n \in \mathbb{N}$, $s'(R) = (s(R) - D_R) \cup A_R$ où :*

$$A_R = \{(o_1, ..., o_n) \in dom(s)^n \mid s \models \alpha^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]\}$$
$$D_R = \{(o_1, ..., o_n) \in dom(s)^n \mid s \models \delta^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]\}$$

— *Pour tout symbole de fonction F d'arité $n \in \mathbb{N}$, $s'(F) = (s(F) - D_F) \cup A_F$ où :*

$$A_F = \{(o_1, ..., o_n, v) \in dom(s)^{n+1} \mid s \models \mu_F^a(x_1, ..., x_n, y)[o_1/x_1, ..., o_n/x_n, v/y]\}$$
$$D_F = \{(o_1, ..., o_n, v) \in dom(s)^{n+1} \mid s \models (\exists y : \mu_F^a(x_1, ..., x_n, y))[o_1/x_1, ..., o_n/x_n]\}$$

Syntaxiquement, les actions sont décrites par quatre groupes différents de clauses. Un premier groupe, désigné par le mot clé « *Precond* », contient une liste de formules logiques dont la conjonction correspond à la condition d'exécutabilité de l'action. Ensuite, trois groupes de clauses sont utilisés pour modéliser les conditions d'ajout, de suppression et de retrait, et sont respectivement désignés par les mots clé « *Add* », « *Delete* » et « *Update* ». Ces clauses ne correspondent pas exactement aux formules de transformation : leur syntaxe, ainsi que les formules équivalentes sont décrites dans la table 2.1.2. Lorsque plusieurs de ces clauses concernent le même symbole de relation ou de fonction, la condition complète est formée à partir de la disjonction des formules correspondant à chaque clause. Enfin, un cinquième élément contenant le terme représentant le coût, et désigné par le mot clé « *Cost* », est ajouté à ces quatre groupes de clauses.

clause	$\alpha_R(x_1, ..., x_n)/\delta_R(x_1, ..., x_n)/\mu_F(x_1, ..., x_n, y)$
$R(t_1, ..., t_n)$	$x_1 = t_1 \wedge ... \wedge x_n = t_n$
$R(t_1, ..., t_n)$ si φ	$x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge \varphi$
$R(t_1, ..., t_n)$ pour tous $z_1, ..., z_k$	$\exists z_1, ..., z_k : x_1 = t_1 \wedge ... \wedge x_n = t_n$
$R(t_1, ..., t_n)$ pour tous $z_1, ..., z_k$ tels que φ	$\exists z_1, ..., z_k : x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge \varphi$
$F(t_1, ..., t_n) = v$	$x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge y = v$
$F(t_1, ..., t_n) = v$ si φ	$x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge y = v \wedge \varphi$
$F(t_1, ..., t_n) = v$ pour tous $z_1, ..., z_k$	$\exists z_1, ..., z_k : x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge y = v$
$F(t_1, ..., t_n) = v$ pour tous $z_1, ..., z_k$ tels que φ	$\exists z_1, ..., z_k : x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge y = v \wedge \varphi$

TABLE 2.1 – Clauses associées aux effets
Clauses d'ajout, de suppression et d'affectation, avec leurs conditions correspondantes. R et F représente respectivement un symbole de relation et un symbole de fonction (tous deux d'arité n).

Afin de permettre la description de toutes les actions d'un domaine sans avoir à toutes les énumérer, celle-ci sont décrites, non pas par les actions elles-mêmes, mais par des « opérateurs ». Ces opérateurs sont des versions paramétrées des actions. Ils modélisent en une seule fois toute une famille d'actions qui ne diffèrent entre elles que par les différentes instanciations possibles des variables correspondant aux paramètres. En plus des éléments déjà cités, ces opérateurs contiennent le nom de la famille d'actions représentée ainsi que la liste des paramètres à instancier. La figure 2.3 représente deux exemples d'opérateurs.

2.1.3 Planification dans l'espace des états

Cette sous-partie passe en revue un premier type d'algorithmes permettant de résoudre un problème de planification. Ces algorithmes ont en commun d'avoir été conçus pour résoudre le problème consistant à trouver un chemin entre deux nœuds d'un graphe. Afin de transposer ces algorithmes aux problèmes de planification automatique, il suffit simplement de considérer le graphe formé par les états reliés entre eux par les actions, et de rechercher un chemin entre l'état initial et un état appartenant à l'ensemble d'états caractérisé par l'objectif à atteindre. Ce chemin est alors constitué d'actions permettant de passer d'un état à un autre, et forme donc un plan.

$SoldierMoveInRoom(s,\ p_1,\ p_2,\ r)$
Precond:
$Soldier(s) \ \wedge \ Position(p_1) \ \wedge \ Position(p_1)$,
$InRoom(p_1,\ r) \ \wedge \ InRoom(p_2,\ r)$,
$SoldierAt(s,\ p_1)$
Add:
$SoldierAt(s,\ p_2)$
Delete:
$SoldierAt(s,\ p_1)$
Cost:
$Distance(p_1,\ p_2)$

$SoldierThrowGrenade(s,\ r_1,\ r_2)$
Precond:
$Soldier(s) \ \wedge \ Room(r_1) \ \wedge \ Room(r_2)$,
$Adjacent(r_1,\ r_2) \ \wedge \ SoldierGrenadeAmmo(s) \ > \ 0$
Add:
$Neutralized(t)$ pour tout t tel que $TargetAtRoom(t,\ r_2)$
Update:
$SoldierGrenadeAmmo(s) \ \leftarrow \ SoldierGrenadeAmmo(s) \ - \ 1$
cost:
5

FIGURE 2.3 – Exemple d'opérateurs décrits avec la syntaxe du langage ADL
Le premier opérateur modélise le déplacement d'un soldat entre deux positions à l'intérieur d'une même pièce. Le deuxième opérateur modélise le jet d'une grenade dans une pièce voisine.

Ces algorithmes sont généralement décrits comme parcourant un espace de recherche correspondant aux nœuds du graphe, et donc aux états dans le cas particulier de la planification d'actions. Pour cette raison, on parlera donc d'algorithmes de recherche dans l'espace des états. Ainsi, en ce qui concerne les différents algorithmes présentés ici, les nœuds de l'espace de recherche correspondront à un état courant et à un plan partiel constitués des actions exécutées depuis l'état initial. Ces algorithmes seront donc spécialisés au cas de la planification « vers l'avant », où les plans sont construits en ajoutant les actions dans l'ordre de leur exécution. Ce procédé apporte un avantage dans le contexte de la planification en ligne, où le temps consacré à la planification est limité : un plan préfixe est disponible à tout instant au cours de la recherche de solution, ce qui permet de retourner des actions à exécuter même si une solution finale n'a pas pu être découverte.

Algorithmes non informés

Les algorithmes présentés ici sont dits « non informés », car ils parcourent l'espace de recherche de façon systématique, sans bénéficier d'information leur permettant de diriger leur recherche vers les solutions possibles. Un premier algorithme commun parcours l'espace de recherche en « largeur d'abord ». Cet algorithme est nommé BFS, pour *Breadth First Search* en anglais. Il explore l'espace de recherche en donnant priorité au nœuds dont la profondeur est la plus faible. Un autre algorithme commun parcours l'espace de recherche en « profondeur d'abord » et est nommé DFS pour *Depth First Search* en anglais. Avec cet algorithme, le prochain nœud à explorer est toujours choisi parmi les

successeurs du dernier nœud exploré.

Une implémentation itérative d'un algorithme non informé générique est décrite par l'algorithme 1. Cet algorithme correspond à DFS lorsque l'insertion et l'extraction d'éléments dans l'ensemble *Ouvert* suit le principe d'une pile LIFO (*Last In, First out*, ce qui signifie « dernier arrivé, premier sortie » en français). Ainsi, l'élément extrait est toujours le dernier élément inséré, et se situe donc au plus profond niveau de l'arbre de recherche. Si à l'opposé, *Ouvert* est implémenté par une pile FIFO (*First In, First out*, ce qui signifie « premier arrivé, premier sortie » en français), alors l'algorithme 1 correspond à BFS. Afin de ne pas boucler infiniment à causes de cycles dans le graphe parcouru, l'ensemble $Fermé$ permet de marquer les états ayant déjà été visités. Ce marquage est superflu dans le cas où l'espace de recherche dispose d'une structure d'arbre orienté.

Algorithm 1 Algorithme de recherche non informé

entrées : un problème de planification P
sorties : un plan solution de P ou *échec*

1: $Ouvert \leftarrow (s_0, \epsilon)$
2: $Fermé \leftarrow \emptyset$
3: **tant que** $Ouvert$ n'est pas vide **faire**
4: $s, \sigma \leftarrow$ extraire un élément de $Ouvert$
5: $Fermé \leftarrow Fermé \cup \{s\}$
6: **si** $s \in G$ **alors**
7: **retourner** σ
8: **fin si**
9: **pour tout** $a \in A$ applicable à s **faire**
10: $s' = \mathcal{T}(s, a)$
11: **si** $s' \notin Fermé$ **alors**
12: $Ouvert \leftarrow Ouvert \cup \{(s', \sigma \circ a)\}$
13: **fin si**
14: **fin pour**
15: **fin tant que**
16: **retourner** *échec*

BFS et DFS ont tous les deux des avantages et des inconvénients. Ainsi BFS consomme une très grande quantité de mémoire, car il doit stocker un nombre de nœuds exponentiel par rapport à la taille du plan retourné, tandis que la taille de la mémoire consommée par DFS est linéaire par rapport à la taille de ce plan. Toutefois BFS permet toujours de trouver un plan solution s'il en existe au moins un, même si l'espace de recherche est infini. DFS au contraire, peut se « perdre » dans une branche infinie de l'espace de recherche ne contenant pas de solution, et donc s'exécuter indéfiniment sans jamais retourner de solution. Une façon de parer ce défaut est d'imposer une limite à la profondeur de l'espace de recherche exploré, mais cela se fait encore au détriment de la complétude. L'algorithme IDDFS [56], pour *Iterative Deepening Depth First Search* en anglais, permet de combiner les avantages de BFS et de DFS tout en se passant de leurs inconvénients. Cet algorithme consiste à exécuter successivement la procédure décrite par algorithme 1 avec une profondeur limitée, mais en augmentant successivement cette limite. Comme DFS, la taille de la mémoire utilisée est toujours linéaire par rapport à la taille du plan retourné, tandis que les nœuds sont explorés dans un ordre équivalent à celui de BFS, ce qui permet de garantir la complétude de l'algorithme. Bien que certains nœuds soient explorés à chaque itération, ce phénomène est négligeable, le temps consommé pour la dernière itération dominant asymptotiquement le temps consommé par les itérations précédentes.

Recherche heuristique

L'algorithme présenté ici est appelé A* [34]. Contrairement aux algorithmes précédents, il a été conçu pour retourner le plus court chemin entre deux nœuds d'un graphe (et non pas n'importe quel chemin), et permet donc de trouver une solution optimale à un problème de planification. Il s'agit aussi d'un algorithme « informé », car la recherche de solution y est guidée par une fonction heuristique. Il fait partie de l'ensemble des algorithmes de recherche *best-first* (ce qui veut dire « meilleur d'abord » en anglais). Ces algorithmes utilisent une fonction d'évaluation, généralement notée f, qui permet d'estimer le coût de la meilleure solution atteignable depuis un plan partiel. Ces plans sont alors explorés en donnant la priorité à ceux dont la valeur de f est la plus faible. Dans le cas de A*, cette fonction est définie par $f(s) = g(s) + h(s)$, où $g(s)$ représente le coût du plan partiellement construit permettant d'atteindre s à partir de l'état initial, et où $h(s)$ est une estimation du coût du plan optimal permettant d'atteindre le but depuis s. On fait référence à h par le nom de fonction heuristique. Une implémentation itérative d'A* est proposée par l'algorithme 2. Un ensemble $Ouvert$ permet de marquer les plans partiels à explorer, tandis qu'un ensemble $Fermé$ permet de marquer les états ayant déjà été visités : tout plan partiel qui atteint un état de $Fermé$ avec un coût plus élevé que le plan précédent est ignoré.

Algorithm 2 Algorithme A*

entrées : un problème de planification P
sorties : un plan solution de P ou *échec*

$g(s_0) \leftarrow 0$
$f(s_0) \leftarrow g(s_0) + h(s_0)$
$Ouvert \leftarrow \{(s_0, \epsilon)\}$
$Fermé \leftarrow \emptyset$
tant que $Ouvert \neq \emptyset$ **faire**
  $s, \sigma \leftarrow$ choisir un élément de $Ouvert$ minimisant f
  $Fermé \leftarrow Fermé \cup \{s\}$
  si $s \in G$ **alors**
    retourner σ
  sinon
    pour tout $a \in A$ applicable à s **faire**
      $s' \leftarrow \mathcal{T}(s, a)$
      si $(s' \notin Fermé$ **et** $\forall \sigma' \in A^*, (s', \sigma') \notin Ouvert)$ **ou** $g(s) + \mathcal{C}(s, a) < g(s')$ **alors**
        $g(s') \leftarrow g(s) + \mathcal{C}(s, a)$
        $f(s') \leftarrow g(s') + h(s')$
        $Ouvert \leftarrow Ouvert \cup \{(s', \sigma \circ a)\}$
      fin si
    fin pour
  fin si
fin tant que
retourner *échec*

A* termine et est complet si tous les coûts sont strictement positifs. L'optimalité de la solution, ainsi que la performance de l'algorithme, dépendent toutefois de la fonction heuristique. Ainsi, plus l'estimation de la fonction heuristique est précise, et plus la recherche est efficace. L'optimalité de la solution retournée est aussi garantie si la fonction heuristique est optimiste, c'est-à-dire qu'elle ne surestime jamais le coût d'une solution optimale atteignable depuis un plan partiel (on parle alors d'une heuristique « admissible »).

Définition 8 (Heuristique admissible)**.** *Une heuristique h est admissible si et seulement si pour tout état $s \in S$:*

$$h(s) \leq min\{\mathcal{C}(s,\sigma) \mid \sigma \in A^* \wedge \mathcal{T}(s,\sigma) \in G\}.$$

Une condition encore plus forte, appelée « consistance » (ou aussi « monotonicité »), peut être appliquée à la fonction heuristique. Cette condition exige que la fonction d'évaluation f soit croissante, c'est-à-dire que la valeur de f pour un état doit être inférieure à celle de ces successeurs dans l'arbre de recherche.

Définition 9 (Heuristique consistante)**.** *Une heuristique h est consistante si et seulement si pour tout état $s \in S$ et toute action $a \in A$ applicable à s :*

$$h(s) \leq \mathcal{C}(s,a) + h(\mathcal{T}(s,a)).$$

Avec cette propriété, non seulement la fonction heuristique est admissible, et donc A* retourne une solution optimale, mais en plus il peut être démontré qu'A* est lui même optimal, c'est-à-dire qu'aucun algorithme bénéficiant d'une heuristique aussi bien informée ne pourra être plus performant. Pour prendre connaissance des preuves relatives à ces propositions, le lecteur est invité à se référer à l'article ayant introduit A* [34].

Il est possible d'améliorer l'efficacité d'A* en utilisant des heuristiques non-admissibles. En surestimant à certains points le coûts d'une solution optimale, ces heuristiques permettent de diriger la recherche davantage en profondeur, et donc d'atteindre une solution plus rapidement. Bien que cela se fasse au détriment de l'optimalité de la solution, il est possible de borner l'erreur obtenue entre le coût de la solution retournée et celui d'une solution optimale. Pour cela, un moyen est d'utiliser une heuristique pondérée [86]. Pour obtenir une telle heuristique, on multiplie simplement la valeur d'une heuristique admissible par un poids $\omega > 1$. La fonction d'évaluation considérée devient donc $f = g + \omega \times h$. Le coût de la solution retournée sera alors au maximum plus élevé d'un facteur $\omega - 1$ par rapport au coût d'une solution optimale (on parle alors d'heuristique ϵ-admissible, où $\epsilon = \omega - 1$). D'autres formes d'heuristiques pondérées ϵ-admissibles ont été étudiées. Il est par exemple possible d'utiliser une pondération dynamique [87], exprimée sous la forme d'une fonction de la profondeur $w(\sigma) = 1 + \epsilon \times (1 - \frac{d_s}{n})$ où s est un état, d_s correspond à la profondeur de cet état dans l'arbre de recherche et n est la profondeur estimée de la solution optimale.

Recherche par séparation et évaluation

La recherche par séparation et évaluation (BB, pour *Branch and Bound* en anglais) est une technique générale en optimisation combinatoire. À ce titre, elle peut être employée pour résoudre des problèmes de recherche du plus court chemin dans un graphe. Cette technique fonctionne en répétant successivement deux étapes jusqu'à trouver une solution. La première étape est appelée « séparation », et consiste à séparer l'ensemble des solutions en plusieurs sous-ensembles. Pour le parcours d'un graphe, chaque plan partiel représente un sous-ensemble de solutions, qui correspond à l'ensemble des solutions dont ce plan partiel est un préfixe. La séparation consiste donc à ajouter une nouvelle action au plan partiel courant. La seconde étape est « l'évaluation », et consiste à comparer une sous-évaluation du coût de la solution optimale comprise dans un ensemble de solutions avec une sur-évaluation. Si la borne inférieure dépasse la borne supérieure, alors l'ensemble de solutions considéré ne contient pas de solution optimal au problème posé, et peut être « élagué », c'est-à-dire supprimé de l'espace de recherche. La borne inférieure est apportée par une fonction d'évaluation, tandis que la borne supérieure correspond aux coûts des solutions non optimales obtenues successivement.

L'algorithme 3 propose une implémentation itérative de cet algorithme en pseudo-code. L'étape déterminant l'ordre de parcours de l'espace de recherche est décrite à la

ligne 5. Cette étape a volontairement été décrite de façon générique. En effet, celle-ci peut être implémentée de multiples façons. Par exemple, l'algorithme de BB peut être combiné avec l'algorithme DFS pour le parcours du graphe (on fera référence à cet algorithme avec l'acronyme DFBB). DFBB peut être implémenté de façon à pouvoir être interrompu à tout moment, et tel que la qualité de la solution retournée s'améliore si le durée de la recherche augmente. Ainsi DFBB ne retourne aucune solution s'il est interrompu trop tôt, puis une succession de solutions de coûts décroissants, jusqu'à atteindre une solution optimale. Cette fonctionnalité désigne une classe d'algorithmes appelée *anytime* [16]. Ces algorithmes sont particulièrement utiles dans le contexte des jeux vidéo compte tenu du temps de calcul limité allouable à la prise de décision [36]. Si en plus une fonction heuristique est disponible, il est possible de réaliser un algorithme *anytime* en combinant A* et BB. Dans ce cas, l'heuristique employée ne doit pas être admissible. Ainsi l'algorithme peut retourner plus tôt une solution sous-optimale, et s'appuyer sur le BB pour améliorer cette solution. C'est ce que propose par exemple l'algorithme *Anytime A**, pour lequel une heuristique admissible permet d'estimer la borne inférieure du BB et aussi, une fois pondérée, de guider la recherche de solution.

Algorithm 3 Algorithme de recherche par séparation et évaluation

entrées : un problème de planification P, une borne supérieur initiale b
sorties : un plan solution de P optimale

1: $borne \leftarrow \infty$
2: $solution \leftarrow$ *échec*
3: $Ouvert \leftarrow \{(s_0, \epsilon)\}$
4: $Fermé \leftarrow \emptyset$
5: **tant que** $Ouvert \neq \emptyset$ **faire**
6: $s, \sigma \leftarrow$ extraire un élément de $Ouvert$
7: $Fermé \leftarrow Fermé \cup \{s\}$
8: **si** $s \in G$ **alors**
9: $borne = \mathcal{C}(s_0, \sigma)$
10: $solution \leftarrow \sigma$
11: **sinon**
12: **pour tout** $a \in A$ applicable à s **faire**
13: $s' \leftarrow \mathcal{T}(s, a)$
14: **si** $s' \notin Fermé$ **et** $f(s') < borne$ **alors**
15: $Ouvert \leftarrow Ouvert \cup \{(s', \sigma \circ a)\}$
16: **fin si**
17: **fin pour**
18: **fin si**
19: **fin tant que**
20: **retourner** *solution*

2.1.4 Planification dans l'espace des plans partiels

Pour tous les algorithmes présentés jusqu'ici, l'espace de recherche est composé de plans partiels. Cependant, le procédé de construction est linéaire : les plans sont formés de façon totalement ordonnée, et les actions sont toujours ajoutées en fin de séquence. Il est toutefois possible d'appliquer une technique non linéaire, où l'ordre des actions peut être partiel, et où les actions peuvent être ajoutées au plan dans un ordre quelconque. Cette méthode est appelée planification non linéaire, ou aussi planification POP (pour *Partial Order Planning* en anglais). Elle a initialement été développée empiriquement

avec les planificateurs hiérarchiques NOAH [91] et NONLIN [99], puis a été formalisée en l'appliquant à la planification classique de type STRIPS (TWEAK [14], SNLP [61]), et a finalement été étendue pour couvrir l'expressivité du langage ADL avec UCPOP [84]. Cette sous-partie passe d'abord en revue les caractéristiques de cette autre façon de représenter les plans, puis aborde les techniques de recherche de solution propre à la planification POP.

Plan partiel

En planification POP, les plans sont donc des ensembles d'actions partiellement ordonnées. Par opposition au séquences d'actions utilisées pour représenter des plans totalement ordonnés, on parlera ici de « réseau d'actions » afin de désigner la structure qui représente un plan partiellement ordonné. En effet, celle-ci peut se représenter graphiquement sous la forme d'un graphe orienté acyclique. Dans un tel graphe, les éléments ne sont pas les actions mêmes, mais des nœuds étiquetés par des actions (ainsi une même action peut apparaitre à plusieurs endroits dans un même plan).

Définition 10 (Réseau d'actions)**.** *Soit A un ensemble d'actions. Un réseau d'actions sur A est un couple $\rho = (N_\rho, \prec_\rho)$ tel que :*

1. *N_ρ est un ensemble fini de nœuds, tel que chaque nœud $n \in N$ est associé à une action $a_n \in A$.*
2. *$\prec_\rho$ est une relation d'ordre partiel définie sur N_ρ. Pour toute paire $n_1, n_2 \in \prec_\rho$, on notera $n_1 \prec_\rho n_2$.*

Un exemple de réseau d'actions est illustré en figure 2.4. Un réseau d'actions étant plus général qu'une séquence d'actions, celui-ci peut faire figure de représentant pour tout un ensemble de séquences. Pour les obtenir, il suffit alors de construire tous les ordres totaux compatibles avec l'ordre partiel du réseau d'actions. Par exemple, le réseau d'actions de la figure 2.4 permet de représenter les 4 séquences d'actions $a_1 \circ a_2 \circ a_3 \circ a_1$, $a_1 \circ a_2 \circ a_1 \circ a_3$, $a_1 \circ a_3 \circ a_2 \circ a_1$ et $a_2 \circ a_1 \circ a_1 \circ a_3$. On parlera alors de « linéarisation » d'un réseau d'actions pour désigner une séquence d'actions quelconque compatible avec son ordre partiel.

Définition 11 (Linéarisation d'un réseau d'actions)**.** *Soit A un ensemble d'actions et ρ un réseau d'action sur A. Une séquence d'actions $a'_1 \circ ... \circ a'_{|\rho|}$ est une linéarisation de ρ si et seulement s'il existe une fonction bijective $\Lambda \in N_\rho \to [\![1, |\rho|]\!]$, appelée fonction de linéarisation, telle que :*

1. *$\forall n \in N_\rho, a'_{\Lambda(n)} = a_n$.*
2. *$\forall n_1, n_2 \in N_\rho, n_1 \prec_\rho n_2 \implies \Lambda(n_1) < \Lambda(n_2)$.*

Lorsqu'un plan contient des actions pouvant être exécutées en parallèle, la nature du réseau d'action permet de n'ajouter que les contraintes nécessaires pour assurer l'exécutabilité du plan. Cette stratégie dite de « moindre engagement » est une des caractéristiques de la planification POP. Elle présente ainsi un avantage par rapport à la recherche de solutions : par exemple, dans le cas de la figure 2.4, si toutes les actions sont indépendantes, c'est alors un seul plan partiel qu'il faut explorer au lieu de 4. Mais elle constitue aussi un avantage au niveau de l'exécution, puisque l'on connait alors quelles sont les actions que l'on peut exécuter en parallèle, ce qui permet d'optimiser le temps d'exécution du plan. Ce dernier point est particulièrement pertinent dans le contexte de la planification multi-agents, où plusieurs agents peuvent agir en même temps.

Il est possible d'aller encore plus loin dans la stratégie de moindre engagement, par exemple en considérant les actions comme étant partiellement instanciées. À nouveau, un seul plan partiellement instancié (et partiellement ordonné) peut représenter à lui seul

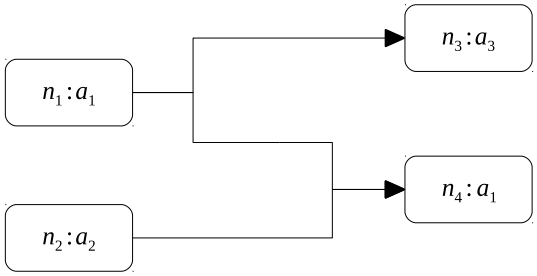


FIGURE 2.4 – Représentation d'un réseau d'actions
Ce réseau d'actions contient 4 nœuds n_1, n_2, n_3 et n_4 tels que $n_1 \prec n_3$, $n_1 \prec n_4$ et $n_2 \prec n_4$, et tel que l'action a_1 est présente deux fois (associée au nœud n_1 et au nœud n_4).

tout un ensemble de plans, correspondant aux différentes instanciations possibles. Pour maintenir les valeurs possible de chaque variable non instanciée, on ajoute au plan une structure de donnée supplémentaire qui répertorie les contraintes portant sur les domaines de ces variables. Un plan peut rester partiellement instancié si sa capacité à être exécuté ne dépend pas d'une valeur spécifique pour au moins une de ses variables. Ce procédé permet de réduire encore davantage la taille de l'espace de recherche.

Pour compléter la définition d'un plan partiel, on y adjoint enfin un ensemble de protections. Ces protections indiquent qu'une formule donnée doit être satisfaite sur un intervalle défini entre deux nœuds donnés du plan. Leur rôle en planification POP est de protéger les liens de causalité entre les sous-buts devant être satisfaits à un instant donné du plan et les actions qui accomplissent ces sous-buts. Ces protections permettent ainsi de tenir indirectement compte des sous-buts valides sur certains intervalles, dans un contexte où la structure non linéaire des plans ne permet pas en général d'identifier précisément l'état du monde à un instant donné, et donc d'évaluer directement la validité des sous-buts.

Compte tenu de tous ces éléments, la définition complète d'un plan partiel est la suivante :

Définition 12 (Plan partiel)**.** *Soit A un ensemble d'actions partiellement instanciées. Un plan partiel est un quadruplet $(N, \prec, C, P)$ tel que :*

1. *$(N, \prec)$ forme un réseau d'actions sur A.*
2. *C est un ensemble de contraintes portant sur les variables partiellement instanciées des actions. Ces contraintes sont de la forme $x = t$ et $x \neq t$, où x est une variable et t est un terme de $\mathcal{L}$.*
3. *P est un ensemble de protections. Une protection est un triplet $p = (\varphi, n_1, n_2)$, où φ est une formule logique de $\mathcal{L}$ et n_1 et n_2 sont des nœuds de N tels que $n_1 \prec n_2$. Si s est l'état initial, une protection (φ, n_1, n_2) impose que $\mathcal{T}(s, a'_1 \circ ... \circ a'_i) \models \varphi$ pour toute linéarisation $a'_1 \circ ... \circ a'_{|N|}$ de $(N, \prec)$ (via une fonction de linéarisation Λ) et tout $i \in [\![\Lambda(n_1) + 1, \Lambda(n_2)]\!]$.*

Recherche dans l'espace des plans partiellement ordonnés

Pour résoudre un problème de planification en parcourant un espace de réseau d'actions, il est d'abord nécessaire de convertir ce problème en un réseau d'actions initial. Un réseau d'actions ne permettant pas de représenter un état de façon explicite, l'état initial est converti en une action fictive. Les effets de cette action fictive correspondent alors aux littéraux satisfaits par l'état initial. Cette action est associée à un nœud initial, qui doit précéder tous les autres nœuds du réseau. De la même façon, le but à atteindre est

représenté par les préconditions d'une seconde action fictive, et le nœud associé à cette action doit être ordonné après tous les autres nœuds du réseau.

Pour être identifié comme une solution, un réseau d'actions doit d'abord être consistant, c'est-à-dire qu'il ne doit pas y avoir d'incohérence dans les définitions de l'ordre partiel et de l'ensemble de contraintes. Par exemple, il ne peut pas y avoir deux nœuds n_1 et n_2 telles que $n_1 \prec n_2$ et $n_2 \prec n_1$, ou deux variables x et y telles que $x = y$ et $x \neq y$. De plus, ce réseau d'actions doit être exempt de « défauts » (*flaws* en anglais). Ces défauts peuvent être de deux natures :

1. Un « sous-but » dont aucun lien causal n'indique par quelle action il est accompli.
2. Une action dont l'un des effets menace un lien causal. Il s'agit là d'une action qui n'est pas encore contrainte à être ordonnée avant l'action qui accomplit le sous-but ou après le sous-but lui-même, et dont l'un des effets peut être unifié avec ce sous-but.

Dans le premier cas, le défaut peut être résolu en ajoutant un lien causal entre le sous-but et une action du plan. Cette action peut déjà faire partie du plan, où y être introduite pour l'occasion. Dans le second cas, on ajoute une nouvelle contrainte au plan afin de protéger le lien causal menacé. Si on considère un lien causal (φ, n_1, n_2) devant être protégé d'une action à un nœud n, alors il est possible d'effectuer les trois types d'opérations suivantes :

1. *Demotion* : on ajoute au réseau d'action la contrainte $n \prec n_1$.
2. *Promotion* : on ajoute au réseau d'action la contrainte $n_2 \prec n$.
3. *Separation* : ce cas intervient dans le cas où il existe une substitution permettant d'unifier φ et l'un des effets de a_n. On ajoute alors une contrainte sur une variable permettant d'empêcher l'unification de ces deux littéraux.

C'est donc en résolvant les défauts d'un réseau d'actions que l'on génère ses successeurs. Un algorithme de planification parcours alors l'espace des réseaux d'actions jusqu'à trouver une solution. Pour explorer cet espace de recherche, les mêmes algorithmes que ceux décrits pour la planification dans les espaces d'états peuvent alors être appliqués (BFS, DFS, A*, BB, etc.), mais en adaptant leurs espaces de recherche pour que leurs éléments correspondent à des réseaux d'actions.

2.1.5 Planification classique pour les jeux vidéo

Les sous-parties précédentes ont proposé une présentation générale de la planification classique dans un contexte académique, afin d'exposer les différentes techniques de planification décrites dans la littérature scientifique. Cette nouvelle sous-partie se place dans un cadre industriel, et traite l'application de la planification classique pour le contrôle d'agents virtuels dans les jeux vidéo.

Cette sous-partie décrit tout d'abord les caractéristiques principales du système de planification GOAP et s'intéresse ensuite à l'utilisation de ce système pour la génération dynamique de comportements coordonnés dans le cade multi-agents.

GOAP

GOAP est un acronyme pour *Goal-Oriented Action Planning*, ce qui signifie « Planification d'actions orientée objectif » en français. Ce système a été conçu par Jeff Orkin dans le cadre de la conception de l'intelligence artificiel du jeu *F.E.A.R First Encounter Assault Recon* [79, 80, 81]. *F.E.A.R* est un jeu de tir en vue subjective, où des équipes de soldats ennemis doivent agir en coordination pour abattre le joueur alors que celui se déplace dans les « niveaux » du jeu. À notre connaissance, GOAP est la première intelligence

artificielle reposant sur les principes de la planification classique ayant été commercialisée dans un jeu vidéo.

La conception de GOAP a été motivée par la nécessité d'obtenir des comportements plus dynamiques pour les personnages non joueur (PNJ), tout en simplifiant la modélisation de ces comportements. En effet les machines à états utilisées à cet escient devenaient alors de plus en plus complexes et difficiles à maintenir. Le rôle de GOAP est donc d'automatiser la découverte de plans optimaux pour contrôler les PNJ, et de ne laisser au modélisateur que le soin de définir les actions, coûts, variables et buts devant être accomplis par ces PNJ.

Pour cela, GOAP utilise un modèle de donné similaire à STRIPS, mais fondé sur le principe des variables d'état, ici encodées sous la forme de tableaux statiques C++ contenant des paires clé/valeur. Mais comme pour STRIPS, les préconditions et effets des actions ne peuvent être formulés que sous forme de conjonction de littéraux. Pour des raisons de performance à l'exécution du système, ces actions sont directement implémentées en C++, en non pas interprétées depuis un langage de planification comme PDDL [63], le langage de planification standard utilisé par la communauté de chercheurs en planification automatique.

Afin de découvrir des plans optimaux, GOAP utilise une implémentation de l'algorithme A* dans l'espace des états. L'heuristique utilisée pour guider la recherche consiste à effectuer la somme des formules non satisfaites par rapport à l'état courant et au but assigné. L'optimalité des solutions apparait comme étant un critère important dans le contexte d'un jeu comme *F.E.A.R.* En effet, le sentiment d'immersion du joueur peut être dégradé si celui-ci est confronté à des entités agissant de façon grossièrement sous-optimale. Mais il s'agit aussi d'un procédé permettant de faciliter la modélisation des comportements : ainsi un modélisateur peut définir plusieurs comportements réalisables, et laisser à la machine le soin de déterminer lequel de ces comportements est le plus adapté par rapport à une situation donnée.

Comportements coordonnés

Avec GOAP, chaque agent dispose de son planificateur et de sa propre représentation de l'environnement, qui est une représentation partielle, centrée sur la perception propre qu'a cet agent de son environnement. Il s'agit donc d'une approche mono-agent de la planification. Dans *F.E.A.R*, les comportement d'équipe nécessitant de la coordination sont simples. Ceux-ci correspondent à un ensemble d'ordres transcris en buts pour chaque membre de l'équipe. Chaque agent fait alors le tri entre les buts qui lui sont attribués par son supérieur, et ceux, parfois plus urgents, provenant d'événements survenant dans leur environnement (comme la chute d'une grenade qui requiert la planification d'actions permettant de s'en protéger).

Pour prendre connaissance d'un système produisant des comportements plus complexes, on peut s'intéresser au projet *Carnival* [85]. Ce projet a consisté à développer une architecture multi-agents par dessus GOAP afin de coordonner une section d'infanterie en suivant le principe de la chaine de commandement. Dans ce projet, les plans d'actions de chaque agent sont toujours déterminés par une instance dédiée de GOAP. Mais, des agents virtuels, correspondant aux différents « preneurs de décisions » situés à chaque niveau de la hiérarchie disposent eux aussi de leur propre planificateur. Contrairement à *F.E.A.R*, les comportements coordonnés ne correspondent plus à des stratégies figées, mais sont composées dynamiquement par planification automatique. À chaque niveau de décision, les actions sont alors transcrites en buts à atteindre pour les unités subordonnées. Afin de fonctionner, cette architecture fait l'hypothèse que la prise de décision au niveau inférieur est totalement indépendante du niveau supérieur. C'est-à-dire qu'une fois son objectif

reçu, l'unité subordonnée a toute la liberté possible pour calculer son plan, sans que le niveau supérieur n'intervienne dans sa prise de décision.

Telle qu'il vient d'être présenté, on peut établir une analogie entre ce système de décomposition d'actions de haut niveau en tâches de planification aux niveaux inférieurs, et un modèle de planification hiérarchique appelé planification HTN, pour lequel ce principe de décomposition de tâches est défini formellement. La description de ce modèle de planification est l'objet de la suite de ce chapitre.

2.2 Planification HTN

Cette partie est consacrée à un domaine particulier de la planification automatique appelé planification HTN, où HTN est un acronyme anglais signifiant *Hierarchical Task Network*, c'est-à-dire « réseau de tâches hiérarchiques» en français. La planification HTN est une forme de planification hiérarchique dont l'objectif est de trouver un plan d'actions permettant d'exécuter un ensemble de tâches de haut niveau. La décomposition des tâches de haut niveau en sous-tâches est modélisée dans le domaine de planification par un ensemble de règles de décomposition. Ces règles, appelées méthodes de décomposition, définissent comment remplacer une tâche de haut niveau par un réseau de sous-tâches accompagnées de contraintes sur l'exécution du plan final.

C'est pour bénéficier de la modélisation de cette structure hiérarchique que l'on s'intéresse ici à la planification HTN. Celle-ci permet en effet de contrôler la recherche de plans solution en suivant le principe de la chaine de commandement utilisée par la section d'infanterie. Ainsi les tâches permettent de modéliser les ordres et modes d'action tels qu'ils sont élaborés aux différents niveaux de décision (chef de section, chefs de groupes, chefs d'équipes et soldats) et le principe de décomposition permet de modéliser la transmission des ordres entre ces échelons.

Cette partie présente dans un premier temps un formalisme général pour la planification HTN, puis l'étudie vis-à-vis de trois contextes plus spécifiques : la planification dans l'espace des plans partiels, la planification dans l'espace des états, et la planification hiérarchique « angélique ». Enfin, les principales propriétés des espaces de recherche en planification HTN sont étudiées, avant d'achever cette partie par la revue de différents exemples d'application de la planification HTN à la simulation militaire.

2.2.1 Formalisme général

La planification HTN a vu le jour en tant que technique de planification partagée par un ensemble de systèmes de planification peu formalisés tel que NOAH [91, 90], NONLIN [98, 99], et SIPE [111, 112]. Ces systèmes font tous partie de la catégorie des planificateurs POP, auxquels ils ajoutent la décomposition de tâches hiérarchiques. Par la suite, un ensemble de travaux ayant abouti au planificateur UMCP [22, 23, 20] ont permis de faire progresser la formalisation de la planification HTN. Cependant, les systèmes conçus ultérieurement n'ont généralement pas suivi ce formalisme à la lettre, en particulier en ce qui concerne la nature des contraintes associées aux décompositions. Ainsi certains planificateurs comme SHOP [73] ont adopté une version simplifiée de ce formalisme, tandis que d'autres approches comme SIADEX [13] ou FAPE [18] l'ont adapté à la planification temporelle.

En conséquence, on peut dire qu'il existe presque autant de formalismes différents pour la planification HTN qu'il n'existe de planificateurs HTN. Dans ce contexte, il a été décidé de présenter ici un formalisme suffisamment général pour traiter ce que tous ces planificateurs ont en commun : la décomposition de tâches en sous-tâches. Ce formalisme

fait donc abstraction des contraintes se rapportant à l'exécution du plan, et qui sont généralement modélisées au côté des contraintes d'ordre dans les réseaux de tâches.

Le principe de la planification HTN consiste à définir des plans formés de tâches correspondant à différents niveaux de hiérarchies. Parmi ces tâches, on distingue les tâches primitives des tâches composées (qu'on appelle aussi parfois tâches non primitives). Les tâches primitives se situent au plus bas niveau de la hiérarchie des tâches, et correspondent aux actions de la planification classique. Elles définissent donc un système de transitions par rapport aux états. Les tâches composées, de leur côté, ne correspondent pas à des transitions d'états mais représentent l'agrégation d'un ensemble de sous-tâches partiellement ordonnées, que l'on nommera réseau de tâches :

Définition 13 (Réseau de tâches)**.** *Soit T un ensemble de tâches. Un réseau de tâches sur T est un couple $\rho = (N_\rho, \prec_\rho)$ où :*

1. *N_ρ est un ensemble fini de nœuds, tel que chaque nœud $n \in N_\rho$ est associé à une tâche $t_n \in T$.*
2. *$\prec_\rho$ est une relation d'ordre partiel définie sur N_ρ. Pour toute paire $n_1, n_2 \in \prec_\rho$, on notera $n_1 \prec_\rho n_2$.*

Pour un ensemble de tâches T, on notera R_T l'ensemble des réseaux de tâches sur T. De plus, on parlera simplement de réseau de tâches, sans préciser l'ensemble de tâches associé, quand l'identité de celui-ci sera évident d'après le contexte. On utilisera aussi le symbole « $\emptyset$ » pour représenter un réseau de tâches vide (qui ne contient aucun nœud et aucun élément dans la relation d'ordre associée). Enfin, la nature des éléments utilisés pour identifier les nœuds des réseaux de tâches n'aura aucune importance dans ce manuscrit. En effet seule l'identité des tâches associées à ces nœuds ainsi que la structure de l'ordre partiel auxquels il servent de support auront une pertinence. En conséquence, on considère que deux réseaux de tâches sont identiques dans la mesure où ils sont isomorphes, selon la définition ci-dessous :

Définition 14 (Réseaux de tâches isomorphes)**.** *Soient ρ_1 et ρ_2 deux réseaux de tâches. On dit que ρ_1 et ρ_2 sont isomorphes s'il existe une application bijective $f \in N_{\rho_1} \to N_{\rho_2}$ telle que :*

1. *$\forall n_1, n_2 \in N_{\rho_1}, n_1 \prec_{\rho_1} n_2 \iff f(n_1) \prec_{\rho_2} f(n_2)$.*
2. *$\forall n \in N_{\rho_1}, t_n = t_{f(n)}$.*

Bien que l'on continuera désormais à parler de réseau de tâches, chaque élément dénommé « réseau de tâches » correspondra formellement à la classe des réseaux de tâches qui lui sont isomorphes, et pour tout ensemble de tâches T, R_T correspondra donc à l'ensemble de ces classes.

Lorsqu'un réseau de tâches est uniquement composé de tâches primitives, on parle alors de réseau de tâches « primitif ». Un réseau de tâches primitif correspond en fait au réseau d'actions introduit par la définition 11. En conséquence, il est aussi possible d'extraire des plans (des séquences d'actions) de ces réseaux de tâches primitifs en les linéarisant (voir définition 11). Un réseau de tâches primitif est par ailleurs considéré comme étant exécutable dans un état donné si on peut en extraire une séquence d'actions exécutable dans cet état :

Définition 15 (Exécutabilité d'un réseau de tâches primitif)**.** *Soit un réseau de tâches primitif $\rho \in R_A$ et un état $s \in S$. ρ est exécutable en s si et seulement s'il existe une linéarisation $\sigma \in A^*$ de ρ exécutable en s.*

Si un réseau de tâches contient au moins une tâche composée, alors celui-ci est dit « non primitif ». L'association d'une tâche composée à un ensemble de sous-tâches est spécifiée dans le domaine de planification HTN par un objet appelé « méthode de décomposition ». Comme le représente la figure 2.5, il est possible de définir plusieurs méthodes de décomposition pour une même tâche composée, ce qui implique qu'il peut exister plusieurs alternatives pour décomposer cette tâche.

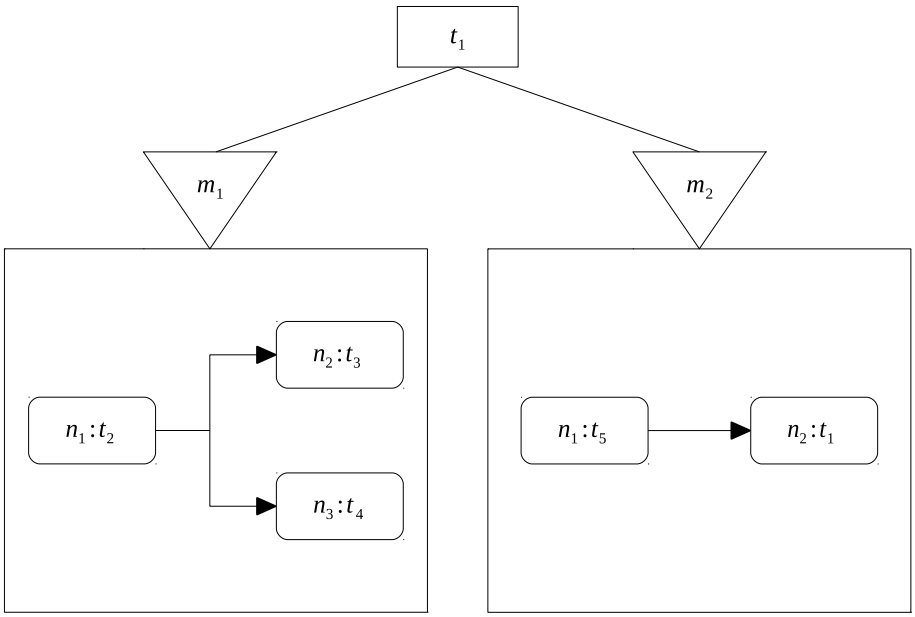


FIGURE 2.5 – Méthodes de décomposition
t_1 est une tâche composée associée à deux méthodes de décomposition m_1 et m_2.

Un domaine de planification HTN peut alors simplement être défini comme l'extension d'un domaine de planification classique, auquel on ajoute un ensemble de tâches composées ainsi qu'un ensemble de méthodes de décomposition :

Définition 16 (Domaine de planification HTN)**.** *Un domaine de planification HTN est un* 6*-uplet* $D = (S, A, C, M, \mathcal{T}, \mathcal{C})$ *où :*

1. S *est un ensemble fini d'états.*
2. A *est un ensemble fini d'actions.*
3. C *est un ensemble fini de tâches composées.*
4. $M \subseteq C \times R_{A \cup C}$ *est un ensemble fini de méthodes de décomposition associées aux éléments de* C*. Chaque méthode de décomposition est un couple* $m = (c_m, \rho_m)$ *où :*
   (a) $c_m \in C$ *est la tâche composée associée à* m*.*
   (b) ρ_m *est un réseau de tâches représentant les sous-tâches de* m*.*
5. $\mathcal{T} \in S \times A \to S$ *est une fonction de transition.*
6. $\mathcal{C} \in S \times A \to \mathbb{R}^+$.

Étant donné un réseau de tâches non primitif, celui-ci peut être transformé en un nouveau réseau de tâches en y décomposant une tâche composée, c'est-à-dire en la remplaçant par un ensemble de sous-tâches désigné par une méthode de décomposition. On appelle cette transformation une « réduction » d'un réseau de tâches en un nouveau réseau de tâche. Le réseau de tâches obtenu correspond alors à celui décrit dans la définition suivante :

Définition 17 (Réduction d'un réseau de tâches)**.** *Soient un réseau de tâches non primitif ρ, un nœud $n \in N_\rho$ tel que $t_n \in C$ et une méthode de décomposition m telle que $c_m = t_n$. Le réseau de tâches obtenu en décomposant la tâche t_n dans ρ selon m, est appelé réduction de ρ, est noté $\mathcal{R}(\rho, n, m)$, et est défini par :*

$$\begin{aligned} N_{\mathcal{R}(\rho,n,m)} &= (N_\rho \setminus \{n\}) \cup N_{\rho_m} \\ \prec_{\mathcal{R}(\rho,n,m)} &= \{(n_1, n_2) \in \prec_\rho \mid n_1 \neq n \wedge n_2 \neq n\} \\ &\quad \cup N_{\rho_m} \times \{n' \in N_\rho \mid n \prec_\rho n'\} \\ &\quad \cup \{n' \in N_\rho \mid n' \prec_\rho n\} \times N_{\rho_m} \end{aligned}$$

Remarque : on considère dans cette définition que $N_\rho \cap N_{\rho_m}$. Si cette condition n'est pas vérifiée, il suffit alors de considéré un réseau de tâches isomorphe à ρ_m dont tous les nœuds sont distincts de ceux de ρ.

La réduction n'est pas la seule transformation qui peut être appliquée à un réseau de tâches. Une transformation plus simple consiste seulement à ajouter une contrainte sur l'ordre des tâches. Cette transformation contient toutefois une subtilité : la relation d'ordre associée à un réseau de tâches étant une relation transitive, chaque ajout de contrainte doit être effectué en y ajoutant aussi l'ensemble des contraintes permettant d'en maintenir la transitivité.

Définition 18 (Ajout d'une contrainte d'ordre dans un réseau de tâches)**.** *Soient un réseau de tâches ρ et deux nœuds $n_1, n_2 \in N_\rho$ distincts tels que $\neg(n_1 \prec_\rho n_2)$ et $\neg(n_2 \prec_\rho n_1)$. Le réseau de tâches obtenu en ajoutant une contrainte d'ordre entre n_1 et n_2 dans ρ est noté $\mathcal{O}(\rho, n_1, n_2)$, et est défini par :*

$$\begin{aligned} N_{\mathcal{O}(\rho,n_1,n_2)} &= N_\rho \\ \prec_{\mathcal{O}(\rho,n_1,n_2)} &= \prec_\rho \cup \{n \in N_\rho \mid n = n_1 \vee n \prec_\rho n_1\} \times \{n \in N_\rho \mid n = n_2 \vee n_2 \prec_\rho n\} \end{aligned}$$

Toutes deux réunies, la réduction et l'ajout de contraintes d'ordre permettent de définir une relation entre les réseaux de tâches. Cette relation, appelée « relation de décomposition » détermine alors le lien qui relie deux réseaux de tâches entre eux :

Définition 19 (Relation de décomposition)**.** *Soient un ensemble de tâches T et un ensemble de méthodes M. On appelle relation de décomposition associée à T et M, notée $\mathcal{D}_{T,M}$, la relation sur R_T définie telle que : $\forall \rho_1, \rho_2 \in R_T, \mathcal{D}_{T,M}(\rho_1, \rho_2) \iff$*

1. *$\exists n_1, n_2 \in N_{\rho_1}$ tels que $\neg(n_1 \prec_{\rho_1} n_2)$ et $\neg(n_2 \prec_{\rho_1} n_1)$ et $\rho_2 = \mathcal{O}(\rho_1, n_1, n_2)$, ou*
2. *$\exists n \in N_{\rho_1}$ et $\exists m \in M$ tels que $c_m = t_n$ et $\rho_2 = \mathcal{R}(\rho_1, n, m)$.*

Ainsi les ensembles de tâches composées et de méthodes de décomposition caractérisent un graphe de réseaux de tâches $(R_{A \cup C}, \mathcal{D}_{A \cup C, M})$ appelé « graphe de décompositions ». Les ensembles de tâches et de méthodes de décomposition considérés étant généralement définis sans ambiguïté par rapport au contexte, on note alors simplement la relation de décomposition avec le symbole $\mathcal{D}$. On utilise aussi la notation $\mathcal{D}^*$ pour désigner la clôture transitive de cette relation, qui détermine l'ensemble des réseaux de tâches accessibles depuis un réseau de tâches donné, et qui est définie de la façon suivante :

$$\forall \rho_1, \rho_2 \in R_{A \cup C} : \mathcal{D}^*(\rho_1, \rho_2) \iff \rho_1 = \rho_2 \vee \exists \rho_3 \in R_{A \cup C} : \mathcal{D}^*(\rho_1, \rho_3) \wedge \mathcal{D}(\rho_3, \rho_2)$$

Le graphe de décompositions est directement lié à la définition d'une solution en planification HTN. En effet le problème posé consiste alors à trouver un réseau de tâches primitif exécutable dans un état initial, et qui soit accessible depuis un réseau de tâches initial :

Définition 20 (Problème de planification HTN)**.** *Un problème de planification HTN est un triplet $P = (D, s_0, \rho_0)$ où :*

1. *D est un domaine de planification HTN.*
2. *$s_0 \in S$ est l'état initial.*
3. *$\rho_0 \in R_{A \cup C}$ est le réseau de tâches initial.*

Ce réseau de tâches initial occupe un rôle similaire à celui du but en planification classique. Toutefois, le rôle d'un plan solution n'est donc plus de satisfaire un objectif final, mais de correspondre à une exécution au niveau primitif d'un ensemble de tâches composées :

Définition 21 (Solution à un problème de planification HTN)**.** *Soit $P = (D, s_0, \rho_0)$ un problème de planification HTN. Une séquence d'actions $\sigma \in A^*$ est un plan solution au problème P si et seulement si :*

1. *σ est exécutable en s_0.*
2. *σ est une linéarisation d'un réseau de tâches primitif ρ tel que $\mathcal{D}^*(\rho_0, \rho)$.*

Enfin, le critère d'optimalité des solutions est le même que pour la planification classique, mis à part que cette fois un plan est optimal par rapport à l'ensemble des solutions défini pour le cadre de la planification HTN :

Définition 22 (Solution optimale à un problème de planification HTN)**.** *Soient $P = (D, s_0, \rho_0)$ un problème de planification HTN, et $\sigma^* \in A^*$ un plan solution de P. σ^* est une solution optimale de P si et seulement si $\mathcal{C}(s_0, \sigma^*) = min\{\mathcal{C}(s_0, \sigma) \mid \sigma$ est une solution de $P\}$.*

2.2.2 Planification HTN dans l'espace des décompositions

Une première approche pour résoudre un problème de planification HTN consiste simplement à parcourir le graphe de décompositions depuis le réseau de tâche initial jusqu'à l'obtention d'un réseau de tâches primitif exécutable. C'est ce qu'effectue la procédure DHTN (*Decomposition* HTN) décrite par l'algorithme 4. Tout comme le formalisme décrit ci-dessus, il s'agit d'une procédure générale, qui ne prend pas en compte les spécificités des planificateurs qui l'implémentent. Pour cette raison, DHTN parcours l'espace des décompositions uniquement par la réduction des réseaux de tâches, qui est la transformation nécessaire et suffisante pour accéder à un réseau de tâches primitif. Les planificateurs qui implémentent DHTN considèrent généralement un ensemble de transformations supplémentaires, désignées sous le nom de « critiques» par Erol [22].

Parmi ces planificateurs, on peut citer NOAH, NONLIN et SIPE, ainsi que d'autres planificateurs combinant planification HTN et planification POP tels que O-Plan [15], O-Plan2 [100], PRIAR [50] et DPOCL [119]. UMCP fait aussi partie de cette catégorie, toutefois le formalisme associé n'est pas directement fondé sur la planification POP. Dans le cas des planificateurs non linéaires, la recherche de solutions s'effectue dans un espace de plans partiels similaires à ceux introduits par la définition 12, mais où le réseau d'actions y est généralisé sous la forme d'un réseau de tâches. Les transformations appliquées aux réseaux de tâches incluent donc la réduction, ainsi que les trois opérations propres à la planification POP (*demotion*, *promotion* et *separation*), qui consistent à ajouter des contraintes sur la relation d'ordre ou sur les variables non instantiées apparaissant dans les noms des tâches. L'espace de recherche parcouru correspond alors à la réunion de l'espace parcouru par DHTN et de l'espace parcouru par les planificateurs POP « classiques ».

Le formalisme de ces planificateurs combinant planification POP et planification HTN définit par ailleurs les tâches composées de la même façon que les actions, c'est-à-dire avec

Algorithm 4 Algorithme DHTN

entrées : un problème de planification HTN P
sorties : un plan solution de P

$Ouvert \leftarrow \{\rho_0\}$
$Fermé \leftarrow \emptyset$
tant que $Ouvert \neq \emptyset$ **faire**
  $\rho \leftarrow$ extraire un élément de $Ouvert$
  $Fermé \leftarrow Fermé \cup \{\rho\}$
  si ρ est primitif **alors**
    si ρ est exécutable en s_0 **alors**
      retourner une linéarisation de ρ
    fin si
  sinon
    choisir $n \in N_\rho$ tel que $t_n \in C$
    pour tout méthode $m \in M$ telle que $c_m = t_n$ **faire**
      $\rho' \leftarrow \mathcal{R}(\rho, n, m)$
      si $\rho' \notin Fermé$ **alors**
        $Ouvert \leftarrow Ouvert \cup \{\rho'\}$
      fin si
    fin pour
  fin si
fin tant que
retourner *échec*

des préconditions et des effets. C'est d'ailleurs le terme « d'action de haut niveau » qui est généralement employé pour désigner les tâches composés dans ce contexte [116]. Ces effets ne permettent pas de déduire une sémantique de transition d'états déterministe pour les actions de haut niveau, contrairement aux actions primitives. Pour autant, il peuvent être utilisés pour identifier des liens de causalité entre des actions de haut niveau et des sous-buts du plan. Cependant, la modélisation de préconditions et d'effets au niveau des tâches composées implique de prendre quelques précautions. En effet, il devient alors nécessaire d'imposer des restrictions sur les méthodes de décomposition afin d'assurer une cohérence entre les actions de haut niveau d'une part, et les préconditions et effets introduits par leurs sous-tâches d'autre part [115, 119, 9].

La planification HTN a longtemps était restreinte au cadre de la planification POP, bénéficiant alors de l'apriori positif, de la part de la communauté scientifique, vis-à-vis de la planification linéaire par rapport aux techniques de planification totalement ordonnée. Cependant, elle a été affectée par le retournement qui s'est opéré au tournant des années 2000 en faveur de cette dernière, et a été adapté avec succès à la recherche de plans totalement ordonnés dans les espaces d'états [10, 64].

2.2.3 Planification HTN dans l'espace de progression d'état

On vient de voir qu'un problème de planification HTN pouvait être résolu en parcourant directement le graphe de décomposition des réseaux de tâches. Mais alors que les planificateurs HTN étudiés précédemment suivent le principe de la planification dans l'espace des plans partiels, il existe une autre catégorie de planificateurs HTN qui, de leur côté, suivent une approche s'appuyant sur le parcours d'un espace d'états. Le principe suivi par ces planificateurs consiste à imposer un ordre total d'exécution sur les tâches primitives et de décomposer les tâches composées en respectant l'ordre de leur exécution (ainsi une tâche ordonnée après une autre ne peut être décomposée avant celle-ci). Ce procédé permet

de déterminer exactement l'état courant pour chaque tâche à décomposer puisque celle-ci n'est alors précédée que par une séquence de tâches primitives, comme illustré en figure 2.6.

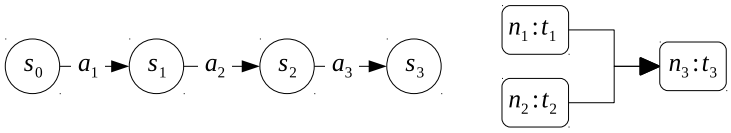


FIGURE 2.6 – Planification HTN par progression des états
Exemple d'un plan partiel dans un contexte de planification HTN dans un espace de progressions des états. À cette étape, seules t_1 et t_2 peuvent être décomposées, car elles ne sont précédées par aucune autre tâche composée, au contraire de t_3. Toutes les tâches primitives qui précèdent t_1 et t_2 (a_1, a_2 et a_3), se sont aussi vu imposer un ordre total d'exécution, ce qui permet de déterminer s_3, l'état précédant t_1 et t_2.

L'espace de recherche exploré ne correspond ainsi plus à l'espace de décompositions considéré précédemment. Ce nouvel espace de recherche combine désormais une forme restreinte de réduction de réseau de tâches (limité aux tâches qui ne sont précédées par aucune tâche composée), avec la progression de l'état courant par les tâches primitives, à la manière de la planification classique dans l'espace des états. Ce type de transformation s'applique donc à la fois au réseau de tâches et à l'état courant, ce qui implique que chaque élément de l'espace de recherche est constitué d'une paire de ces éléments. On appelle cet espace de recherche « espace de progressions », et le graphe associé est défini par la relation suivante :

Définition 23 (Relation de progression)**.** *Soient un ensemble de tâches T et un ensemble de méthodes M. On appelle relation de progression associée à T et M, notée $\mathcal{P}_{T,M}$, la relation sur $S \times R_T$ définie telle que :* $\forall (s_1, \rho_1), (s_2, \rho_2) \in S \times R_T, \mathcal{P}_{T,M}((s_1, \rho_1), (s_2, \rho_2)) \iff \exists n \in N_{\rho_1}$ *tel que* $\forall n' \in N_{\rho_1}, \neg(n' \prec_{\rho_1} n)$ *et :*

1. $\exists m \in M$ *telle que* $c_m = t_n$ *et* $\rho_2 = \mathcal{R}(\rho_1, n, m)$*, ou*
2. $t_n \in A$*,* $s_2 = \mathcal{T}(s_1, t_n)$ *et* $\rho_2 = (N_{\rho_1} \setminus \{n\}, \{(n_1, n_2) \in \prec_{\rho_1} |\ n_1 \neq n \wedge n_2 \neq n\})$

À nouveau, au notera simplement cette relation par le symbole $\mathcal{P}$, en omettant les ensembles de tâches et de méthodes associées, lorsque ceux-ci seront clairement définis par le contexte. On utilisera aussi à nouveau la notation $\mathcal{P}^*$.

Parmi les planificateurs parcourant un espace de progression d'état, on peut citer SHOP [73, 74] et son successeur SHOP2 [77, 75], ainsi que des systèmes plus récents comme HTNPLAN-P [95, 94] et HATP [57]. Avec cette catégorie de planificateurs HTN, les contraintes modélisées dans les méthodes de décompositions prennent la forme de formules de préconditions. L'état courant étant toujours connu au niveau des tâches à décomposer, la validité de ces préconditions peut être évaluée efficacement. Les méthodes dont les préconditions ne sont pas satisfaites sont alors considérées inapplicables, et ne sont pas employées pour décomposer la tâche sélectionnée. Ce procédé permet aux modélisateurs d'employer les méthodes de décompositions pour programmer des procédures de planification spécialisées, et ainsi améliorer l'efficacité de la recherche de solutions. À titre d'exemple, la figure 2.7 représente l'utilisation des préconditions associées aux méthodes pour programmer récursivement la neutralisation, par les membres d'un trinôme de fantassins, d'un ensemble de cibles situées dans une pièce d'un bâtiment. La méthode 1, qui ne contient pas de sous-tâches, fait ici figure de cas de base, et est utilisée pour terminer la récursion lorsqu'il ne reste plus de cibles à neutraliser.

La procédure PHTN (*Progression* HTN), décrite par l'algorithme 5, généralise les différentes procédures utilisées par ces planificateurs. Cette procédure applique les transformations qui caractérisent la relation de progression afin de parcourir l'espace correspondant, jusqu'à

$FireteamAchieveAllTargetsNeutralized(f, r)$

$Method1$

`Precond:`

$\forall\ t,\ p,\ TargetAt(t,p)\ \wedge\ InRoom(p,r)\ \Longrightarrow\ Neutralized(t)$

$Method2(s, t, p1,\ p2)$

`Precond:`

$MemberOfFireteam(s, f)$

$InRoom(p_1,\ r)\ \wedge\ HaveLineOfFire(p_1, p_2)$

$TargetAt(t, p_2)\ \wedge\ InRoom(p_2, r)\ \wedge\ \neg\ Neutralized(t)$

`Subtasks:`

$n_1\ :\ SoldierAchieveAt(s, p)$

$n_2\ :\ SoldierNeutralizeTarget(s, t, r)$

$n_3\ :\ FireteamAchieveAllTargetsNeutralized(f, r)$

`Order:`

$n_1\ \prec\ n_2$

FIGURE 2.7 – Méthodes de décomposition avec préconditions
Exemple de tâche composée récursive dont les méthodes de décomposition sont modélisées avec des formules de préconditions. On peut remarquer que les méthodes y sont modélisées avec des paramètres. À la manière des opérateurs vis-à-vis des actions, ce procédé permet de modéliser en une seule fois tout un ensemble de méthodes que l'on obtiendrait en instanciant ces paramètres.

atteindre un élément correspondant à une solution, identifiable lorsque le réseau de tâches est vide. Un ensemble *Fermé* enregistre aussi les sous-problèmes visités afin d'éviter de les traiter plusieurs fois et garantit que PHTN termine lorsque l'espace de recherche est fini.

Les différents planificateurs mentionnés plus haut implémentent cette procédure de diverses façons. Ainsi, SHOP implémente les retours-arrière en parcourant l'espace de recherche en profondeur d'abord, et restreint l'usage des réseaux de tâches à des séquences de tâches totalement ordonnées. Son successeur SHOP2 (ainsi que HATP) généralise l'usage des réseaux de tâches partiellement ordonnés, et ajoute en plus une étape de séparation et évaluation en comparant le coût du plan courant avec celui du meilleur plan afin d'élaguer l'espace de recherche et de retourner une succession de plans de qualités croissantes. Enfin HTNPLAN-P combine aussi une technique de séparation et évaluation, mais cette fois avec un parcours de l'espace de recherche en meilleur d'abord. La fonction utilisée pour évaluer la qualité des plans, ainsi que les heuristiques utilisées pour guider la recherche et l'évaluation de la borne pour l'étape de séparation et évaluation, reposent ici sur un ensemble de préférences modélisées dans le domaine de planification (et qui ne sont pas liées aux coûts des actions).

La décomposition des tâches dans l'ordre de leur exécution présente encore d'autres avantages : ainsi, la possibilité d'évaluer les préconditions des actions et des méthodes directement sur l'état courant permet d'en étendre l'expressivité en y incluant des calculs numériques complexes, mais aussi d'y évaluer les résultats de procédures externes. Toutes ces propriétés ont fait qu'en pratique, les planificateurs HTN comme SHOP et SHOP2 se sont révélés très bien adaptés pour résoudre des problèmes de planification issus de cas d'applications concrets [78, 71, 76].

Algorithm 5 Algorithme PHTN

entrées : un problème de planification HTN P
sorties : un plan solution de P ou *échec*

$Ouvert \leftarrow \{(s_0, epsilon, \rho_0)\}$
$Fermé \leftarrow \emptyset$
tant que $Ouvert \neq \emptyset$ **faire**
  $s, \sigma, \rho \leftarrow$ extraire un élément de $Ouvert$
  $Fermé \leftarrow Fermé \cup \{(s, \rho)\}$
  si $N_\rho = \emptyset$ **alors**
    retourner σ
  sinon
    pour tout $n \in N_\rho$ tel que $\forall n' \in N_\rho, \neg(n' \prec n)$ **faire**
      si $t_n \in A$ **alors**
        si t_n est applicable à s **alors**
          $s' \leftarrow \mathcal{T}(s, t_n)$
          $\rho' \leftarrow (N_\rho \setminus \{n\}, \{(n_1, n_2) \in \prec_\rho |\ n_1 \neq n \wedge n_2 \neq n\})$
          si $(s', \rho') \notin Fermé$ **alors**
            $Ouvert \leftarrow Ouvert \cup \{(s', \sigma \circ t_n, \rho')\}$
          fin si
        fin si
      sinon
        pour toute méthode $m \in M$ applicable à s et telle que $c_m = t_n$ **faire**
          $\rho' \leftarrow \mathcal{R}(\rho, n, m)$
          si $(s, \rho') \notin Fermé$ **alors**
            $Ouvert \leftarrow Ouvert \cup \{(s, \sigma, \rho')\}$
          fin si
        fin pour
      fin si
    fin pour
  fin si
fin tant que
retourner *échec*

2.2.4 Planification hiérarchique angélique

Si on considère les deux approches de la planification HTN étudiées jusqu'ici, aucune d'elles n'étend la fonction de transition, définie sur l'ensemble des actions, aux tâches composées. Cela se traduit par l'impossibilité d'exécuter les tâches composées, et donc de raisonner sur l'exécutabilité d'un plan non primitif. Or, cette capacité apparait très séduisante pour améliorer l'efficacité d'un algorithme de planification HTN. Par exemple, elle pourrait permettre d'identifier qu'un plan de haut niveau n'est pas exécutable avant de l'avoir entièrement décomposé, et donc de réduire la taille de l'espace de recherche en évitant d'explorer ses successeurs. Mais même en ce qui concerne les « actions de haut niveau » utilisées parfois en planification HTN non linéaire, il n'est pas possible d'interpréter les effets des tâches composées par une fonction de transition déterministe. En effet, une tâche non primitive est potentiellement décomposable en tout un ensemble de séquences d'actions exécutables, et chaque séquence d'actions peut produire un état final différent. La planification hiérarchique angélique [60, 59, 114] (aussi appelée AHP, pour *Angelic Hierarchical Planning*) propose alors d'adopter une sémantique de transition non-déterministe pour les tâches composées. Le terme « angélique » fait ici référence au fait que ce non-déterministe et un non-déterministe contrôlable, dans la mesure où l'état

final est déterminé par le choix de la séquence d'actions utilisée pour décomposer cette tâche, et que ce choix appartient au planificateur.

La fonction de transition est donc désormais définie telle que $\mathcal{T} \in S \times A \cup C \to \mathcal{P}(S)$, où $\mathcal{P}(S)$ est l'ensemble des parties de S. Cette définition inclut les actions, cependant celles-ci demeurent déterministes, ce qui implique que pour tout état s et action $a \in A$, $\mathcal{T}(s, a)$ est un singleton. Afin de décrire de façon compacte cette fonction de transition, AHP propose deux procédés. Le premier consiste à utiliser un langage de description des effets appelé NCSTRIPS (Nondeterministic Conditional STRIPS), et qui permet de modéliser des effets « possibles » en plus des effets classiques. Le deuxième procédé consiste à ne pas décrire directement cette fonction de transition, mais d'en décrire des approximations plus compactes. Ces approximations sont au nombre de deux. La première est dite « optimiste », dans la mesure où elle surestime l'ensemble des états atteignables, c'est-à-dire que pour tout état s, et toute tâche t, on a $\mathcal{T}(s,t) \subseteq \mathcal{T}_{opt}(s,t)$, où $\mathcal{T}_{opt}$ et la fonction de transition optimiste. À l'opposé, la seconde approximation est dite « pessimiste », car elle sous-estime l'ensemble des états atteignables, c'est-à-dire que pour tout état s et toute tâche t, on a $\mathcal{T}_{pes}(s,t) \subseteq \mathcal{T}(s,t)$, où $\mathcal{T}_{pes}$ et la fonction de transition pessimiste. Ces fonctions de transitions optimiste et pessimiste sont décrites par deux jeux d'effets distincts, appelés respectivement effets optimistes et effets pessimistes.

AHP fait usage de ces fonctions de transition pour déterminer l'exécutabilité des plans de haut niveau. Pour cela, ces plans sont contraints à être des séquences de tâches, et non pas des réseaux de tâches (partiellement ordonnés par définition). Ainsi ces fonctions de transitions peuvent elles-aussi être généralisées aux séquences de tâches composées. Une séquence $\sigma \in (A \cup C)^*$ peut alors être identifiée comme étant non exécutable en un état initial s_0 si $\mathcal{T}_{opt}(s_0, \sigma) = \emptyset$. Si de plus cette séquence vérifie $\mathcal{T}_{pes}(s_0, \sigma) \neq \emptyset$, on peut alors en déduire que le plan est exécutable avec certitude, et donc que ce plan de haut niveau est nécessairement décomposable en une séquence de tâches primitives exécutable. Ce dernier point est particulièrement intéressant. En effet, il permet à la procédure de recherche de s'engager dans une branche de l'espace de recherche, en ayant la garantie que celle-ci mènera à une solution.

Ce principe d'approximation optimiste et pessimiste est aussi appliqué à la fonction de coût $\mathcal{C}$. Pour cela, on considère tout d'abord une version de $\mathcal{C}$ étendue aux tâches composées : pour toute tâche $t \in C$ et état s tels que $\mathcal{T}(s,t) \neq \emptyset$, le coût réel $\mathcal{C}(s,t)$ correspond au minimum des coûts de toutes les séquences d'actions exécutables en s et que l'on peut obtenir en décomposant successivement t. À nouveau, cette fonction pouvant difficilement être décrite de façon compacte, AHP propose d'en modéliser une approximation optimiste $\mathcal{C}_{opt}$ et une approximation pessimiste $\mathcal{C}_{pes}$. La fonction de coût optimiste correspond ici à une sous-estimation du coût réel d'une tâche. On a donc $\mathcal{C}_{opt}(s,t) \leq \mathcal{C}(s,t)$ pour toute tâche t et état s tels que $\mathcal{T}_{opt}(s,t) \neq \emptyset$. À l'opposé, la fonction de coût pessimiste constitue une surestimation du coût réel, c'est-à-dire qu'on a $\mathcal{C}(s,t) \leq \mathcal{C}_{pes}(s,t)$ pour toute tâche t et état s tels que $\mathcal{T}_{pes}(s,t) \neq \emptyset$. Tout comme les fonctions de transitions, les fonctions de coût optimiste et pessimiste sont généralisables aux séquences de tâches. Cette généralisation peut-être formulée récursivement. Ainsi, pour tout état s, on a $\mathcal{C}_{opt}(s,\epsilon) = 0$ et $\mathcal{C}_{pes}(s,\epsilon) = 0$, ainsi que pour toute tâche t et séquence de tâches σ telles que $\mathcal{T}_{opt}(s, t \circ \sigma) \neq \emptyset$:

$$\mathcal{C}_{opt}(s, t \circ \sigma) = \mathcal{C}_{opt}(s,t) + \min\{C_{opt}(s', \sigma) \mid s' \in \mathcal{T}_{opt}(s,t)\}$$

et de même, pour toute tâche t et séquence de tâches σ telles que $\mathcal{C}_{pes}(s, t \circ \sigma) \neq \emptyset$:

$$\mathcal{C}_{pes}(s, t \circ \sigma) = \mathcal{C}_{pes}(s,t) + \min\{C_{pes}(s', \sigma) \mid s' \in \mathcal{T}_{pes}(s,t)\}.$$

Ces fonctions de coût optimiste et pessimiste permettent d'appliquer AHP à la planification optimale. C'est par exemple le cas de l'algorithme *Angelic Hierarchical A** (AHA*),

présenté dans [59]. Cet algorithme peut être interprété comme étant A* dans l'espace des décompositions, mais pour lequel les réseaux de tâches sont remplacés par des séquences d'actions de haut niveau (on peut aussi considérer qu'il s'agit simplement de réseaux de tâches totalement ordonnés). Ici, la fonction heuristique correspond au coût optimiste d'une séquence de tâches donnée. Il s'agit donc d'une heuristique admissible, puisque ce coût sous-estime le coût réel d'une solution finale, ce qui implique que la solution retournée par AHA* est optimale. La fonction de coût pessimiste n'est pas utilisée pour guider la recherche de solution, mais pour élaguer l'espace de recherche. En effet, si une séquence de tâches admet un coût pessimiste défini (ce qui signifie qu'elle possède au moins une décomposition primitive exécutable), alors tout autre séquence dont le coût optimiste est supérieur à ce coût pessimiste peut être élaguée sans sacrifier l'optimalité de la solution finale.

Toutefois, la modélisation des effets et coûts optimistes et pessimistes demeure l'inconvénient de l'approche proposée par AHP. En effet, pour chaque tâche composée, le modélisateur doit lui même effectuer la synthèse des effets ainsi que la somme des coûts au niveau de chacune de ses décompositions. De plus, ce travail doit être répété sur une partie de la hiérarchie de tâches à chaque modification du domaine de planification HTN (par exemple avec l'ajout d'une méthode de décomposition supplémentaire, ou la modification d'un coût). Un algorithme permettant de calculer automatiquement les descriptions correspondant à la fonction de transition pour les tâches composées est proposé par [114]. Mais il s'agit là d'un algorithme théorique, qui calcule ces descriptions pour chaque instanciation de tâches, alors qu'en pratique les tâches sont définies sous une forme paramétrée dans les domaines HTN.

2.2.5 Propriété des espaces de décompositions

L'ajout de tâches composées rend la planification HTN plus expressive que la planification classique. Cet avantage sur le plan de l'expressivité se paye toutefois en terme de complexité. En effet, la planification HTN en général est indécidable, même lorsque les états et les actions sont décrits avec un langage logique propositionnel, alors que la planification classique est décidable dans ce cadre [26, 25]. Ce résultat provient de la possibilité de modéliser des tâches composées récursives. Il s'agit de tâches pouvant réapparaître parmi les sous-tâches issues de leur propre décomposition (éventuellement après une succession de décompositions). Par exemple, la tâche $Achieve(l)$, dont la description se trouve en figure 2.8, est une tâche récursive car elle est à nouveau présente parmi les sous-tâches de la méthode 2. À cause de ces tâches récursives, l'espace de décomposition se trouve être infini. Une procédure de recherche comme DHTN ou PHTN risque alors d'explorer une des branches infinies de cet espace sans jamais retourner de solution.

Pour régler ce problème, on peut imposer des restrictions sur la nature des réseaux de tâches modélisés dans un domaine de planification. La restriction la plus évidente est de ne pas utiliser de tâches récursives, ce qui impose un nombre fini de décompositions successives pour chaque réseau de tâches. Cependant, il s'agit là d'une contrainte forte pour le modélisateur, qui ne peut par exemple plus modéliser de problème de recherche de chemin tel que celui décrit en figure 2.8. D'autres restrictions moins contraignantes sont toutefois envisageables. Ainsi les réseaux de tâches utilisés pour la modélisation du problème initial et des méthodes de décomposition peuvent être totalement ordonnés (ce qui, par exemple, est le cas dans AHP). Il est alors possible d'utiliser une technique de programmation dynamique nécessitant une structure de donnée calculable en 2-EXPTIME (« seulement » EXPTIME dans le cadre propositionnel) [21, 24] et qui permet de garantir la terminaison de l'algorithme.

Un autre type de restriction qui s'applique au cas général des ordres partiels, et autorise

$Move(l_1,\ l_2)$
    `Precond:`
        $At(l_1)\ \wedge\ Adjacent(l_1, l_2)$
    `Add:`
        $At(l_2)$
    `Delete:`
        $At(l_1)$

$AchieveAt(l)$
    $Method1$
        `Precond:`
            $At(l)$
    $Method2(l_1, l_2)$
        `Precond:`
            $Location(l_1)\ \wedge\ Location(l_2)$
            $Adjacent(l_1, l_2)$
        `Subtasks:`
            $n_1\ :\ Move(l_1, l_2)$
            $n_2\ :\ AchieveAt(l)$
        `Order:`
            $n_1\ \prec\ n_2$

FIGURE 2.8 – Tâche récursive
Exemple de tâche récursive. La décomposition récursive de cette tâche permet de calculer un plan permettant d'atteindre un nœud dans un graphe de navigation.

un usage limité des tâches récursives, est proposée par [3]. Ce type de restriction est fondé sur une répartition des tâches en strates, et se décline en deux versions : l'une étant applicable au cas de la planification HTN dans un espace de décompositions (comme DHTN), et l'autre étant applicable à la planification HTN dans un espace de progressions. Dans le premier cas, il est imposé que pour chaque méthode de décomposition, chacune des sous-tâches doit appartenir à une strate inférieure à celle de la tâche composée associée, sauf si l'ensemble des sous-tâches est réduit à un singleton, et dans ce cas l'unique sous-tâche ne doit pas appartenir à une strate supérieure. Lorsqu'un domaine vérifie cette propriété, on dit, suivant [3], qu'il est « $\leq_1 -stratifiable$ », où $\leq_1$ est un pré-ordre total sur les tâches qui modélise leur stratification. Cette propriété est à la fois nécessaire et suffisante pour garantir qu'un espace de décompositions est fini. Dans le cas de la planification HTN dans un espace de progression, chaque sous-tâche doit aussi appartenir à une strate inférieure, à l'exception d'une unique sous-tâche pouvant être associée à la même strate, mais devant être ordonnée après toutes les autres sous-tâches. À titre d'exemple, cette propriété est respectée par la tâche *AchieveAt* présenté en figure 2.8. Un domaine de planification vérifiant cette propriété est dit « $\leq_r -stratifiable$ », où $\leq_r$ correspond à nouveau au pré-ordre total qui modélise la stratification des tâches. Dans ce cas, cette propriété est suffisante pour garantir que l'espace de recherche est fini, mais il n'a toutefois pas été démontré qu'elle était aussi une condition nécessaire. Pour l'algorithme PHTN, la complexité de la résolution d'un problème de planification HTN dont le domaine est « $\leq -stratifiable$ » correspond à la classe 2-EXPSPACE en général et à la classe EXPSPACE si tous les réseaux de tâches sont de plus totalement ordonnés [2].

Une solution alternative pour limiter la taille des espaces de décomposition sans restreindre l'utilisation des tâches récursives consiste à fixer une profondeur d'appel maximale

pour chaque tâche récursive. Ce système peut être implémenté en incrémentant un compteur qui est ensuite transmis aux sous-tâches par l'intermédiaire d'un paramètre, et dont la valeur est comparé à la profondeur d'appel maximale par une contrainte (par exemple une précondition de méthode avec SHOP, ou une précondition d'action de haut niveau dans le cas de la planification HTN non linéaire ou d'AHP). Ainsi toute décomposition contenant une tâche récursive ayant dépassée sa profondeur d'appel maximale peut être directement élaguée dans l'espace de recherche puisque cette contrainte ne peut pas y être satisfaite. Un avantage supplémentaire de cette pratique est qu'il n'est plus nécessaire de maintenir un ensemble de sous-problèmes visités pour garantir qu'un algorithme termine. À titre d'exemple, la figure 2.9 décrit comment ce système peut être implémenté pour la tâche $AchieveAt(l)$ décrite précédemment en figure 2.8. La valeur maximale de la profondeur d'appel peut être définie dans le modèle sans pour autant remettre en cause sa complétude. Par exemple, la profondeur d'une tâche récursive destinée à parcourir un graphe peut être limitée par la taille du graphe, puisque aucune séquence d'actions passant deux fois par le même nœud ne peut faire partie d'une solution optimale. Ou encore, si on prend l'exemple de la tâche récursive de la figure 2.7, dont l'objectif est de neutraliser un ensemble de cibles par les membres d'un trinôme de fantassins, la profondeur d'appel maximale peut simplement être limitée par le nombre de cibles à neutraliser.

$AchieveAt(l,\ k)$
  $Method1$
    `Precond:`
      $At(l)$
  $Method2(l_1, l_2)$
    `Precond:`
      $Location(l_1)\ \wedge\ Location(l_2)$
      $Adjacent(l_1, l_2)$
      $k\ <\ GraphSize$
    `Subtasks:`
      $n_1\ :\ Move(l_1, l_2)$
      $n_2\ :\ AchieveAt(l, k\ +\ 1)$
    `Order:`
      $n_1\ \prec\ n_2$

FIGURE 2.9 – Limitation de la profondeur d'appel d'une tâche récursive
Exemple de tâche récursive dont la profondeur d'appel est limitée grâce à un compteur k et une profondeur maximale déterminée par la taille du graphe parcouru (encodée ici dans la constante $GraphSize$).

2.2.6 Application de la planification HTN à la simulation militaire

Avant de clore cette partie, il paraît essentiel de retourner vers la raison qui a motivée ici l'étude de la planification HTN, et donc de s'intéresser à l'application de cette technique pour la génération automatique de plans permettant d'animer des unités d'infanterie dans des environnements virtuels en temps réel.

Applications des planificateurs HTN non linéaires

On peut tout d'abord trouver des exemples d'applications parmi les planificateurs HTN non linéaires. C'est par exemple le cas de SOCAP, une application du planificateur

SIPE-2 à la planification d'opérations interarmées [17, 113, 8], ainsi que de l'utilisation du planificateur O-Plan pour un domaine de planification lié aux opérations militaires en zone urbaine [101]. Bien que ces deux exemples d'applications confirment la pertinence de l'utilisation de la planification HTN pour traiter des problèmes de planification militaire, ils n'apportent que très peu d'enseignement en ce qui concerne l'objectif que l'on cherche ici à atteindre. En effet, la planification n'y est pas du tout envisagée dans le contexte de la simulation pour contrôler des agents autonomes, mais dans le but d'assister la prises de décision d'opérateurs humains à la suite d'une interaction entre celui-ci et le système de planification.

Avec *PlannedAssault* [108], on se rapproche davantage de la simulation. Ce planificateur a en effet été conçu pour générer des plans de missions exécutables dans les jeux Arma/Arma2 et dans VBS2, une version de ces jeux dédiée à un usage professionnel. Néanmoins, l'étape de planification y est encore effectuée « hors-ligne », et les unités concernées se situent au niveau de la section ou du groupe d'infanterie. Cependant cet exemple montre à nouveau la pertinence de la planification HTN non linéaire pour ce type d'application, où le plan d'actions doit coordonner entre-elles de nombreuses unités organisées hiérarchiquement et devant parfois agir simultanément. Cet exemple met aussi en valeur l'importance de disposer de plans de bonne qualité, et propose une approche de la planification HTN reposant sur l'algorithme A*, utilisant un système d'heuristique similaire aux coûts optimistes d'AHP.

Applications de SHOP

Finalement, les cas d'application de la planification HTN pour la coordination et l'animation de combattants dans des environnements virtuels ne reposent pas sur des planificateurs HTN non linéaires, mais sur des systèmes dérivés du planificateur SHOP. Ce constat peut paraître étonnant. En effet ces planificateurs ne peuvent retourner que des plans totalement ordonnés, et qui ne sont donc pas adaptés pour coordonner des soldats devant agir en parallèle les uns aux autres. Néanmoins, cette technique de planification HTN semble actuellement être la seule dont l'efficacité computationnel permet une utilisation dans des environnements virtuels temps réel. Une première question que l'on se pose alors concerne le procédé employé par ces applications pour contourner ce problème de plans totalement ordonnés. Un autre sujet d'intérêt concerne aussi la méthode utilisée pour adapter les plans d'actions calculés à l'évolution dynamique de l'environnement.

Un premier cas d'application concerne des travaux de recherche dont le but est d'évaluer l'utilisation de la planification HTN pour contrôler des combattants (appelés ici *bots*) dans le jeux de tir *Unreal Tournament* [70, 40, 39]. Dans ce projet, le planificateur JSHOP (une implémentation de SHOP en Java) est utilisé pour sélectionner dynamiquement les stratégies utilisées dans des parties de domination, un mode du jeu où les joueurs gagnent des points en contrôlant des zones le plus longtemps possible. La planification HTN ne prend pas en charge l'intégralité du contrôle des agents, mais se contente de déterminer des tâches générales à exécuter pour chaque *bot* dans le cadre de la stratégie sélectionnée. Le planificateur cohabite alors avec l'utilisation de machines à états finies, qui sont employées pour modéliser les comportements réactifs. Ainsi chaque *bot* peut suivre les instructions qui lui sont transmises par le planificateur, tout en suivant sa machine à états en parallèle pour réagir face aux évènements imprévus survenant dans son environnement. Dans ce contexte, la re-planification ne semble être effectuée que lorsqu'une tâche attribuée à un *bot* n'est plus applicable (par exemple lorsqu'une position à atteindre est déjà occupée). Pour augmenter l'efficacité du planificateur face à l'évolution dynamique de la situation, celui-ci peut être complété par un système de gestion des objectifs capable d'analyser l'exécution du plan en cours, de détecter les divergences entre la situation obtenue et

l'objectif attendu, et de sélectionner des buts précis pour lesquels la re-planification est nécessaire [69].

Un second exemple d'application de SHOP décrit dans la littérature est *SquadSmart* [31]. Cette fois, la planification est utilisée pour animer les membres de groupes de combat dans un jeux sérieux dédié à l'entraînement des troupes. Le problème posé par les plans totalement ordonnés y est contourné par un procédé de dé-linéarisation. Pour cela, les actions associées à des agents distincts sont exécutées en parallèle. Pour modéliser la synchronisation parfois nécessaire entre les différents soldats, des actions spéciales sont ajoutées aux plans. Par exemple, l'une de ces actions permet de bloquer l'exécution des actions d'un soldat donné en attendant qu'un autre soldat ait achevé l'exécution d'une action particulière. Afin de permettre aux soldats de réagir face à l'évolution dynamique de leur environnement, le planificateur est exécuté à intervalles réguliers, ou lorsque des événements particuliers, tels que la perte d'un soldat, sont détectés.

Enfin, on trouve d'autres applications de SHOP au sein de jeux vidéo grand public. C'est par exemple le cas de *Killzone 2* [109], mais aussi de *Transformers : Fall of Cybertron* [46]. Cependant dans ces deux cas, le planificateur est utilisé pour générer des plans à l'échelle individuelle, et y remplit donc un rôle semblable à celui tenu par GOAP dans *F.E.A.R.*. Ainsi dans *Killzone 2*, la chaine de commandement est modélisée par une architecture multi-niveau similaire à celle décrite pour *Carnival* (voir la sous-partie 2.1.5). Ici la technique utilisée pour adapter dynamiquement les plans consiste, comme pour *SquadSmart*, à re-planifier à intervalles réguliers (de l'ordre de 5 fois par seconde pour *Killzone 2*).

La planification HTN semble donc être le cadre idéal pour implémenter un système de décision adapté à un ensemble d'agents organisés selon plusieurs niveaux de hiérarchie et dont le processus de décision suit une chaine de commandement. Mais si ce modèle de planification correspond bien au principe de découpage des activités et de transmissions des ordres, il y a un aspect de la prise de décision hiérarchique qu'il ne couvre pas, et qui correspond aux différents niveaux de représentation de l'environnement nécessaires pour raisonner efficacement à chaque niveau de la chaine de commandement militaire.

2.3 Planification par abstraction hiérarchique

En intelligence artificielle, l'abstraction est un procédé qui consiste à modifier la représentation d'un problème en le simplifiant, dans l'espoir de résoudre ce problème abstrait plus efficacement, et de s'appuyer sur la solution abstraite pour obtenir une solution concrète, résolvant le problème initial [92]. Une illustration de ce mécanisme est présentée en figure 2.10. La simplification de la représentation d'un problème implique ici la suppression de détails dans la formulation du problème. Cette simplification est par ailleurs ce qui distingue l'abstraction parmi l'ensemble des procédés de reformulation d'un problème [41]. Lorsque ce procédé d'abstraction est appliqué à la planification automatique, on parle de planification par abstraction.

Si ce procédé d'abstraction du problème concret et d'affinement de la solution abstraite est répété à plusieurs reprises, à travers des niveaux de représentation de plus en plus abstrait, on parle alors de planification par abstraction hiérarchique. L'utilisation d'une hiérarchie d'abstractions pour contrôler la recherche en planification a été introduite pour la première fois par Earl Sacerdoti avec le système ABSTRIPS [89]. Bien qu'Earl Sacerdoti ait aussi introduit NOAH, le tout premier planificateur HTN (voir la partie précédente), ces deux systèmes sont tout à fait différents. En effet la planification HTN est un formalisme de planification hiérarchique où le concept de « hiérarchie » se manifeste à travers des tâches décomposables en sous-tâches. Dans le cas de la planification par abstractions hiérarchiques, la hiérarchie se manifeste par une série de niveaux de représentation

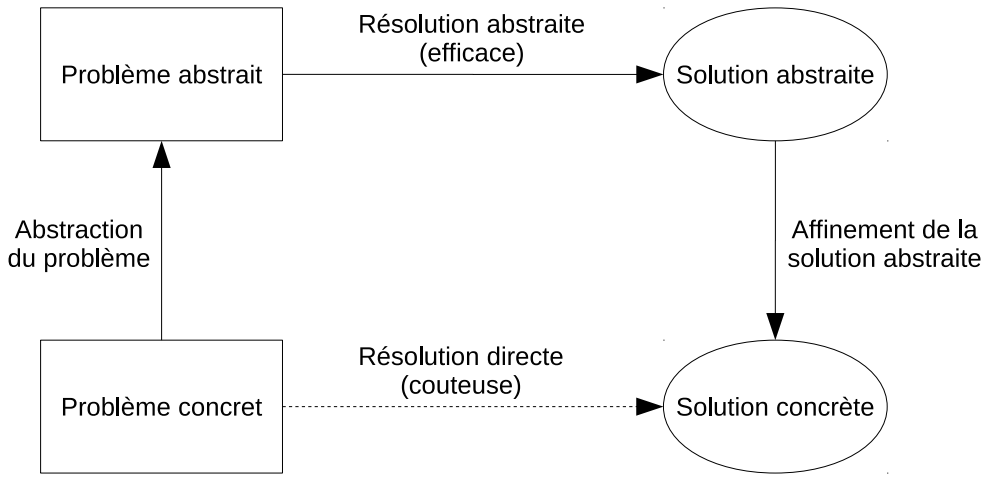


FIGURE 2.10 – Planification par abstraction
Illustration du mécanisme d'abstraction tel qu'il est généralement employé en intelligence artificielle.

du problème de plus en plus abstraits. Chaque formulation abstraite du problème, en commençant par la plus abstraite, permet d'obtenir une solution qui est ensuite utilisée pour guider la résolution d'une formulation moins abstraite, jusqu'à trouver une solution au problème concret.

Ce principe de hiérarchie de niveaux de représentation de l'environnement semble bien adapté aux chaines de commandement militaires. En effet, si on considère le cas de la section d'infanterie, le chef de section ne peut pas raisonner en prenant en compte toute la complexité de l'environnement. Des informations telles que la position précise de chaque soldat, ou la quantité exacte de munitions détenues par ceux-ci constituent un ensemble de données trop complexe pour lui permettre de raisonner efficacement. En conséquence, il adoptera un niveau de représentation plus abstrait, par exemple en ne considérant que les positions globales des groupes de combats ainsi qu'une synthèse de l'état des munitions dans chaque groupe. De plus, une fois que le plan du chef de section a été transmis aux chefs de groupes sous la forme d'ordres, ces derniers vont faire usage de ces instructions pour synthétiser leurs propres plans et ainsi compléter celui de leur chef, ce qui s'apparente à nouveau au principe de la planification par abstraction. Ce procédé se répétant à chaque échelon de la hiérarchie (chef de section, chef de groupe, chef d'équipe et soldat), la planification par abstractions hiérarchiques apparaît alors tout à fait pertinente vis-à-vis de ce contexte.

Cette partie a donc pour objectif de présenter l'application des abstractions hiérarchiques à la planification d'actions. Elle formalise tout d'abord le concept d'abstraction d'un problème de planification, et donc d'un système de transition d'états. Elle passe ensuite en revue différents procédés permettant de spécifier une abstraction par rapport au langage logique utilisé pour décrire un problème de planification. Enfin, cette partie aborde l'utilisation de hiérarchies d'espaces abstraits dans les algorithmes de planification, et s'achève par une étude des principales propriétés des abstractions identifiées en planification, et de leurs effets sur la recherche de solutions.

2.3.1 Abstraction d'un problème de planification

Cette première sous-partie se propose de répondre à la question suivante : qu'est-ce que l'abstraction dans le contexte de la planification d'actions ? L'abstraction en général est un concept délicat à définir. Toutefois, on en a déjà rapidement proposé une définition en introduction : l'abstraction consisterait à supprimer des détails dans la formulation d'un

problème. Mais il s'agit là d'une définition encore vague. Dans [120], Jean-Daniel Zucker propose une définition de l'abstraction en intelligence artificielle que nous pouvons tenter d'appliquer à la planification d'actions. Cette définition est la suivante : « *An abstraction is a change of representation, in a same formalism, that hides details and preserves desirable properties* ». Cette définition se traduit en français de la façon suivante : « Une abstraction est un changement de représentation, dans un même formalisme, qui masque des détails et préserve des propriétés souhaitables ».

En planification d'actions, il existe de nombreux exemples dans la littérature où l'abstraction est définie comme étant un morphisme de systèmes de transition d'états déterministes [51, 42, 45, 44, 38, 37]. Pour rappel, on a vu dans la sous-partie 2.1.1, qu'un problème de planification classique peut être formalisé sous la forme d'un système de transition d'états déterministe, auquel on ajoute un état initial, un ensemble d'états à atteindre et éventuellement une fonction de coût. Par exemple, pour un problème de planification $P = (D, s_0, G)$ où le domaine de planification est $D = (S, A, \mathcal{T}, \mathcal{C})$, le système de transition d'états déterministe correspondant est le triplet $\Sigma_P = (S, A, \mathcal{T})$. Un morphisme entre deux systèmes de transition d'états déterministes est une fonction qui relie chaque état du premier système à un état du second, et qui conserve l'accessibilité des états : si un état du premier système est accessible depuis un autre état, alors l'image de cet état est aussi accessible depuis l'image de l'autre état dans le nouveau système.

Définition 24 (Morphisme de systèmes de transition d'états déterministes)**.** *Soient* $\Sigma = (S, A, \mathcal{T})$ *et* $\Sigma' = (S', A', \mathcal{T}')$ *deux systèmes de transition d'états déterministes. Une application* $f \in S \to S'$ *est un morphisme entre* Σ *et* Σ' *si et seulement si pour tout état* $s \in S$ *et action* $a \in A$*, a est applicable en* $s \implies \exists a' \in A'$ *applicable en* $f(s)$ *et telle que* $\mathcal{T}'(f(s), a') = f(\mathcal{T}(s, a))$*.*

Au niveau d'un problème de planification, on complète cette définition en y ajoutant que l'image de l'état initial doit correspondre à l'état initial du problème abstrait, et que toute image d'un état à atteindre doit aussi correspondre à un état à atteindre dans le problème abstrait. Cette définition de l'abstraction, qui repose donc sur un morphisme de systèmes de transition d'états, satisfait-elle la définition proposée par Jean-Daniel Zucker ? Tout d'abord, on est bien en présence d'un changement de représentation dans un même formalisme, dans la mesure où cette abstraction est formulée par rapport à deux problèmes de planification classique dont elle modifie la représentation des états et des actions. Mais encore, cette transformation préserve bien une propriété souhaitable, puisqu'elle préserve l'accessibilité des états, ce qui est, comme on le verra plus loin, une condition essentielle, puisque les algorithmes de planification par abstraction reposent sur l'existence d'une solution au niveau abstrait pour guider la recherche de la solution concrète.

Mais pour faire de cette transformation une abstraction, la définition initiale précise qu'il est aussi nécessaire de « masquer des détails ». Dans certains exemples de la littérature, seuls les états sont affectés par le morphisme, tandis que l'ensemble des actions n'est pas modifié (c'est par exemple le cas d'ABSTRIPS, que l'on présente dans la partie suivante). Dans ce cas, il suffit d'imposer au morphisme d'être non-injectif, c'est-à-dire de faire en sorte qu'il transforme au moins deux états concrets différents en un même état abstrait. Ainsi, on a bien masqué des détails, puisque on a supprimé l'information permettant de distinguer au moins deux états. Toutefois cette contrainte devient trop restrictive lorsque l'ensemble des actions peut aussi être simplifié. En effet, tout en restant dans le cadre d'un morphisme de systèmes de transition déterministe, il est aussi possible de diminuer la quantité d'information en diminuant le nombre d'actions sans modifier les états (le morphisme correspond alors à la fonction identité, qui est injective). C'est par exemple le cas dans [4], où le mécanisme d'abstraction présenté ne dépend pas nécessairement d'une simplification de la structure des états, mais fusionne des opérateurs après une

généralisation de leurs préconditions. La dissimulation des détails peut donc se manifester à la fois par une réduction du nombre d'états ou par une réduction du nombre d'actions. Dans chacun des deux cas, l'effet produit est la réduction du nombre de transitions dans l'espace de recherche abstrait. C'est donc ce dernier critère que l'on retient ici en ce qui concerne la réduction des détails.

Reste encore à considérer ce qu'il advient de la fonction de coût au niveau du problème de planification abstrait. Chaque transition étant associée à un coût, et chaque transition abstraite correspondant à plusieurs transitions concrètes, on choisira ici d'associer à la transition abstraite le plus petit coût associé aux transitions concrètes qui lui correspondent. Ce choix permet l'utilisation de ces coûts abstraits afin de former des heuristiques admissibles pour l'application de l'abstraction aux algorithmes de recherche heuristique.

La définition complète de l'abstraction d'un problème de planification classique, telle qu'on la considère dans ce manuscrit, est donc la suivante :

Définition 25 (Abstraction d'un problème de planification classique)**.** *Soit $P = (D, s_0, G)$ un problème de planification classique où $D = (S, A, \mathcal{T}, \mathcal{C})$. Une abstraction de P est un problème $P' = (D', s'_0, G')$ où $D' = (S', A', \mathcal{T}', \mathcal{C}')$ et tel que :*

1. *il existe un morphisme f entre Σ_P et $\Sigma_{P'}$ tel que $f(s_0) = s'_0$ et $f(G) \subseteq G'$.*
2. *$|E_P| < |E'_P|$, où pour un problème de planification P, $E_P = \{(s_1, s_2, a) \in S \times S \times A \mid a$ est applicable en $s_1 \wedge \mathcal{T}(s_1, a) = s_2\}$.*
3. *Pour tout état $s' \in S'$ et toute action $a' \in A'$ applicable à s', $\mathcal{C}'(s', a') = \min\{\mathcal{C}(s, a) \mid s \in f^{-1}(s'), a \in A$ tels que $\mathcal{T}(s, a)$ est défini et $f(\mathcal{T}(s, a)) = \mathcal{T}'(s', a')\}$.*

La fonction f est alors appelée fonction d'abstraction entre P et P'. Afin d'illustrer l'abstraction d'un problème de planification telle que l'on vient de la définir, la figure 2.11 présente un exemple d'abstraction du problème de planification initialement présenté en figure 2.1.

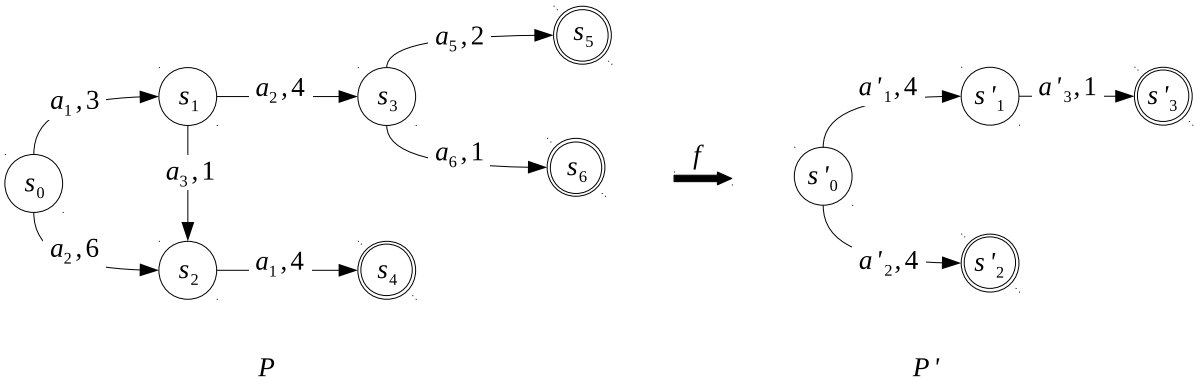


FIGURE 2.11 – Fonction d'abstraction
Abstraction d'un problème P en un problème abstrait P' par une fonction d'abstraction f définie telle que $f^{-1}(s'_0) = \{s_0, s_1, s_2\}$, $f^{-1}(s'_1) = \{s_3\}$, $f^{-1}(s'_2) = \{s_4\}$ et $f^{-1}(s'_3) = \{s_5, s_6\}$.

2.3.2 Décrire l'abstraction d'un problème de planification d'actions

Dans la partie précédente, nous avons défini l'abstraction à partir du modèle général d'un problème de planification classique. Or en pratique, un problème de planification classique est modélisé à partir d'une description reposant sur un langage logique (par exemple avec ADL pour une logique du premier ordre). À partir de ce constat, il est légitime de s'intéresser à la façon dont l'abstraction d'un problème de planification peut

être décrite par rapport à sa description en logique, et non pas à partir de l'énumération des états et des actions qui correspondent à son modèle. Afin d'apporter un aperçu de la façon dont cet objectif peut être atteint, cette partie propose une revue de différents exemples d'applications de l'abstraction à la planification d'actions que l'on peut trouver dans la littérature.

Le tout premier système qui applique le principe de l'abstraction en planification est ABSTRIPS [89]. Comme son nom l'indique, ce système est basé sur le planificateur STRIPS, mais aussi sur le formalisme associé, où les états, préconditions et buts sont représentés par des ensembles de formules logiques. La planification y est effectuée à travers une hiérarchie d'espaces de recherche abstraits. Chacun de ces espaces est obtenu à partir du précédant en supprimant certaines formules dans les préconditions des opérateurs. Pour cela, un nombre entier positif, appelé « criticité » est affecté à chaque précondition dans chaque opérateur. De plus, chaque espace abstrait correspond à un niveau en fonction de sa position dans la hiérarchie : 0 pour l'espace concret, 1 pour l'espace abstrait suivant, et ainsi de suite. Lors du passage entre deux niveaux, toutes les préconditions restantes dont la criticité est inférieure au nouveau niveau sont alors supprimées. Ainsi au dernier niveau, seules les préconditions les plus « critiques » restent présentes pour résoudre le problème de planification.

On peut remarquer que ce système ne correspond pas à la définition de l'abstraction telle qu'elle a été définie dans la partie précédente. Bien que le principe des criticités réduise des détails du modèle en supprimant des formules, celui-ci n'a pas pour effet de réduire la taille du système de transition correspondant au problème de planification. En effet, ni les états, ni les effets des actions ne sont modifiés par cette transformation. Celle-ci correspond en fait à un relâchement des préconditions, et donc à une augmentation du domaine d'application des actions. C'est la difficulté du problème qui se trouve ici simplifiée, et non pas sa représentation (du moins, la simplification du problème n'est pas une conséquence de la simplification de sa représentation).

Une utilisation de ces criticités qui est conforme au principe de l'abstraction telle que nous l'avons définie est proposée par Craig Knoblock [52, 54]. Le principe des criticités n'est alors plus de relâcher le modèle, mais de le réduire. Pour cela, on se place toujours dans le cadre de STRIPS, mais en restreignant l'usage des formules logiques : les états sont décrits par un ensemble de formules atomiques, les préconditions et les buts sont décrits par des ensembles de littéraux et les effets sont décrits par des listes de formules atomiques. Chaque formule atomique est alors associée à un niveau de criticité, qui ne dépend donc plus des préconditions dans lesquels cette formule apparait. Lorsque la criticité d'une formule atomique devient inférieur au niveau d'abstraction considéré, celle-ci est supprimée des états, préconditions, buts et effets dans lesquels elle apparaît. On retrouve aussi ce principe de la réduction d'information en planification optimale avec les « bases de patterns » [19, 35], où les patterns sont constituées d'ensemble de formules atomiques à supprimer du modèle initial pour le simplifier et en dériver automatiquement des heuristiques. Josh Tenenberg [102, 103] propose aussi une version de l'abstraction par réduction adaptée à un cadre plus étendu, où les formules logiques ne sont plus limitées aux conjonctions de littéraux. Les criticités y sont directement appliquées aux symboles de prédicats, ce qui définit une abstraction moins « fine » que dans les cas précédents, puisqu'elle entraine la suppression de toute l'interprétation d'un symbole de prédicat, au lien du seul élément représenté par une formule atomique. L'abstraction y est appliquée aux formules à partir de leur forme clausale, et une contrainte forte est imposée aux criticités : deux symboles de prédicat qui apparaissent dans la même clause doivent être associées à la même criticité, qui vaut alors pour toute la clause. Ainsi les formules logiques sont abstraites en supprimant les clauses dont la criticité n'est pas assez importante au regard du niveau d'abstraction considéré.

Tenenberg propose aussi d'appliquer une méthode d'abstraction alternative, qui n'est pas fondée sur de la réduction de prédicats ou de formules atomiques, mais qui associe plusieurs prédicats concrets à un même prédicat abstrait [104, 105]. L'intérêt de ce type d'abstraction est de regrouper les objets du domaine dans des classes abstraites en fonction d'une hiérarchie d'héritage, et ainsi de regrouper certaines relations et actions associées à ces objets. Dans le cadre d'un problème de combat d'infanterie, ce type d'abstraction peut par exemple être exploité pour simplifier la représentation d'un problème en généralisant des équipements (par exemple différents types d'armes) ou en généralisant des unités correspondant à différentes spécialités.

Enfin, pour les problèmes de planification contenant de nombreux objets, Juan-Antonio Fernandez-Madrigal et Javier Gonzalez [27, 28] proposent une représentation des objets et de leur relations sous la forme de graphes hiérarchiques qui agrègent des ensembles d'objets en « super-objets ». Ils appliquent cette structure au cas de la navigation pour des robots dans des bâtiments, et ces graphes hiérarchiques permettent alors de représenter les relations de composition entre les différents éléments de l'espace dans lequel naviguent les robots (points de passages, pièces, étages, etc.). Ce dernier type d'abstraction semble bien adapté au cas des hiérarchies militaires, puisque l'un de leur principe est d'agréger les unités sur plusieurs échelons (par exemple, les soldats sont regroupés en trinômes, les trinômes en groupes de combat, etc.).

2.3.3 Planification et hiérarchie d'abstractions

En planification par abstraction, il est possible d'utiliser une, mais aussi plusieurs abstractions du problème de planification initial, comme on l'a vu pour les systèmes qui associent différent niveaux de criticités aux prédicats. On considère alors une suite finie de problèmes de plus en plus abstraits que l'on appelle une hiérarchie d'abstractions. Chaque élément de cette suite correspond ainsi à un niveau d'abstraction dans cette hiérarchie. Le premier élément correspond au problème concret, tandis que chaque élément suivant correspond à une abstraction du problème de planification précédent. Dans la suite de ce manuscrit on aura pour convention de noter X^i tout élément situé à un niveau i d'une hiérarchie d'abstraction. Par exemple S^i correspond à l'ensemble des états au niveau i, et A^i correspond à l'ensemble des actions au même niveau.

Définition 26 (Hiérarchie d'abstractions d'un problème de planification classique)**.** *Une hiérarchie d'abstractions d'un problème de planification classique* $P^0 = (D^0, s_0^0, G^0)$ *est une suite finie* $(P^i, f^i)_{i\in[\![1,n]\!]}$*, où* $n > 0$ *et telle que* $\forall i \in [\![1, n]\!]$*,* P^i *est une abstraction de* P^{i-1} *et* $f^i \in S^{i-1} \to f^i$ *est une fonction d'abstraction entre* P^{i-1} *et* P^i*.*

Le principe de la planification par hiérarchie d'abstractions consiste à rechercher une solution à chaque problème de planification abstrait en s'appuyant sur la solution retournée au niveau supérieur. Pour cela, la recherche de solutions s'effectue tout d'abord dans l'espace de recherche le plus abstrait, c'est-à-dire celui qui se situe au sommet de la hiérarchie d'abstractions (cette hiérarchie étant une séquence, il s'agit donc du dernier élément de cette séquence). On fait alors appel à un planificateur classique affin de retourner une solution. Si une telle solution existe et est retournée, celle-ci sert alors de guide pour résoudre le problème au niveau inférieur.

Il existe plusieurs façons de guider la résolution du problème de planification à un niveau donné en se servant de la solution au niveau supérieur. Une première méthode consiste à calculer la séquence d'états intermédiaires auxquels le plan permet d'accéder en exécutant successivement les actions qui le compose. Les images réciproques de chacun de ces états par la fonction d'abstraction forment alors une séquence de buts intermédiaires. En réduisant l'ensemble des solutions aux séquences d'actions qui satisfont ces buts,

l'espace de recherche se trouve alors réduit, ce qui permet d'améliorer l'efficacité de la recherche de solution. Cette séquence de buts intermédiaires permet ainsi de résoudre le problème en considérant une suite de sous-problèmes, où le but correspond à un but intermédiaire, et où l'état initial correspond à l'état obtenu par la séquence d'actions qui accomplit le but intermédiaire précédent. On retrouve cette méthode dans des algorithmes de planification d'itinéraires appliqués aux jeux vidéo, où elle se présente sous la forme de versions hiérarchiques de l'algorithme A* [11, 96], mais aussi en robotique, pour de la planification d'actions et de la navigation dans des environnements hiérarchisés [30].

Mais on trouve aussi une autre méthode pour guider la résolution d'un problème à un niveau donné en se servant de la solution retournée au niveau supérieur. Cette seconde méthode consiste, dans un premier temps, à remplacer chaque action du plan obtenu au niveau supérieur par une nouvelle action, ou une séquence d'actions, du niveau considéré. Cette opération est possible pour les systèmes où l'abstraction d'un problème de planification classique permet aussi de définir une correspondance entre chaque action abstraite et un ensemble de séquences d'actions au niveau directement inférieur. Ce cas correspond par exemple à celui de l'abstraction fondée sur les criticités, où chaque opérateur est défini à tous les niveaux d'abstraction, et où chaque action est alors « remplacée » par elle-même. Dans le cas de la généralisation d'opérateurs, une action abstraite peut correspondre à plusieurs actions au niveau inférieur : il faut donc remplacer chaque action abstraite en choisissant un candidat possible (cela correspond alors à un point de choix auquel il faut effectuer un retour arrière en cas d'échec). Enfin, dans le cadre de l'utilisation de « macro-opérateurs » abstraits, une action est remplacée par la séquence d'actions associée. Le plan obtenu après avoir remplacé chaque action n'est généralement pas complet : en effet la concrétisation des états et des actions entraîne l'apparition de nouvelles préconditions et de nouveaux buts qui ne sont pas encore satisfaits. On parle alors de « squelette de plan », constitué d'intervalles entre l'état initial, chaque action qui le compose et le but à atteindre. On retrouve ainsi la situation décrite pour la première méthode au paragraphe précédent, car il faut désormais résoudre une séquence de sous-problèmes pour compléter chaque intervalle, mais où cette fois les buts intermédiaires correspondent aux préconditions des actions. Cette seconde méthode est celle généralement employée par les planificateurs qui parcourent un espace d'états et qui utilisent une hiérarchie d'abstractions fondée sur des criticités. On la retrouve donc pour ABSTRIPS [89], où les sous-problèmes sont résolus à chaque niveau en faisant appel au planificateur STRIPS, ainsi qu'a un système similaire mais reposant cette fois sur le planificateur PRODIGY [53, 12].

Quelle que soit la méthode employée, le plan obtenu correspond alors à une concrétisation du plan précédent. Par concrétisation d'un plan abstrait, on entend ici que le plan au niveau inférieur « suit le chemin » indiqué par le plan abstrait. Ou encore, pour l'exprimer d'un façon plus formelle, que si on applique la fonction d'abstraction à la suite des états traversés par le plan obtenu, alors la séquence d'états traversés par le plan abstrait en constitue une sous-suite, comme le montre la figure 2.12. Lorsque la nouvelle solution est retournée, celle-ci est transmise une fois de plus au niveau inférieur afin d'y guider la recherche de solution, jusqu'à atteindre le niveau concret, qui correspond au problème initial, et où la solution obtenue correspond à une solution de ce problème.

L'algorithme 6 présente une implémentation possible de la procédure que l'on vient de décrire, et que l'on nomme ABPLAN. Cet algorithme décrit un « méta-planificateur », dans la mesure ou l'une des données attendues en entré est un planificateur désigné par le nom PLAN. Le méta-planificateur supervise la transmission des solutions entre les niveaux d'abstraction, ainsi que leur décomposition en sous-problèmes, mais délègue la résolution de ces sous-problèmes à PLAN. Afin d'assurer la complétude de cet algorithme, il est nécessaire que PLAN ne retourne pas qu'une seule solution au problème posé, mais soit

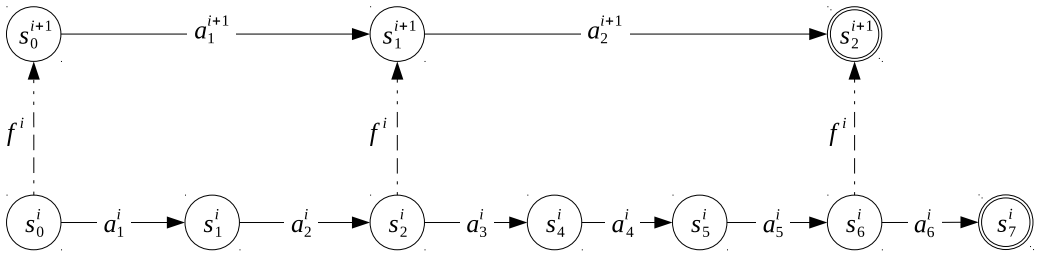


FIGURE 2.12 – Concrétisation d'un plan abstrait
Concrétisation d'un plan abstrait à un niveau d'abstraction $i+1$ en un nouveau plan au niveau i. Les états traversés par le plan abstrait correspondent à des états concrets traversés dans le même ordre par le plan concret.

capable d'en retourner successivement toutes les solutions pour fournir des alternatives en cas de retour arrière. Cet ensemble de solutions doit aussi être fini, sans quoi la terminaison de cet algorithme ne peut pas être assurée pour toute stratégie d'exploration de l'espace de recherche (en profondeur d'abord, largeur d'abord ou meilleur d'abord, etc). L'espace de recherche (correspondant aux états accessibles) de PLAN étant fini, il suffit pour cela de veiller à exclure de l'ensemble des solutions les plans qui visitent plusieurs fois le même état (ce qui se fait, par exemple, en maintenant un ensemble d'états visités et en élaguant les plans qui atteignent ces états). Enfin, l'implémentation d'ABPLAN proposée ici correspond à la première des deux méthodes de concrétisation des plans abstraits que l'on a décrit plus haut. Pour adapter cet algorithme à la seconde méthode, il suffit simplement de concrétiser chacune des actions du plan abstrait, et de décomposer les sous-problèmes en se servant des préconditions des actions en guise de buts intermédiaires.

L'algorithme ABPLAN est une implémentation de la planification par hiérarchie d'abstractions adaptée au contexte de la recherche de solutions en progressant les états par les actions. Bien que PLAN puisse tout à fait être implémenté par un planificateur non-linéaire dans l'algorithme 6, il existe une variante d'ABPLAN plus adaptée au contexte de la recherche de solution dans un espace de plans partiels. Cette variante est par exemple implémentée par ABTWEAK [117, 118], un planificateur qui utilise le principe de l'abstraction par les criticités et repose sur les opérations de modification de plan définies pour TWEAK. Au lieu de décomposer une solution à un niveau donné en une séquence de sous-problèmes totalement ordonnés, ce qui contredirait le principe du moindre engagement propre à la planification non-linéaire, ABTWEAK utilise les criticités pour ordonner l'établissement des liens causaux entre les effets des actions et les sous-buts dans le plan. Ainsi, ABTWEAK fait tout d'abord appel aux opérations de modification de plan de TWEAK afin d'obtenir un plan partiellement ordonné complet par rapport au plus haut niveau d'abstraction (où un certain nombre de sous-buts sont ignorés). Une fois qu'un plan abstrait complet est obtenu, les sous-buts correspondant au niveau inférieur sont réintroduits et il est à nouveau fait appel aux opérations de modification de plans pour compléter le plan au niveau inférieur. Une fois de plus, ce processus se répète à travers les niveaux d'abstraction jusqu'à l'obtention d'une solution, qui correspond au plan complet obtenu au niveau concret.

Enfin, les hiérarchies d'abstractions peuvent aussi être utilisées dans le cadre de l'application de la recherche heuristique à la planification optimale. Il est ainsi possible de concevoir une version hiérarchique de l'algorithme A* qui utilise les différents niveaux d'abstraction pour obtenir une heuristique admissible de façon automatique [45, 43]. L'heuristique au niveau concret y est calculée à partir du coût du plan optimal obtenu en résolvant le problème au niveau d'abstraction supérieur. Au niveau supérieur, le même procédé est employé, et ainsi de suite, jusqu'à atteindre le niveau le plus abstrait, où la recherche du plan optimal est effectuée sans heuristique (par exemple avec une recherche en largeur

Algorithm 6 Algorithme de planification par abstraction hiérarchique ABPLAN

entrées : Un problème de planification classique $P^0 = (D^0, s_0^0, G^0)$, une hiérarchie d'abstractions $(P^i, f^i)_{i \in [\![1,n]\!]}$ de taille $n > 0$ et une procédure de planification classique PLAN.

sorties : un plan solution de P ou *échec*.

$Ouvert \leftarrow \{(s_0^n, \epsilon, \epsilon, n)\}$
tant que $Ouvert \neq \emptyset$ **faire**
  $s^i, \sigma^i, \sigma^{i+1}, i \leftarrow$ extraire un élément de $Ouvert$
  si $\sigma^{i+1} = \epsilon$ **et** $s^i \in G^i$ **alors**
    si $i > 0$ **alors**
      $Ouvert \leftarrow Ouvert \cup \{(s_0^{i-1}, \epsilon, \sigma^i, i-1)\}$
    sinon
      retourner σ^i
    fin si
  sinon
    si $\sigma^{i+1} = \epsilon$ **alors**
      $S_i \leftarrow G^i$
    sinon
      $a^{i+1}\sigma^{i+1} \leftarrow \sigma^{i+1}$
      $S_i \leftarrow f^{i+1^{-1}}(\mathcal{T}^{i+1}(f^{i+1}(s^i), a^{i+1}))$
    fin si
    pour chaque solution σ de $PLAN(D^i, s^i, S_i)$ **faire**
      $Ouvert \leftarrow Ouvert \cup \{(\mathcal{T}^i(s^i, \sigma), \sigma^i\sigma, \sigma^{i+1}, i)\}$
    fin pour
  fin si
fin tant que
retourner *échec*

d'abord). Comme ce procédé doit être répété à chaque étape de la recherche, il est possible d'utiliser un cache qui enregistre le meilleur coût optimal obtenu pour chaque paire d'états (chaque solution abstraite retournée permet d'enregistrer le coût optimal pour chaque paire d'états traversée par le plan). Ce cache permet à cet algorithme d'être beaucoup plus efficace en évitant de recalculer plusieurs fois la valeur de l'heuristique entre deux états. On remarque que dans cette utilisation des hiérarchies d'abstractions, seul le coût de la solution abstraite est utilisé pour guider la recherche, et que contrairement aux méthodes décrites précédemment, le plan obtenu n'est pas contraint d'être une concrétisation des plans retournés aux niveaux supérieurs.

2.3.4 Propriétés des hiérarchies d'abstraction

Nous avons introduit la procédure ABPLAN dans la partie précédente, et avons aussi discuté d'autres techniques de planification s'appuyant sur une hiérarchie d'abstraction. Nous allons désormais introduire une successions de propriétés liés aux hiérarchies d'abstractions, et discuter de leur effet sur les procédures de planification.

La première de ces propriétés est désignée par Qiang Yang dans [116] sous le nom d'*Upward Solution Property* (USP). D'après Qiang Yang, une hiérarchie d'abstraction vérifie l'USP si pour tout niveau d'abstraction, l'existence d'une solution à un problème de ce niveau implique l'existence d'une solution à l'abstraction de ce problème. Cette propriété est nécessaire et suffisante pour assurer le fonctionnement des planificateurs qui ne se servent de l'abstraction que pour calculer des heuristiques. En effet, elle assure au moins l'existence d'un plan au niveau supérieur dont le coût fournit alors la valeur de

l'heuristique. Or, c'est tout ce dont ces planificateurs ont besoin, puisqu'ils n'imposent aucun lien entre les solutions retournées à chaque niveau. Il est possible de montrer que lorsque l'abstraction correspond à un morphisme de systèmes de transition d'états, c'est-à-dire lorsqu'elle correspond à la définition 25, cette propriété est toujours vraie.

Théorème 1 (USP)**.** *Soient P et P' deux problèmes de planification classique tels que P' est une abstraction de P. Si P est résoluble, alors P' est résoluble.*

Démonstration. Soit $f \in S \to S'$ une fonction d'abstraction entre P et P'. Montrons par récurrence que la propriété suivante, notée $P(k)$ est vrai pour tout $k \in \mathbb{N}$: $\forall s \in S$ si (D, s, G) admet une solution de taille k, alors $(D', f(s), G')$ admet une solution.

1. $k = 0$: soit $s \in S$ tel que ϵ est une solution de (D, s, G). $\mathcal{T}(s, \epsilon) = s$ donc $s \in G$. $f(s) \in f(G)$ et $f(G) \subseteq G'$ (d'après la définition 25) donc $f(s) \in G'$. ϵ est exécutable en $f(s)$ et $\mathcal{T}'(f(s), \epsilon) = f(s)$ donc $\mathcal{T}'(f(s), \epsilon) \in G'$, et donc ϵ est une solution de $(D', f(s), G')$. $P(0)$ est donc vraie.
2. Soit $k \in \mathbb{N}$, on suppose que $P(k)$ est vraie. Soit $s \in S$ tel que (D, s, G) admet une solution de taille $k+1$ notée $a_1 \circ ... \circ a_{k+1}$. $a_2 \circ ... \circ a_{k+1}$ étant une solution de taille k de $(D, \mathcal{T}(s, a_1), G)$, il existe donc d'après $P(k)$ une solution de $(D', f(\mathcal{T}(s, a_1)), G')$ que l'on note σ'. De plus, il existe d'après la définition 24 une action a' applicable en $f(s)$ et telle que $\mathcal{T}'(f(s), a') = f(\mathcal{T}(s, a_1))$. Ainsi, $a' \circ \sigma'$ est exécutable en $f(s)$ et $\mathcal{T}(f(s), a' \circ \sigma') \in G'$, donc $a' \circ \sigma'$ est une solution de $(D, f(s), G')$ et donc $P(k+1)$ est vraie.

$P(0)$ est vraie, et $\forall k \in \mathbb{N}, P(k) \implies P(k+1)$. Par récurrence, $P(k)$ est donc vraie pour tout $k \in \mathbb{N}$. En conséquence, si (D, s_0, G) admet une solution de taille quelconque, alors $(D, f(s_0), G')$ admet une solution. □

Cependant, cette propriété n'est pas suffisante dans le cas d'ABPLAN. En effet, ABPLAN planifie en concrétisant successivement une suite de solutions de moins en moins abstraites jusqu'à obtenir la solution concrète. Cette dernière ne peut donc être atteinte qu'à la condition qu'à chaque niveau d'abstraction (excepté le dernier) il existe au moins une solution au niveau plus abstrait à partir de laquelle elle peut être concrétisée. Dans le cadre d'ABPLAN, la concrétisation consiste à transformer la séquence d'états traversés par la solution abstraite en une séquence de sous-buts et à construire une solution concrète qui satisfait chacun de ces sous-buts :

Définition 27 (Concrétisation d'une solution)**.** *Soient P et P' deux problèmes de planification classique résolubles tels qu'il existe une fonction d'abstraction f entre P et P'. Soit $\sigma' \in A'^*$ une solution de P'. Une concrétisation de σ' par f est une solution σ de P telle que la séquence $s'_1 \circ ... \circ s'_{|\sigma'|}$ est une sous-suite de la séquence $f(s_1)...f(s_{|\sigma|})$ où :*

1. $\forall i \in [\![1, |\sigma|]\!], s_i = \mathcal{T}(s_{i-1}, a_i)$ *où* $\sigma = a_1 \circ ... \circ a_{|\sigma|}$.
2. $\forall i \in [\![1, |\sigma'|]\!], s'_i = \mathcal{T}'(s'_{i-1}, a'_i)$ *où* $\sigma' = a'_1 \circ ... \circ a'_{|\sigma|}$.

À nouveau, il est possible de montrer que dans le cadre des abstractions présentées ici, il est toujours possible de construire cette suite de solutions dont chacune est une concrétisation de la précédente, et d'en déduire la complétude d'ABPLAN. On pourrait être tenté d'effectuer cette démonstration en montrant que pour un problème concret P et une abstraction P' de ce problème, toute solution de P peut être obtenue en concrétisant une solution de P'. Cependant, cela ne serait pas suffisant. En effet, on a précisé dans la partie précédente que pour assurer la terminaison d'ABPLAN, on considère que PLAN ne retourne pas de solution cyclique (qui visite plusieurs fois le même état). Or, une conséquence de ce choix est de restreindre l'ensemble des solutions produite à chaque niveau. Il faut donc s'assurer que la concrétisation est possible malgré l'absence de ces

solutions. Pour cela, il suffit de montrer qu'il est possible d'obtenir n'importe quelle solution en concrétisant une suite de solutions acycliques, qui font partie des solutions pouvant être construites à chaque niveau par ABPLAN. En guise d'étape intermédiaire, montrons tout d'abord que toute solution concrète peut être concrétisée à partir d'une solution abstraite acyclique :

Proposition 1. *Soient P et P' deux problèmes de planification classique tels qu'il existe une fonction d'abstraction f entre P et P'. Supposons qu'il existe une solution $\sigma \in A^*$ de P. Alors il existe une solution $\sigma' \in A'^*$ acyclique dont σ est une concrétisation par f.*

Démonstration. Montrons par récurrence que la propriété suivante, baptisée $P(k)$, est vraie pour tout $k \in \mathbb{N}$: $\forall s \in S$, s'il existe une solution $\sigma \in A^*$ de (D, s, G) telle que $|\sigma| = k$, alors il existe une solution $\sigma' \in A'^*$ de $(D', f(s), G')$ acyclique dont σ et une concrétisation.

1. $k = 0$: soit un état $s \in S$ tel que ϵ est une solution de (D, s, G). Alors, ϵ est une solution de $(D', f(s), G')$ (voir la preuve du théorème 1). Or, ϵ est acyclique, donc $P(0)$ est vraie.
2. Soit $k \in \mathbb{N}$, supposons que $P(k)$ est vraie.

   Soient $s \in S$, et $a_1 \circ ... \circ a_{k+1}$ une solution de (D, s, G) (dont on suppose l'existence). On note $s_1, ..., s_{k+1}$ la suite des états traversés par $a_1 \circ ... \circ a_{k+1}$. $a_2 \circ ... \circ a_{k+1}$ étant une solution de (D, s_1, G) de taille k, il existe, d'après $P(k)$, une solution de $(D', f(s_1), G')$ acyclique que l'on note σ' et dont $a_2 \circ ... \circ a_{k+1}$ est une concrétisation.

   (a) Supposons dans un premier temps que $f(s) = f(s_1)$. Dans ce cas, σ' est une solution de $(D, f(s), G')$ acyclique. $a_2 \circ ... \circ a_{k+1}$ étant une concrétisation de σ', la suite des états traversés par σ' est une sous-suite de $f(s_2)...f(s_{k+1})$ et est donc une sous-suite de $f(s_1), f(s_2), ..., f(s_{k+1})$, la suite des états traversés par $a_1 \circ ... \circ a_{k+1}$. $a_1 \circ ... \circ a_{k+1}$ est donc une concrétisation de σ', et donc $P(k+1)$ est vraie.

   (b) Supposons désormais que $f(s) \neq f(s_1)$. Notons $s'_1, ..., s_{|\sigma'|}$ la suite des états traversés par σ'.

      i. Supposons que $\forall i \in [\![1, |\sigma'|]\!], f(s) \neq s'_i$. D'après la définition 24, il existe alors $a' \in A'$ applicable en $f(s)$ et telle que $\mathcal{T}'(f(s), a') = f(s_1)$. $a'\sigma'$ est donc une solution de $(D', f(s), G')$ acyclique. À nouveau, $a_2 \circ ... \circ a_{k+1}$ est une concrétisation de σ', donc la suite des états $s'_1, ..., s'_{|\sigma'|}$ est une sous-suite de $f(s_2), ..., f(s_{k+1})$. En conséquence, $f(s_1), s'_1, ..., s'_{|\sigma'|}$ est une sous-suite de $f(s_1), f(s_2), ..., f(s_{k+1})$, et donc $a_1 \circ ... \circ a_{k+1}$ est une concrétisation de $a' \circ \sigma$. $P(k+1)$ est donc vraie.

      ii. Sinon, il existe alors $k \in [\![1, |\sigma'|]\!]$ tel que $f(s) = s'_k$. Notons alors $\sigma' = a'_1 \circ ... \circ a'_{|\sigma'|}$. La séquence d'actions $a'_{k+1} \circ ... \circ a'_{|\sigma'|}$ est donc une solution de $(D', f(s), G')$ acyclique. De plus, $s'_{k+1}, ..., s'_{|\sigma'|}$ est une sous-suite de $s'_1, ..., s'_{|\sigma'|}$ qui est une sous-suite de $f(s_2), ..., f(s_{k+1})$. $s'_{k+1}, ..., s'_{|\sigma'|}$ est donc une sous-suite de $f(s_2), ..., f(s_{k+1})$ et $a_1 \circ ... \circ a_{k+1}$ est donc une concrétisation de $a'_{k+1} \circ ... \circ a'_{|\sigma'|}$. $P(k+1)$ est donc vraie.

$P(0)$ est vraie, et $\forall k \in \mathbb{N}, P(k) \implies P(k+1)$. Par récurrence, $P(k)$ est vrai pour tout $k \in \mathbb{N}$. Ainsi, s'il existe une solution $\sigma \in A^*$ de (D, s_0, G) de taille quelconque, alors il existe une solution de $(D', f(s_0), G')$ acyclique dont σ et une concrétisation. □

À partir de cette proposition, il est possible de justifier la complétude d'ABPLAN. Cette complétude dépend toutefois des propriétés de PLAN, la procédure de planification à laquelle ABPLAN fait appel pour planifier à chaque niveau. Il est en effet nécessaire

de préciser que PLAN doit être capable d'énumérer successivement toutes les solutions acycliques aux problèmes de planification qui lui sont posés. Comme nous l'avons déjà vu, la restriction portant sur les plans acycliques est un moyen raisonnable de garantir la terminaison d'ABPLAN. Le fait que PLAN soit en plus complet vis-à-vis de ces solutions (« complet » signifiant ici qu'aucune solution acyclique n'est élaguée de l'espace de recherche) permet de garantir que toutes les concrétisations acycliques d'un plan abstrait sont visitées par ABPLAN.

Théorème 2 (Complétude d'ABPLAN)**.** *Soient P^0 un problème de planification classique, un entier $n > 0$ et $(P^i, f^i)_{i\in[\![1,n]\!]}$ une hiérarchie d'abstractions de P^0. Si pour tout problème de planification classique, PLAN retourne l'ensemble des solutions acycliques, alors ABPLAN retourne au moins l'ensemble des solutions acycliques de P^0.*

Démonstration. Supposons que P^0 est résoluble. Il existe alors au moins une solution acyclique σ^0 de P^0. D'après, la proposition 1, on peut déduire qu'il existe une suite $\sigma_1 \circ ... \circ \sigma_n$ telle que $\forall i \in [\![1, n]\!]$, σ_i est une solution acyclique de P^i et σ_{i-1} est une concrétisation de σ_i par f^i. Comme PLAN retourne l'ensemble des solutions acycliques, chacune des solutions σ^i peut être retournée à chaque niveau i (éventuellement en effectuant des retours arrières). Il est donc bien possible pour ABPLAN de retourner la solution σ^0. ABPLAN est donc bien complet. □

L'USP est une propriété qui a été énoncée dans le contexte d'ABSTRIPS, où l'abstraction s'effectue en supprimant des préconditions dans les opérateurs. Nous avons déjà vu que ce procédé ne correspond pas à la définition que nous avons donné ici de l'abstraction. Mais l'USP (ainsi que l'extension correspondant à la proposition 1) étant essentielle au fonctionnement des algorithmes présentés, nous avons tenu à en démontrer l'application aux hiérarchies d'abstraction telles que nous les avons définies dans ce manuscrit.

Une autre propriété importante des abstractions, nommé *Downward Refinement Property* (DRP) est introduite par Qiang Yang et Fahiem Bacchus [5]. Celle-ci a aussi été énoncée dans le cadre d'ABSRIPS, mais reste à nouveau tout à fait pertinente vis-à-vis de notre contexte. Cette propriété exprime le phénomène opposé de la proposition 1, et énonce donc que toute solution à un problème abstrait est concrétisable en une solution d'un problème plus concret. La définition suivante introduit directement la DRP au niveau d'une hiérarchie d'abstractions.

Définition 28 (DRP)**.** *Soient un entier $n > 0$, un problème de planification classique P^0 et $H^n_{P^0} = (P^i, f^i)_{i\in[\![1,n]\!]}$ une hiérarchie d'abstractions de P^0 de taille n. On dit que $H^n_{P^0}$ satisfait la DRP si et seulement si pour tout $i \in [\![1, n]\!]$, pour toute solution de $\sigma^i \in A^{i*}$ de P^i, il existe une solution de P^{i-1} qui est une concrétisation de σ^i par f^i.*

La DRP est particulièrement utile pour les procédures de planification qui, comme ABPLAN, se contentent de retourner une solution exécutable (et pas nécessairement optimale). Pour ces procédures, elle garantit que la concrétisation d'une solution abstraite vers la solution concrète s'effectue sans retour arrière entre les niveaux d'abstraction, ce qui permet de réduire la complexité du problème. Toutefois la DRP est une contrainte très forte sur la structure du problème de planification, et tous les problèmes de planification ne se prêtent pas à une décomposition hiérarchique qui la satisfait. Pour contourner ce problème, Bacchus et Yang propose une procédure pour approximer cette propriété (qu'ils appellent alors *near-DRP*). Ce générateur de hiérarchie, nommé HIGHPOINT [6], associe les niveaux de criticités aux symboles de prédicats de façon à maximiser la probabilité qu'un plan abstrait soit concrétisable au niveau inférieur, et de façon à ce que cette probabilité augmente à mesure que l'on se rapproche du niveau concret.

La DRP est donc un indicateur de la qualité d'une hiérarchie d'abstraction par rapport à sa capacité à limiter les retours arrières entre les niveaux d'abstractions. Un autre

indicateur est le niveau d'indépendance des sous-problèmes. Les sous-problèmes sont indépendants lorsque l'exécutabilité de la solution d'un premier sous-problème n'a pas d'influence sur l'exécutabilité de la solution du suivant. Dans ce cas, la complexité de la résolution est à nouveau fortement réduite. En effet dans le pire des cas, la complexité est égale à la somme des complexités de chaque sous-problème, au lieu du produit entre chacun d'entre eux.

Craig Knoblock propose une évaluation de la réduction de la complexité qui permet d'appréhender l'efficacité de la planification par abstraction lorsque la hiérarchie utilisée combine à la fois la DRP et l'indépendance des sous-problèmes [55]. En considérant un facteur de branchement moyen b, et une longueur du plan final l, il montre que la complexité est réduite de $O(b^l)$ (cas correspondant à une recherche sans abstraction) à $O(lb^{\sqrt[n]{l}})$ (n étant la taille de la hiérarchie d'abstractions considérée). Cette estimation ne prend par ailleurs pas en compte la réduction du facteur de branchement induite par l'abstraction, et correspond donc à une sous-estimation du gain réel apportée par les hiérarchie d'abstraction, quand elles vérifient ces deux propriétés idéales.

2.4 Conclusion

Dans ce chapitre, nous avons décrit les principes généraux de la planification d'actions classique et avons introduit ADL le langage de planification sur lequel les travaux présentés dans ce manuscrit vont s'appuyer. Nous avons aussi passé en revue les deux principaux formalismes de planification hiérarchiques appliqués à la planification classique. Le premier est la planification HTN, qui repose sur le principe d'une hiérarchie de tâches décomposables, et le second correspond à la planification par hiérarchie d'abstractions, dont le principe repose sur une hiérarchie d'espaces de transitions abstraits.

En ce qui concerne le langage de planification, nous nous posions la question suivante : pourquoi s'appuyer sur le langage ADL plutôt qu'un autre langage de planification plus simple (comme STRIPS) ? Nous avons montré que la modélisation d'actions pour une unité militaire pouvait exiger de formuler des expressions logiques mentionnant les liens de subordination entre les unités, afin de vérifier des propriétés pour un ensemble de subordonnés, et que ces formules s'exprimaient plus aisément en utilisant des quantificateurs logiques. Or, seul ADL apporte ce niveau d'expressivité en planification d'actions.

Nous souhaitions aussi justifier le choix du modèle de planification employé, ce qui nous a amené à formuler la question suivante : en quoi la planification HTN est-elle adéquate pour la modélisation de problèmes appliqués aux unités militaires ? Ainsi, nous avons montré que la planification HTN est particulièrement adaptée à la modélisation de chaines de commandement, car elle permet de modéliser la transmission des ordres, ainsi que la coordination des unités, sous la forme de tâches décomposables. Elle a de plus déjà été appliquée avec succès pour l'animation de soldats dans les jeux sérieux ainsi que dans les jeux vidéos avec le planificateurs SHOP. En conséquence, les travaux de cette thèse se sont appuyés sur ce système de planification, et le chapitre suivant apporte une description de notre propre implémentation de ce planificateur, ainsi que de l'architecture globale dans lequel celui-ci s'insère.

Enfin, notre dernière question concernait la technique de planification vers laquelle nous nous sommes orientés : en quoi la planification par abstraction dispose-t-elle d'un potentiel pour résoudre efficacement les problèmes de planification d'actions appliqués aux unités militaires ? Nous avons ainsi montré que les hiérarchies d'abstractions permettent de modéliser un aspect complémentaire du principe de décision utilisé par une chaine de commandement, à savoir l'utilisation de différents niveaux de représentation de l'environnement pour raisonner sur la faisabilité et la qualité des plans. À cet effet, les travaux de cette

thèse ont été orientés vers la combinaison de la planification HTN avec la planification par abstraction.

Chapitre 3

Système de planification en ligne pour les jeux vidéo

Ce chapitre décrit un système de planification développé pour la simulation tactique, et dans lequel s'insèrent les travaux sur la planification hiérarchique décrits dans ce manuscrit.

La première partie décrit une architecture de planification en ligne. Nous tâcherons alors de répondre à la question suivante : en quoi l'architecture proposée est-elle adaptée à la planification en ligne dans un environnement temps réel de type jeu vidéo ? Nous montrerons alors que les différents composants de cette architecture permettent de réagir à l'évolution dynamique de l'environnement de simulation.

La seconde partie porte plus particulièrement sur le composant de planification tactique et introduit SHPE (*Simple Hierarchical Planning Engine*), un planificateur HTN inspiré de SHOP, ainsi qu'un langage de planification HTN baptisé HTDL. SHPE est fondé sur une technique de pré-compilation des éléments de planification décrits en code C++ statique, afin de minimiser son temps d'exécution. Nous verrons alors pourquoi l'implémentation de SHPE en fait un système de planification adapté aux contraintes posées par le fonctionnement d'un jeu vidéo.

Enfin, nous nous poserons la question des performances de ce système : celles-ci sont-elles suffisantes pour retourner des plans en ligne permettant d'animer une section entière ? Afin de répondre à cette question, SHPE est évalué sur un domaine de planification HTN représentatif des tâches à planifier dans un jeu vidéo, et ses performances sont comparées avec JSHOP2, un planificateur HTN qui implémente le même algorithme que SHPE, mais qui n'a pas été conçu spécifiquement pour une application aux jeux vidéo.

Le contenu de cette section a fait l'objet d'une publication au workshop « Planning in Game » lors de la conférence ICAPS 2013 [65], ainsi que d'une publication au workshop « Computer Games » qui s'est déroulé lors de la conférence ECAI 2014 [66].

3.1 Architecture de planification réactive

Une vue d'ensemble de l'architecture de planification et de contrôle de l'exécution est illustrée en figure 3.1. Elle présente deux niveaux différents de planification. Au niveau supérieur, le gestionnaire de mission résout le problème de planification à partir de l'échelon de la brigade et jusqu'à la section d'infanterie. En dessous de ce niveau, la génération de plans tactiques pour les groupes de combat et soldats est confiée à un planificateur HTN. Ce composant central détermine la qualité du comportement des soldats virtuels, et est détaillé dans une section dédiée.

L'architecture de planification est conçue pour être intégrée dans un simulateur temps réel, en conséquence il n'est pas possible de consommer trop de puissance de calcul au

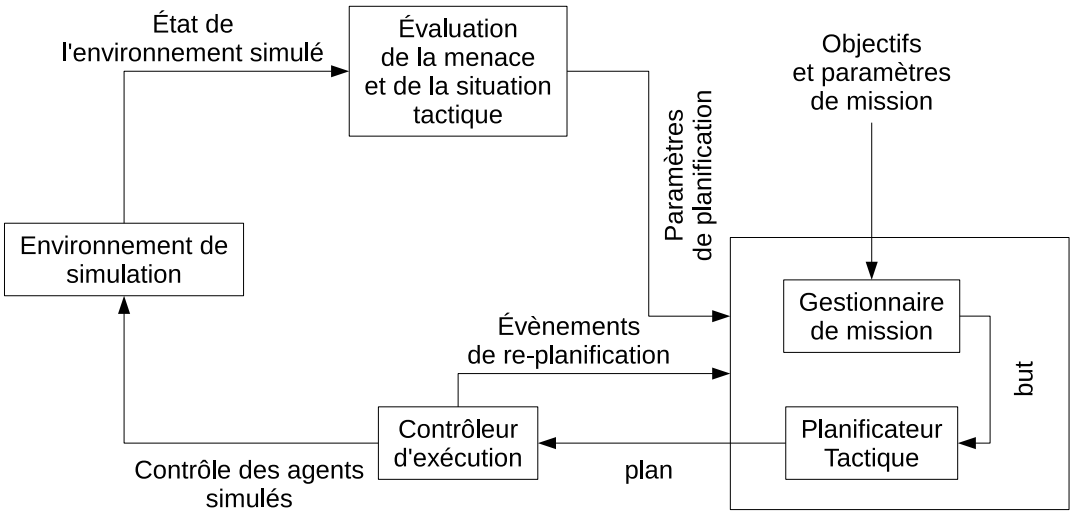


FIGURE 3.1 – Architecture de planification globale pour jeu sérieux

niveau de la planification, au détriment du nombre d'images par seconde, de la jouabilité ou du rendu graphique. De plus, l'architecture doit permettre de gérer plusieurs douzaines d'entités en parallèle, potentiellement en traitant plusieurs requêtes de planification simultanément.

Les deux planificateurs prennent en entrée l'état synthétisé par le composant d'évaluation de la menace et de la situation tactique, qui fusionne les données extraites de l'environnement de simulation. Le contrôleur d'exécution prend en charge la réalisation des plans d'actions au cœur du simulateur, et émet un évènement de replanification à chaque fois qu'une séquence d'actions ne peut être correctement exécutée dans l'environnement de simulation.

Le gestionnaire de mission, l'évaluation de la menace et de la situation tactique, ainsi que le contrôleur d'exécution sont des composants logiciels fondamentaux pour la conception de cette architecture. Toutefois, la description détaillée de leur fonctionnement n'est pas l'objet de ce manuscrit.

3.1.1 Gestionnaire de mission

Le gestionnaire de mission combine la planification de mission, l'ordonnancement, et la décomposition des plans pour les unités élémentaires. Étant donné les conditions initiales (provenant à la fois unités amies et ennemies), la planification et l'ordonnancement (P&O) définissent l'enchainement des actions permettant d'accomplir les objectifs de la mission. La P&O prennent en considération la structure du terrain, les capacités des unités et leur coordination :

- La représentation du terrain inclut les axes d'approche, les points d'observation, les couverts et les masques.
- Les capacités des unités correspondent à leur mobilité, à leur capacité d'engagement, de communication et d'observation.
- La coordination des actions est nécessaire dans le temps et dans l'espace pour l'appui mutuel, ou pour obtenir un effet particulier sur l'ennemi.

Le problème résolu par la planification et l'ordonnancement au niveau du gestionnaire de mission est de trouver, pour chaque unité, un enchainement d'actions et de déplacements (c'est-à-dire un plan) associé à un ensemble de dates, de la position initiale jusqu'à l'objectif. Le problème de planification de mission peut être modélisé et résolu d'après une approche reposant sur la programmation par contraintes [32].

Le gestionnaire de mission doit aussi prendre en compte la décomposition des plans à travers la chaine de commandement tactique. Chaque niveau hiérarchique définit un ordre d'opération (OPORD), à partir de ces propres objectifs de mission. L'OPORD

affecte et coordonne les tâches des unités de l'échelon inférieur et répartit les ressources. Cette partie n'est pas encore automatisée, dans la mesure où de nombreux paramètres doivent être simultanément pris en compte (organisation, logistique, estimation de la situation ennemie et de son mode d'action, procédures complexes de coordination, etc.). Cependant, il est possible de structurer les différents OPORD en utilisant à la fois les résultats du planificateur de mission et de l'organisation des unités.

Lorsqu'un évènement majeur se produit durant l'exécution de la mission, une replanification globale peut s'avérer nécessaire. Cela entraine deux impacts :

— Un nouveau problème est transmis au planificateur. Des techniques de réparation de plan ou de recherche locale peuvent alors être utilisées pour déterminer un plan répondant à la nouvelle situation.
— L'OPORD est mis à jour via l'émission d'un ordre de conduite (un FRAGO, pour *FRAGmentary Order* en anglais).

3.1.2 Contrôleur d'exécution

Le contrôleur d'exécution exécute les actions fournies par le planificateur tactique (le planificateur HTN). Il interagit de plusieurs façons avec l'environnement de simulation en contrôlant les soldats virtuels. Pour une action donnée, il détermine le déplacement de chaque soldat, son orientation et sa position. Via le simulateur, le contrôleur peut effectuer des requêtes pour obtenir un point de passage à atteindre, l'intervisibilité entre deux zones, ou pour identifier les menaces dans son champs de vision. Le contrôleur évalue le succès d'une action d'après la logique suivante :

— Les préconditions sont vérifiées. Celles-ci incluent la mobilité (points de passage), l'achèvement des actions précédentes, l'observation des résultats, ou des évènements se produisant dans l'environnement.
— Les requêtes émises au simulateur sont traitées avec succès (par exemple, le planificateur d'itinéraire à bien été capable de trouver un point de passage).
— La coordination dans le temps et l'espace peut être satisfaite. Cette coordination peut être définie par le gestionnaire de mission aussi bien que par des actions collaboratives (par exemple pour un appui mutuel ou un déplacement en formation).

À chaque fois qu'une action ne peut être achevée avec succès, un évènement de replanification est émis.

3.1.3 Perception de l'environnement

L'estimation de la situation tactique est maintenue par un système de *Red Force Tracking* (RFT) directement inspiré des systèmes d'acquisition et de poursuite de cible [93]. Le service de RFT prend en charge la gestion des rapports d'observation, la désignation des cibles, l'estimation des positions à court terme, ainsi que la prédiction du plan d'actions ennemi sur le long terme. Des algorithmes de fusion de données tels que les filtres de Kalman, IMM (*Interactive Multiple Models*), ou MHT (*Multiple Hypothesis Tracker*) sont intégrés pour associer les observations, supprimer les incohérences, et pour gérer une liste de pistes actives. Les comptes rendus différés et le pistage sont aussi simulés afin de ne pas obtenir la même perception immédiate de l'environnement au niveau de deux unités différentes (à noter que dans la réalité, ces informations sont actualisées en transmettant un OPORD via la chaine de commandement).

À partir de ces données en sortie, et à chaque modification de la situation ennemie, chaque soldat ou groupe de combat évalue sa propre évaluation de la menace ennemie afin de recalculer un nouveau plan d'actions si nécessaire.

3.2 SHPE : un moteur de planification HTN

Cette section décrit SHPE (*Simple Hierarchical Planning Engine*). Par rapport à l'architecture présentée dans la section précédente, SHPE est l'implémentation du planificateur tactique. Il s'agit d'un système de planification HTN s'inscrivant dans la lignée de SHOP et SHOP2, et donc de la planification HTN simplifiée présentée dans la sous-section 2.2.3. Pour rappel, en planification HTN simplifiée, les contraintes associées aux méthodes sont exprimées exclusivement sous la forme de préconditions. Toutefois, ce système a été conçu spécifiquement pour générer des plans pour un environnement dynamique temps réel, et bénéficie d'une implémentation particulière lui permettant de traiter des requêtes de planification non triviales en quelques millisecondes.

La première sous-section décrit cette implémentation en détail. La deuxième sous-section présente le principe de précompilation qui permet d'utiliser SHPE avec un langage de modélisation de haut niveau et enfin la dernière sous-section décrit HTDL, le langage de modélisation mis au point pour être utilisé conjointement avec SHPE.

3.2.1 Description de l'implémentation

La majorité des systèmes de planification disponibles ont été influencés, lors de leur conception, par les règles de l'IPC (*International Planning Competition*). Ils sont conçus pour prendre en entrée un fichier contenant la modélisation d'un domaine de planification, ainsi qu'un fichier contenant une modélisation d'un problème associé à ce domaine. Ces deux fichiers sont modélisés via le langage PDDL [63] (*Planning Domain Definition Language*), et doivent être dynamiquement interprétés par le planificateur. L'implémentation de cette étape est donc générique, et la nécessité de pouvoir s'adapter à des domaines et des problèmes de n'importe quelle taille (différents nombres de prédicats, d'actions ou d'objets) impose d'allouer dynamiquement ces structures de données. Or, dans le cadre de l'architecture de planification en ligne présentée dans ce manuscrit, le planificateur tactique doit résoudre des problèmes qui sont toujours associés au même domaine de planification. Il peut donc s'avérer judicieux de tenter d'optimiser l'implémentation du planificateur en fonction de ce domaine, et ainsi d'en améliorer les performances.

Afin de mettre en pratique une telle approche, SHPE s'inspire de Pyhop [72]. Pyhop est une implémentation de SHOP dans le langage Python qui présente deux caractéristiques remarquables : elle est très courte (environ 150 lignes de code), et les données de planification ne sont pas toutes implémentées sous la forme de structure de données, mais sous la forme de procédures en code Python pour les actions et les méthodes, comme illustré par la figure 3.2. La première de ces caractéristiques est par ailleurs liée à la simplicité algorithmique de la planification HTN simplifiée, mais est aussi fortement renforcée par l'absence de moteur d'inférence (utilisé dans SHOP et SHOP2 pour générer les substitutions) qui est une conséquence de la seconde caractéristique. La possibilité d'implémenter les éléments de planification via un langage impératif est rendu possible par le fonctionnement des planificateurs HTN simplifiées. En effet ces planificateurs fonctionnent en représentant explicitement les états et en les « progressant » par les actions, c'est-à-dire en appliquant une action sur un état pour obtenir l'état suivant. Ainsi il est possible d'encoder la structure des états sous la forme d'une structure de données d'un langage impératif, et d'encoder les actions directement comme des fonctions prenant en entrée un état et retournant un autre état en sortie. De même, les méthodes étant associées à des préconditions, celles-ci peuvent être encodées sous forme de fonctions prenant en entrée un état et retournant une liste de décompositions en sortie.

Pyhop est mis en avant comme étant un système de planification adapté au développement des jeux vidéo, car il permet à un programmeur d'encoder les données de planification dans

```
def shoot_player(state, npc, gun):
    if state.visible[state.at[npc]][state.player_at] and \
       state.holding[npc][gun] and state.loaded[gun]:
        state.loaded[gun] = False
        state.player_wounded = True
        return state
    else: return False
```

FIGURE 3.2 – Exemple d'action encodée en Python pour Pyhop

un langage impératif « classique », et lui évite d'apprendre un langage de planification qui pourrait lui paraitre « ésotérique ». Ce n'est cependant pas cet argument qui a motivé la conception de SHPE. En effet, c'est pour améliorer les performances du planificateur que SHPE suit une approche similaire à celle de Pyhop. Puisque le domaine traité par le planificateur est connu à l'avance, il n'est donc pas nécessaire d'avoir recours à des données allouées dynamiquement, ni même à des données statiques qui devront encore être interprétées par une procédure générique non optimisée, mais il est possible de les représenter sous la forme de procédures spécialisées directement exécutables. Ainsi, la quantité d'opérations à exécuter pour chaque nœud de l'espace de recherche peut être réduite, de même que la quantité de mémoire à allouer dynamiquement (ce qui peut aussi avoir un impact sur le temps de calcul, du fait de la fragmentation de la mémoire).

Le système se présente donc, non pas comme un planificateur autonome, mais comme une bibliothèque C++ offrant un moteur de planification. Les différentes structures, classes et fonctions associées à ce moteur possèdent ainsi un paramètre « template » C++ correspondant à la structure des états, et qui doit donc être défini au préalable. Il est ainsi possible d'implémenter n'importe quelle structure de données pour représenter les états, et celle-ci peut donc être adaptée au problème traité pour minimiser la quantité de mémoire nécessaire ou accélérer l'accès aux données. Par exemple, il est ainsi possible de représenter l'état d'une partie de jeu d'échec simplement par un tableau en deux dimensions pour représenter le plateau et un type énuméré pour identifier l'état de chaque case (vide ou contenant une pièce d'un type et d'une couleur donnés). Afin d'encoder les tâches, ce moteur de planification exhibe une interface pouvant être implémentée pour définir les tâches primitives et composées du domaine. Cette interface est présentée en figure 3.3. Elle permet de définir trois méthodes pour les tâches primitives (« méthodes » dans le sens de la programmation orientée objet, c'est-à-dire de procédures attachées à une classe, et non pas dans le sens de « méthodes de décomposition ») :

- une méthode *applicable*, qui évalue si la précondition de l'action est satisfaite dans l'état courant ;
- une méthode *apply* qui applique l'action à l'état courant pour obtenir un nouvel état ;
- une méthode *cost* qui retourne le coût de cette action en fonction de l'état courant.

Pour les tâches non-primitives, cette interface propose une méthode *decompose* qui prend en entrée l'état courant et retourne un ensemble de décompositions.

L'algorithmie au cœur de SHPE correspond à la théorie de la planification HTN simplifiée et totalement ordonnée : un plan est construit en appliquant des actions à l'état initiale et ces actions sont obtenues en décomposant les tâches du réseau de tâches qui ne sont pas contraintes à être ordonnées après d'autres tâches. Lorsqu'une action ou une méthode ne peut être appliquée, le planificateur effectue un retour arrière dans un espace de recherche parcouru en profondeur d'abord. De plus, SHPE implémente la procédure de séparation et évaluation de SHOP2 permettant d'améliorer l'efficacité de la recherche

```
template <typename State>
class Task {
public:
    Task(bool primitive);
    virtual ~Task();
    bool is_primitive() const;
    virtual bool applicable(const State&) const;
    virtual void apply(State& state) const;
    virtual double cost(const State& state) const;
    virtual void decompose(
        const State& state,
        TaskNetworks<State>& task_networks
    ) const;

protected:
    bool primitive;
};
```

FIGURE 3.3 – Interface C++ pour l'implémentation des tâches dans SHPE

d'une solution optimale. Enfin, l'algorithme de planification est implémenté en suivant une structure itérative permettant de l'interrompre et de le redémarrer après un nombre donné d'itérations ou une limite de temps. Cette propriété lui permet d'être utilisé avec des moteurs de jeu qui imposent que tous les calculs s'exécutent entre deux rafraichissements de l'image de l'environnement virtuel. La procédure exécutée à chaque itération est décrite dans l'algorithme 7, tandis que les procédures pour la recherche de solutions non-optimales et optimales sont respectueusement présentées par les algorithmes 8 et 9.

3.2.2 Précompilation du domaine

L'écriture du domaine de planification en C++ pose cependant un problème. En effet, C++ est un langage « verbeux ». A titre d'exemple, la version HTN du domaine *Rover* encodée en LISP et distribuée avec la version publique de JSHOP2 contient environ 170 lignes de codes, alors que le même domaine encodé en C++ pour SHPE contient près de 800 lignes. Dans le cas de Pyhop, l'utilisation de Python au lieu de C++ limite le problème. Python étant un langage de haut niveau, un programme Python est en effet deux fois plus court en moyenne qu'un programme équivalent écrit en C++ [88]. Pour résoudre ce problème, une solution est d'avoir recours à un langage de haut-niveau, qui pourra ensuite être compilé en C++. Ce langage aurait pu être un langage impératif, comme l'est par exemple Python. Mais afin de rendre ce langage de modélisation plus synthétique encore, il a été choisi de mettre au point un langage de planification. Ce langage a été nommé HTDL, et est décrit dans la prochaine sous-section. À première vue, utiliser un langage de planification, plus abstrait qu'un langage impératif, peut apparaitre comme une contradiction avec la liberté mise en avant pour SHPE d'optimiser à bas niveau la consommation en mémoire et en temps de calcul. Cependant, le processus de compilation permet toujours de bénéficier de l'optimisation du code, obtenue en convertissant les éléments de planification en structures et procédures impératives, même si ce processus n'est plus spécialisé pour chaque domaine. De plus, dans le cas où ce niveau d'optimisation ne serait pas suffisant, la possibilité d'encoder manuellement un domaine reste disponible, en plus de la possibilité désormais de s'appuyer en premier lieu sur un code complet généré automatiquement par un compilateur.

Algorithm 7 suivant(*pile*, *meilleur_plan*, *meilleur_coût*)

```
(plan, coût, état, réseau) ← dépiler pile
si coût ≥ meilleur_coût alors
    retourner
fin si
si réseau est vide alors
    meilleur_plan ← plan
    meilleur_coût ← cost
    retourner
fin si
tâches ← toutes les tâches de réseau n'ayant pas de prédécesseur
pour tout t ∈ tâches faire
    si t est une tâche primitive alors
        si t.applicable(état) alors
            plan ← ajouter t à la fin de plan
            coût ← coût + t.cost(état)
            supprimer t de réseau
            empiler (plan, coût, t.apply(état), réseau) sur pile
        fin si
    fin si
    si t est une tâche composée alors
        pour tout tn ∈ t.decompose(état) faire
            remplacer t dans réseau par tn
            empiler (plan, cost, état, réseau) sur pile
        fin pour
    fin si
fin pour
```

Cette étape de conversion du domaine depuis un langage de planification vers un langage de bas niveau qui sera ensuite compilé pour obtenir un planificateur est appelée précompilation du domaine. Il s'agit d'un procédé qui a déjà été employé dans le contexte de la planification HTN. En effet, le système JSHOP2 utilise un procédé similaire pour compiler un planificateur à partir d'un domaine et d'un problème de planification [48]. Dans le cas de JSHOP2, la précompilation permet de limiter l'usage de l'allocation dynamique à l'exécution en fixant la taille des structures de données employées pour encoder les éléments du domaine (listes de préconditions, de sous-tâches, etc.). Le processus n'est pas tout a fait identique dans le cas de SHPE, dans la mesure où seul le domaine de planification est précompilé et qu'une partie des éléments de planification sont convertis en procédures et non pas en structures de données. Le processus complet de précompilation du domaine en C++ et de compilation du planificateur est illustré en figure 3.4.

3.2.3 HTDL : un langage de haut niveau pour la modélisation de domaines HTN

HTDL signifie *Hierarchical Task Description Language*, soit « langage de description de tâches hiérarchiques » en français. Il s'agit d'un langage permettant de modéliser des domaines de planification HTN simplifiée. Sa sémantique correspond donc à celle de ce type particulier de problème de planification HTN, tel qu'il est décrit dans la sous-section 2.2.3.

En ce qui concerne la syntaxe, HTDL a été conçu dans l'objectif d'être à la fois

Algorithm 8 trouver_premier_plan(*état*, *réseau*)

1: *pile* ← []
2: *meilleur_plan* ← []
3: *meilleur_coût* ← ∞
4: empiler ([], 0, *état*, *réseau*) sur *pile*
5: **tant que** *meilleur_coût* = ∞ **et** *pile* n'est pas vide **faire**
6: suivant(*pile*, *meilleur_plan*, *meilleur_coût*)
7: **fin tant que**

Algorithm 9 trouver_meilleur_plan(*état*, *réseau*)

1: *pile* ← []
2: *meilleur_plan* ← []
3: *meilleur_coût* ← ∞
4: empiler ([], 0, *état*, *réseau*) sur *pile*
5: **tant que** *pile* n'est pas vide **faire**
6: suivant(*pile*, *meilleur_plan*, *meilleur_coût*)
7: **fin tant que**

synthétique et lisible. Pour cela, HTDL s'inspire largement de la syntaxe du langage Python, par exemple en forçant l'usage de l'indentation pour identifier les blocs constituant les descriptions des éléments de planification. Cette syntaxe est illustrée dans les paragraphes qui suivent à l'aide d'exemples, et une description plus formelle, utilisant la forme de Backus-Naur étendue, est disponible en annexe A.

Symboles

HTDL distingue différents types de symboles pour sa syntaxe. Il dispose tout d'abord d'un ensemble de mots réservés pour représenter les connecteurs logiques (`not`, `and`, `or`, `imply`, `iff`, `exists`, `forall`), des structures du langage (`task`, `if`, `precond`, etc.) ou des opérateurs (`+`, `*`, `==`, `<`, etc.). Ensuite, les symboles représentant les variables sont constitués de caractères alpha-numériques et du symbole « `_` », doivent commencer par une lettre minuscule, et peuvent se terminer par un ou plusieurs signes prime (exemple : `x`, `x'`, `soldier`, `teamAlpha`, `team_bravo`, `squad2`, etc.). Les symboles utilisés pour nommer les fonctions, les prédicats et les tâches sont seulement constitués de caractères alpha-numériques, et doivent commencer par une lettre majuscule (exemple : `Leader`, `AtArea`, `Move`, etc.). Enfin des nombres entiers et décimaux peuvent être utilisés pour représenter des constantes numériques (exemple : `0`, `-7`, `152`, `3.14`, etc.).

Termes

Suivant les règles définissant la logique du premier ordre, les termes sont constitués des variables, des constantes numériques et des fonctions. Les fonctions sont constituées d'un symbole de fonction (décrit au paragraphe précédent) et d'une liste de termes, placés entre parenthèses et séparés par des virgules (exemple : `Distance(area1, Area(soldier1))`). Les fonctions d'arité nulle (appelées aussi constantes) peuvent être représentées avec des parenthèses vides, ou sans parenthèse (exemple : `MaxDistance()` ou `MaxDistance`).

Un certain nombre de constantes et de fonctions sont prédéfinies dans le langage. C'est par exemple le cas de la constante `Undefined` et des fonctions `Min` et `Max` d'arité 2, et qui sont interprétées en faisant appel à des procédures. D'autres fonctions de ce type sont intégrées au langage, mais sont appelées via les opérateurs numériques `+`, `-`,

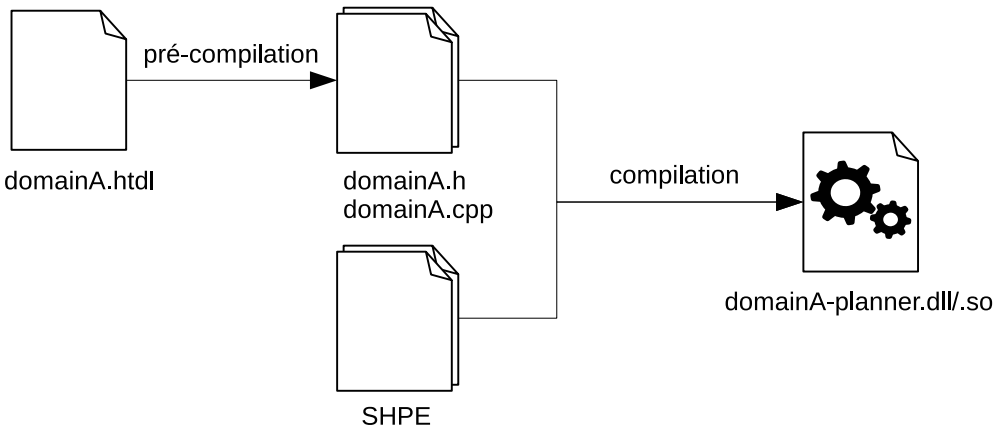


FIGURE 3.4 – Précompilation de domaine
Précompilation d'un domaine de haut-niveau depuis le langage HTDL en langage C++, suivie de la compilation vers le planificateur final.

`*`, `/` et `**`. Ces fonctions sont aussi d'arité 2 mais s'utilisent sans parenthèse (exemple : `Speed(vehicle) * Distance(route)`).

Formules logiques

Les formules logiques sont constituées des formules atomiques, des conjonctions, disjonctions, des négations, implications, équivalences, et des formules quantifiées universellement et existentiellement. De plus une formule logique placée entre parenthèses est aussi une formule logique.

Les formules atomiques sont constituées de la même façon que les fonctions, c'est-à-dire par un symbole de prédicat suivi par une liste de termes entre parenthèse, séparée par des virgules. Les formules atomiques d'arité 0 (les propositions) peuvent être représentées avec ou sans parenthèse, et les opérateur logiques `==`, `!=` (différent), `<`, `<=`, `>` et `>=` sont prédéfinis et à utiliser sans parenthèse. Exemple :

```
InFireteam(soldier1, TeamAlpha)

Ammunition(weapon1) == 12

Distance(area1, area2) <= MaxDistance

MissionAccomplished
```

Les conjonctions sont formées par deux formules logiques séparées par le mot clé « `and` » ou par un saut de ligne. L'opérateur « `and` » étant associatif, il est possible d'enchainer les conjonctions. Exemple :

```
Soldier(soldier1) and Weapon(weapon1)
Holding(soldier1, weapon1) and Loaded(weapon1) and Antitank(weapon1)
```

Les disjonctions sont formées par deux formules logiques séparées par le mot clé « `or` ». À nouveau, cet opérateur étant associatif, on pourra enchainer les disjonctions. Exemple :

```
NoEnemyAt(area) or AllEnemyNeutralizedAt(area)
```

Les négations sont formées en faisant précéder une formule logique par le mot clé « `not` ». Exemples :

```
not (Neutralized(soldier) or Wounded(soldier))

not Neutralized(soldier) and not Wounded(sodlier)
```

Les implications et les équivalences sont formées par deux formules logiques séparées respectivement par les mots clés « `imply` » et « `iff` ». Exemples :

```
NoEnemyAt(area) or AllEnemyNeutralizedAt(area) implies Safe(area)

AtArea(soldier, area) iff InArea(Position(soldier), area)
```

Les formules universellement quantifiées sont formées à partir du mot clé « `forall` », suivi d'une liste de variables, du mot clé « `if` », d'une première formule logique permettant de contraindre ces variables et enfin du symbole « `:` » suivi de la formule logique quantifiée, ou d'un nouveau bloc contenant cette formule. Exemples :

```
forall soldier if Soldier(soldier): Available(soldier)

forall soldier, weapon if Holding(soldier, weapon):
    Loaded(weapon)
    Available(soldier)
```

Enfin, les formules avec quantificateurs existentiels sont formées à partir du mot clé « `exists` » suivi d'une liste de variables, du mot clé « `:` » et d'une formule logique, ou d'un nouveau bloc contenant cette formule. Exemples :

```
exists soldier, weapon: Holding(soldier, weapon) and Loaded(weapon)

exists soldier, area, route:
    Area(area1) and Area(area2) and Route(route)
    From(route) == AtArea(soldier) and To(route) == area
    Distance(route) < 10
```

Effets

Les effets sont regroupés dans trois types de blocs : « `add` » et « `delete` » pour les prédicats et « `update` » pour les fonctions. Les effets simples sont exprimés par des formules atomiques pour les prédicats ou par des affectations pour les fonctions. Les affectations sont des expressions formées d'un terme fonctionnel suivi de l'opérateur « `=` » et d'un second terme. Un effet conditionnel est formé d'un effet simple suivi du mot clé « `if` » suivi d'une formule logique. Enfin, un effet quantifié est formé d'un effet simple suivi du mot clé « `forall` », suivi d'une liste de variables séparées par des virgules et suivie du mot clé « `if` » et d'une formule logique contenant les variables quantifiées. Exemples :

```
add:
    FireteamAt(fireteam, area_to)
    At(soldier, area_to) forall soldier if InFireteam(soldier, fireteam)
delete:
    FireteamAt(fireteam, area_from)
    At(soldier, area_from) forall soldier if InFireteam(soldier, fireteam)
update:
    Ammunition(fireteam) = Low if Ammunition(fireteam) == High
```

Méthodes

Les méthodes sont des blocs formés par le mot clé « `method` », d'une liste optionnelle de variables séparées par des virgules, du mot clé « : » et de trois sous-blocs exprimant les préconditions, les sous-tâches et les contraintes d'ordre (chaque bloc étant optionnel). Exemple :

```
method soldier1, soldier2, soldier3:
    precond:
        InFireteam(soldier1, fireteam)
        InFireteam(soldier2, fireteam)
        InFireteam(soldier3, fireteam)
        soldier1 != soldier2 and soldier2 != soldier3 and soldier3 != soldier1
    subtasks:
        n1 : Cover(soldier1, route)
        n2 : Move(soldier2, route)
        n3 : Move(sodlier3, route)
    order:
        n1 < n2, n1 < n3
```

Tâches

Toutes les tâches sont formées par le mot clé « `task` » suivi d'un symbole de tâche, d'une liste de variables placées entre parenthèses et séparées par des virgules (les parenthèses sont optionnelles si cette liste est vide), du mot clé « : » suivi par plusieurs blocs. De la nature de ces blocs dépend le type de la tâche (primitive ou composée).

Les tâches primitives peuvent contenir un bloc de préconditions, des blocs d'effets de différents types (ajout, retrait et mise à jour) ainsi qu'un bloc de coût. Ce bloc de coût contient un terme devant être de type numérique. Exemple :

```
task SoldierMove(soldier, route):
    precond:
        Soldier(soldier) and Route(route)
        At(soldier, From(route))
    add:
        At(soldier, To(route))
    delete:
        At(soldier, From(route))
    cost:
        Distance(route)
```

Les tâches composées ne peuvent contenir que des blocs correspondant aux méthodes de décompositions. Exemple :

```
task SoldierAchieveAt(soldier, area):
    method:
        precond:
            At(soldier, area)
    method route:
        precond:
            Route(route) and From(route) != area
            At(soldier, From(route))
        subtasks:
            n1 : SoldierMove(soldier, route)
```

```
            n2 : SoldierAchieveAt(soldier, area)
        order:
            n1 < n2
```

Axiomes

Certains planificateurs, tels SHOP ou UCPOP supportent la modélisation de « prédicats dérivés ». Par l'intermédiaire d'axiomes, les valeurs de ces prédicats dérivés sont liées à celles d'autres prédicats de base ou dérivés eux-mêmes. Ces prédicats dérivés apportent davantage d'expressivité à un langage de planification [106]. En effet, ils permettent par exemple d'exprimer des conditions complexes plus simplement en les modélisant hiérarchiquement, ou de déduire logiquement les conséquences de certaines actions, sans avoir à recourir à des effets complexes ou à l'introduction d'actions supplémentaires. Dans la modélisation d'un domaine traitant d'unités militaires hiérarchisées, ces axiomes permettent ainsi d'exprimer des conditions aux niveaux hiérarchiques supérieurs en fonction des conditions aux niveaux subordonnés.

HTDL intègre donc ces axiomes afin d'exprimer des prédicats ainsi que des fonctions dérivées. La syntaxe de ces axiomes est identique à celle des effets. Exemples :

```
FireteamAt(fireteam, area) forall fireteam, area if \
    Fireteam(fireteam) and Area(area)
    forall soldier if InFireteam(soldier, fireteam):
        SoldierAt(soldier, area)

FireteamAmmo(fireteam) = \
    SoldierAmmo(soldier1) + SoldierAmmo(soldier2) + SoldierAmmo(soldier3) \
        forall fireteam, soldier1, soldier2, soldier3 if \
            InFireteam(soldier1, fireteam)
            InFireteam(soldier2, fireteam)
            InFireteam(soldier3, fireteam)
            soldier1 != soldier2
            soldier2 != soldier3
            soldier3 != soldier1
```

Domaine

Un domaine HTDL est simplement une liste d'axiomes et de tâches. HTDL ne permet pas de décrire des problèmes de planification. En effet, le but de HTDL est seulement de permettre la précompilation du domaine HTN. Les problèmes HTN doivent donc être encodés directement en C++. Des exemples de domaines HTN encodés en HTDL sont disponibles en annexe C (ces exemples intègrent toutefois des extensions qui seront introduites dans les chapitres suivants).

3.3 Évaluation des performances de SHPE

Cette section décrit une expérience ayant été mise en œuvre afin d'évaluer les performances de SHPE et de valider l'intérêt de son implémentation particulière. La première sous-section présente le domaine utilisé pour effectuer l'expérience, la sous-section suivante décrit le protocole expérimental et la dernière présente et discute les résultats.

3.3.1 *SimpleFPS* : un domaine de planification pour l'animation d'un PNJ

SimpleFPS est un domaine de planification PDDL permettant de modéliser des problèmes de planification proches de ceux posés par certains jeux vidéo [110]. Ce domaine s'inspire des mécanismes de planification introduits dans le jeu F.E.A.R. Il permet de modéliser un environnement dans lequel un personnage non joueur (PNJ, mais aussi NPC en anglais, pour *Non Playable Character*) doit patrouiller afin de trouver et attaquer le personnage contrôlé par le joueur. Chaque état inclut une description de l'environnement sous la forme d'un ensemble de zones connectées entre elles par des points de passages. Ces points de passages peuvent être bloqués et rendus accessibles grâce à certains objets (par exemple de simples clés). Les zones contiennent aussi d'autres points d'intérêts entre lesquels il est possible de naviguer librement : armes, munitions, kits de soin, interrupteurs (pour éclairer une zone ou la plonger dans l'obscurité), ou encore la position du joueur à attaquer. Un exemple d'état pour un problème de planification correspondant à SimpleFPS est illustré en figure 3.5. Parmi les actions disponibles, le PNJ peut ramasser les objets et s'en servir afin d'attaquer le joueur, de se soigner, de déverrouiller des points de passage ou de recharger une arme. Afin d'y ramasser un objet ou d'y attaquer le joueur, le PNJ peut aussi modifier l'éclairage d'une zone. Enfin il peut se déplacer entre les zones et les points d'intérêts.

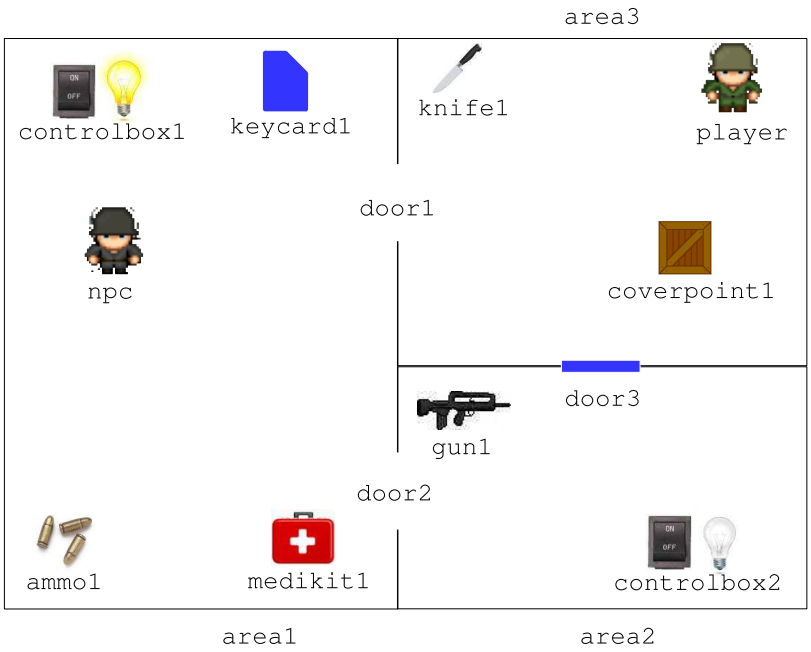


FIGURE 3.5 – Une représentation d'une situation pour un problème du domaine *SimpleFPS*.

SimpleFPS a été choisi pour cette expérience pour plusieurs raisons. Tout d'abord, il s'agit à notre connaissance du seul domaine de planification disponible dans la littérature qui permette d'évaluer des tâches de planification dont le but est d'animer un « soldat » virtuel. Ainsi ce domaine permet au moins d'évaluer le temps de calcul nécessaire pour prendre une décision au niveau d'un soldat. De plus, il ne s'agit pas d'un domaine de planification « trivial », car la résolution des problèmes proposés doit faire face à de nombreux points de choix : modes d'action pour attaquer le joueur, objets à utiliser, itinéraires pour se déplacer entre les zones, etc.

La version HTN du domaine utilisée pour cette expérience a été développée en s'appuyant sur la version PDDL disponible. Elle intègre donc à la fois les prédicats et les actions du

domaine d'origine. La structure HTN ajoutée suit quant à elle les règles de la planification HTN simplifiée, et intègre certains éléments propres au langage de planification de SHOP2, tels que les branches des méthodes, les préconditions de type « *first* », qui permettent de ne tester qu'une seule instanciation de variable valide, et les préconditions de type « *sort-by* » qui permettent d'instancier les variables suivant un ordre particulier. Ces dernières préconditions sont par exemple utilisées pour améliorer l'efficacité de la navigation en explorant en priorité les zones les plus proches de la destination à atteindre. Pour cela, la distance entre chaque pair de zones (en nombre de zones intermédiaires à parcourir) est encodée dans chaque problème sous la forme d'un prédicat supplémentaire. La description complète de ce domaine dans le langage de planification de SHOP2 est disponible en annexe B.

3.3.2 Protocole expérimental

L'expérience mise en œuvre consiste à comparer les performances de SHPE avec celles de JSHOP2. Ces deux systèmes de planification partagent les mêmes principes algorithmiques : les domaines de planification sont de type « HTN simplifié » et la recherche de solution se fait en profondeur d'abord dans l'arbre des solutions partielles, combinée à une étape de « séparation et évaluation » pour la recherche de solution optimale, l'évaluation se fondant sur le coût du plan partiel. Ces deux systèmes diffèrent donc par leur implémentation. Tout d'abord par le langage de programmation utilisé : avec Java pour JSHOP2 et C++ pour SHPE. Mais aussi via le mécanisme de précompilation : avec JSHOP2, ce sont à la fois le domaine et le problème qui sont précompilés tandis que pour SHPE seul le domaine l'est. De plus, le mécanisme de précompilation est différent : SHPE convertit tous les éléments de planification en procédures à l'échelle de chaque tâche, tandis que pour JSHOP2, chaque élément du domaine et du problème est converti en structures de données statiques. Pour JSHOP2, le domaine de planification utilisé est celui présenté ci-dessus. Dans le cas de SHPE, une version équivalente de ce domaine a été encodée manuellement et utilise l'interface de programmation de SHPE afin de compiler le planificateur.

Le banc d'essai mis en place pour comparer les performances de ces deux planificateurs est constitué de six séries de problèmes issus du domaine *SimpleFPS*. Chaque série contient une centaine d'instances résolubles générées aléatoirement. Le nombre de points d'intérêt (armes, munitions, kits de soin, points de couverture, etc.) de chaque instance varie entre 100 et 350 avec un pas de 50 et ces points d'intérêt sont répartis aléatoirement entre les zones. Le nombre de zones est fixe (10 zones), mais la structure du graphe d'interconnexion entre ces zones est aussi générée aléatoirement.

Enfin, la machine utilisée pour exécuter cette expérience est équipée d'un processeur Intel core i5 CPU (2.66 GHz), de 4 GB de mémoire RAM, et tourne sous une version 64 bits de Debian 7.0. Cette configuration est équivalente à une machine de moyenne gamme adaptée au jeu vidéo dans le grand public.

3.3.3 Résultats

Les résultats de l'expérience sont présentés graphiquement en figure 3.6. Pour chaque série, ils représentent le temps moyen nécessaire pour retourner une solution. On peut voir qu'en moyenne SHPE est capable de résoudre les problèmes de planification posés en dix à quinze fois moins de temps que JSHOP2. De plus, ces résultats montrent que le temps d'exécution nécessaire atteint l'ordre de la milliseconde, ce qui permet de planifier en ligne pour une section composée d'une trentaine d'entités simultanément, soit l'effectif d'une section d'infanterie.

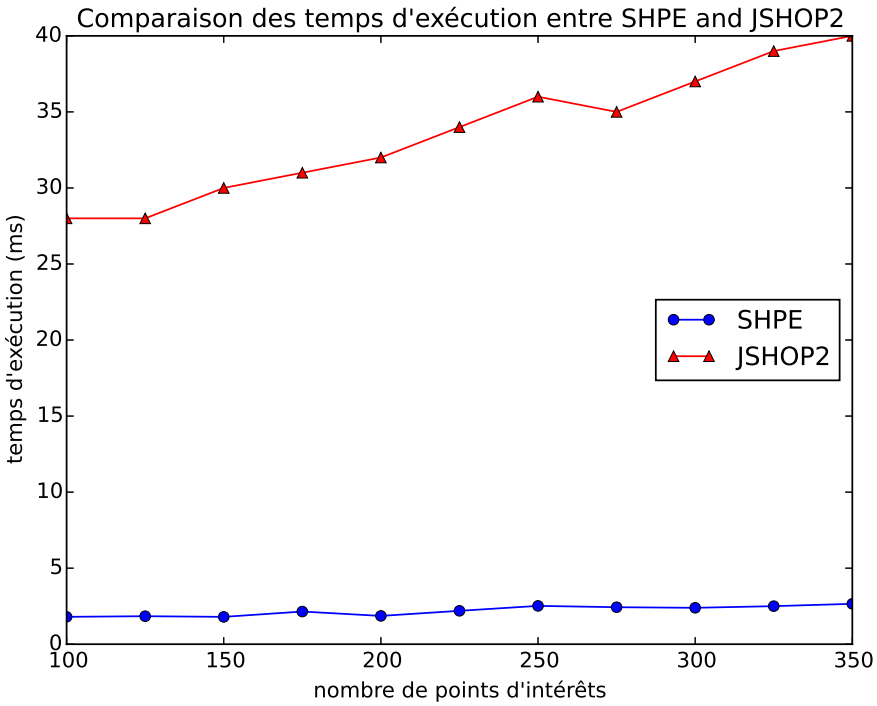


FIGURE 3.6 – Comparaison des performances de SHPE et JSHOP2
Comparaison des performances de SHPE et JSHOP2 pour des problèmes de différentes tailles avec le domaine SimpleFPS. Les instances générées contiennent dix zones et un nombre variable de points d'intérêts. Pour ces scénarios, SHPE retourne une solution au moins dix fois plus rapidement que JSHOP2.

Cependant, ces résultats n'indiquent rien sur la qualité des solutions. JSHOP2 n'étant pas suffisamment rapide pour permettre de planifier en ligne, il n'est pas apparu nécessaire de s'intéresser à la qualité des solutions qu'il pouvait retourner. Pour SHPE, une expérience supplémentaire a toutefois été conduite pour évaluer la capacité de ce système à retourner des solutions de bonne qualité rapidement. Cette expérience a été conduite uniquement sur la première série (celle avec 100 points d'intérêt), et en retournant pour chaque problème le meilleur plan obtenu au bout de 1000 secondes. Ce plan sert alors de référence pour déterminer le temps de calcul nécessaire pour retourner des plans dont le coût est 50 % puis 25 % plus coûteux. Aucune garantie ne prouve que le plan de référence est la solution optimale. Les résultats obtenus sont donc de nature optimiste, car ils sous-estiment la durée réelle nécessaire pour obtenir ces plans de qualités intermédiaires.

Les résultats sont donnés dans le tableau 3.1. Ainsi en moyenne, il faut au moins 1.6 secondes pour retourner une solution 50 % plus coûteuse que la meilleure solution, ce qui pourrait encore permettre de planifier en ligne pour une poignée de soldats. Toutefois il faut au moins 44 secondes pour obtenir une solution de bonne qualité, soit au plus 25 % plus coûteuse que la solution de référence, et cette durée n'est pas compatible avec la planification en ligne.

3.4 Conclusion

La première question traitée dans ce chapitre est la suivante : en quoi l'architecture de planification que nous avons proposée est-elle adaptée à la planification en ligne dans un environnement temps réel de type jeu vidéo ? L'architecture proposée permet de conserver une représentation à jour de l'environnement grâce à un composant assurant la perception

première solution	solution à 50 %	solution à 25 %	solution de référence
3.84 ms	1.63 s	44.80 s	248.24 s

TABLE 3.1 – Temps de calculs pour retourner une solution sous-optimale
Temps de calcul moyens obtenus pour la première solution retournée, puis pour des solutions de coûts 50 % puis 25 % plus élevés que celui de la meilleure solution retournée au bout de 1000 secondes. Ces résultats correspondent à une série de problèmes générés aléatoirement avec 100 points d'intérêt.

de l'environnement et de la situation tactique. Le composant de contrôle de l'exécution proposé permet ensuite la détection des évènements qui requièrent une adaptation du plan actuel. Enfin, en proposant une séparation du module de planification en un planificateur de missions à long terme et un planificateur tactique à court terme, il devient possible de mettre à jour les plans tactiques en continue sans remettre en cause les objectifs principaux qui doivent être atteints pour accomplir une mission.

On a ensuite présenté le planificateur responsable des plans d'actions tactiques, appelé SHPE, ce qui nous permet de répondre à la question suivante : pourquoi l'implémentation de SHPE en fait un système de planification adapté aux contraintes posées par le fonctionnement d'un jeu vidéo ? La principale contrainte à laquelle il faut faire face est le temps d'exécution disponible. Or, nous avons montré qu'en précompilant les éléments d'un domaine de planification, à partir d'une description dans un langage nommé HTDL, en structure de données statiques et sous forme procédurale, il est possible d'optimiser l'implémentation du planificateur, et ainsi d'en réduire le temps d'exécution.

Enfin, la dernière question traitée porte sur les performances de ce système, à savoir si celles-ci sont bien suffisantes pour retourner des plans en ligne permettant d'animer une section entière ? Nous avons mesuré ces performances par rapport à des problèmes de planification représentatifs des jeux vidéo, et avons constaté que ce système était en mesure de retourner un plan d'actions pour un soldat en seulement quelques millisecondes, une échelle satisfaisante pour effectuer de la planification en ligne.

Toutefois, les performances de ce système ne sont pas encore suffisantes pour lui permettre de retourner des plans de bonne qualité (c'est-à-dire avec un coût raisonnable) pour de la planification en ligne. D'autres améliorations pourraient être implémentées au niveau du procédé de précompilation, comme une gestion dédiée des allocations dynamiques via des allocateurs spécialisés. Cependant, ces améliorations ne feraient que diviser le temps de calcul par un facteur constant, or une solution vraiment efficace devrait intervenir au niveau de l'explosion combinatoire.

À cet effet, les chapitres suivants étudient les possibilités d'exploiter la hiérarchie de niveaux de décision dans l'organisation de la section d'infanterie. Ainsi, ils proposent de les interpréter comme des niveaux d'abstraction permettant de guider la recherche de solutions à plusieurs niveaux, et ainsi améliorer l'efficacité de l'algorithme de recherche.

Chapitre 4

Planification HTN avec sémantique d'exécution pour les tâches composées

Une extension possible de SHPE consisterait à modéliser les tâches composées sous la forme de transitions d'états, et ainsi d'obtenir la capacité de calculer les états intermédiaires dans un plan non primitif. Une telle extension aurait au moins deux avantages. Le premier avantage est valable pour n'importe quel planificateur HTN : il devient possible de raisonner sur l'exécutabilité des plans avant d'atteindre le niveau primitif. Des plans partiels ne menant qu'à des plans non exécutables peuvent ainsi être identifiés en amont, et être élagués de l'espace de recherche, entrainant une réduction du temps de calcul. Le deuxième avantage est propre aux planificateurs HTN fondés sur SHOP. Ces planificateurs nécessitent d'accéder à l'état courant pour évaluer une méthode de décomposition, et sont donc limités à la stratégie de décomposition qui sélectionne les tâches dans l'ordre de leur exécution. Si l'état courant pouvait aussi être accédé en appliquant les tâches composées, les mêmes mécanismes pourraient être conservés tout en permettant d'implémenter des stratégies de décomposition des tâches plus adaptées pour résoudre certains problèmes. À titre d'exemple, il deviendrait possible de différer la décomposition de certaines tâches composées, pour décomposer en priorité des tâches dont l'exécution intervient plus tard, mais qui sont plus critiques vis-à-vis de la faisabilité du plan. Un autre exemple est la prise de décision multi-niveau, qu'il devient possible d'implémenter en décomposant un plan intégralement jusqu'à un niveau de décision donné, avant de le décomposer au niveau subordonné.

Bien que pour plusieurs systèmes de planification HTN, les tâches composées soient modélisables à l'identique des actions (par exemple dans NOAH ou NONLIN), c'est-à-dire avec préconditions et effets (ou postconditions), il n'est pour autant pas possible de les interpréter comme des actions déterministes, comme c'est le cas avec les actions primitives. En effet, une « action de haut niveau » est potentiellement décomposable en plusieurs séquences d'actions primitives qui n'atteignent pas nécessairement le même état final. Dans ces conditions, considérer que cette action de haut niveau est déterministe ne serait pas cohérent avec toutes ses décompositions. Au plus, les préconditions et postconditions peuvent être interprétées comme étant des contraintes à satisfaire par chaque décomposition primitive d'une action de haut niveau [9]. AHP [60] propose aussi de décrire les tâches composées sous la forme d'actions non-déterministes. Ainsi, une action de haut niveau peut accéder à l'ensemble des états correspondant aux états finaux atteints par chacune de ses décompositions primitives. Cependant, les descriptions de telles actions non-déterministes ne sont généralement pas compactes, ce qui amène AHP à introduire une paire d'approximations optimiste et pessimiste pour chacune d'entre elles.

Ce principe nous éloigne alors de notre intention initiale, qui est de compléter SHPE, et non pas d'implémenter un système alternatif capable de traiter ce type de description.

Dans ce chapitre, nous proposons une solution alternative qui s'appuie sur l'abstraction des états. L'abstraction permet de fusionner l'ensemble des états atteignables par une action de haut niveau sous la forme d'un unique état abstrait, et ainsi d'interpréter et de décrire les tâches composées comme des actions déterministes dans un espaces d'états abstrait. Comme nous allons le voir dans ce chapitre, cette solution à l'avantage de s'intégrer aisément avec des planificateurs comme SHOP ou SHPE.

Dans la première partie, nous proposons d'appliquer le principe des hiérarchies d'abstractions à la planification HTN, en définissant certaines tâches composées comme étant des tâches primitives vis-à-vis d'un niveau d'abstraction donné. Nous traitons alors la question des avantages de cette sémantique de transitions d'états abstraits pour la planification HTN.

Dans la seconde partie, nous illustrons l'utilisation de ces effets abstraits dans un exemple appliqué au combat d'infanterie, et nous nous posons la question suivante : quel est le gain potentiel de performances que l'on peut obtenir grâce à ces effets abstraits ? Pour y répondre, nous comparons les temps d'exécution obtenus entre plusieurs stratégies de décomposition des tâches qui exploitent les effets abstraits pour vérifier l'exécutabilité des plans abstraits à des horizons différents.

Enfin en troisième partie, nous introduisons une nouvelle forme d'effets abstraits. Au lieu de reposer sur une hiérarchie d'espaces abstraits, la sémantique qui est présentée repose sur un système de transitions d'états défini à partir d'un langage du premier ordre reposant sur une logique trivaluée, où la troisième valeur exprime l'indéterminé introduit par l'abstraction. Nous verrons alors pourquoi cette définition alternative des effets abstraits permet de résoudre certains problèmes introduits par les hiérarchies d'abstractions.

4.1 Planification HTN avec hiérarchie d'abstractions

Cette première partie introduit un système de planification hiérarchique qui combine la planification HTN avec la planification par hiérarchie d'abstractions. Ce système repose sur le langage ADL, et le procédé d'abstraction employé consiste à réduire le nombre de symboles de prédicats et de fonctions entre deux niveaux d'abstraction consécutifs. Le cadre théorique est introduit dans un premier temps, suivi par la description des fonctions d'abstraction ainsi que leur intégration dans HTDL, le langage de description HTN du système SHPE. Enfin, cette partie présente un algorithme adapté à ce nouveau cadre de planification hiérarchique.

4.1.1 Sémantiques de transitions abstraites pour les tâches composées

Dans la partie 2.3, nous avons présenté les principes de la planification par hiérarchie d'abstractions appliquée à la planification classique. Nous allons désormais présenter un cadre théorique pour l'application de la planification par hiérarchie d'abstractions à la planification HTN. Pour rappel, le principe de la planification par hiérarchie d'abstractions consiste à appliquer une suite de morphismes de système de transition d'états au problème de planification posé. Les états sont alors associés à de nouveaux états abstraits, tandis que les actions abstraites sont définis de façon à conserver l'accessibilité des états dans le système de transition. Nous proposons ici d'appliquer le même principe, mais en utilisant les tâches d'un domaine de planification HTN pour définir les actions abstraites. Une tâche composée peut alors correspondre à une action dans un espace d'états abstraits, tandis que les tâches contenues dans ses décompositions peuvent correspondre à des actions par rapport à un espaces d'états plus concret. La concrétisation d'une action abstraite

s'effectue alors en appliquant les méthodes de décomposition définies dans le domaine HTN, et non plus en faisant appel à un planificateur d'actions classique.

Cependant, il n'est pas toujours désirable d'associer chaque tâche composée avec une action abstraite. Par exemple, un modélisateur peut souhaiter définir une tâche correspondant à un but à atteindre et non pas à une action à exécuter. De façon générale, il est souhaitable de pouvoir introduire des tâches intermédiaires afin d'ajouter des niveaux de décomposition entre deux niveaux de planification, sans pour autant associer ces tâches avec un système de transition d'états déterministe s'inscrivant dans une suite de niveaux d'abstraction. Chaque niveau d'abstraction contient ainsi son propre ensemble de tâches composées et d'actions, les tâches primitives étant les actions du niveau le plus concret. Chaque niveau contient donc son propre système de transition d'états, ainsi que son propre système de décomposition de tâches. Un planificateur HTN « classique » peut alors être exécuté pour résoudre un problème formulé à chaque niveau. En considérant que les actions abstraites à un niveau donné sont aussi des tâches au niveau inférieur, chaque solution abstraite peut alors y être convertie en réseau de tâches initial. En ajoutant l'état initial sous sa forme abstraite, on obtient un nouveau problème de planification au niveau inférieur, et ainsi de suite, jusqu'à parvenir au niveau primitif. La figure 4.1 propose une illustration de ce principe de décomposition des tâches à travers deux niveaux dans une hiérarchie d'abstractions.

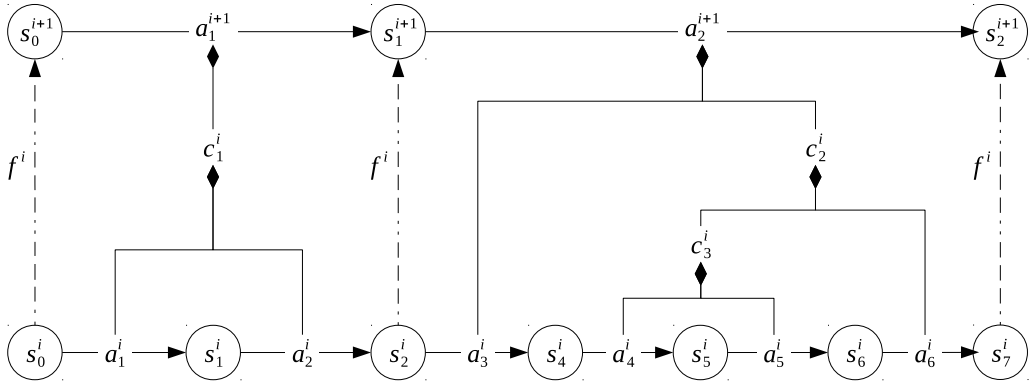


FIGURE 4.1 – Décomposition d'un plan abstrait
Décomposition du plan abstrait $a_1^{i+1}a_2^{i+1}$ au niveau $i+1$ en un plan abstrait $a_1^i a_2^i a_3^i a_4^i a_5^i a_6^i$ au niveau i. Les connecteurs en forme de diamant représentent la relation de décomposition entre une tâche et ses sous-tâches. Les tâches c_1^i, c_2^i et c_3^i ne correspondent pas à des actions abstraites, mais permettent d'introduire des niveaux de décomposition supplémentaires entre les niveaux d'abstraction.

En planification HTN, il est donc possible de formuler une hiérarchie d'abstractions sous la forme d'une suite de domaines de planification HTN, associée à une suite de fonctions d'abstraction d'états.

Définition 29 (Domaine de planification HTN avec hiérarchie d'abstractions)**.** *Soit un entier $n > 0$. Un domaine de planification HTN avec hiérarchie abstractions de taille n est un $(n+1)$-uplet $((D^0, f^0), ..., (D^n, f^n))$ tel que :*

1. $\forall i \in [\![0, n]\!], D^i = (S^i, A^i, C^i, M^i, \mathcal{T}^i, \mathcal{C}^i)$ *est un domaine de planification HTN.*
2. $f^0 \in S^0 \to S^0$ *est la fonction identité sur* S^0.
3. $\forall i \in [\![1, n]\!], f^i \in S^{i-1} \to S^i$ *et* $A^i \subseteq A^{i-1} \cup C^{i-1}$.

On remarque que cette définition n'impose pas de contraintes particulières sur les fonctions d'abstraction f^i, alors que celles-ci devaient être des morphismes de systèmes de transitions d'états dans le cas de la planification classique. Cette différence est liée à la

définition des solutions. En planification classique, toute séquence d'actions exécutable et qui satisfait le but est une solution. Comme un morphisme de système de transition d'états conserve l'accessibilité des états, il apporte l'assurance que chaque solution concrète correspond à une solution abstraite. Dans le cas de la planification HTN, une solution est une séquence d'actions exécutable qui doit être obtenue en décomposant successivement un réseau de tâches initial, et qui doit éventuellement satisfaire un certain nombre de contraintes définies dans les méthodes de décomposition. Pour une solution à un niveau d'abstraction donné, il faut donc s'assurer qu'il existe au moins un plan abstrait exécutable permettant d'accéder à cette solution par une succession de décompositions. Considérer un morphisme de système de transition d'états n'est ici pas suffisant puisque cela n'impose pas qu'un plan soit relié par une succession de décompositions à un plan abstrait correspondant. Une condition suffisante pour assurer l'existence de solution abstraite est la cohérence des actions abstraites par rapport à leur décomposition. Cette cohérence se définit vis à vis de deux aspects. Tout d'abord, si une décomposition est exécutable en un état, alors l'action abstraite doit être applicable à l'abstraction de cet état. Ensuite, pour tout état atteint par une décomposition, l'abstraction de cet état doit aussi être accessible en appliquant l'action abstraite.

Définition 30 (Correspondance entre séquences de tâches et réseaux de tâches)**.** *Soient T un ensemble de tâches. $\varrho \in T^* \to R_T$ est une fonction telle que :*

1. $\varrho(\epsilon) = \emptyset$.
2. *Pour tout $k > 0$ et pour tout $(\theta_1, ..., \theta_k) \in T^k$, il existe k nœuds distincts n_1, ..., n_k tels que :*

$$\begin{aligned} N_{\varrho(\theta_1 ... \theta_k)} &= \{n_1, ..., n_k\} \\ \prec_{\varrho(\theta_1 ... \theta_k)} &= \{(n_i, n_j) \mid (i,j) \in [\![1,k]\!]^2 \wedge i < j\} \\ t_{n_i} &= \theta_i \quad \forall i \in [\![1,k]\!] \end{aligned}$$

Définition 31 (Cohérence d'une action abstraite)**.** *Soient un entier $n > 0$ et un domaine de planification HTN avec abstraction de taille n $((D^0, f^0), ..., (D^n, f^n))$. Soient un entier $i \in [\![1,n]\!]$ ainsi qu'une action abstraite $a \in A^i$. On dit que a est cohérente si et seulement si pour toute séquence d'actions $\sigma \in A^{i-1^*}$ telle que $\mathcal{D}^*(\varrho(a), \varrho(\sigma))$:*

$\forall s \in S^{i-1}, \sigma$ *est exécutable en* $s \implies a$ *est applicable en* $s \wedge \mathcal{T}^i(f^i(s), a) = f^i(\mathcal{T}^{i-1}(s, \sigma))$.

La cohérence d'une action abstraite est similaire au concept d'action optimiste introduit par AHP. On montrera plus tard (au moment d'introduire les algorithmes de planification) que cette propriété permet d'en assurer la complétude vis-à-vis des solutions définies pour un planificateur HTN « classique ». Autrement, une alternative possible consiste à redéfinir l'ensemble des solutions pour ne considérer que celles pour lesquelles l'action abstraite est cohérente. De cette façon, l'état atteint par l'action abstraite fait figure de « contrat » devant être rempli par ses décompositions pour constituer des solutions valides. Cette alternative s'inscrit dans l'esprit de la planification HTN, puisque les méthodes de décompositions permettent généralement de spécifier des contraintes sur l'exécution des solutions. Toutefois, cette alternative impose de rajouter des étapes supplémentaires dans l'algorithme de planification : en effet, les préconditions des actions abstraites doivent être transmises et maintenues dans les réseaux de tâches lorsque ces actions sont décomposées, et les états obtenus en exécutant les décompositions doivent être comparées aux états obtenus précédemment par l'action abstraite correspondante. Il est alors nécessaire de consommer davantage de mémoire et de temps de calcul. Pour cette raison, on préfère ici s'en tenir à la première alternative, et considérer que les actions abstraites modélisées sont cohérentes avec leurs décompositions.

Un domaine de planification HTN avec hiérarchie d'abstractions est un cas particulier de domaine de planification HTN. En effet, pour un domaine de planification HTN avec hiérarchie d'abstractions $(D^i, f^i)_{i\in[\![0,n]\!]}$, le domaine de planification HTN correspondant est simplement $(S^0, A^0, C^0 \cup A^1 \cup C^1 \cup ... \cup A^n \cup C^n, M^0 \cup ... \cup M^n, \mathcal{T}^0, \mathcal{C}^0)$. La formulation d'un problème de planification peut donc reposer sur un domaine de planification HTN avec hiérarchie d'abstractions. Toutefois, tout problème de planification n'est pas adapté au concept des hiérarchies d'abstractions. En effet, on s'attend à ce que le planificateur traverse les niveaux d'abstraction du haut (le niveau le plus abstrait) vers le bas (niveau concret). Pour appliquer ce principe, il est nécessaire de formuler un problème de planification correspondant au niveau le plus abstrait, afin d'initialiser la recherche à ce niveau. La suite de fonctions d'abstraction permet bien d'abstraire l'état initial jusqu'au niveau final, mais aucun mécanisme n'a été décrit pour en faire autant avec les tâches (celles-ci peuvent être concrétisées par leur décomposition, mais pas abstraites). Pour cette raison, on considère un ensemble de problèmes de planification spécifiques, pour lesquels les tâches composant le réseau de tâches initial appartiennent nécessairement au plus haut niveau de la hiérarchie d'abstractions, et que l'on appelle « problèmes de planification HTN avec hiérarchie d'abstractions ».

Définition 32 (Problème de planification HTN avec hiérarchie d'abstrations)**.** *Soit un entier $n > 0$. Un problème de planification HTN avec hiérarchie d'abstractions de taille n est un triplet $(((D^0, f^0), ..., (D^n, f^n)), s_0, \rho_0)$ tel que :*

1. *$((D^0, f^0), ..., (D^n, f^n))$ est un domaine de planification HTN avec hiérarchie abstractions de taille n.*
2. *$s_0 \in S^0$ est l'état initial.*
3. *$\rho_0 \in R_{A^n \cup C^n}$ est le réseau de tâches initial.*

Un problème de planification HTN avec hiérarchie d'abstractions $P' = (((D^0, f^0), ..., (D^n, f^n)), s_0, \rho_0)$ est donc simplement équivalent au problème de planification HTN $P = (D, s_0, \rho_0)$, où $D = (S^0, A^0, C^0 \cup A^1 \cup C^1 \cup ... \cup A^n \cup C^n, M^0 \cup ... \cup^n, \mathcal{T}^0, \mathcal{C}^0)$. L'ensemble des solutions de P' est donc équivalent à l'ensemble des solutions de P.

4.1.2 Description des fonctions d'abstraction et des effets abstraits

Après la définition théorique d'un domaine de planification HTN avec hiérarchie d'abstractions, se pose la question de la modélisation en pratique d'un tel domaine. Un domaine de planification HTN étant une suite finie de domaines HTN « classiques », associée à une suite de fonctions d'abstraction, il suffit simplement de modéliser chacun de ces domaines, et de trouver un procédé supplémentaire pour modéliser les fonctions d'abstraction. En ce qui concerne les domaines HTN, nous disposons déjà d'un langage de modélisation avec HTDL, que nous avons présenté dans la partie 3.2.3. Pour une hiérarchie d'abstractions de taille n, on peut alors modéliser $n + 1$ domaines de planification HTN en HTDL, chaque domaine à un niveau i étant fondé sur son propre langage logique $\mathcal{L}^i$, contenant des ensembles de symboles de fonctions et de prédicats adaptés à chaque niveau d'abstraction. Certaines règles doivent toutefois être respectées : ainsi chaque tâche primitive d'un domaine abstrait doit être définie en tant que tâche dans le domaine qui correspond au niveau inférieur.

Or, une action abstraite peut apparaitre à plusieurs niveaux : un niveau maximal et un niveau minimal, qui correspondent respectivement au niveau le plus abstrait (et respectivement le moins abstrait) auquel cette action apparait en tant que tâche exécutable, ainsi que tous les autres niveaux situés entre ces deux niveaux, où cette tâche y figure nécessairement comme tâche exécutable. De plus, si cette action abstraite n'est pas

primitive, elle apparait encore au niveau inférieur au niveau minimal, où elle joue le rôle de tâche composée. Modéliser une telle tâche à l'identique, dans chaque domaine où elle apparait pourrait donc être contreproductif. Pour éviter cela, nous choisissons de décrire l'ensemble des tâches dans un unique domaine, et de préciser pour chacune d'entre elles à quels niveaux elles appartiennent. Nous introduisons alors un nouveau champ pour les actions abstraites, baptisé « `level:` ». Ce champ est succédé, sur la même ligne, ou dans un nouveau bloc d'indentation, d'une expression constituée de deux termes numériques dont la valeur est indépendante de l'état courant, et séparés par « `...` ». Le premier de ces termes indique la valeur du niveau minimal, tandis que le second indique la valeur du niveau maximal. Dans le cas où le niveau minimal et égal au niveau maximal, cette expression peut être représentée simplement par une unique expression numérique. Lorsqu'une tâche n'est décomposable à aucun niveau (et correspond alors à une action concrète), ce champ est optionnel, la tâche est alors considérée comme n'appartenant qu'au niveau 0. S'il est spécifié, il n'est valide que si le niveau minimal défini par l'expression associée correspond à 0. Le même champ est aussi utilisé pour définir le niveau auquel chaque tâche composée « pure » appartient. Mais comme chacune de ces tâches ne peut par définition apparaitre qu'à un seul niveau, on représente simplement l'expression associée au champ « `level:` » par une unique expression numérique. La figure 4.2 représente quelques tâches modélisées en HTDL avec leur champ « `level:` ». Dans cet exemple, la tâche `MoveToArea` est une action abstraite au niveau 1, ainsi qu'une tâche composée pour le niveau 0. De plus, les tâches `TakeCover` et `MoveToArea` sont des actions abstraites définies au niveau 0, mais `TakeCover` est aussi une tâche abstraite au niveau 1.

Mais en plus des $n+1$ domaines de planification HTN, il faut encore modéliser n fonctions d'abstraction (comme celle qui est associée au domaine concret correspond à la fonction identité, il n'est pas nécessaire de la modéliser). Pour cela, nous réutilisons le mécanisme des criticités introduit par ABSTRIPS, appliqué aux symboles de prédicat et de fonction. L'abstraction correspond donc à une suppression d'information, chaque langage logique associé à un niveau d'abstraction contenant un sous-ensemble des symboles contenus dans le langage associé au niveau inférieur. Ainsi, si on note $\mathcal{L}^0, ..., \mathcal{L}^n$ la suite de ces langages logiques, où $\mathcal{L}^i$ est le langage associé au niveau i, on obtient :

$$PRED(\mathcal{L}^n) \subseteq ... \subseteq PRED(\mathcal{L}^0)$$
$$FUNC(\mathcal{L}^n) \subseteq ... \subseteq FUNC(\mathcal{L}^0)$$

Chaque symbole de prédicat ou de fonction est donc associé à un nombre entier (sa criticité), qui correspond au plus haut niveau auquel il appartient. Ainsi, pour tout symbole de fonction F et tout symbole de prédicat R, leurs criticités, notées respectivement $crit(R)$ et $crit(F)$, sont définies de la façon suivante :

$$crit(R) = \max\{i \in [\![0, n]\!] \mid R \in PRED(\mathcal{L}^i)\}$$
$$crit(F) = \max\{i \in [\![0, n]\!] \mid F \in FUNC(\mathcal{L}^i)\}$$

Pour déterminer un état abstrait $f^i(s)$ à un niveau $i > 0$ à partir d'un état $s \in S^{i-1}$, on supprime simplement l'interprétation de tous les symboles dont la criticité est strictement inférieure au niveau i. Ainsi, si on note X un symbole de prédicat ou de fonction quelconque, on détermine son interprétation dans $f^i(s)$ de la façon suivante :

$$f^i(s)(X) = s(X) \text{ si } crit(X) \geq i \text{ et n'est pas défini sinon}$$

```
task MoveToArea(s, a1, a2)
    precond:
        Soldier(s) and AtArea(s, a1)
    add:
        AtArea(s, a2)
    delete:
        AtArea(s, a1)
    method pt1, pt2:
        precond
            AtPoint(s, pt1)
            InArea(pt2, a2)
        subtasks:
            MoveToPoint(s, pt1, pt2, a1, a2)
    level: 1

task TakeCover(s):
    precond:
        not Covered(s)
    add:
        Covered(s)
    level: 0...1

task MoveToPoint(s, pt1, pt2, a1, a2)
    precond:
        Soldier(s) and AtPoint(pt1)
        InArea(pt1, a1) and InArea(pt2, a2)
    add:
        AtPoint(s, pt2)
        AtArea(s, a2)
    delete:
        AtPoint(s, pt1)
        AtArea(s, a1)
```

FIGURE 4.2 – Domain de planification HTN avec hiérarchie d'abstraction
Description d'un domaine de planification HTN avec hiérarchie d'abstractions à 2 niveaux. Chaque tâche contient un nouveau champ « `level:` » qui permet d'indiquer les niveaux d'abstraction auxquels elle appartient, sauf `MoveToArea`, pour laquelle le niveau correspond par défaut à 0.

Les criticités ne concerne que les symboles de prédicat et de fonction. Le domaine du discours n'est donc pas modifié par la fonction d'abstraction :

$$dom(f^i(s)) = dom(s)$$

Pour obtenir la criticité de chaque symbole, on peut simplement s'appuyer sur les niveaux des tâches dans la description du domaine en HTDL. En effet, les symboles utilisés pour décrire une tâche doivent nécessairement appartenir au langage logique correspondant à son niveau d'abstraction. La criticité d'un symbole correspond alors simplement au plus haut niveau pour lequel il existe une tâche où ce symbole apparait (dans une formule de précondition, dans un effet, ou une condition d'effet). Ainsi, si on considère à nouveau l'exemple de la figure 4.2, on en déduit que les symboles de prédicat `AtPoint` et `InArea` sont tous les deux de criticité 0, alors que les symboles de prédicat `AtArea` et `Covered` sont de criticité 1.

Ce type d'opération d'abstraction fondé sur des criticités correspond à une suppression d'information. Or, dans la partie 2.3.2, on avait constaté que d'autres types d'opérations, qui par exemple généralisent ou agrègent l'information contenue dans les états plutôt que de la supprimer, étaient plus adaptées dans un contexte de planification appliquée au comportement tactique d'unités d'infanterie. Une solution pour implémenter ces opérations

est de modéliser des « règles » permettant de déterminer l'interprétation d'un symbole abstrait à partir de l'interprétation des symboles présents au niveau inférieur. L'ensemble des règles qui correspondent aux symboles d'un niveau donné, ainsi que le mécanisme des criticités, caractérisent alors chaque fonction d'abstraction. Il est toutefois possible d'implémenter un système équivalent en utilisant uniquement le principe des criticités pour définir ces fonctions d'abstraction. Pour cela, on définit les symboles abstraits dès le niveau concret sous la forme de symboles dérivés. Les règles qui définissent ces prédicats et ces fonctions à partir des symboles plus concrets sont alors modélisées sous la forme d'axiomes. Il devient ainsi possible d'exprimer n'importe quelle opération d'abstraction entre un symbole abstrait et un ensemble de symboles plus concrets, à condition que cette opération s'exprime sous la forme d'une formule en logique du premier ordre. La figure 4.3 présente un exemple d'axiome décrivant une opération d'agrégation (un ensemble de soldats est agrégé en une équipe afin de synthétiser leur position).

Chaque symbole est alors associé à un niveau, qui correspond au plus bas niveau d'abstraction à partir duquel l'interprétation de ce symbole n'est plus dérivée, mais peut être modifiée par les effets des actions. Comme pour les criticités, il n'est pas nécessaire de définir ces niveaux de façon explicite en HTDL, puisque ceux-ci correspondent aux plus petits niveaux pour lequel chaque symbole apparait dans les effets d'une action. Pour être cohérente, la description en HTDL de ces symboles dérivés doit cependant respecter la règle suivante : si un axiome définit un symbole dérivé à un niveau $i > 0$, alors les criticités de tous les symboles (autres que le symbole dérivé concerné) qui apparaissent dans cet axiome doivent être supérieur ou égale à $i - 1$. Ainsi, si un symbole est défini avec un niveau i et une criticité $j \geq i$, ce symbole sera géré comme un symbole dérivé pour tous les niveaux strictement inférieurs à i (un symbole de niveau 0 ne sera donc jamais traité comme un symbole dérivé : on retrouve donc le fonctionnement des symboles concrets), puis comme un symbole non dérivé entre les niveaux i et j compris, où il fait parti des effets de certaines actions, et enfin ne fera plus parti de la définition des états au delà du niveau j.

La gestion des prédicats dérivés peut s'effectuer de plusieurs façons. Une première possibilité est de suivre la méthode proposée par JSHOP2 [47], qui impose de modéliser les axiomes sous la forme de clauses de Horn, puis utilise un moteur d'inférence pour calculer les substitutions permettant de satisfaire une clause par rapport à un état donné. Autrement, on peut remarquer que par définition, ces axiomes ne peuvent pas être récursifs (à cause de la restriction portant sur les valeurs des criticités pour les symboles contenus dans les axiomes), ce qui permet de les pré-compiler en une procédure permettant de mettre à jour l'interprétation de chaque symbole dérivé après l'application des effets d'une action, et cela quelque soit la forme d'un axiome. C'est cette dernière solution qui a été retenue pour l'algorithme implémenté.

4.1.3 Algorithmes

Cette partie présente deux algorithmes de planification qui exploitent la hiérarchie d'abstractions modélisée dans le domaine de planification HTN. Dans un premier temps, nous décrivons un algorithme simple, conçu sous la forme d'un méta-planificateur. À partir de cet algorithme, et après avoir discuter des conditions à respecter pour en assurer le fonctionnement, nous présentons un nouvel algorithme qui apporte davantage de flexibilité pour l'implémentation de stratégies de décomposition des tâches. Nous démontrons finalement que sous certaines conditions, cet algorithme termine, et est correct et complet.

```
FireteamAt(ft, a) forall ft, a if \
    Fireteam(ft) and Area(a)
    forall s if InFireteam(s, ft):
        SoldierAt(s, a)

task FireteamMove(ft, a1, a2):
    precond:
        Fireteam(ft)
        Area(a1) and Area(a2)
        FiretealAt(ft, a1)
    add:
        FireteamAt(ft, a2)
    delete:
        FireteamAt(ft, a1)
    method:
        subtasks:
            AchieveAtArea(f, a2)
    level: 1

task AchieveFireteamAt(ft, a):
    method:
        precond:
            FireteamAt(ft, a)
    method s, a':
        precond:
            InFireteam(s, ft)
            SoldierAt(s, a') and a' != a
        subtasks:
            n1 : SoldierMove(s, a1, a2)
            n2 : AchieveFireteamAt(ft, a)
        order:
            n1 < n2
    level: 0

task SoldierMove(s, a1, a2):
    precond:
        Soldier(s)
        Area(a1) and Area(a2)
        SoldierAt(s, a1)
    add:
        SoldierAt(s, a2)
    delete:
        SoldierAt(s, a1)
```

FIGURE 4.3 – Prédicat abstrait dérivé
Description d'un domaine de planification HTN avec hiérarchie d'abstractions à 2 niveaux où un axiome défini le prédicat abstrait `FireteamAt`. Ce symbole apparaissant comme effet de la tâche `FireteamMove`, au niveau 1, mais pas de `SoldierMove` au niveau 1, on en déduit que `FireteamAt` est un symbole de niveau 1.

ABHTNPLAN

En suivant le même principe que pour la planification avec hiérarchie d'abstractions, il est possible de concevoir un « méta-planificateur » pour résoudre les problèmes de planification HTN avec hiérarchie d'abstractions. Ce méta-planificateur est décrit par l'algorithme 10, et est appelé ABHTNPLAN. Il s'appuie sur un planificateur HTN (désigné sous le nom de HTNPLAN) pour résoudre le problème de planification posé à chaque niveau d'abstraction. ABHTNPLAN prend en charge, à chaque niveau, la conversion du plan obtenu en réseau de tâches initial qu'il transmet au niveau inférieur, et prend aussi en charge la gestion des retours arrières entre les niveaux d'abstraction, lorsque HTNPLAN échoue à trouver de solution à un niveau donné. Contrairement à ABPLAN, le méta-planificateur adapté au contexte de la planification classique présenté en partie 2.3.3, la décomposition en sous-problèmes à chaque niveau d'abstraction n'est pas gérée par le méta-planificateur. En effet, celle-ci fait déjà partie du processus de décomposition de la planification HTN, puisque chaque tâche y joue le rôle des sous-problèmes en planification

classique. La décomposition en sous problèmes est donc gérée en interne par la procédure de planification HTN employée.

Algorithm 10 Algorithme de planification HTN avec abstraction hiérarchique ABHTNPLAN

entrées : Un problème de planification HTN avec hiérarchie d'abstractions $P = ((D^0, f^0), ..., (D^n, f^n))$ de taille n, où n est un entier strictement positif. Une procédure de planification HTN nommée HTNPLAN.

sorties : un plan solution de P ou *échec*.

1: $Ouvert \leftarrow \{(\epsilon, \rho_0^0, n)\}$
2: **tant que** $Ouvert \neq \emptyset$ **faire**
3: $\sigma^k, \rho^k, k \leftarrow$ extraire un élément de $Ouvert$
4: $s_0^k \leftarrow f^k \circ ... \circ f^0(s_0^0)$
5: **si** N_{ρ^k} est vide **alors**
6: **si** $k > 0$ **alors**
7: $\rho^{k-1} \leftarrow \varrho(\sigma^k)$
8: $Ouvert \leftarrow Ouvert \cup \{(\epsilon, \rho^k, k-1)\}$
9: **sinon**
10: **retourner** σ^k
11: **fin si**
12: **sinon**
13: **pour** chaque solution σ de HTNPLAN(D^k, s_0^k, ρ^k) **faire**
14: $Ouvert \leftarrow Ouvert \cup \{(\sigma, \emptyset, k)\}$
15: **fin pour**
16: **fin si**
17: **fin tant que**
18: **retourner** *échec*

Tout comme ABPLAN, ABHTNPLAN impose certaines restrictions sur le planificateur utilisé. Ainsi HTNPLAN doit être en mesure de retourner successivement l'ensemble des solutions au problème posé. En conséquence, l'utilisation d'un mécanisme d'élagage des plans non optimaux (comme le *branch & bound*) au niveau de HTNPLAN peut compromettre l'optimalité d'ABHTNPLAN (en effet une solution concrète optimale ne l'est pas nécessairement au niveau abstrait), voire la complétude, puisque toutes les décompositions concrètes d'une solution abstraite ne sont pas nécessairement exécutables. Par conséquent, la technique de *branch & bound* implémenté dans SHOP2 et SHPE doit être « désactivée » afin d'utiliser ces planificateurs. Si on souhaite bénéficier des avantages du *branch & bound*, par exemple pour réaliser une version optimale de ABHTNPLAN, cette technique peut toutefois être implémentée au niveau du méta-planificateur, en utilisant uniquement les coût des solutions concrètes.

Dans le cas d'ABPLAN, il était nécessaire que PLAN ne retourne qu'un nombre fini de solutions. Pour un nombre fini d'états et d'actions, cette contrainte pouvait aisément être satisfaite en implémentant un mécanisme permettant de supprimer les plans acycliques dans l'espace de recherche de PLAN. La même contrainte n'est cependant pas exactement applicable à la planification HTN. En effet, les plans n'y sont plus des solutions parce qu'ils permettent d'atteindre un état « but », mais parce qu'ils correspondent à des décompositions, et ces décompositions sont potentiellement cycliques. C'est par exemple le cas d'un problème de navigation pour lequel une tâche imposerait de traverser deux fois le même nœud d'un graphe. Nous avons toutefois vu que la structure de certains espaces de planification HTN permettait de garantir que ceux-ci sont finis. De façon générale, si un espace de recherche est fini, et si une procédure permet de contrôler qu'un élément n'est pas visiter plus d'une fois, l'ensemble des solutions est aussi fini. Dans le cas où

HTNPLAN correspond à SHPE, l'espace de recherche (qui correspond alors à un espace de progression) est fini si le problème de planification est « $\leq_r -stratifiable$ » (d'après [3]). Dans le cas d'un problème de planification HTN avec hiérarchie d'abstractions, si celui est « $\leq_r -stratifiable$ », alors chaque problème généré à chaque niveau d'abstraction est aussi « $\leq_r -stratifiable$ ». Ainsi si une procédure de vérification des sous-problèmes visités est implémentée dans SHPE, on obtient alors la garantie que ABHTNPLAN termine.

Cependant, une telle procédure ne permettrait plus de garantir la complétude d'ABHTNPLAN. En effet, les sous-problèmes des espaces de progressions (qui est l'espace parcouru par SHPE) correspondent aux paires constituées de l'état courant et du réseau des tâches restant à planifier. À cause de l'abstraction des états, un planificateur à un niveau abstrait pourrait alors supprimer un sous-problème déjà visité, alors que ce sous-problème correspondrait à un état différent au niveau concret. La figure 4.4 présente un domaine de planification HTN avec hiérarchie d'abstractions décrit qui permet d'illustrer cette situation. Ce domaine modélise un problème consistant à trouver un plan pour effectuer un voyage en visitant un ensemble de lieux puis retourner au lieu de départ. Le langage logique permettant de décrire les états du monde est constitué de la constante `At`, qui représente le lieu auquel se trouve le voyageur, de la constante `Start`, qui représente le lieu de départ, et du symbole de prédicat `Visited` qui représente les lieux qui ont été visités par le voyageur. Ce dernier prédicat est abstrait au niveau des tâches `AbTravel` et `EndTrip`. En conséquence, pour tout plan au niveau concret, l'état initial et l'état final sont différents car aucun lieu n'est visité initialement alors qu'ils le sont tous dans l'état final. Toutefois au niveau abstrait, tous les plans se décomposant en solution seront rejetés, car ils imposeront de visiter deux fois le sous problème correspondant à l'état abstrait pour lequel `At` et `Start` sont égaux, et au réseau de tâches ne contenant que la tâche `AchieveTrip`.

Ce problème ne se pose pas si le planificateur qui implémente HTNPLAN parcours un espace de décomposition (ce que font les planificateurs HTN non linéaires par exemple). Ces planificateurs vérifieront alors que le plan global n'a pas été visité plus d'une fois, et assureront la terminaison de l'algorithme tout en maintenant la complétude (mais dans des conditions plus restrictive vis à vis de la structure des réseaux de tâches). Dans le cas d'ABHTNPLAN, on peut remarquer qu'au niveau du méta-planificateur, l'espace de recherche ne correspond plus à un espace de progression, puisque les hiérarchies d'abstraction permettent de décomposer certaines tâches alors que toutes les tâches précédentes n'ont pas encore été réduites au niveau primitif. Cependant, l'espace de recherche au niveau du méta-planificateur correspond être à un cas particulier d'espace de décomposition. Une solution serait donc de contrôler les sous-problèmes visités au niveau d'ABHTNPLAN, et non pas au niveau de HTNPLAN. Toutefois, cette modification ne semble pas compatible avec une approche fondée sur un méta-planificateur, puisque celui-ci n'a alors pas accès à l'espace de recherche exploré par la procédure de planification exécutée à chaque niveau.

L'utilisation d'effets à différents niveaux d'abstraction porte aussi atteinte à la complétude d'ABHTNPLAN. En effet, à chaque niveau, le planificateur impose un ordre total sur les actions abstraites afin de vérifier l'exécutabilité du plan. Ce procédé a pour conséquence la suppression des plans finaux dont l'ordre des tâches primitives n'est compatible avec aucune linéarisation des actions abstraites correspondantes au niveau supérieur (ce qui advient par exemple lorsque les sous-tâches de deux actions abstraites sont « entremêlées »). Une solution à ce problème serait de dé-linéariser les plans obtenus à chaque niveau au moment de les transformer en réseaux de tâches, par exemple en ne conservant que les éléments de la relation d'ordre spécifiés dans les méthodes de décompositions utilisées. Une autre solution plus efficace serait d'utiliser un planificateur de type POP pour implémenter HTNPLAN, et retourner le plan directement sous la forme d'un réseau de tâches partiellement ordonné. Cependant, cette dernière solution n'assurerait toujours pas

```
task AchieveTrip
    method x:
        subtasks:
            n1 : AbTravel(x)
            n2 : AchieveTrip
        order:
            n1 < n2
    method x:
        subtasks:
            AbEndTrip

task AbTravel(x)
    update:
        At = x
    method:
        subtasks:
            subtasks:
                Travel(x, y)
    level: 1

task Travel(x)
    update:
        At = x
    add:
        Visited(x)

task AbEndTrip
    precond:
        At == StartAt
    method:
        subtasks:
            EndTrip
    level: 1

task EndTrip(x)
    precond:
        Start(x) and At(x)
        forall y if Location(y):
            Visited(y)
```

FIGURE 4.4 – Domaine de planification HTN pour accomplir un voyage entre plusieurs lieux
Domaine HTDL (avec l'extension permettant de modéliser une hiérarchie d'abstraction) pour un exemple de problème de voyage entre plusieurs lieux. Ce domaine est modélisé avec une hiérarchie d'abstractions à deux niveaux. Le niveau concret contient les deux tâches primitives `Travel` et `EndTrip`, et le niveau 1 contient les deux actions abstraites `AbTravel` et `AbEndTrip` ainsi que la tâche composée `AchieveTrip`. Les états du niveau 1 sont obtenus en ignorant l'interprétation du symbole de prédicat `Visited`.

la complétude pour ABHTNPLAN. Qiang Yang a en effet montré que l'utilisation d'effets au niveau des tâches non primitives entraine des situations où un plan non exécutable au niveau non-primitif (qui serait donc élagué par ABHTNPLAN) pouvait néanmoins être décomposé en un plan exécutable [116]. L'un des exemples utilisés par Qiang Yang pour illustrer ce phénomène est présenté en figure 4.5. Dans cet exemple, il est possible de former un plan composé de deux actions abstraites qui n'est pas exécutable, car chacune des deux actions contient un effet qui entraine la négation d'une précondition de l'autre. Or, il est possible de décomposer ces deux actions et d'entremêler leurs sous-tâches de façon à obtenir un plan exécutable.

On peut noter que le problème d'incomplétude posé par l'exemple de la figure 4.5 ne se pose pas si on attribue une durée aux tâches (mais on sort alors du cadre de la planification classique pour entrer dans celui de la planification temporelle, qui n'est pas l'objet de ce manuscrit). Si une durée non nulle sépare les préconditions des effets de chacune des deux actions abstraites, alors il suffit de les ordonnancer de façon à ce que l'instant où s'applique les effets de l'une intervient après l'instant où les préconditions de

```
task TaskA
    precond:
        X
    add:
        U
    delete:
        Y
    method:
        subtasks:
          n1 : TaskA1
          n2 : TaskA2
        order:
            n1 < n2
    level: 1

task TaskA1
    precond:
        X

task TaskA2
    add:
        U
    delete:
        Y
```

```
task TaskB
    precond:
        Y
    add:
        W
    delete:
        X
    method:
        subtasks:
            n1 : TaskB1
            n2 : TaskB2
        order:
            n1 < n2
    level: 1

task TaskB1
    precond:
        Y

task TaskB2
    add:
        W
    delete:
        X
```

FIGURE 4.5 – Exemple d'incomplétude pour les effets abstraits
Exemple de domaine de planification HTN avec effet abstrait pour lequel ABHTNPLAN peut échouer à retourner une solution. Dans ce domaine, $TaskA$ et $TaskB$ sont des actions abstraites, tandis que $TaskA1$, $TaskA2$, $TaskB1$ et $TaskB2$ sont des actions primitives, et la fonction d'abstraction correspond à la fonction identité sur l'ensemble des états concrets. Si on considère un état initial vérifiant la formule $X \wedge Y \wedge \neg U \wedge \neg W$, alors aucun plan abstrait contenant $TaskA$ et $TaskB$ (peut importe dans quel ordre) n'est exécutable, alors qu'au niveau primitif, le plan formé par la séquence d'actions $TaskA1, TaskB1, TaskA2, TaskB2$ est bien exécutable.

l'autre doivent être satisfaites.

L'intérêt principale de ABHTNPLAN par rapport à un planificateur HTN comme SHOP ou SHPE est de permettre l'utilisation d'une stratégie de décomposition des tâches alternative, tout en maintenant le mécanisme de progression des états introduit par SHOP. Au lieu de toujours décomposer en priorité les premières tâches non primitives, la stratégie de sélection alternative décompose en priorité les tâches au plus haut niveau d'abstraction. Mais on pourrait aussi souhaiter introduire des stratégies de décompositions des tâches intermédiaires vis-à-vis de ces deux stratégies radicalement opposées. Par exemple, on pourrait souhaiter implémenter une stratégie qui décomposerait les tâches au niveau le plus abstrait jusqu'à une profondeur de plan donné, puis décomposerait les actions abstraites obtenues, sans pour autant attendre d'avoir obtenu un plan complet au niveau abstrait. Ainsi on pourrait atteindre un compromis entre la décomposition des premières tâches composées et la décomposition des tâches plus abstraites. Ce compromis permet d'accéder plus rapidement au premières tâches primitives, que l'on peut alors retourner si le temps d'exécution affecté au planificateur est expiré, mais aussi d'obtenir un aperçu

du plan à plus long terme, et éventuellement de détecter son infaisabilité.

Réseaux de tâches totalement ordonnés

Avant de décrire un nouvel algorithme qui implémente les différentes remarques que nous avons énumérées ci-dessus, nous allons nous intéresser aux réseaux de tâches totalement ordonnés. L'intérêt des réseaux de tâches totalement ordonnés est qu'ils permettent d'éviter le problème d'incomplétude lié aux effets abstraits, puisque l'ordre total imposé à leurs tâches ne permet pas d'entremêler leurs sous-tâches. La complétude de l'algorithme qui sera présentée plus loin reposera donc sur l'hypothèse que tous les réseaux de tâches (le réseau de tâches initial et les réseaux de tâches définis dans les méthodes de décomposition) sont totalement ordonnés. Le but de cette sous-partie est donc d'introduire des éléments de théories relatifs aux réseaux de tâches totalement ordonnés qui nous permettrons de justifier les propriétés de cet algorithme.

On considère ici un ensemble de tâches T ainsi qu'un ensemble de méthodes de décomposition M. Le premier élément que l'on introduit est une fonction qui permet de concaténer deux réseaux de tâches. Cette fonction prend deux réseaux de tâches en entrée, et retourne un réseau de tâches pour lequel toutes les tâches du premier se trouvent ordonnées avant toutes les tâches du second :

Définition 33 (Concaténation de réseau de tâches)**.** *Soit un ensemble de tâches T. On note $* \in R_T \times R_T \to R_T$ l'opérateur de concaténation définit tel que :*

$$\forall \rho_1, \rho_2 \in R_T : \rho_1 * \rho_2 = (N_{\rho_1} \cup N_{\rho_2}, \prec_{\rho_1} \cup \prec_{\rho_2} \cup N_{\rho_1} \times N_{\rho_2})$$

Remarque : comme pour la réduction de réseau de tâches, on considère dans cette définition que ρ_1 et ρ_2 sont deux représentants de leur classe d'isomorphisme pour lesquels N_{ρ_1} et N_{ρ_2} sont disjoints.

Avec le réseau de tâches vide, cette opération caractérise un monoïde sur l'ensemble des réseaux de tâches. La caractérisation d'un monoïde nous permet d'obtenir des propriétés intéressantes pour manipuler cet opérateur, comme l'associativité. De plus, ce monoïde est régulier à gauche et à droite, une autre propriété intéressante qui permet de simplifier certains calculs :

Proposition 2 (Monoïde associé à la concaténation)**.** *Soit T un ensemble de tâches. $(R_T, *, \emptyset)$ est un monoïde régulier à gauche et à droite.*

Démonstration. Soient $(\rho_1, \rho_2, \rho_3) \in {R_T}^3$.

1. Loi interne : $*$ est une loi interne d'après la définition 33.

2. Associativité :

$$\begin{aligned}
N_{\rho_1*(\rho_2*\rho_3)} &= N_{\rho_1} \cup N_{\rho_2*\rho_3} \\
&= N_{\rho_1} \cup N_{\rho_2} \cup N_{\rho_3} \\
N_{(\rho_1*\rho_2)*\rho_3)} &= N_{\rho_1*\rho_2} \cup N_{\rho_3} \\
&= N_{\rho_1} \cup N_{\rho_2} \cup N_{\rho_3} \\
\prec_{\rho_1*(\rho_2*\rho_3)} &= \prec_{\rho_1} \cup \prec_{\rho_2*\rho_3} \cup (N_{\rho_1} \times \ N_{\rho_2*\rho_3}) \\
&= \prec_{\rho_1} \cup \prec_{\rho_2} \cup \prec_{\rho_3} \cup (N_{\rho_2} \times N_{\rho_3}) \cup (N_{\rho_1} \times N_{\rho_2} \cup N_{\rho_3}) \\
&= \prec_{\rho_1} \cup \prec_{\rho_2} \cup \prec_{\rho_3} \cup (N_{\rho_2} \times N_{\rho_3}) \cup (N_{\rho_1} \times N_{\rho_2}) \cup (N_{\rho_1} \times N_{\rho_3}) \\
\prec_{(\rho_1*\rho_2)*\rho_3)} &= \prec_{\rho_1*\rho_2} \cup \prec_{\rho_3} \cup (N_{\rho_1*\rho_2} \times N_{\rho_3}) \\
&= \prec_{\rho_1} \cup \prec_{\rho_2} \cup (N_{\rho_1} \times N_{\rho_2}) \cup \ \prec_{\rho_3} \cup (N_{\rho_1} \cup N_{\rho_2} \times N_{\rho_3}) \\
&= \prec_{\rho_1} \cup \prec_{\rho_2} \cup (N_{\rho_1} \times N_{\rho_2}) \cup \ \prec_{\rho_3} \cup (N_{\rho_1} \times N_{\rho_3}) \cup (N_{\rho_2} \times N_{\rho_3})
\end{aligned}$$

Ces calculs permettent d'en déduire que :

$$N_{\rho_1*(\rho_2*\rho_3)} = N_{(\rho_1*\rho_2)*\rho_3}$$
$$\prec_{\rho_1*(\rho_2*\rho_3)} = \prec_{(\rho_1*\rho_2)*\rho_3}$$

On en conclut donc que $*$ est un opérateur associatif.

3. Élément neutre :

$$\begin{aligned}\rho_1 * \emptyset &= (N_{\rho_1} \cup \emptyset, \prec_{\rho_1} \cup \emptyset \cup (N_{\rho_1} \times \emptyset)) \\ &= (N_{\rho_1}, \prec_{\rho_1}) \\ &= \rho_1 \\ \emptyset * \rho_1 &= (\emptyset \cup N_{\rho_1}, \emptyset \cup \prec_{\rho_1} \cup (\emptyset \times N_{\rho_1})) \\ &= (N_{\rho_1}, \prec_{\rho_1}) \\ &= \rho_1\end{aligned}$$

$\emptyset$ est donc bien un élément neutre pour $*$.

4. Régularité à gauche :
ρ_1, ρ_2 et ρ_3 étant trois réseaux de tâches distincts, on peut en déduire que :

$$\begin{aligned}\rho_1 * \rho_2 = \rho_1 * \rho_3 \implies & N_{\rho_1} \cup N_{\rho_2} = N_{\rho_1} \cup N_{\rho_3} \wedge \\ & \prec_{\rho_1} \cup \prec_{\rho_2} \cup (N_{\rho_1} \times N_{\rho_2}) = \prec_{\rho_1} \cup \prec_{\rho_3} \cup (N_{\rho_1} \times N_{\rho_3}) \\ \implies & N_{\rho_2} = N_{\rho_3} \wedge \\ & \prec_{\rho_2} \cup (N_{\rho_1} \times N_{\rho_2}) = \prec_{\rho_3} \cup (N_{\rho_1} \times N_{\rho_3}) \\ \implies & N_{\rho_2} = N_{\rho_3} \wedge \prec_{\rho_2} = \prec_{\rho_3} \\ \implies & \rho_2 = \rho_3\end{aligned}$$

$(R_T, *, \emptyset)$ est donc bien régulier à gauche.

5. Régularité à droite :
La démonstration étant identique à la régularité à gauche, on en déduit que $(R_T, *, \emptyset)$ est régulier à droite.

□

Jusqu'ici, nous avons considéré l'ensemble complet des réseaux de tâches. Nous allons maintenant nous restreindre à l'ensemble des réseaux de tâches totalement ordonnés. On peut remarquer que chaque réseau de tâches totalement ordonné correspond à une séquence de tâches. On peut donc définir l'ensemble des réseaux de tâches totalement ordonnés comme étant l'image de T^* par la fonction ϱ, qui se trouve être un isomorphisme entre ces deux ensembles. Pour justifier cette affirmation, on montre dans un premier temps que chaque séquence de tâches est l'unique linéarisation d'un réseau de tâches totalement ordonné :

Proposition 3 (Linéarisation d'un réseau de tâches totalement ordonné)**.** *Soit $\sigma \in T^*$. σ est l'unique linéarisation de $\varrho(\sigma)$.*

Démonstration. 1. Si $\sigma = \epsilon$, alors σ est trivialement l'unique linéarisation de $\emptyset$.

2. Sinon, on note $k = |\sigma|$. Il existe alors $(\theta_0, ..., \theta_k) \in T^k$ tel que $\sigma = \theta_1 ... \theta_k$.
D'après la définition 30 il existe k nœuds distincts n_1, ..., n_k tels que :

$$\begin{aligned}N_{\varrho(\theta_1 ... \theta_k)} &= \{n_1, ..., n_k\} \\ \prec_{\varrho(\theta_1 ... \theta_k)} &= \{(n_i, n_j) \mid (i,j) \in [\![1,k]\!]^2 \wedge i < j\} \\ t_{n_i} &= \theta_i \quad \forall i \in [\![1,k]\!]\end{aligned}$$

On considère alors la fonction $\Lambda \in N_{\varrho(\sigma)} \to [\![1,k]\!]$ telle que pour tout $i \in [\![1,k]\!]$: $\Lambda(n_i) = i$.
Par définition, Λ est bijective et vérifie :

$$\begin{aligned} t_{n_i} &= \theta_{\Lambda(n_i)} \quad \forall i \in [\![1,k]\!] \\ n_i \prec_{\varrho(\sigma)} n_j \implies \Lambda(n_i) &< \Lambda(n_j) \quad \forall (i,j) \in [\![1,k]\!]^2 \end{aligned}$$

D'après la définition 11, Λ est donc une fonction de linéarisation, et σ est donc bien une linéarisation de $\varrho(\sigma)$.
Supposons maintenant qu'il existe une autre linéarisation $\sigma' \in T^*$ de $\varrho(\sigma)$.
D'après la définition 11, on en déduit que $|\sigma'| = k$. Il existe donc $(\theta'_0, ..., \theta'_k) \in T^k$ tel que $\sigma' = \theta'_1 ... \theta'_k$.
Soit Λ' la fonction de linéarisation associée à σ'.
Supposons que $\sigma' \neq \sigma$. Il existe alors un plus petit $i_0 \in [\![1, k-1]\!]$ tel que $\Lambda'(n_{i_0}) > i_0$.
Λ' étant bijective, il existe donc un $i_1 \in [\![i_0+1, k]\!]$ tel que $\Lambda'(n_{i_1}) = i_0$.
On en déduit donc que $\Lambda'(n_{i_1}) < \Lambda'(n_{i_0})$.
$\prec_{\varrho(\sigma)}$ étant un ordre total, on peut montrer que $\forall (n_1, n_2) \in N_{\varrho(\sigma)} : n_1 \prec_{\varrho(\sigma)} n_2 \iff \Lambda'(n_1) < \Lambda'(n_2)$.
On en déduit donc que $n_{i_1} \prec_{\varrho(\sigma)} n_{i_0}$ alors que $i_0 < i_1$, ce qui est une contradiction avec la définition de $\prec_{\varrho(\sigma)}$.
On a donc $\sigma' = \sigma$.
σ est donc bien l'unique linéarisation de $\varrho(\sigma)$. □

On en déduit donc que ϱ est un isomorphisme entre le monoïde $(T^*, \circ, \epsilon)$ et le sous-monoïde $(\varrho(T^*), *, \emptyset)$ de R_T :

Proposition 4 (Isomorphisme entre séquence de tâches et réseaux de tâches totalement ordonnés)**.** *ϱ est un isomorphisme entre les monoïdes $(T^*, \circ, \epsilon)$ et $(\varrho(T^*), *, \emptyset)$.*

Démonstration. On montre dans un premier temps que ϱ est un morphisme entre $(T^*, \circ, \epsilon)$ et $(\varrho(T^*), *, \emptyset)$, et dans un second temps que ϱ est une application bijective entre T^* et $\varrho(T^*)$.

1. ϱ est un morphisme entre $(T^*, \circ, \epsilon)$ et $(\varrho(T^*), *, \emptyset)$:
   ϱ est un morphisme entre les monoïdes $(T^*, \epsilon, \circ)$ et $(R_T, \emptyset, *)$ si et seulement si les propriétés suivantes sont satisfaites :

$$\varrho(\epsilon) = \emptyset \tag{4.1}$$
$$\forall \sigma_1, \sigma_2 \in T^* : \varrho(\sigma_1 \circ \sigma_2) = \varrho(\sigma_1) * \varrho(\sigma_2) \tag{4.2}$$

   (a) 4.1 :
   $\varrho(\epsilon) = \emptyset$ est satisfaite d'après la définition 30.

   (b) 4.2 :
   Soient $\sigma_1, \sigma_2 \in T^*$ et $(\theta_1, ..., \theta_{|\sigma_1|+|\sigma_2|}) \in T^{|\sigma_1|+|\sigma_2|}$ telles que $\sigma_1 = \theta_1 ... \theta_{|\sigma_1|}$ et $\sigma_2 = \theta_{|\sigma_1|+1} ... \theta_{|\sigma_1|+|\sigma_2|}$.
   Notons de plus $|\sigma_1| + |\sigma_2|$ nœuds distincts n_1, ..., $n_{|\sigma_1|+|\sigma_2|}$ tel que :

$$\forall i \in [\![1, |\sigma_1| + |\sigma_2|]\!] : t_{n_i} = \theta_i$$

   D'après la définition 30, on peut définir les réseaux de tâches $\varrho(\sigma_1)$, $\varrho(\sigma_2)$ et

$\varrho(\sigma_1 \circ \sigma_2)$ de la façon suivante :

$$\begin{aligned}
N_{\varrho(\sigma_1)} &= \{n_1, ..., n_{|\sigma_1|}\} \\
\prec_{\varrho(\sigma_1)} &= \{(n_i, n_j) \mid (i,j) \in [\![1, |\sigma_1|]\!]^2 \wedge i < j\} \\
N_{\varrho(\sigma_2)} &= \{n_{|\sigma_1|+1}, ..., n_{|\sigma_1|+|\sigma_2|}\} \\
\prec_{\varrho(\sigma_2)} &= \{(n_i, n_j) \mid (i,j) \in [\![|\sigma_1|+1, |\sigma_1|+|\sigma_2|]\!]^2 \wedge i < j\} \\
N_{\varrho(\sigma_1 \circ \sigma_2)} &= \{n_1, ..., n_{|\sigma_1|+|\sigma_2|}\} \\
\prec_{\varrho(\sigma_1 \circ \sigma_2))} &= \{(n_i, n_j) \mid (i,j) \in [\![1, |\sigma_1|+|\sigma_2|]\!]^2 \wedge i < j\}
\end{aligned}$$

On en déduit donc :

$$\begin{aligned}
N_{\varrho(\sigma_1 \circ \sigma_2)} &= N_{\varrho(\sigma_1)} \cup N_{\varrho(\sigma_2)} \\
\prec_{\varrho(\sigma_1 \circ \sigma_2))} &= \{(n_i, n_j) \mid (i,j) \in [\![1, |\sigma_1|+|\sigma_2|]\!]^2 \wedge i < j\} \\
&= \{(n_i, n_j) \mid (i,j) \in [\![1, |\sigma_1|]\!]^2 \wedge i < j\} \\
&\quad \cup \{(n_i, n_j) \mid (i,j) \in [\![|\sigma_1|+1, |\sigma_1|+|\sigma_2|]\!]^2 \wedge i < j\} \\
&\quad \cup \{(n_i, n_j) \mid (i,j) \in [\![1, |\sigma_1|]\!] \times [\![|\sigma_1|+1, |\sigma_1|+|\sigma_2|]\!] \wedge i < j\} \\
&\quad \cup \{(n_i, n_j) \mid (i,j) \in [\![|\sigma_1|+1, |\sigma_1|+|\sigma_2|]\!] \times [\![1, |\sigma_1|]\!] \wedge i < j\} \\
&= \prec_{\varrho(\sigma_1)} \cup \prec_{\varrho(\sigma_2)} \cup (N_{\varrho(\sigma_1)} \cup N_{\varrho(\sigma_2)}) \cup \emptyset
\end{aligned}$$

D'après la définition 33, on en déduit donc que $\varrho(\sigma_1 \circ \sigma_2) = \varrho(\sigma_1) * \varrho(\sigma_2)$.

2. ϱ est une bijection entre T^* et $\varrho(T^*)$:
   Soit $\rho \in \varrho(T^*)$. On suppose qu'il existe σ_1 et σ_2 tel que $\varrho(\sigma_1) = \rho$ et $\varrho(\sigma_2) = \rho$. D'après la proposition 3, σ_1 et σ_2 sont des linéarisations de ρ, et ρ n'admet qu'une unique linéarisation. On en déduit donc que $\sigma_1 = \sigma_2$.
   Pour tout $\rho \in \varrho(T^*)$, il existe donc un unique $\sigma \in T^*$ tel que $\varrho(\sigma) = \rho$.
   On en déduit donc que ϱ est une bijection entre T^* et $\varrho(T^*)$.

□

Ce dernier résultat nous permet finalement de manipuler les réseaux de tâches totalement ordonnés à la façon des séquences de tâches. Ainsi, pour tout réseau de tâches totalement ordonné ρ, on pourra toujours exhiber une séquence de tâches $(\theta_1, ..., \theta_{|N_\rho|}) \in T^{N_\rho|}$ telle que

$$\rho = \varrho(\theta_1) * ... * \varrho(\theta_{|N_\rho|}).$$

Dans un espace de réseaux de tâches totalement ordonnés, il n'est par définition pas possible de transformer un réseau de tâches en un nouveau réseau de tâches simplement en lui ajoutant des contraintes d'ordre. En effet, chaque nœud étant déjà ordonné par rapport aux autres nœuds du réseau, l'ajout de nouvelles contraintes aurait pour effet soit de ne pas modifier la relation d'ordre (dans le cas où celle-ci contient déjà ces contraintes) ou de la rendre incohérente. On en déduit donc que l'unique transformation que l'on peut appliquer à un réseau de tâches totalement ordonné est la décomposition d'une tâche non primitive, c'est-à-dire la réduction de réseau de tâches :

Proposition 5 (Espace de décomposition d'un réseau de tâches totalement ordonné)**.** *Soit $(\rho_1, \rho_2) \in \varrho(T^*)$ tel que $\mathcal{D}(\rho_1, \rho_2)$. Il existe $n \in N_{\rho_1}$ et $m \in M$ tels que $c_m = t_n$ et $\rho_2 = \mathcal{R}(\rho_1, n, m)$.*

Démonstration. ρ_1 et ρ_2 étant totalement ordonnés, il n'est pas possible d'obtenir l'un à partir de l'autre en ajoutant de nouvelles contraintes d'ordre. L'unique possibilité est donc de décomposer un nœud de ρ_1 en appliquant une méthode de décomposition. □

La réduction étant l'unique transformation applicable au réseaux de tâches totalement ordonnés, il devient utile de s'intéresser à la façon dont cette transformation interagit avec la concaténation de réseaux de tâches. Entre autre, on peut montrer que la concaténation d'un réseau de tâches avec la réduction d'un second réseau de tâches est une réduction de la concaténation du premier réseau de tâches avec le second (à noter que ce résultat est valable pour tout réseau de tâches, et donc pas seulement pour des réseaux de tâches totalement ordonnés) :

Proposition 6. *Soient un réseau de tâches $\rho \in R_T$ non primitif, un nœud $n \in N_\rho$ tel que $t_n \in C$ est une méthode de décomposition $m \in M$ telle que $c_m = t_n$.*

$$\forall \rho' \in \varrho(T^*) : \mathcal{R}(\rho * \rho', n, m) = \mathcal{R}(\rho, n, m) * \rho' \wedge \mathcal{R}(\rho' * \rho, n, m) = \rho' * \mathcal{R}(\rho, n, m)$$

Démonstration. Soit $\rho' \in R_T$.
D'après la définition 17, on a d'une part :

$$\begin{aligned} N_{\mathcal{R}(\rho*\rho',n,m)} &= (N_{\rho*\rho'} \setminus \{n\}) \cup N_{\rho_m} \\ &= ((N_\rho \cup N_{\rho'}) \setminus \{n\}) \cup N_{\rho_m} \end{aligned}$$

et d'autre part :

$$\begin{aligned} \prec_{\mathcal{R}(\rho*\rho',n,m)} &= \{(n_1, n_2) \in \prec_{\rho*\rho'} \mid n_1 \neq n \wedge n_2 \neq n\} \\ &\quad \cup N_{\rho_m} \times \{n' \in N_{\rho*\rho'} \mid n' \prec_{\rho*\rho'} n_2\} \\ &\quad \cup \{n' \in N_{\rho*\rho'} \mid n_1 \prec_{\rho*\rho'} n\} \times N_{\rho_m} \\ &= \{(n_1, n_2) \in \prec_{\rho*\rho'} \mid n_1 \neq n \wedge n_2 \neq n\} \\ &\quad \cup N_{\rho_m} \times \{n' \in (N_\rho \cup N_{\rho'}) \mid n \prec_{\rho*\rho'} n'\} \\ &\quad \cup \{n' \in (N_\rho \cup N_{\rho'}) \mid n' \prec_{\rho*\rho'} n\} \times N_{\rho_m} \\ &= \{(n_1, n_2) \in \prec_{\rho*\rho'} \mid n_1 \neq n \wedge n_2 \neq n\} \\ &\quad \cup N_{\rho_m} \times \{n \in N_\rho \mid n' \prec_{\rho*\rho'} n_2\} \\ &\quad \cup N_{\rho_m} \times \{n \in N_{\rho'} \mid n' \prec_{\rho*\rho'} n_2\} \\ &\quad \cup \{n' \in N_\rho \mid n' \prec_{\rho*\rho'} n\} \times N_{\rho_m} \\ &\quad \cup \{n' \in N_{\rho'} \mid n' \prec_{\rho*\rho'} n\} \times N_{\rho_m} \end{aligned}$$

Or, $n \notin N_{\rho'}$, et $\prec_{\rho*\rho'} = \prec_\rho \cup \prec_{\rho'} \cup (N_\rho \times N_{\rho'})$, ce qui permet d'en déduire les éléments suivants :

$$\begin{aligned} (N_\rho \cup N_{\rho'}) \setminus \{n\} &= (N_\rho \setminus \{n\}) \cup N_{\rho'} \\ \{(n_1, n_2) \in \prec_{\rho*\rho'} \mid n_1 \neq n \wedge n_2 \neq n\} &= \{(n_1, n_2) \in \prec_\rho \mid n_1 \neq n \wedge n_2 \neq n\} \\ &\quad \cup \prec_{\rho'} \cup ((N_\rho \setminus \{n\}) \times N_{\rho'}) \\ \{n' \in N_\rho \mid n \prec_{\rho*\rho'} n'\} &= \{n' \in N_\rho \mid n \prec_\rho n'\} \\ \{n' \in N_{\rho'} \mid n \prec_{\rho*\rho'} n'\} &= N_{\rho'} \\ \{n' \in N_\rho \mid n' \prec_{\rho*\rho'} n\} &= \{n' \in N_\rho \mid n' \prec_\rho n\} \\ \{n' \in N_{\rho'} \mid n' \prec_{\rho*\rho'} n\} &= \emptyset \end{aligned}$$

On en conclut donc d'une part :

$$\begin{aligned} N_{\mathcal{R}(\rho*\rho',n,m)} &= (N_\rho \setminus \{n\}) \cup N_{\rho_m} \cup N_{\rho'} \\ &= N_{\mathcal{R}(\rho,n,m)} \cup N_{\rho'} \\ &= N_{\mathcal{R}(\rho,n,m)*\rho'} \end{aligned}$$

et d'autre part :

$$\begin{aligned}
\prec_{\mathcal{R}(\rho*\rho',n,m)} &= \{(n_1,n_2) \in \prec_\rho \mid n_1 \neq n \wedge n_2 \neq n\} \\
&\quad \cup \prec_{\rho'} \cup ((N_\rho \setminus \{n\}) \times N_{\rho'}) \\
&\quad \cup N_{\rho_m} \times \{n' \in N_\rho \mid n \prec_\rho n'\} \\
&\quad \cup N_{\rho_m} \times N_{\rho'} \\
&\quad \cup \{n' \in N_\rho \mid n' \prec_\rho n\} \times N_{\rho_m} \\
&\quad \cup \emptyset \times N_{\rho_m} \\
&= \{(n_1,n_2) \in \prec_\rho \mid n_1 \neq n \wedge n_2 \neq n\} \\
&\quad \cup N_{\rho_m} \times \{n' \in N_\rho \mid n \prec_\rho n'\} \\
&\quad \cup \{n' \in N_\rho \mid n' \prec_\rho n\} \times N_{\rho_m} \\
&\quad \cup \prec_{\rho'} \cup (((N_\rho \setminus \{n\}) \cup N_{\rho_m}) \times N_{\rho'}) \\
&= \prec_{\mathcal{R}(\rho,n,m)} \cup \prec_{\rho'} \cup (N_{\mathcal{R}(\rho,n,m)} \times N_{\rho'}) \\
&= \prec_{\mathcal{R}(\rho,n,m)*\rho'}
\end{aligned}$$

On a donc bien $\mathcal{R}(\rho * \rho', n, m) = \mathcal{R}(\rho, n, m) * \rho'$.
La preuve pour $\mathcal{R}(\rho' * \rho, n, m) = \rho' * \mathcal{R}(\rho, n, m)$ étant similaire, celle-ci est omise. □

On peut alors en déduire deux autres résultats essentiels pour la suite de ce manuscrit. Le premier énonce que si deux réseaux de tâches peuvent être respectivement atteints par une suite de décompositions à partir de deux réseaux de tâches initiaux, alors la concaténation de ces deux réseaux de tâches peut être atteinte par une suite de décompositions à partir de la concaténation des deux réseaux de tâches initiaux :

Proposition 7. *Soit* $(\rho_1, \rho_1, \rho_3, \rho_4) \in \varrho(T^*)^4$.
Si $\mathcal{D}^*(\rho_1, \rho_3)$ *et* $\mathcal{D}^*(\rho_2, \rho_4)$, *alors* $\mathcal{D}^*(\rho_1 * \rho_2, \rho_3 * \rho_4)$.

Démonstration. Commençons par montrer que la propriété suivante est satisfaite :

$$\forall(\rho, \rho', \rho'') \in \varrho(T^*)^3 : \mathcal{D}^*(\rho, \rho') \implies \mathcal{D}^*(\rho * \rho'', \rho' * \rho'') \wedge \mathcal{D}^*(\rho'' * \rho, \rho'' * \rho') \tag{4.3}$$

Soit $(\rho, \rho', \rho'') \in \varrho(T^*)^3$ tel que $\mathcal{D}^*(\rho, \rho')$.

1. Si $\rho = \rho'$:
   Alors $\rho'' * \rho = \rho'' * \rho$ et $\rho * \rho'' = \rho' * \rho''$, donc $\mathcal{D}^*(\rho'' * \rho, \rho'' * \rho') \wedge \mathcal{D}^*(\rho * \rho'', \rho' * \rho'')$.
2. Si $\exists r \in \varrho(T^*) : \mathcal{D}^*(\rho, r) \wedge \mathcal{D}(r, \rho')$: On suppose que $\mathcal{D}^*(\rho * \rho'', r * \rho'') \wedge \mathcal{D}^*(\rho'' * \rho, \rho'' * r)$. Comme $\mathcal{D}(r, \rho')$, on en déduit d'après la définition 32 qu'il existe $n \in N_r$ et $m \in M$ tels que $c_m = t_n$ et $\rho' = \mathcal{R}(r, n, m)$. D'après la proposition 6, on en déduit :

   $$\begin{aligned}
   \mathcal{R}(\rho'' * r, n, m) &= \rho'' * \mathcal{R}(r, n, m) \\
   &= \rho'' * \rho' \\
   \mathcal{R}(r * \rho'', n, m) &= \mathcal{R}(r, n, m) * \rho'' \\
   &= \rho' * \rho''
   \end{aligned}$$

   D'après la définition 19, on en déduit donc que $\mathcal{D}(\rho'' * r, \rho'' * \rho')$ et que $\mathcal{D}(r * \rho'', \rho' * \rho'')$. $\mathcal{D}^*$ étant transitive, on en déduit que :

   $$\begin{aligned}
   \mathcal{D}^*(\rho'' * \rho, \rho'' * r) \wedge \mathcal{D}(\rho'' * r, \rho'' * \rho') &\implies \mathcal{D}^*(\rho'' * \rho, \rho'' * \rho') \\
   \mathcal{D}^*(\rho * \rho'', r * \rho'') \wedge \mathcal{D}(r * \rho'', \rho' * \rho'') &\implies \mathcal{D}^*(\rho * \rho'', \rho' * \rho'')
   \end{aligned}$$

   On a donc bien $\mathcal{D}^*(\rho'' * \rho, \rho'' * \rho') \wedge \mathcal{D}^*(\rho * \rho'', \rho' * \rho'')$.

Par récurrence structurelle sur $\mathcal{D}^*$, on en déduit donc que 4.3 est satisfaite.
Par application de 4.3 on en déduit que :

$$\begin{aligned}\mathcal{D}^*(\rho_1, \rho_3) &\implies \mathcal{D}^*(\rho_1 * \rho_2, \rho_3 * \rho_2)\\ \mathcal{D}^*(\rho_2, \rho_4) &\implies \mathcal{D}^*(\rho_3 * \rho_2, \rho_3 * \rho_4)\end{aligned}$$

$\mathcal{D}^*$ étant une relation transitive, on obtient finalement $\mathcal{D}^*(\rho_1 * \rho_2, \rho_3 * \rho_2) \wedge \mathcal{D}^*(\rho_3 * \rho_2, \rho_3 * \rho_4) \implies \mathcal{D}^*(\rho_1 * \rho_2, \rho_3 * \rho_4)$. □

Le second résultat est similaire. Pour un réseau de tâches atteignable par une suite de décompositions à partir de la concaténation de deux réseaux de tâches initiaux, il établit l'existence de deux réseaux de tâches, chacun atteignable à partir des réseaux de tâches initiaux, et dont la concaténation correspond au premier réseau de tâches :

Proposition 8. *Soient deux réseaux de tâches totalement ordonnés ρ et ρ' tels que $\mathcal{D}^*(\rho, \rho')$.*
S'il existe $(\rho_1, \rho_2) \in \varrho(T^*)^2$ tel que $\rho = \rho_1 * \rho_2$, alors il existe $(\rho'_1, \rho'_2) \in \varrho(T^*)^2$ tel que $\mathcal{D}^*(\rho_1, \rho'_1) \wedge \mathcal{D}^*(\rho_2, \rho'_2) \wedge \rho' = \rho'_1 * \rho'_2$.

Démonstration. Supposons qu'il existe $(\rho_1, \rho_2) \in \varrho(T^*)^2$ tels que $\rho = \rho_1 * \rho_2$.

1. Si $\rho = \rho'$:
   En posant $\rho'_1 = \rho_1$ et $\rho'_2 = \rho_2$, on obtient bien $\mathcal{D}^*(\rho_1, \rho'_1) \wedge \mathcal{D}^*(\rho_2, \rho'_2) \wedge \rho' = \rho'_1 * \rho'_2$.
2. Si $\exists r \in \varrho(T^*) : \mathcal{D}^*(\rho, r) \wedge \mathcal{D}(r, \rho')$:
   On suppose qu'il existe $(r_1, r_2) \in \varrho(T^*)^2$ tel que $\mathcal{D}^*(\rho_1, r_1) \wedge \mathcal{D}^*(\rho_2, r_2) \wedge r = r_1 * r_2$.
   Comme $\mathcal{D}(r, \rho')$, on en déduit d'après la proposition 5 qu'il existe $n \in N_r$ et $m \in M$ tels que $c_m = t_n$ et $r = \mathcal{R}(\rho', n, m)$.
   (a) si $n \in N_{r_1}$:
   Alors $\mathcal{R}(r, n, m) = \mathcal{R}(r_1, n, m) * r_2$ d'après la proposition 6.
   En posant :

$$\begin{aligned}\rho'_1 &= \mathcal{R}(r_1, n, m)\\ \rho'_2 &= r_2\end{aligned}$$

   On obtient donc $\rho' = \rho'_1 * \rho'_2$ avec $\mathcal{D}^*(r_1, \rho'_1)$ et $\mathcal{D}^*(r_2, \rho'_2)$. $\mathcal{D}^*$ étant transitive, on en déduit que :

$$\begin{aligned}\mathcal{D}^*(\rho_1, r_1) \wedge \mathcal{D}^*(r_1, \rho'_1) &\implies \mathcal{D}^*(\rho_1, \rho'_1)\\ \mathcal{D}^*(\rho_2, r_2) \wedge \mathcal{D}^*(r_2, \rho'_2) &\implies \mathcal{D}^*(\rho_1, r_{i+1,1})\end{aligned}$$

   On a donc $\rho' = \rho'_1 * \rho'_2 \wedge \mathcal{D}^*(\rho_1, \rho'_1) \wedge \mathcal{D}^*(\rho_2, \rho'_2)$.
   (b) si $n \in N_{r_{i,2}}$:
   La démonstration est alors similaire au cas $n \in N_{r_{i,1}}$ et est donc omise.

Par récurrence structurelle, on en conclut donc qu'il existe $(\rho'_1, \rho'_2) \in \varrho(T^*)^2$ tel que $\mathcal{D}^*(\rho_1, \rho'_1) \wedge \mathcal{D}^*(\rho_2, \rho'_2) \wedge \rho' = \rho'_1 * \rho'_2$. □

Cette dernière proposition nous permet de démontrer le lemme suivant. Dans l'hypothèse où on considère des actions abstraites cohérentes, et des méthodes de décomposition dont tous les réseaux de tâches sont totalement ordonnés, on peut montrer que si un plan abstrait se décompose en un plan au niveau d'abstraction inférieur, alors l'exécutabilité en un état de ce dernier plan garantit l'exécutabilité du premier pour la version abstraite de cet état :

Lemme 1. *Soient un entier $n > 0$, un domaine de planification HTN avec hiérarchie d'abstractions $((D^0, f^0), ..., (D^n, f^n))$ totalement ordonné, un entier $k \in [\![1, n]\!]$, deux séquences de tâches $\sigma^k \in A^{k*}$ et $\sigma^{k-1} \in A^{k-1*}$ telles que $\mathcal{D}^*(\varrho(\sigma^k), \varrho(\sigma^{k-1}))$ ainsi qu'un état $s \in S^{k-1}$.*
Si σ^{k-1} est exécutable en s, alors σ^k est exécutable en $f^k(s)$ et $\mathcal{T}^k(f^k(s), \sigma^k) = f^k(\mathcal{T}^{k-1}(s, \sigma^{k-1}))$.

Démonstration. Supposons que σ^{k-1} est exécutable en s.
Soit $(\theta_1, ..., \theta_{|\sigma^k|}) \in A^{k|\sigma^k|}$ tel que $\sigma^k = \theta_1 ... \theta_{|\sigma^k|}$.
On peut déduire des propositions 3 et 8 qu'il existe $(\sigma_1, ..., \sigma_{|\sigma^k|}) \in A^{k-1*|\sigma^k|}$ tel que $\sigma^{k-1} = \sigma_1 \circ ... \circ \sigma_{|\sigma^k|}$ et tels que pour tout $i \in [\![1, |\sigma^k|]\!]$, $\mathcal{D}^*(\varrho(\theta_i), \varrho(\sigma_i))$.
Notons $s_0 = s$, et $s_i = \mathcal{T}^{k-1}(s_{i-1}, \sigma_{i-1})$ pour tout $i \in [\![1, |\sigma^k|]\!]$.
σ^{k-1} étant exécutable en s, on peut donc en déduire que pour tout $i \in [\![1, |\sigma^k|]\!]$, σ^i est exécutable en s_{i-1}.
Comme on suppose que toutes les actions abstraites sont cohérentes, on en déduit d'après la définition 31 que pour tout $i \in [\![1, |\sigma^k|]\!]$, θ_i est applicable en $f^k(s_{i-1})$ et que $\mathcal{T}^k(f^k(s_{i-1}), \theta_i) = f^k(\mathcal{T}^{k-1}(s_{i-1}, \sigma_i))$.
Or pour tout $i \in [\![1, |\sigma^k|]\!]$, $f^k(\mathcal{T}^{k-1}(s_{i-1}, \sigma_i)) = f^k(s_i)$, donc $\mathcal{T}^k(f^k(s_{i-1}), \theta_i) = f^k(s_i)$. On en déduit donc que σ^k est exécutable en $f^k(s)$ et que $\mathcal{T}^k(f^k(s), \sigma^k) = f^k(s_{|\sigma_k|})$. Comme $s_{|\sigma_k|} = \mathcal{T}^{k-1}(s, \sigma^{k-1})$, on en déduit finalement que $\mathcal{T}^k(f^k(s), \sigma^k) = f^k(\mathcal{T}^{k-1}(s, \sigma^{k-1}))$. □

Ce résultat peut être reproduit à travers toute une hiérarchie d'abstractions. On en déduit alors une propriété qui est le pendant de l'USP (propriété définie pour la planification par abstraction « classique ») pour la planification HTN avec hiérarchie d'abstractions :

Proposition 9. *Soient un entier $n > 0$, un domaine de planification HTN avec hiérarchie d'abstractions $((D^0, f^0), ..., (D^n, f^n))$ totalement ordonné, deux entiers $(k_1, k_2) \in [\![0, n]\!]^2$ tel que $k_2 > k_1$, deux séquences de tâches $\sigma^{k_1} \in A^{k_1*}$ et $\sigma^{k_2} \in A^{k_2*}$ telles que $\mathcal{D}^*(\varrho(\sigma^{k_2}), \varrho(\sigma^{k_1}))$, ainsi qu'un état $s \in S^{k_1}$.*
Si σ^{k_1} est exécutable en s, alors σ^{k_2} est exécutable en $f^{k_2} \circ ... \circ f^{k_1+1}(s)$ et $\mathcal{T}^{k_2}(f^{k_2} \circ ... \circ f^{k_1+1}(s), \sigma^{k_2}) = f^{k_2} \circ ... \circ f^{k_1+1}(\mathcal{T}^{k_1}(s, \sigma^{k_1}))$.

Démonstration. Se déduit du lemme 1 en raisonnant par récurrence sur k_2. □

Tout comme l'USP, cette propriété est à la base de la complétude pour les algorithmes de planification HTN avec hiérarchie d'abstractions.

ABPHTN

Il est maintenant temps d'introduire notre nouvel algorithme. Celui-ci se nomme simplement ABPHTN. Cet algorithme n'est plus un méta-planificateur reposant sur un planificateur HTN « boîte noir » comme l'est ABHTNPLAN, mais reprend les mécanismes de planification d'un algorithme de type PHTN (comme SHPE par exemple) et les intègre dans une procédure qui permet de gérer les différents niveaux d'exécution offerts par la hiérarchie d'abstractions. Tout comme ABHTNPLAN, ABPHTN prend en entrée un problème de planification HTN avec hiérarchie d'abstractions. Il prend aussi en entrée une suite $(h_1, ..., h_n)$. Chaque élément h_i de cette suite est appelé un « horizon » et correspond à la profondeur de plan maximale pouvant être atteinte avant de décomposer les actions abstraites contenues dans le plan au niveau i. Cette suite fait partie d'un mécanisme permettant de moduler la stratégie de décomposition des tâches. Lorsque tous les horizons valent 1, on obtient la stratégie de décomposition employée par les planificateurs de type PHTN, puisqu'ABPHTN décompose alors toujours la première action abstraite obtenue

avant de planifier les suivantes. À l'opposé, lorsque ces horizons ont des valeurs très élevées (tendant vers l'infini), la stratégie de décomposition des tâches suivie par ABPHTN tend vers celle de ABHTNPLAN, car une action abstraite n'est alors décomposée que lorsqu'un plan complet a été obtenu au niveau qui lui correspond.

Le choix de ne pas implémenter ABPHTN sous la forme d'un méta-planificateur permet de faciliter la gestion des accès aux différents niveaux qui, à cause de l'utilisation des horizons, ne se fait plus exclusivement de haut en bas, mais effectue des allers et des retours successifs. Ainsi le planificateur « descend » dans la hiérarchie lorsqu'il atteint l'horizon au niveau courant, ou si les réseaux de tâches à ce niveau et aux niveaux supérieurs sont vides. Inversement, il « remonte » la hiérarchie lorsque le réseau de tâches est vide et qu'il peux obtenir de nouvelles tâches à partir d'un niveau supérieur. Lorsque l'horizon n'est pas atteint, et qu'il reste des tâches à accomplir au niveau courant, ABPHTN se comporte alors comme un planificateur PHTN à ce niveau. L'algorithme 11 présente ABPHTN en détails. La notation $\min \rho$ y représente le premier nœud d'un réseau de tâches totalement ordonné ρ. Cette notation est utilisée dans ce sens dans la suite de ce manuscrit.

Nous proposons maintenant une analyse en détails d'ABPHTN, et en démontrons, sous certaines conditions, les propriétés de terminaison, de correction et de complétude. Pour cela nous considérons un entier strictement positif n, ainsi qu'un domaine de planification HTN avec hiérarchie d'abstractions de taille n que l'on note $((D^0, f^0), ..., (D^n, f^n))$, où pour chaque $i \in [\![0, n]\!]$, $D^i = (S^i, A^i, C^i, M^i, \mathcal{T}, \mathcal{C})$. On considère que chaque méthode y est associée à un réseau de tâches totalement ordonné, et que chaque action abstraite y est cohérente. On regroupe aussi l'ensemble des tâches de ce domaine dans un ensemble noté $T = \bigcup_{i \in [\![0,n]\!]} A^i \cup C^i$. Enfin, on considère une suite de n horizons $(h_0, ..., h_n)$, où pour tout $i \in [\![1, n]\!]$, l'horizon h_i est un entier strictement positif.

L'espace de recherche exploré par ABPHN est un espace de plans. Ces plans sont représentés pas une structure que nous appelons une « hiérarchisation ». Un plan est réparti sur $n + 1$ niveaux en fonction de l'appartenance des tâches qui le composent aux différents niveaux d'abstraction. De plus, les tâches situées à chaque niveau doivent précéder les tâches situées au niveau supérieur. En conséquence, seuls les plans dont le niveau d'abstraction des tâches est croissant par rapport à leur ordre dans le plan peuvent être représentés sous la forme d'une hiérarchisation. De plus, les tâches sont aussi réparties parmi deux éléments à chaque niveau : une séquence d'actions abstraites dont on a déjà contrôlé l'exécution d'une part, et un réseau de tâche à planifier d'autre part. Enfin, l'état atteint par l'exécution des actions abstraites est aussi maintenu à chaque niveau. Un exemple de hiérarchisation est représenté en figure 4.6.

Lorsque toutes les séquences d'actions et réseaux de tâches contenus dans une hiérarchisation sont concaténées, on obtient un réseau de tâches qui représente le plan complet auquel correspond cette hiérarchisation.

Définition 34 (Hiérarchisation d'un réseau de tâche)**.** *Soit $\rho \in \varrho(T^*)$. Une hiérarchisation de ρ est un $(n+1)$-uplet $((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n)) \in \prod_{i=0}^{n} S^i \times A^{i^*} \times \varrho(A^i \cup C^i)$ tel que $\rho = \varrho(\sigma^0) * \rho^0 * ... * \varrho(\sigma^n) * \rho^n$.*
Pour tout réseau de tâches $\rho \in \varrho(T^*)$, on note H_ρ l'ensemble des hiérarchisations de ρ, et on note $H = \bigcup_{\rho \in \varrho(T^*)} H_\rho$ l'ensemble de toutes les hiérarchisations.

On dit qu'une hiérarchisation est primitive lorsque que toutes les tâches qu'elle contient sont regroupées dans la séquence d'actions correspondant au niveau concret.

Définition 35 (Hiérarchisation primitive)**.** *Soit $\rho \in \varrho(T^*)$ et $\eta = ((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n))$ une hiérarchisation de ρ. On dit que η est primitive si et seulement si :*

$$\{i \in [\![0, n]\!] \mid \rho^i \neq \emptyset \vee (i > 0 \wedge \sigma^i \neq \epsilon)\} = \emptyset$$

Algorithm 11 Algorithme ABPHTN

entrées : Un problème de planification HTN avec hiérarchie d'abstractions $P = (((D^0, f^0), ..., (D^n, f^n)), s_0, \rho_0)$ et une suite d'entiers strictement positifs $(h^1, ..., h^n)$.

sorties : un plan solution de P

$Ouvert \leftarrow \{(f^i \circ ... \circ f^0(s_0), \epsilon, \rho_0 \textbf{ si } i = n \textbf{ sinon } (\emptyset, \emptyset))_{i \in [\![0,n]\!]}\}$
$Fermé \leftarrow \emptyset$
tant que $Ouvert \neq \emptyset$ **faire**
  $((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n)) \leftarrow$ extraire un élément de $Ouvert$
  $Fermé \leftarrow Fermé \cup \{(\sigma^i, \rho^i)_{i \in [\![0,n]\!]}\}$
  si $\{i \in [\![0, n]\!] \mid \rho^i \neq \emptyset \vee (i > 0 \wedge \sigma^i \neq \epsilon)\} = \emptyset$ **alors**
    retourner σ^0
  sinon
    $k \leftarrow min\{i \in [\![0, n]\!] \mid \rho^i \neq \emptyset \vee (i > 0 \wedge \sigma^i \neq \epsilon)\}$
    si $k > 0$ **et** $(|\sigma^k| \geq h^k$ **ou** $\rho^k = \emptyset)$ **alors**
      $\rho^{k-1} \leftarrow \varrho(\sigma^k)$
      $\sigma^k \leftarrow \epsilon$
      si $((\sigma^0, \rho^0), ..., (\sigma^n, \rho^n)) \notin Fermé$ **alors**
        $Ouvert \leftarrow Ouvert \cup \{((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n))\}$
      fin si
    sinon
      $s, \sigma, \rho \leftarrow s^k, \sigma^k, \rho^k$
      si $t_{\min \rho} \in A^k$ **alors**
        si $t_{\min \rho}$ est applicable en s **alors**
          $s^k, \sigma^k, \rho^k \leftarrow \mathcal{T}^k(s, t_{\min \rho}), \sigma \circ t_{\min \rho}, (N_\rho \setminus \{\min \rho\}, \prec_\rho \setminus (\{\min \rho\} \times N_\rho))$
          si $((\sigma^0, \rho^0), ..., (\sigma^n, \rho^n)) \notin Fermé$ **alors**
            $Ouvert \leftarrow Ouvert \cup \{((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n))\}$
          fin si
        fin si
      sinon
        pour toute méthode $m \in M^k$ telle que $c_m = t_{\min \rho}$ **faire**
          $\rho^k \leftarrow \mathcal{R}^k(\rho, \min \rho, m)$
          si $((\sigma^0, \rho^0), ..., (\sigma^n, \rho^n)) \notin Fermé$ **alors**
            $Ouvert \leftarrow Ouvert \cup \{((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n))\}$
          fin si
        fin pour
      fin si
    fin si
  fin si
fin tant que
retourner *échec*

Si la hiérarchisation d'un réseau de tâches est primitive, on peut alors en déduire que ce réseau de tâches est primitif, puisque toutes les tâches qu'il contient sont alors nécessairement des tâches primitives :

Proposition 10. *Soit $\rho \in \varrho(T^*)$ et $\eta = ((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n))$ une hiérarchisation de ρ.*
Si η est primitive, alors ρ est un réseau de tâches primitif.

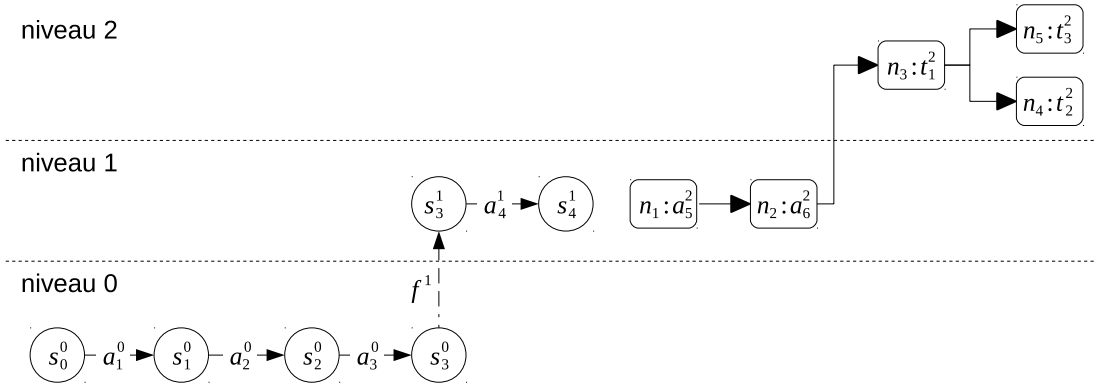


FIGURE 4.6 – Plan multi-niveaux
Illustration d'un plan structuré par rapport à une hiérarchie d'abstractions à trois niveaux dans ABPHTN. Chaque niveau est composé d'une séquence d'actions exécutée et d'un réseau de tâches. Ce réseau de tâches peut être vide, comme au niveau 0. Le réseau de tâches du niveau 1 ne contient que des actions abstraites du niveau 2, car il provient directement de la dernière séquence d'actions exécutée au niveau 2.

Démonstration. η est primitive, donc d'après la définition 35, on a :

$$\sigma^i = \epsilon \quad \forall i \in [\![1, n]\!]$$
$$\rho^i = \emptyset \quad \forall i \in [\![0, n]\!]$$

D'après la définition 34, on en déduit donc que $\rho = \varrho(\sigma^0)$. Or $\sigma^0 \in A^{0*}$, donc $\rho \in R_{A^0}$. ρ est donc un réseau de tâches primitif. □

Une hiérarchisation primitive correspond donc à une solution potentielle, et constitue donc un élément final de l'espace de recherche d'ABPHTN. Tout autre hiérarchisation doit donc pouvoir subir une transformation permettant d'explorer cet espace de recherche. Pour toute hiérarchisation non primitive, il existe donc des niveaux pour lesquels les réseaux de tâches à planifier ou les séquences d'actions abstraites ne sont pas vides. On appelle alors niveau courant le plus petit de ces niveaux. Il s'agit du niveau auquel ABPHTN opère à chaque étape de planification.

Le premier type de transformation appliqué par ABPHTN est nommé « concrétisation ». La concrétisation consiste à convertir les actions abstraites exécutées au niveau courant en tâches à planifier au niveau inférieur. Pour que cette transformation soit applicable, il est donc nécessaire que le niveau courant ne corresponde pas au niveau concret. De plus, cette transformation n'est applicable par principe que dans deux conditions : soit il n'existe plus de tâches à planifier au niveau courant, soit le nombre d'actions abstraites exécutées a atteint la valeur définie par l'horizon.

Définition 36 (Concrétisation d'une hiérarchisation)**.** *Soient* $\eta_1 = ((s_1^0, \sigma_1^0, \rho_1^0), ..., (s_1^n, \sigma_1^n, \rho_1^n))$ *et* $\eta_2 = ((s_2^0, \sigma_2^0, \rho_2^0), ..., (s_2^n, \sigma_2^n, \rho_2^n))$ *deux hiérarchisations telles que* η_1 *est non primitive. On note de plus* $k = \min\{i \in [\![0, n]\!] \mid \rho_1^i \neq \emptyset \vee (i > 0 \wedge \sigma_1^i \neq \epsilon)\}$.
On dit que η_2 *est une concrétisation de* η_1*, noté* $\mathcal{K}(\eta_1, \eta_2)$*, si et seulement si :*

1. $k > 0 \wedge (|\sigma_1^k| \geq h_k \vee \rho_1^k = \emptyset)$.
2. $\forall i \in [\![0, n]\!] : s_1^i = s_2^i$.
3. $\forall i \in [\![0, n]\!] \setminus \{k-1, k\} : \sigma_1^i = \sigma_2^i \wedge \rho_1^i = \rho_2^i$
4. $\sigma_2^k = \epsilon \wedge \rho_1^{k-1} = \emptyset \wedge \rho_2^k = \rho_1^k \wedge \sigma_1^{k-1} = \sigma_2^{k-1} \wedge \rho_2^{k-1} = \varrho(\sigma_1^k)$.

Le second type de transformation consiste à appliquer les mécanismes de la planification HTN dans les espaces de progression. Ces transformations s'appliquent lorsque le réseau de

tâches à planifier contient des éléments au niveau courant. On considère alors le premier élément de ce réseau de tâche (celui-ci étant totalement ordonné, il contient donc un unique élément minimal). Si cet élément correspond à une action abstraite, alors celui-ci est supprimé du réseau de tâches et est ajoutée à la séquence d'actions déjà construite (dans l'hypothèse où cette action est bien applicable). Si au contraire cet élément est associé à une tâche composée pour le niveau courant, on choisit alors une méthode de décomposition permettant de décomposer ce nœud afin de réduire le réseau de tâches.

Définition 37 (Progression d'une hiérarchisation)**.** *Soient* $\eta_1 = ((s_1^0, \sigma_1^0, \rho_1^0), ..., (s_1^n, \sigma_1^n, \rho_1^n))$ *et* $\eta_2 = ((s_2^0, \sigma_2^0, \rho_2^0), ..., (s_2^n, \sigma_2^n, \rho_2^n))$ *deux hiérarchisations telles que* η_1 *est non primitive.*
On note de plus $k = \min\{i \in [\![0, n]\!] \mid \rho_1^i \neq \emptyset \vee (i > 0 \wedge \sigma_1^i \neq \epsilon)\}$.
On dit que η_2 *est une progression de* η_1*, noté* $\mathcal{P}(\eta_1, \eta_2)$*, si et seulement si :*

1. $(k = 0 \vee |\sigma_1^k| < h_k) \wedge \rho^k \neq \emptyset$.
2. $\forall i \in [\![0, n]\!] \setminus \{k\} : s_1^i = s_2^i \wedge \sigma_1^i = \sigma_2^i \wedge \rho_1^i = \rho_2^i$.
3. *Si* $t_{\min \rho_1^k} \in A^k$ *:*
   (a) $t_{\min \rho_1^k}$ *est applicable en* s_1^k *et* $s_2^k = \mathcal{T}^k(s_1^k, t_{\min \rho_1^k})$.
   (b) $\rho_1^k = \varrho(t_{\min \rho_1^k}) * \rho_2^k$.
   (c) $\sigma_2^k = \sigma_1^k \circ t_{\min \rho_1^k}$.
   Sinon ($t_{\min \rho_1^k} \in C^k$) *:*
   (a) $s_2^k = s_1^k$.
   (b) $\sigma_2^k = \sigma_1^k$.
   (c) $\exists m \in M^k : c_m = t_{\min \rho_1^k} \wedge \rho_2^k = \mathcal{R}(\rho_1^k, \min \rho_1^k, m)$.

La concrétisation et la progression au niveau courant forment les deux types de transformations appliquées par ABPHTN aux hiérarchisations. On introduit alors une relation, que l'on note Θ, et qui définit l'ensemble des couples de hiérarchisations dont la seconde s'obtient à partir d'une transformation de la première.

Définition 38 (Transformation d'une hiérarchisation)**.** *Soient* η_1 *et* η_2 *deux hiérarchisations tels que* η_1 *est non primitive.*
On dit que η_2 *est une transformation de* η_1*, noté* $\Theta(\eta_1, \eta_2)$*, si et seulement si* $\mathcal{K}(\eta_1, \eta_2) \vee \mathcal{P}(\eta_1, \eta_2)$.

L'espace de recherche de ABPHTN correspond ainsi au graphe (H, Θ). À nouveau, on utilise la notation Θ^* pour représenter la clôture transitive de Θ. Ainsi pour toute paire de hiérarchisations $(\eta_1, \eta_2) \in H^2$ on écrit que $\Theta^*(\eta_1, \eta_2)$ si et seulement si $\eta_1 = \eta_2$ ou qu'il existe une hiérarchisation $\eta_3 \in H$ telle que $\Theta^*(\eta_1, \eta_3)$ et $\Theta(\eta_3, \eta_2)$.

On peut alors montrer qu'à partir d'une hiérarchisation initiale, toute hiérarchisation atteignable dans l'espace de recherche d'ABPHTN correspond à un réseau de tâches atteignable, dans l'espace de décomposition des réseaux de tâches, à partir du réseau de tâches associé à la hiérarchisation initiale :

Proposition 11. *Soient* $(\rho_1, \rho_2) \in \varrho(T^*)$*, tel qu'il existe une hiérarchisation de* ρ_1*, notée* η_1*, ainsi qu'une hiérarchisation de* ρ_2*, notée* η_2.
Si $\Theta^*(\eta_1, \eta_2)$*, alors* $\mathcal{D}^*(\rho_1, \rho_2)$.

Démonstration. On note $\eta_1 = ((s_1^0, \sigma_1^0, \rho_1^0), ..., (s_1^n, \sigma_1^n, \rho_1^n))$ et $\eta_2 = ((s_2^0, \sigma_2^0, \rho_2^0), ..., (s_2^n, \sigma_2^n, \rho_2^n))$.

1. Si $\eta_1 = \eta_2$:
   Dans ce cas, on en déduit d'après la définition 34 que $\rho_1 = \rho_2$, et donc que $\mathcal{D}^*(\rho_1, \rho_2)$.
2. Si $\exists \eta_3 \in H : \Theta^*(\eta_1, \eta_3) \wedge \Theta(\eta_3, \eta_2)$:
   On note alors ρ_3 le réseau de tâches totalement ordonné dont η_3 est une hiérarchisation, et on note $\eta_3 = ((s_3^0, \sigma_3^0, \rho_3^0), ..., (s_3^n, \sigma_3^n, \rho_3^n))$ On suppose que $\mathcal{D}^*(\rho_1, \rho_3)$.
   Comme $\mathcal{D}(\eta_3, \eta_2)$, on en déduit d'après la définition 38 que $\mathcal{K}(\eta_3, \eta_2) \vee \mathcal{P}(\eta_3, \eta_2)$.
   D'après les définitions 36 et 37, on peut en déduire que η_3 n'est pas primitive.
   On note alors $k = \min\{i \in [\![0, n]\!] \mid \rho_3^i \neq \emptyset \vee (i > 0 \wedge \sigma_3^i \neq \epsilon)\}$.
   (a) Supposons que $\mathcal{K}(\eta_3, \eta_2)$:
   D'après la définition 36, on a $k > 0 \wedge \forall i \in [\![0, n]\!] \setminus \{k-1, k\} : \sigma_3^i = \sigma_2^i \wedge \rho_3^i = \rho_2^i$.
   On en déduit donc d'après la régularité de $(\varrho(T^*), *, \emptyset)$ à gauche et à droite que :

$$\rho_3 = \rho_2 \iff \varrho(\sigma_3^{k-1}) * \rho_3^{k-1} * \varrho(\sigma_3^k) * \rho_3^k = \varrho(\sigma_2^{k-1}) * \rho_2^{k-1} * \varrho(\sigma_2^k) * \rho_2^k$$

   Or, toujours d'après la définition 36, on a aussi $\sigma_2^k = \epsilon \wedge \rho_3^{k-1} = \emptyset \wedge \rho_2^k = \rho_3^k \wedge \sigma_2^{k-1} = \sigma_3^{k-1} \wedge \rho_2^{k-1} = \varrho(\sigma_3^k)$.
   On en déduit donc que :

$$\begin{aligned}
\rho_3 = \rho_2 &\iff \varrho(\sigma_3^{k-1}) * \rho_3^{k-1} * \varrho(\sigma_3^k) * \rho_3^k = \varrho(\sigma_2^{k-1}) * \rho_2^{k-1} * \varrho(\sigma_2^k) * \rho_2^k \\
&\iff \varrho(\sigma_3^{k-1}) * \emptyset * \varrho(\sigma_3^k) * \rho_3^k = \varrho(\sigma_3^{k-1}) * \varrho(\sigma_3^k) * \emptyset * \rho_3^k \\
&\iff \varrho(\sigma_3^{k-1}) * \varrho(\sigma_3^k) * \rho_3^k = \varrho(\sigma_3^{k-1}) * \varrho(\sigma_3^k) * \emptyset * \rho_3^k
\end{aligned}$$

   On en conclut donc que $\rho_3 = \rho_2$, et donc que $\mathcal{D}^*(\rho_3, \rho_2)$.
   (b) Supposons que $\mathcal{P}(\eta_3, \eta_2)$:

   i. Si $t_{\min \rho_3^k} \in A^k$:
   D'après la définition 35, on a $\forall i \in [\![0, n]\!] \setminus \{k\} : \sigma_3^i = \sigma_2^i \wedge \rho_3^i = \rho_2^i$.
   On en déduit donc d'après la régularité de $(\varrho(T^*), *, \emptyset)$ à gauche et à droite que :

$$\rho_3 = \rho_2 \iff \varrho(\sigma_3^k) * \rho_3^k = \varrho(\sigma_2^k) * \rho_2^k$$

   Or, toujours d'après la définition 36, on a aussi $\rho_3^k = \varrho(t_{\min \rho_3^k}) * \rho_2^k$ et $\sigma_2^k = \sigma_3^k \circ t_{\min \rho_3^k}$. On en déduit donc que :

$$\begin{aligned}
\rho_3 = \rho_2 &\iff \varrho(\sigma_3^k) * \rho_3^k = \varrho(\sigma_2^k) * \rho_2^k \\
&\iff \varrho(\sigma_3^k) * \varrho(t_{\min \rho_3^k}) * \rho_2^k = \varrho(\sigma_3^k \circ t_{\min \rho_3^k}) * \rho_2^k \\
&\iff \varrho(\sigma_3^k) * \varrho(t_{\min \rho_3^k}) * \rho_2^k = \varrho(\sigma_3^k) * \varrho(t_{\min \rho_3^k}) * \rho_2^k
\end{aligned}$$

   On en conclut donc que $\rho_3 = \rho_2$, et donc que $\mathcal{D}^*(\rho_3, \rho_2)$.
   ii. Sinon ($t_{\min \rho_3^k} \in C^k$) :
   Notons :

$$\begin{aligned}
\rho_g &= \varrho(\sigma_3^0) * \rho_3^0 * ... \varrho(\sigma_3^{k-1}) * \rho_3^{k-1} * \varrho(\sigma_3^k) \\
\rho_d &= \varrho(\sigma_3^{k+1}) * \rho_3^{k+1} * ... \varrho(\sigma_3^n) * \rho_3^n
\end{aligned}$$

   D'après la définition 36, il existe une méthode de décomposition $m \in M^k$ telle que :

$$\begin{aligned}
\rho_3 &= \rho_g * \rho_3^k * \rho_d \\
\rho_2 &= \rho_g * \mathcal{R}(\rho_3^k, \min \rho_3^k, m) * \rho_d
\end{aligned}$$

D'après la proposition 6, on en déduit donc que :

$$\begin{aligned}\rho_2 &= \rho_g * \mathcal{R}(\rho_3^k, \min \rho_3^k, m) * \rho_d \\ &= \mathcal{R}(\rho_g * \rho_3^k * \rho_d, \min \rho_3^k, m) \\ &= \mathcal{R}(\rho_3, \min \rho_3^k, m)\end{aligned}$$

D'après la définition 19, on en déduit donc que $\mathcal{D}(\rho_3, \rho_2)$, ce qui implique que $\mathcal{D}^*(\rho_3, \rho_2)$.

On a donc $\mathcal{D}^*(\eta_3, \eta_2)$. Comme on a aussi par hypothèse que $\mathcal{D}^*(\eta_1, \eta_3)$, et que $\mathcal{D}^*$ est une relation transitive, on en déduit donc que $\mathcal{D}^*(\eta_1, \eta_2)$.

Par récurrence structurelle sur Θ^*, on en déduit alors que $\Theta^*(\eta_1, \eta_2) \implies \mathcal{D}^*(\rho_1, \rho_2)$. □

Par l'intermédiaire de ce résultat, on peut montrer qu'ABPHTN termine lorsque l'espace de décompositions du problème est fini. En effet, on peut alors montrer que pour chaque réseau de tâches accessible, il existe un nombre fini de hiérarchisations correspondantes accessibles.

Théorème 3 (Terminaison de ABPHTN)**.** *Soit un problème de planification avec hiérarchie d'abstractions* $P = (((D^0, f^0), ..., (D^n, f^n)), s_0, \rho_0)$. *Si* $\{\rho \in \varrho(T^*) \mid \mathcal{D}^*(\rho_0, \rho)\}$ *est fini, alors la résolution de* P *par ABPHTN termine.*

Démonstration. On note $\eta_0 = ((f^0(s_0), \epsilon, \emptyset), ..., (f^{n-1} \circ ... \circ f^0(s_0), \epsilon, \emptyset), (f^n \circ ... \circ f^0(s_0), \epsilon, \rho_0))$ la hiérarchisation de ρ_0 qui correspond au premier élément visité par ABPHTN.
Pour toute hiérarchisation $\eta = ((s^0, \sigma^0, \rho^0), ...(s^n, \sigma^n, \rho^n))$, on note aussi $\chi(\eta) = ((\sigma^0, \rho^0), ..., (\sigma^n, \rho^n))$.
L'ensemble $Fermé$ permet de s'assurer qu'il n'est pas possible de visiter deux hiérarchisations dont l'image par χ est identique. En conséquence, il suffit de montrer que $\chi(\{\eta \in H \mid \Theta^*(\eta_0, \eta)\})$ est un ensemble fini pour s'assurer qu'ABPHTN termine.
D'après la proposition 11, on a :

$$\begin{aligned}\{\eta \in H \mid \Theta^*(\eta_0, \eta)\} &= \bigcup_{\rho \in \varrho(T^*)} \{\eta \in H_\rho \mid \Theta^*(\eta_0, \eta)\} \\ &\subseteq \bigcup_{\rho \in \varrho(T^*)} \{\eta \in H_\rho \mid \mathcal{D}^*(\rho_0, \rho)\} \\ &\subseteq \bigcup_{\rho \in \varrho(T^*) \mid \mathcal{D}^*(\rho_0, \rho)} H_\rho\end{aligned}$$

On en déduit donc :

$$\chi(\{\eta \in H \mid \Theta^*(\eta_0, \eta)\}) \subseteq \bigcup_{\rho \in \varrho(T^*) \mid \mathcal{D}^*(\rho_0, \rho)} \chi(H_\rho)$$

$\{\rho \in \varrho(T^*) \mid \mathcal{D}^*(\rho_0, \rho)\}$ étant un ensemble fini par hypothèse, il suffit donc de démontrer que $\chi(H_\rho)$ est fini pour tout $\rho \in \varrho(T^*)$.
Soit $\rho \in \varrho$. Notons $T_\rho = \{t_{n_0} \mid n_0 \in N_\rho\}$.
On peut alors montrer que :

$$\chi(H_\rho) \subseteq (\bigcup_{i=0}^{|N_\rho|} {T_\rho}^i) \times \varrho(\bigcup_{i=0}^{|N_\rho|} {T_\rho}^i)$$

Or, $\bigcup_{i=0}^{|N_\rho|} {T_\rho}^i$ est un ensemble fini, donc $\chi(H_\rho)$ est bien un ensemble fini. On en conclut donc qu'ABPHTN termine. □

Ainsi, on obtient la garantie qu'ABPHTN termine dans les conditions qui garantissent que l'espace de décomposition du problème est fini. D'après [3], une condition nécessaire est suffisante consiste à ce que chaque réseau de tâches défini dans le problème de planification HTN soit « $\leq_1$ $-stratifiable$ ». Si le problème vérifie cette propriété, on obtient la garantie que ABPHTN retourne une solution. Cependant, nous ne sommes pas en mesure d'affirmer que cette condition est nécessaire pour garantir la terminaison d'ABPHTN, mais seulement qu'elle est suffisante.

Nous nous intéressons maintenant à la correction d'ABPHTN. On cherche donc à savoir si toute objet retourné par cet algorithme est bien une solution du problème de planification posé. ABPHTN retourne une solution lorsqu'il atteint une hiérarchisation primitive, et retourne alors la séquence de tâches primitive associée. Pour s'assurer que cette séquence est une solution, il faut donc s'assurer que celle-ci est bien une décomposition du réseau de tâches initiale, et qu'elle est exécutable dans l'état initial. La première de ces deux conditions a déjà été démontrée précédemment, mais il faut encore justifier la seconde. Pour cela, nous introduisons une notion d'exécutabilité associée aux hiérarchisations. L'une des conséquence de l'utilisation d'une hiérarchie d'abstractions est de permettre de contrôler l'exécutabilité d'un plan avant d'atteindre le niveau primitif. On considère donc qu'une hiérarchisation est exécutable si les états qu'elle contient sont bien le résultat de l'exécution des actions abstraites applicables obtenu au cours des transformations qui lui ont été appliquées.

Définition 39 (Hiérarchisation exécutable)**.** *Soient une hiérarchisation* $\eta = ((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n))$ *et un état* $s \in S^0$*. On dit que* η *est exécutable en* s *si et seulement si il existe* $(\sigma_0, ..., \sigma_n) \in A^{0*n+1}$ *tel que :*

1. $\mathcal{D}^*(\varrho(\sigma_0), \varrho(\sigma^0))$.
2. $\forall i \in [\![1, n]\!] : \mathcal{D}^*(\varrho(\sigma_i), \varrho(\sigma^0) * \rho^0 * ... * \varrho(\sigma^{i-1}) * \rho^{i-1} * \varrho(\sigma^i))$.
3. $\forall i \in [\![0, n]\!] : \sigma_i$ *est exécutable en* $f^i \circ ... \circ f^0(s) \wedge s^i = \mathcal{T}^i(f^i \circ ... \circ f^0(s), \sigma_i)$.

On peut alors montrer que toute hiérarchisation accessible à partir d'une hiérarchisation exécutable est elle-même une hiérarchisation exécutable :

Proposition 12. *Soient un état* $s \in S^0$ *et* $(\eta_1, \eta_2) \in H^2$ *tel que* $\Theta^*(\eta_1, \eta_2)$.
Si η_1 *est exécutable en* s*, alors* η_2 *est exécutable en* s.

Démonstration. On suppose que η_1 est exécutable en s.

1. Si $\eta_1 = \eta_2$:
   η_1 est exécutable en s, donc η_2 est exécutable en s.
2. Si $\exists \eta_3 \in H : \Theta^*(\eta_1, \eta_3) \wedge \Theta(\eta_3, \eta_2)$:
   On suppose que η_3 est exécutable en s.
   On note $\eta_2 = ((s_2^0, \sigma_2^0, \rho_2^0), ..., (s_2^n, \sigma_2^n, \rho_2^n))$ et $\eta_2 = ((s_3^0, \sigma_3^0, \rho_3^0), ..., (s_3^n, \sigma_3^n, \rho_3^n))$.
   Comme $\Theta(\eta_3, \eta_2)$, on en déduit d'après la définition 38 que $\mathcal{K}(\eta_1, \eta_2) \vee \mathcal{P}(\eta_1, \eta_2)$.
   D'après les définitions 36 et 37, on en déduit aussi que η_3 n'est pas primitive.
   On note alors $k = \min\{i \in [\![0, n]\!] \mid \rho_3^i \neq \emptyset \vee (i > 0 \wedge \sigma_3^i \neq \epsilon)\}$.
   Comme η_3 est exécutable en s, on en déduit d'après la définition 39 qu'il existe $(\sigma_0, ..., \sigma_n) \in A^{0*n+1}$ tel que :

$$\mathcal{D}^*(\varrho(\sigma_0), \varrho(\sigma_3^0))$$
$$\forall i \in [\![1, n]\!] : \mathcal{D}^*(\varrho(\sigma_i), \varrho(\sigma_3^0) * \rho_3^0 * ... * \varrho(\sigma_3^{i-1}) * \rho_3^{i-1} * \varrho(\sigma_3^i))$$
$$\forall i \in [\![0, n]\!] : s_3^i = \mathcal{T}^i(f^i \circ ... \circ f^0(s), \sigma_i)$$

(a) Supposons que $\mathcal{K}(\eta_3, \eta_2)$:
D'après la définition 36, on a :

$$\begin{aligned} \sigma_3^0 = \sigma_3^0 &\implies \mathcal{D}^*(\varrho(\sigma_0), \varrho(\sigma_2^0)) \\ \forall i \in [\![0, n]\!] : s_2^i = s_2^i &\implies \sigma_i \text{ est exécutable en } f^i \circ^0 (s) \wedge s_2^i = \mathcal{T}^i(f^i \circ f^0(s), \sigma_i) \end{aligned}$$

De plus, on a :

$$\begin{aligned} (\forall i \in [\![1, n]\!] \setminus \{k-1, k\} : \sigma_3^i = \sigma_2^i \wedge \rho_3^i = \rho_2^i) &\implies \\ (\forall i \in [\![0, n]\!] \setminus \{k\} : \rho_2^{i-1} * \varrho(\sigma_2^i) = \rho_3^{i-1} * \varrho(\sigma_3^i) & \end{aligned}$$

Ainsi que :

$$\rho_3^{k-1} = \emptyset \wedge \sigma_2^k = \epsilon \wedge \rho_2^{k-1} = \varrho(\sigma_3^k) \implies \rho_2^{k-1} * \varrho(\sigma_2^k)) = \rho_3^{k-1} * \varrho(\sigma_3^k))$$

On en déduit donc que :

$$\forall i \in [\![1, n]\!] : \mathcal{D}^*(\varrho(\sigma_i), \varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{i-1}) * \rho_2^{i-1} * \varrho(\sigma_2^i))$$

D'après la définition 39, on en déduit donc que η_2 est exécutable en s.

(b) Supposons que $\mathcal{P}(\eta_3, \eta_2)$:

i. Si $t_{\min \rho_3^k} \in A^k$:
On note :

$$\begin{aligned} \sigma'_k &= \sigma_k \circ t_{\min \rho_3^k} \\ \sigma'_i &= \sigma_i \quad \forall i \in [\![0, n]\!] \setminus \{k\} \end{aligned}$$

D'après la définition 36, on a :

$$\begin{aligned} \sigma_2^i = \sigma_3^i \wedge \rho_2^i = \rho_3^i \quad \forall i \in [\![0, n]\!] \setminus \{k\} \\ \sigma_2^k = \sigma_3^k \circ t_{\min \rho_3^k} \wedge \rho_3^k = \varrho(t_{\min \rho_3^k}) \circ \rho_2^k \end{aligned}$$

Pour tout $i \in [\![1, n]\!] \setminus \{k\}$, on en déduit donc :

$$\varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{i-1}) * \rho_2^{i-1} * \varrho(\sigma_2^i) = \varrho(\sigma_3^0) * \rho_3^0 * ... * \varrho(\sigma_3^{i-1}) * \rho_3^{i-1} * \varrho(\sigma_3^i)$$

Et donc que :

$$\forall i \in [\![1, n]\!] \setminus \{k\} : \mathcal{D}^*(\varrho(\sigma'_i), \varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{i-1}) * \rho_2^{i-1} * \varrho(\sigma_2^i))$$

De plus, si $k \neq 0$, on a aussi que $\varrho(\sigma_2^0) = \varrho(\sigma_3^0)$, et donc que $\mathcal{D}^*(\varrho(\sigma_0), \varrho(\sigma_2^0))$.
Notons $\rho = \varrho(\sigma_3^0) * \rho_3^0 * ... * \varrho(\sigma_3^{k-1}) * \rho_3^{k-1}$ si $k > 0$ et $\rho = \emptyset$ sinon.
D'après la proposition 7 on en déduit que :

$$\begin{aligned} \mathcal{D}^*(\varrho(\sigma_k), \rho * \varrho(\sigma_3^k)) &\implies \\ \mathcal{D}^*(\varrho(\sigma_k) * \varrho(t_{\min \rho_3^k}), \rho * \varrho(\sigma_3^k) * \varrho(t_{\min \rho_3^k})) &\implies \\ \mathcal{D}^*(\varrho(\sigma'_k), \rho * \varrho(\sigma_2^k)) & \end{aligned}$$

Or d'après la définition 36, on a aussi $\rho = \varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{k-1}) * \rho_2^{k-1}$ si $k > 0$. Si $k > 0$, on en déduit donc que :

$$\mathcal{D}^*(\varrho(\sigma'_k), \varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{k-1}) * \rho_2^{k-1} * \varrho(\sigma_2^k))$$

Et sinon que $\mathcal{D}^*(\varrho(\sigma_0), \varrho(\sigma_2^0))$.
Enfin, toujours d'après la définition 36, on a :

$$s_2^i = s_3^i \quad \forall i \in [\![0, n]\!] \setminus \{k\}$$
$$t_{\min \rho_3^k} \text{ est exécutable en } \mathcal{T}^k(f^k \circ ... \circ f^0(s), \sigma_k) \wedge s_2^k = \mathcal{T}^k(s_3^k, t_{\min \rho_3^k})$$

On en déduit donc que σ'_k est exécutable en $f^k \circ ... \circ f^0(s)$ et que $s_2^k = \mathcal{T}^k(f^k \circ ... \circ f^0(s), \sigma'_k)$.
On note alors :

$$s_2^k = \mathcal{T}^k(f^k \circ ... \circ f^0(s), \sigma'_k)$$
$$s_2^i = s_3^i \quad \forall i \in [\![0, n]\!] \setminus \{k\}$$

On a donc montré que :

$$\mathcal{D}^*(\varrho(\sigma'_0), \varrho(\sigma_2^0))$$
$$\forall i \in [\![1, n]\!] : \mathcal{D}^*(\varrho(\sigma'_i), \varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{i-1}) * \rho_2^{i-1} * \varrho(\sigma_2^i))$$
$$\forall i \in [\![0, n]\!] : s_2^i = \mathcal{T}^i(f^i \circ ... \circ f^0(s), \sigma'_i)$$

D'après la définition 39, on en déduit donc que η_2 est exécutable en s.

ii. Sinon ($t_{\min \rho_1^k} \in C^k$) :
D'après la définition 36, on a :

$$\sigma_2^i = \sigma_3^i \wedge \rho_2^i = \rho_3^i \quad \forall i \in [\![0, n]\!] \setminus \{k\}$$
$$\sigma_2^k = \sigma_3^k$$
$$\exists n_3 \in N_{\rho_3^k}, m \in M : c_m = t_{n_3} \wedge \rho_2^k = \mathcal{R}(\rho_3^k, n_3, m)$$

A. Pour tout $i \in [\![0, k]\!]$, on a :

$$\varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{i-1}) * \rho_2^{i-1} * \varrho(\sigma_2^i) = \varrho(\sigma_3^0) * \rho_3^0 * ... * \varrho(\sigma_3^{i-1}) * \rho_3^{i-1} * \varrho(\sigma_3^i)$$

On en déduit donc que pour tout $i \in [\![0, k]\!]$:

$$\mathcal{D}^*(\varrho(\sigma_i), \varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{i-1}) * \rho_2^{i-1} * \varrho(\sigma_2^i))$$

B. Pour tout $i \in [\![k+1, n]\!]$, on a (si $k < n$) :
On note :

$$\begin{aligned}
\rho_g &= \varrho(\sigma_3^0) * \rho_3^0 * ... * \varrho(\sigma_3^{k-1}) * \rho_3^{k-1} * \varrho(\sigma_3^k) \\
&= \varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{k-1}) * \rho_2^{k-1} * \varrho(\sigma_2^k) \\
\rho_d &= \varrho(\sigma_3^{k+1}) * \rho_3^{k+1} * ... * \varrho(\sigma_3^{i-1}) * \rho_3^{i-1} * \sigma_3^k \\
&= \varrho(\sigma_2^{k+1}) * \rho_2^{k+1} * ... * \varrho(\sigma_2^{i-1}) * \rho_2^{i-1} * \sigma_2^k
\end{aligned}$$

On a donc $\mathcal{D}^*(\varrho(\sigma_i), \rho_g * \rho_3^k * \rho_d)$. Or $\rho_2^k = \mathcal{R}(\rho_3^k, n_3, m)$, donc $\mathcal{D}^*(\rho_3^k, \rho_2^k)$ d'après la définition 19.
D'après la proposition 7, on obtient :

$$\begin{aligned}
\mathcal{D}^*(\rho_3^k, \rho_2^k) &\implies \mathcal{D}^*(\rho_g * \rho_3^k, \rho_g * \rho_2^k) \\
&\implies \mathcal{D}^*(\rho_g * \rho_3^k * \rho_d, \rho_g * \rho_2^k * \rho_d)
\end{aligned}$$

On en déduit donc que $\mathcal{D}^*(\varrho(\sigma_i), \rho_g * \rho_2^k * \rho_d)$, et donc que :

$$\mathcal{D}^*(\varrho(\sigma_i), \varrho(\sigma_2^0) * \rho_2^0 * ... * \varrho(\sigma_2^{i-1}) * \rho_2^{i-1} * \varrho(\sigma_2^i))$$

Comme de plus pour tout $i \in [\![0, n]\!]$, on a $s_2^i = s_3^i$ (d'après la définition 36), on en déduit d'après la définition 39, que η_2 est exécutable en s.

Par récurrence structurelle sur Θ^*, on en déduit donc que η_2 est exécutable en s. □

Lors de la résolution d'un problème de planification donné, la hiérarchisation initiale (celle qui correspond au réseau de tâches initial) étant définie de façon à être exécutable, on en déduit donc que toute hiérarchisation visitée par ABPHTN est exécutable. Il suffit alors de montrer qu'une hiérarchisation primitive et exécutable correspond nécessairement à un réseau de tâches exécutable pour établir la correction d'ABPHTN :

Théorème 4 (Correction de ABPHTN)**.** *Soit un problème de planification avec hiérarchie d'abstractions $P = ((D^0, f^0), ..., (D^n, f^n), s_0, \rho_0)$. Toute séquence d'actions retournée par ABPHTN en résolvant P est une solution de P.*

Démonstration. On note $\eta_0 = ((f^0(s_0), \epsilon, \emptyset), ..., (f^{n-1} \circ ... \circ f^0(s_0), \epsilon, \emptyset), (f^n \circ ... \circ f^0(s_0), \epsilon, \rho_0))$ la hiérarchisation de ρ_0 qui correspond au premier élément visité par ABPHTN et on note $\eta = ((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n))$ la hiérarchisation visitée par ABPHTN lorsqu'il retourne une solution (qui correspond alors à σ_0).
D'après l'algorithme 11, η vérifie $\{i \in [\![0, n]\!] \mid \rho^i \neq \emptyset \vee (i > 0 \wedge \sigma^i \neq \epsilon)\} = \emptyset$, ainsi que $\Theta^*(\eta_0, \eta)$.
On en déduit alors que η est une hiérarchisation de $\varrho(\sigma^0)$ et que η est une hiérarchisation primitive (d'après la définition 35).
On en déduit donc d'après la proposition 10 que $\varrho(\sigma_0)$ est un réseau de tâches primitif. Comme $\Theta^*(\eta_0, \eta)$, on en déduit d'après la proposition 11 que $\mathcal{D}^*(\rho_0, \varrho(\sigma^0))$.
De plus, η_0 est une hiérarchisation exécutable en s_0, puisque le $(n+1)$-uplet $(\epsilon, ..., \epsilon)$ satisfait les conditions de la définition 39.
D'après la proposition 12, on en déduit donc que η est exécutable. On en déduit alors de la définition 39 qu'il existe $\sigma_0 \in A^{0^*}$ exécutable en s_0 tel que $\mathcal{D}^*(\varrho(\sigma_0), \varrho(\sigma^0))$.
Comme $\varrho(\sigma_0)$ est un réseau de tâches primitif, on en déduit que $\varrho(\sigma_0) = \varrho(\sigma^0)$, et donc que σ^0 est exécutable en s_0.
Ainsi, σ^0 est une séquence d'actions primitives exécutable en s_0 et est une linéarisation d'un réseau de tâches $\varrho(\sigma^0)$ vérifiant $\mathcal{D}^*(\rho_0, \varrho(\sigma^0))$. D'après la définition 21, on en déduit donc que σ^0 est une solution de P, et donc que ABPHTN est un algorithme correct. □

Le dernier résultat que nous souhaitons établir est la complétude de ABPHTN. C'est-à-dire que s'il existe une solution au problème de planification posé, alors ABPHTN doit en retourner une solution (en temps fini si les conditions permettent en plus d'en garantir la terminaison). Ce résultat va être démontré en plusieurs étapes. En première étape, on démontre qu'au sein des hiérarchisations d'un même réseau de tâches, l'exploration de l'espace de recherche par ABPHTN atteint nécessairement un point fixe :

Lemme 2. *Soient $\rho \in \varrho(T^*)$ ainsi qu'une suite $(\eta_i)_{i \in \mathbb{N}}$ telle que pour tout $i \in \mathbb{N}$:*

1. *$\eta_i \in H_\rho$.*
2. *si $\{\eta \in H_\rho \mid \Theta(\eta_i, \eta)\} = \emptyset$ alors $\eta_{i+1} = \eta_i$, et sinon $\Theta(\eta_i, \eta_{i+1})$.*

Il existe alors un $i_0 \geq 0$ tel que $\forall i \geq i_0 : \eta_i = \eta_{i_0}$.

Démonstration. Pour toute hiérarchisation $\eta = ((s^0, \sigma^0, \rho^0), ..., (s^n, \sigma^n, \rho^n))$ de ρ, on pose :

$$p(\eta) = \sum_{i=0}^{n} 2i \times |\sigma^i| + (2i+1) \times |\rho^i|$$

Montrons que la proposition suivante est satisfaite :

$$\forall (\eta_1, \eta_2) \in H_\rho^2 : \Theta(\eta_1, \eta_2) \implies p(\eta_1) > p(\eta_2) \vee \eta_1 = \eta_2 \tag{4.4}$$

Soit $(\eta_1, \eta_2) \in H_\rho^2$ tel que $\Theta(\eta_1, \eta_2)$.
On note $\eta_1 = ((s_1^0, \sigma_1^0, \rho_1^0), ..., (s_1^n, \sigma_1^n, \rho_1^n))$ et $\eta_2 = ((s_2^0, \sigma_2^0, \rho_2^0), ..., (s_2^n, \sigma_2^n, \rho_2^n))$. Comme $\Theta(\eta_1, \eta_2)$, on en déduit d'après la définition 38, que $\mathcal{K}(\eta_1, \eta_2) \vee \mathcal{P}(\eta_1, \eta_2)$.

1. Supposons que $\mathcal{K}(\eta_1, \eta_2)$. D'après la définition 36, on peut alors en déduire que η_1 n'est pas primitive.
   On note alors $k = \min\{i \in [\![0, n]\!] \mid \rho_1^i \neq \emptyset \vee (i > 0 \wedge \sigma_1^i \neq \epsilon)\}$.
   On en déduit aussi que :

$$\begin{aligned} p(\eta_1) > p(\eta_2) & \iff \\ 2k \times |\sigma_1^k| > (2(k-1)+1) \times |N_{\rho_2^{k-1}}| & \iff \\ 2k \times |\sigma_1^k| > (2(k-1)+1) \times |\sigma_1^k| & \iff \\ |\sigma_1^k| > 0 & \end{aligned}$$

   Or, toujours d'après la définition 36 on a :

$$(\rho_1^k \neq \emptyset \vee (k > 0 \wedge \sigma_1^i \neq \epsilon)) \wedge k > 0 \wedge (|\sigma_1^k| \geq h_k \vee \rho_1^k = \emptyset)$$

   Si $\rho_1^k = \emptyset$, on en déduit donc que $\sigma_1^k \neq \epsilon$, et donc que $|\sigma_1^k| > 0$.
   Si en revanche on a $\rho_1^k \neq \emptyset$, on en déduit que $|\sigma_1^k| \geq h_k$. Or, $h_k > 0$, donc $|\sigma_1^k| > 0$.
   On a donc $|\sigma_1^k| > 0$ dans tous les cas, et donc $p(\eta_1) > p(\eta_2)$.
2. Supposons que $\mathcal{P}(ph_1, ph_2)$.

   (a) Supposons que $t_{\min \rho_1^k} \in A^k$.
   D'après la définition 37, on en déduit que :

$$\eta_1 > \eta_2 \iff$$
$$2k \times |\sigma_1^k| + (2k+1) \times |N_{\rho_1^k}| > 2k \times |\sigma_2^k| + (2k+1) \times |N_{\rho_2^k}|$$

   Mais aussi que :

$$|N_{\rho_1^k}| = 1 + |N_{\rho_2^k}|$$
$$|\sigma_2^k| = |\sigma_1^k| + 1$$

   On en déduit donc que :

$$\begin{aligned} \eta_1 > \eta_2 & \iff \\ (2k+1) > 2k & \iff \\ 1 > 0 & \end{aligned}$$

   On a donc bien $p(\eta_1) > p(\eta_2)$.

   (b) Sinon, on a $t_{\min \rho_1^k} \in C^k$. D'après la définition 37, il existe une méthode $m \in M^k$ telle que $c_m = t_{\min \rho_1^k}$ et $\rho_2^k = \mathcal{R}(\rho_1^k, \min \rho_1^k, m)$.
   Or, comme η_1 et η_2 sont toutes deux des hiérarchisations de ρ, on déduit de la définition 37 et de la régularité à gauche et à droite du monoïde $(\varrho(T^*), *\emptyset)$, que $\rho_1^k = \mathcal{R}(\rho_1^k, \min \rho_1^k, m)$, et donc que $\eta_1 = \eta_2$.

La proposition 4.4 est donc bien satisfaite.
La suite $(p(\eta_i))_{i \in \mathbb{N}}$ est donc une suite décroissante. De plus, pour tout $i \in \mathbb{N}$, on a $p(\eta_i) \geq 0$. $(p(\eta_i))_{i \in \mathbb{N}}$ est donc une suite décroissante ayant une borne inférieure. On en déduit donc que cette suite converge.
Il existe donc un entier $i_0 \geq 0$ tel que pour tout $i \geq i_0$, $p(\eta_i) = p(\eta_{i_0})$.
Supposons qu'il existe un plus petit entier $i > i_0$ tel que $\eta_i \neq \eta_{i_0}$. On aurait alors $\eta_i \neq \eta_{i-1}$ et $p(\eta_i) = p(\eta_{i-1})$. Par définition de la suite $(\eta_i)_{i \in \mathbb{N}}$, on en déduit alors que $\Theta(\eta_{i-1}, \eta_i)$. D'après la proposition 4.4, on en déduit donc que $p(\eta_{i-1}) > p(\eta)$, ce qui est une contradiction.
On en déduit donc que pour tout entier $i \geq i_0$, $\eta_i = \eta_{i_0}$.

□

On montre ensuite que dans ce cas, le point fixe est soit une hiérarchisation primitive, soit ne mène à aucune hiérarchisation exécutable, ou impose la décomposition d'une tâche composée :

Lemme 3. *Soient un état $s \in S^0$, une séquence d'actions $\sigma \in A^{0*}$, un réseau de tâches $\rho \in \varrho(T^*)$ et une hiérarchisation de ρ $\eta = ((s^0, \sigma^0, \rho^0), ...(s^n, \sigma^n, \rho^n))$ tels que σ est exécutable en s, η est exécutable en s, $\mathcal{D}^*(\rho, \varrho(\sigma))$ et $\{\eta' \in H_\rho \mid \Theta(\eta, \eta')\} \subseteq \{\eta\}$.*
Si η n'est pas une hiérarchisation primitive alors $(k = 0 \vee |\sigma^k| < h_k) \wedge \rho^k \neq \emptyset \wedge t_{min\rho^k} \in C^k$, où :

$$k = \min \{i \in [\![0, n]\!] \mid \rho^i \neq \emptyset \vee (i > 0 \wedge \sigma^i \neq \epsilon)\}$$

Démonstration. Supposons que η n'est pas primitive.
Soit $\eta \in H_\rho$ tel que $\Theta(\eta, \eta_0)$. Par hypothèse, on a $\eta = \eta_0$.
On note $\eta_0 = ((s_0^0, \sigma_0^0, \rho_0^0), ..., (s_0^n, \sigma_0^n, \rho_0^n))$.
D'après la définition 38, $\Theta(\eta, \eta_0) \implies \mathcal{K}(\eta, \eta_0) \vee \mathcal{P}(\eta, \eta_0)$.
Supposons que $\mathcal{K}(\eta, \eta_0)$.
D'après la définition 36, on a alors :

$$\begin{aligned} \rho^{k-1} &= \emptyset \\ \rho_0^{k-1} &= \varrho(\sigma^k) \\ \sigma^k &\neq \epsilon \end{aligned}$$

On en déduit donc que $\rho^{k-1} \neq \rho_0^{k-1}$, et donc que $\eta \neq \eta_{i_0}$, ce qui est une contradiction.
On en déduit donc que $\neg\mathcal{K}(\eta, \eta_0)$, et donc que $\mathcal{P}(\eta, \eta_0)$, ce qui implique que $(i = 0 \vee |\sigma^k| < h_k) \wedge \rho^k \neq \emptyset$.
Supposons que $t_{\min \rho_{i_0}^k} \in A^k$:
η étant exécutable en s, il existe d'après la définition 39 une séquence d'action $\sigma_k \in A^{k*}$ exécutable en $f^k \circ ... \circ f^0(s)$ et telle que :

$$s^k = \mathcal{T}^k(f^k \circ ... \circ f^0(s), \sigma_k)$$
$$(k = 0 \wedge \mathcal{D}^*(\varrho(\sigma_k), \varrho(\sigma^0))) \vee (k > 0 \wedge \mathcal{D}^*(\varrho(\sigma_k), \varrho(\sigma^0) * \rho^0 * ... * \varrho(\sigma^{k-1}) * \rho^{k-1} * \varrho(\sigma^k))$$

On peut donc dire qu'il existe $(\rho_1, \rho_2) \in \varrho(T^*)^2$ tel que $\rho = \rho_1 * \varrho(t_{min\rho^k}) * \rho_2$ et $\mathcal{D}^*(\varrho(\sigma_k), \rho_1)$.
D'après la proposition 7, on en déduit donc que $\mathcal{D}^*(\varrho(\sigma_k) * \varrho(t_{min\rho^k}) * \rho_2, \rho)$, et donc que $\mathcal{D}^*(\varrho(\sigma_k) * \varrho(t_{min\rho^k}) * \rho_2, \varrho(\sigma))$.
D'après la proposition 8, on en déduit donc qu'il existe $(\sigma_1', \sigma_2', \sigma_3') \in T^{0*3}$ tel que :

$$\begin{gathered} \mathcal{D}^*(\varrho(\sigma_k), \varrho(\sigma_1')) \\ \mathcal{D}^*(\varrho(t_{min\rho^k}), \varrho(\sigma_2')) \\ \mathcal{D}^*(\varrho(\rho_2), \varrho(\sigma_3')) \\ \sigma = \sigma_1' \circ \sigma_2' \circ \sigma_3' \end{gathered}$$

Or σ_1' est exécutable en s, σ_2' est exécutable en $\mathcal{T}^0(s, \sigma_1')$ et toutes les actions abstraites sont cohérentes. D'après la proposition 9, on en déduit donc que $s^k = f^k \circ ... \circ f^0(\mathcal{T}^0(s, \sigma_1'))$ et que $t_{min\rho^k}$ est donc applicable en s^k.
D'après la définition 37, on a alors $\sigma_0^k = \sigma^k \circ t_{\min \rho_{i_0}^k}$, donc $\sigma_0^k \neq \sigma^k$ et donc $\eta \neq \eta_{i_0}$, ce qui est à nouveau une contradiction.
On en déduit donc que $t_{min\rho^k} \notin A^k$, et donc que $t_{min\rho^k} \in C^k$.
En conclusion, on a donc bien $(k = 0 \vee |\sigma^k| < h_k) \wedge \rho^k \neq \emptyset \wedge t_{min\rho^k} \in C^k$. □

On en déduit ainsi qu'entre chaque réseau de tâche, le parcours des hiérarchisations se fait en un nombre d'étapes fini. On peut alors montrer qu'à partir d'un réseau de tâches initial, tout réseau de tâches accessible dans l'espace de décomposition est aussi accessible dans l'espace des hiérarchisations exploré par ABPHTN :

Lemme 4. *Soient un état $s \in S^0$, une séquence d'actions $\sigma \in A^{0*}$ et un réseau de tâches $\rho \in \varrho(T^*)$ tels que σ est exécutable en s et $\mathcal{D}^*(\rho, \varrho(\sigma))$.*
S'il existe $\eta \in N_\rho$ exécutable en s, alors il existe $\eta' \in N_{\varrho(\sigma)}$ telle que $\Theta^*(\eta, \eta')$.

Démonstration. Pour commencer, montrons par récurrence que la propriété suivante, notée $P(i)$, est satisfaite pour tout $i > 0$:
Pour tout $\rho \in \varrho(T^*)$, exécutable en s, si $i = 1$ et $\rho = \varrho(\sigma)$ ou si $i > 1$ et il existe $(r_1, ..., r_i) \in \varrho(T^*)$ tels que :

$$r_1 = \rho$$
$$r_i = \varrho(\sigma)$$
$$\forall j \in [\![0, i-1]\!] : \mathcal{D}(r_j, r_{j+1})$$

alors s'il existe $\eta \in H_\rho$ exécutable en s, il existe $\eta' \in H_{\varrho(\sigma)}$ telle que $\Theta^*(\eta, \eta')$.

1. $P(1)$:
   Soit $\rho \in \varrho(T^*)$ tel que $\rho = \varrho(\sigma)$.
   On suppose qu'il existe $\eta \in H_\rho$. Dans ce cas, η est aussi une hiérarchisation de $\varrho(\sigma)$, et η vérifie $\Theta^*(\eta, \eta)$ par définition de Θ. $P(1)$ est donc bien satisfaite.
2. $\forall i > 0 : P(i) \implies P(i+1)$:
   Soit $i > 0$. On suppose que $P(i)$ est satisfaite.
   Soit $\rho \in \varrho$ tel qu'il existe $(r_1, ..., r_{i+1}) \in \varrho(T^*)$ tels que :

   $$r_1 = \rho$$
   $$r_{i+1} = \varrho(\sigma)$$
   $$\forall j \in [\![0, i]\!] : \mathcal{D}(r_j, r_{j+1})$$

   On suppose qu'il existe une hiérarchisation de $\eta \in H_\rho$ exécutable en s.
   D'après la proposition 5, pour tout $j \in [\![0, i]\!]$, $\mathcal{D}(r_j, r_{j+1})$ implique qu'il existe i nœuds $n_j \in N_{r_j}$ et une méthode de décomposition m_j tels que $c_{m_j} = t_{n_j}$ et tels que $r_{j+1} = \mathcal{R}(r_j, n_j, m_j)$.
   Soit $(\eta_j)_{j \in \mathbb{N}}$ une suite telle que $\eta_0 = \eta$ et telle que pour tout $j \in \mathbb{N}$, on a $\eta_j \in H_{r_1}$ et $\eta_{j+1} = \eta_j$ si $\{\eta' \in H_{r_1} \mid \Theta(\eta_j, \eta')\} = \emptyset$ et $\Theta(\eta_j, \eta_{j+1})$ sinon.
   Pour tout $j \in \mathbb{N}$, on note $\eta_j = ((s_j^0, \sigma_j^0, \rho_j^i), ..., (s_j^n, \sigma_j^n, \rho_j^n))$.
   D'après le lemme 2, il existe un entier j_0 tel que pour tout $j \geq j_0$, $\eta_j = \eta_{j_0}$.
   On en déduit donc que $\{\eta' \in H_\rho \mid \Theta(\eta_{j_0}, \eta')\} \subseteq \{\eta_{j_0}\}$
   r_1 n'étant pas primitif (puisqu'on a $r_2 = \mathcal{R}(r_1, n_1, m_1)$), on en déduit, d'après le lemme 3, que $(k = 0 \vee |\sigma_{j_0}^k| < h_k) \wedge \rho_{j_0}^k \neq \emptyset \wedge t_{min\rho_{j_0}^k} \in C^k$, où :

   $$k = \min \{j \in [\![0, n]\!] \mid \rho_{j_0}^j \neq \emptyset \vee (j > 0 \wedge \sigma_{j_0}^j \neq \epsilon)\}$$

   Or $\min \rho_{j_0}^k \in N_{r_1}$. Il existe donc un $i_1 \in [\![1, i]\!]$ tel que $\min \rho_{j_0}^k = n_{i_1}$.
   On peut donc définir $(r'_1, ..., r'_{j+1}) \in \varrho(T^*)^{j+1}$ tel que :

   $$r'_1 = \rho$$
   $$r'_2 = \mathcal{R}(r'_1, n_{i_1}, m_{i_1})$$
   $$r'_{i+1} = \varrho(\sigma)$$
   $$\forall j \in [\![2, i_1]\!] : r_{j+1} = \mathcal{R}(j_i, n_{j-1}, m_{j-1})$$
   $$\forall j \in [\![i_1 + 1, i]\!] : r_{j+1} = \mathcal{R}(r_j, n_j, m_j)$$

Comme η est exécutable en s et que $\Theta^*(\eta, \eta_{i_0})$, on peut en déduire d'après la proposition 12 que η_{i_0} est exécutable en s.
De plus, on a $(r'_2, ..., r'_{k+1}) \in \varrho(T^*)^k$ tels que :

$$r'_2 = \rho$$
$$r'_{i+1} = \varrho(\sigma)$$
$$\forall j \in [\![0, i]\!] : \mathcal{D}(r_{1+j}, r_{1+j+1})$$

$P(i)$ étant satisfaite par hypothèse, on en déduit donc qu'il existe $\eta' \in H_\varrho$ tel que $\Theta^*(\eta_{i_0}, \eta')$.
Comme $\Theta(\eta, \eta_{i_0})$, on en déduit donc que $\Theta(\eta, \eta')$.

Ainsi $P(1)$ est satisfaite et $\forall i > 0 : P(i) \implies P(i+1)$. Par récurrence, on en déduit donc que $P(i)$ est satisfaite pour tout $i > 0$.
Soit $\rho \in \varrho$ tel que $\mathcal{D}^*(\rho, \varrho(\sigma))$. Par définition de $\mathcal{D}^*$, on en déduit que $\rho = \varrho(\sigma)$ ou qu'il existe $i > 1$ et $(r_1, ..., r_i) \in \varrho(T^*)^i$ tels que :

$$r_1 = \rho$$
$$r_i = \varrho(\sigma)$$
$$\forall j \in [\![0, i-1]\!] : \mathcal{D}(r_j, r_{j+1})$$

D'après $P(1)$ et $P(i)$, on en déduit donc qu'il existe $\eta' \in H_{\varrho(\sigma)}$ tel que $\Theta^*(\eta, \eta')$. □

On en déduit finalement que ABPHTN est un algorithme complet pour des problèmes de planification HTN totalement ordonnés et dont toutes les actions abstraites sont cohérentes :

Théorème 5 (Complétude de ABPHTN)**.** *Soit un problème de planification avec hiérarchie d'abstractions $P = ((D^0, f^0), ..., (D^n, f^n)), s_0, \rho_0)$. Si P est résoluble, alors ABPHTN retourne une solution.*

Démonstration. On note $\eta_0 = ((f^0(s_0), \epsilon, \emptyset), ..., (f^{n-1} \circ ... \circ f^0(s_0), \epsilon, \emptyset), (f^n \circ ... \circ f^0(s_0), \epsilon, \rho_0))$ la hiérarchisation de ρ_0 qui correspond au premier élément visité par ABPHTN.
η_0 est une hiérarchisation exécutable en s_0, puisque le $(n+1)$-uplet $(\epsilon, ..., \epsilon)$ satisfait les conditions de la définition 39.
Soit $\sigma \in A^{0*}$ une solution de P. σ est donc exécutable en s_0 et vérifie $\mathcal{D}^*(\rho_0, \varrho(\sigma))$.
D'après le lemme 4, il existe donc une hiérarchisation de $\varrho(\sigma)$, noté $\eta = ((s^0, \sigma^0, \rho_0), ..., , (s^n, \sigma^n, \rho_n))$, telle que $\Theta^*(\eta_0, \eta)$.
D'après la proposition 12, on en déduit donc que η est exécutable en s_0.
Soit $(\eta_i)_{i \in \mathbb{N}}$ une suite telle que $\eta_0 = \eta$ et telle que pour tout $i \in \mathbb{N}$, on a $\eta_i \in H_{\varrho(\sigma)}$ et $\eta_{i+1} = \eta_i$ si $\{\eta' \in H_{\varrho(\sigma)} \mid \Theta(\eta_i, \eta')\} = \emptyset$ et $\Theta(\eta_i, \eta_{i+1})$ sinon.
D'après le lemme 2, il existe un entier i_0 tel que pour tout $i \geq i_0$, $\eta_i = \eta_{i_0}$.
On en déduit donc que $\{\eta' \in H_{\varrho(\sigma)} \mid \Theta(\eta_{i_0}, \eta')\} \subseteq \{\eta_{i_0}\}$
D'après le lemme 3, on en déduit donc que η_{i_0} est primitive, ou que $(i = 0 \vee |\sigma_0^k| < h_k) \wedge \rho_0^k \neq \emptyset \wedge t_{min\rho_0^k} \in C^k$, où :

$$\eta_{i_0} = ((s_0^0, \sigma_0^0, \rho_0^0), ...(s_0^n, \sigma_0^n, \rho_0^n))$$
$$k = \min \{j \in [\![0, n]\!] \mid \rho_0^j \neq \emptyset \vee (j > 0 \wedge \sigma_0^j \neq \epsilon)\}$$

Comme $\varrho(\sigma) \in \varrho(A^{0*})$, on en déduit que $t_{min\rho_0^k} \in A^0$, et donc que η_{i_0} est primitive.
Comme $\Theta^*(\eta_0, \eta)$ et $\Theta^*(\eta, \eta_{i_0})$, on en déduit donc que $\Theta^*(\eta_0, \eta_{i_0})$.
Ainsi, on a montré qu'à partir de η_0, ABPHTN peut visiter une hiérarchisation primitive de $\varrho(\sigma)$, et donc retourner la solution σ.
ABPHTN est donc un algorithme complet. □

4.2 Modélisation et expériences

Cette seconde partie présente des exemples de domaines de planification HTN avec hiérarchie d'abstractions. Trois différentes modélisations sont décrites : un exemple de reconnaissance de bâtiment pour le combat d'infanterie en zone urbaine, un exemple de problème d'affectation de cibles à neutraliser pour une section d'infanterie en position « statique », ainsi qu'un problème de recherche de chemin dans un graphe hiérarchique à deux niveaux.

De ces trois exemples, les deux derniers font l'objet d'expériences permettant de mesurer les performances de l'algorithme ABPHTN pour différentes valeurs d'horizon (la profondeur maximale du plan à chaque niveau). Ces expériences mettent en avant les conditions pour lesquelles la modélisation d'effets abstraits permet d'améliorer l'efficacité de la recherche de plans solutions.

4.2.1 Domaine de planification pour la reconnaissance de bâtiment

Le premier exemple est une modélisation d'un problème de reconnaissance de bâtiment, dans un contexte de combat d'infanterie en zone urbaine. Ce problème est extrait du *Manuel d'emploi du groupe de combat d'infanterie en zone urbaine* [68]. Pour reconnaître un bâtiment, chaque pièce qui le constitue doit être reconnue par une unité. Chaque fois qu'un trinôme pénètre dans une pièce, celui-ci la sécurise en y neutralisant les combattants ennemis qui l'occupe, y neutralise d'éventuels pièges, puis la tient en plaçant ses soldats en couvertures face aux accès par lesquels des renforts ennemis pourraient s'introduire. Lorsque les zones vers lesquels mènent ces accès ont été reconnues et sont tenues par d'autres unités, le trinôme peut être redéployé afin d'effectuer la reconnaissance d'une nouvelle pièce.

Le modèle de ce problème que nous proposons ici est décrit en HTDL dans l'annexe C.1. Ce domaine modélise la reconnaissance d'un petit bâtiment (moins de 8 pièces) ou d'une section d'un bâtiment plus grand (par exemple un étage) par un groupe de combat formé de deux trinômes. Les tâches y sont réparties sur trois niveaux d'abstraction : un adapté au niveau des soldats, un autre correspondant au niveau des trinômes et enfin un dernier qui correspond au niveau du groupe. Le problème est soumis à un ensemble de simplifications : aucune présence ennemie et aucun piège ne sont modélisés, il suffit de pénétrer une pièce pour la reconnaître, et les trinômes ne peuvent pas être réorganisés (il n'est pas possible de prélever des soldats de plusieurs trinômes pour en former un nouveau).

Au niveau concret, les soldats se déplacent sur des points de passage. Le déplacement des soldats est libre entre deux points de passage appartenant à la même pièce. Si deux points de passage sont situés dans deux pièces différentes, le déplacement de l'un à l'autre ne peut alors être exécuté que si ces deux points sont explicitement reliés. Enfin, certains points de passage sont aussi des points de couvertures, et sont alors associés à des secteurs d'observation.

Au niveau des trinômes, les déplacements s'effectuent entre les pièces. Une pièce est accessible à partir d'une autre lorsqu'il existe deux points de passage reliés entre elles. Il existe deux types de déplacement pour les trinômes. Le premier s'effectue uniquement au sein des pièces ayant déjà été reconnues, tandis que le second correspond à la reconnaissance d'une pièce. Un trinôme peut de plus tenir une pièce. Une pièce est dite « tenue » par un trinôme si tous ses points de passage menant vers des zones non reconnues sont situés dans le secteur d'observation d'un membre du trinôme en couverture. Un trinôme qui tient une pièce peut mettre fin à cette action si toutes les pièces voisines ont été reconnues. De plus, un trinôme qui tient une pièce ne peut pas se déplacer, sauf si seulement une pièce voisine n'a pas été sécurisée, et que le déplacement correspond à la reconnaissance de cette pièce.

Enfin, les quelques tâches qui correspondent au dernier niveau modélisent uniquement la reconnaissance d'une partie d'un bâtiment par le groupe de combat. Un exemple de problème, avec les différents niveaux d'abstraction, et représenté par la figure 4.7. Ce domaine fait usage des symboles dérivés pour décrire la situation aux niveaux abstraits : ainsi la position d'un trinôme parmi les pièces se déduit de la position de ses membres parmi les points de passage, ou encore, le fait qu'une pièce soit tenue par un trinôme se déduit des points de couvertures occupés par les soldats.

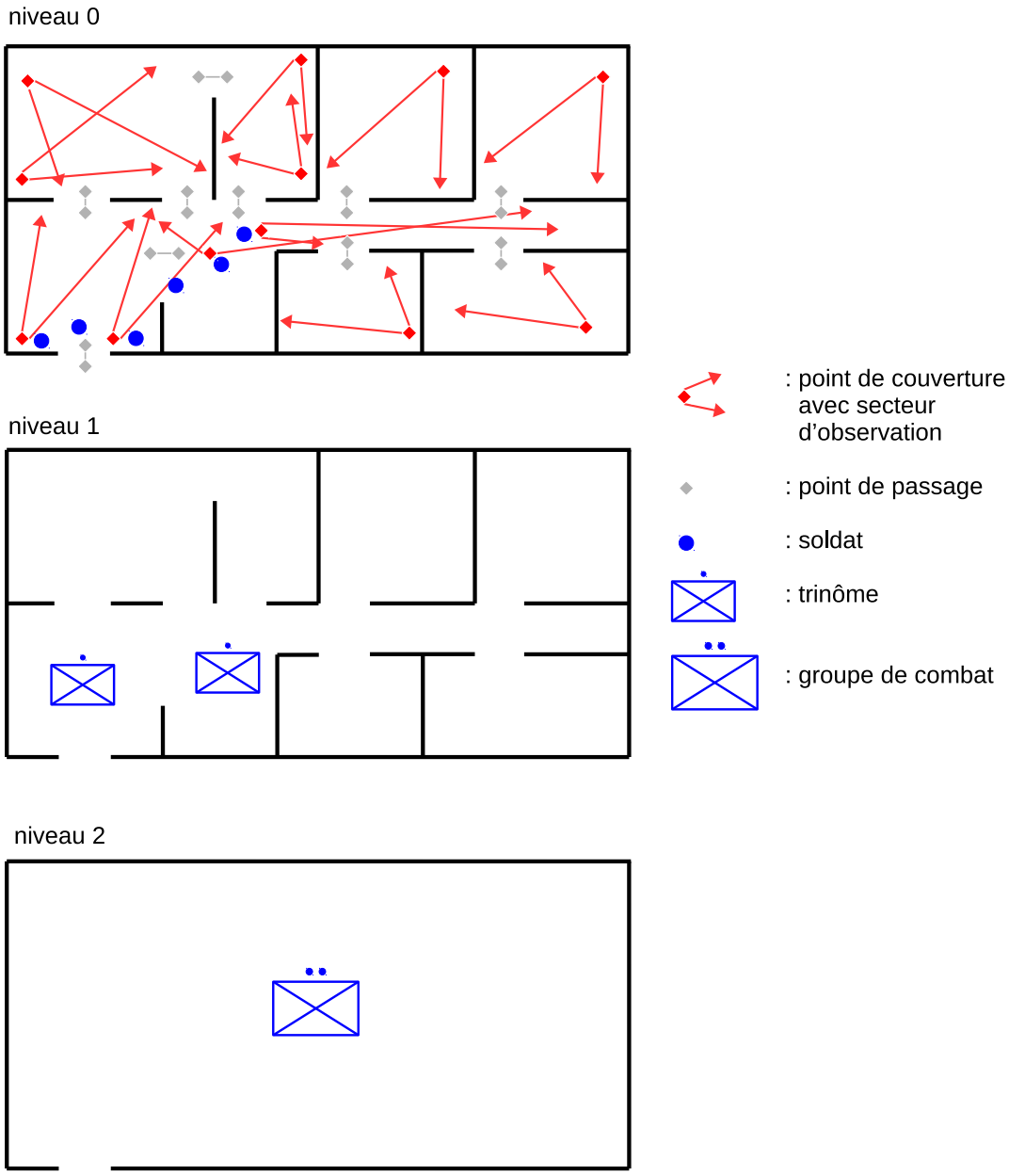


FIGURE 4.7 – Problème de reconnaissance de bâtiment
Représentation d'une situation pour le problème de reconnaissance de bâtiment sur trois niveaux. Le niveau 0 correspond au niveau des soldats, le niveau 1 correspond aux trinômes et le niveau 2 correspond au groupe de combat.

Ce domaine n'a pas fait l'objet d'expériences significatives, par manque d'un générateur d'instances aléatoires fiable. À titre d'indication, quelques résultats ont été obtenus sur des instances modélisées manuellement. Le planificateur utilisé est un prototype d'ABPHTN implémenté en langage Python. Il est exécuté sur un unique cœur d'un processeur Intel Core i7-4510U cadencé à 2.00 GHz. Par exemple, on obtient pour l'instance illustrée

en figure 4.7 un temps de calcul de l'ordre de 100 millisecondes, quelque soit la valeur de l'horizon. La modélisation d'effets abstraits ne semble donc pas être pertinente pour cette instance. Toutefois, ce résultat est encourageant vis-à-vis de la contrainte en temps d'exécution, puisqu'on obtient une moyenne de 15 millisecondes par soldat, chiffre qui pourrait encore être divisé avec une implémentation du planificateur en C++.

Ce domaine pourrait être enrichi avec l'introduction de tâches supplémentaires au niveau du groupe, par exemple pour reconnaître une succession d'étages d'un même bâtiment avec plusieurs groupes de combats, et ainsi obtenir un domaine modélisant la reconnaissance d'un bâtiment complet par une section d'infanterie. Il pourrait aussi être complété au niveau concret, avec l'introduction d'ennemis et de pièges, ce qui introduirait de la complexité à ce niveau et pourrait renforcer l'intérêt de la hiérarchie d'abstractions.

Enfin, la modélisation d'effets abstraits s'appuyant sur une séquence d'espaces de transitions abstraits montre des limites. Par exemple, les trinômes ne sont pas figés dans le cas réel, mais peuvent-être réorganisés afin d'apporter plus de flexibilité à l'exécution de la mission. Le système de séquence d'abstractions, en masquant l'information contenue aux niveaux inférieurs par rapport aux trinômes, ne permet pas d'appliquer ce procédé, ce qui dégrade la pertinence du modèle par rapport au cas réel.

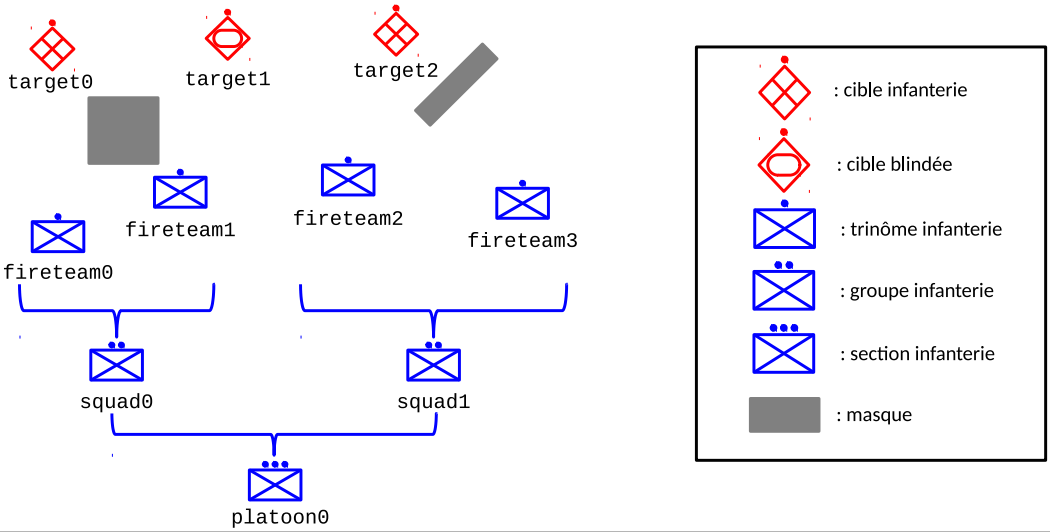
4.2.2 Évaluation du système pour un domaine de répartition des cibles

Le second domaine modélise un problème de répartition de cibles à neutraliser pour une section d'infanterie. Contrairement au domaine précédent, ce domaine a pu faire l'objet d'expériences dont le protocole expérimental et les résultats sont décrits dans cette sous-partie, à la suite de la présentation du domaine.

Description du domaine

Pour une section d'infanterie, l'application de tirs sur un ensemble de cibles ennemies, à partir d'une position statique, est une activité qui a lieu dans le cadre de plusieurs missions, telle que défendre une position ou fixer une unité ennemie, et dont ont trouve la description précise dans le *manuel d'emploi de la section d'infanterie* [67]. La modélisation du domaine représente une section d'infanterie constituée de plusieurs groupes de combat eux-même divisés en trinômes. Les trinômes sont considérés comme unité de base, ce qui signifie qu'il est fait abstraction des soldats dans la modélisation du problème. La hiérarchie d'abstractions utilisée est donc composée de trois niveaux : trinôme, groupe de combat et section. Les cibles à neutraliser correspondent à des équipes de soldats ennemis ou à des véhicules, éventuellement blindés. Pour chacune d'entre elles, une grandeur numérique appelée « capacité » modélise son potentiel de combat. Pour être neutralisée, cette cible doit subit un tir dont la puissance de feu est supérieure à sa capacité. Différents types d'armes sont aussi modélisés, avec leur portée pratique, leur capacité à percer les blindages, ainsi qu'une grandeur numérique appelée « efficacité », et qui modélise la puissance de feu de cette arme par rapport à une unité de munition. Chaque trinôme est alors décrit par la quantité de munitions qu'il possède par type d'arme, aux distances qui le sépare de chaque cible, ainsi que par l'ensemble des cibles visibles dans son secteur de tir (certaines cibles peuvent être en dehors de son secteur, ou être dissimulée derrière un masque). La figure 4.8 illustre un exemple de situation pour ce domaine.

L'abstraction au niveau des groupes de combats et des sections s'effectue par agrégation des données au niveau inférieur. Ainsi, la distance qui sépare un groupe de combat avec une cible correspond à la distance minimale entre l'ensemble des trinômes qui le composent et cette cible. De plus, le secteur de tir d'un groupe de combat correspond à la réunion



platoon0							
squad0				squad1			
fireteam0		fireteam1		fireteam2		fireteam3	
weapon-type	ammo-count	weapon-type	ammo-count	weapon-type	ammo-count	weapon-type	ammo-count
FAMAS	148	FAMAS	156	FAMAS	256	FAMAS	124
AT4	1	ANF1	123	AT4	3	ANF1	52
		LGI	2			LGI	3

FIGURE 4.8 – Problème d'affectation de cible
Représentation d'une situation pour le problème d'affectation de cible.

des secteurs de tir de chacun de ces trinômes et le nombre de munitions par type d'arme correspond à la somme des munitions détenues par chaque trinôme composant le groupe de combat. La même opération d'abstraction est répétée au niveau des sections à partir des données décrites au niveau des groupes de combat.

Une seule action est modélisée par niveau. Cette action modélise un tir de la part d'une unité, pour un type d'arme donné et sur une cible désignée. Pour que cette action soit applicable, l'unité concernée doit se trouver à distance de tir de la cible, celle-ci doit être visible dans son secteur d'observation, l'arme sélectionnée doit être adaptée à la nature de la cible (par rapport au blindage) et disposer de suffisamment de munitions pour la neutraliser. Si ces conditions sont satisfaites, le nombre de munitions pour l'arme concernée est diminué de la quantité consommée, et la cible est marquée comme étant neutralisée. À chaque niveau, cette action se décompose en une unique sous-action au niveau inférieur, qui correspond à un tir effectué par un membre de l'unité concernée. Une description en HTDL de ce domaine est disponible en annexe C.2.

Protocole expérimentale

Avec ce domaine, nous souhaitons évaluer deux approches de planification HTN différentes. La première correspond aux planificateurs de types PHTN, dont SHOP et SHPE sont des représentants. Pour rappel, la particularité de ces planificateurs et que leur stratégie de décomposition des tâches sélectionne toujours la première tâche non-primitive dans un réseau de tâches donné, et qu'ainsi l'exécutabilité des premières tâches primitives peut être évaluée. La seconde approche est celle pour laquelle ABPHTN a été conçu. Elle consiste à décomposer les tâches en suivant la hiérarchie d'abstractions. Pour cela, toutes les tâches du réseau sont décomposées jusqu'à obtenir un plan ne contenant que des actions abstraites au plus haut niveau. Ces tâches sont ensuite décomposées jusqu'à obtenir un plan abstrait au niveau inférieur, et ainsi de suite, jusqu'au niveau concret. Cette stratégie de décomposition est combinée à une sémantique de transition pour les actions

abstraites à chaque niveau, ce qui permet d'élaguer les plans abstraits non exécutables de l'espace de recherche. La question qui se pose et donc la suivante : obtient-on de meilleures performances (mesurées par le temps de calcul) en modélisant des effets abstraits et en résolvant les problèmes de planification avec un planificateur de type ABPHN, ou en les résolvant avec un planificateur de type PHTN, qui ne nécessite pas la modélisation supplémentaire d'effets abstraits ?

On peut par avance raisonner sur deux phénomènes susceptibles de se produire et qui sont contradictoires en terme de temps de calcul. D'un côté, les effets abstraits permettent de détecter des plans non exécutables avant d'atteindre le niveau concret. Il est donc possible d'élaguer des solutions partielles de l'espace de recherche et donc d'accélérer la recherche d'une solution exécutable. D'un autre côté, cette abstraction dissimulant aussi des informations, il est possible d'explorer une multitude de plans abstraits exécutables alors que ceux-ci ne mènent à aucune solution au niveau concret, alors que les planificateurs de type PHTN ont l'avantage de bénéficier rapidement d'un préfixe du plan concret, et donc d'informations exactes. Si un plan n'est pas exécutable à cause des premières tâches qui le composent, ces planificateurs pourront le détecter rapidement, et sans qu'il soit nécessaire de décomposer les tâches suivantes, ce qui pourrait les rendre plus performants.

Afin de comparer ces deux approches, on les évalue via une implémentation d'ABPHTN en Python (celle déjà mentionnée dans la sous-partie précédente). Pour obtenir une stratégie de décomposition qui construit des plans complets à chaque niveau d'abstraction avant de les décomposer, on attribue une unique valeur $h = \infty$ à tous les éléments de la suite d'horizons. Le même planificateur permet aussi de représenter les planificateurs de type PHTN, en attribuant pour cela une valeur unique $h = 1$ à toutes les horizons. On obtient bien dans ce cas la même stratégie de décomposition, puisqu'il n'est alors pas possible d'enchainer l'exécution de plus d'une action abstraite à un niveau d'abstraction donné. On évalue aussi les performances de valeurs intermédiaires pour les horizons. Ainsi la valeur $h = 2$ apporte un aperçu des performances de l'algorithme lorsqu'on exploite les effets abstraits à minima, et la valeur $h = 4$ permet d'obtenir des résultats pour une stratégie de décomposition exploitant les effets abstraits de façon significative, mais pas absolue. Ces différents planificateurs sont exécutés sur un seul cœur d'un processeur Intel Core i7-4510U cadencé à 2.00 GHz. La machine ayant servi à exécuter ces expériences est aussi dotée de 8 Go de mémoire vive, mais seule une petite proportion de cette mémoire (inférieure à 0.5 %) a été employée pour résoudre chaque instance.

Enfin, pour évaluer les performances de ces différents planificateurs dans ce domaine, on les exécute sur plusieurs séries d'instances générées aléatoirement. Chaque série est constituée de 500 instances résolubles, et diffère par rapport aux autres par le nombre de cibles à neutraliser, qui s'étend de 1 à 8 (soient huit séries d'instances de tailles croissantes). Pour chaque instance, la situation modélisée représente une seule section dont l'organisation respecte la description d'une section d'infanterie qui nous est indiquée par le *manuel d'emploi de la section d'infanterie* [67], avec pour seule exception l'absence de groupe antichar. Cette section est donc composée de trois groupes, et chaque groupe est lui-même composé de deux trinômes, l'un équipé avec des armes portant à 300 mètres (fusil d'assaut FAMAS et lance roquette AT-4) et l'autre équipé avec des armes portant à 600 mètres (FAMAS, mitrailleuse ANF-1 et lance grenade LGI). Les munitions pour chaque type d'arme sont affectées aléatoirement (selon une loi uniforme) entre 0 et la quantité maximale emportée par un trinôme. La nature des cibles est aussi déterminée aléatoirement : ainsi une cible est classée comme véhicule blindé avec une probabilité de 33.33 %, et la capacité de chaque cible est définie aléatoirement entre des valeurs minimales et maximales cohérentes par rapport à leur nature. Enfin, l'intervisibilité entre un trinôme et une cible est définie avec une probabilité de 75 %, et les distances sont

sélectionnées aléatoirement entre 50 mètres et 400 mètres.

Analyse des résultats

Les résultats obtenus sont présentés en figures 4.9. Chaque courbe est associée à un planificateur ABPHTN avec une valeur donnée pour les horizons ($h = 1$, $h = 2$, $h = 4$, et $h = \infty$). Chaque point représente la moyenne des temps CPU obtenus pour résoudre les instances qui correspondent à un nombre de cibles déterminé. Si on observe dans un premier temps les deux cas extrêmes, à savoir $h = 1$ et $h = \infty$, on en conclut que la planification HTN par hiérarchie d'abstractions est contreproductive, puisque la courbe pour $h = \infty$ domine celle pour $h = 1$, avec des temps de calcul presque 10 fois supérieurs à partir de 4 cibles, puis 100 fois supérieurs à partir de 6 cibles. Si on s'intéresse aux cas intermédiaires ($h = 2$ et $h = 4$), on peut alors proposer un jugement plus nuancé. En effet, on voit apparaitre des points d'intersection entre chacune de ces deux courbes et la courbe $h = 1$. Ces points d'intersection apparaissent après 7 cibles, et à partir de ces points, les temps de calcul moyens deviennent plus élevés pour le planificateur où $h = 1$. Ce résultat pourrait nous amener à conclure que la construction de plan abstrait avec une profondeur limitée à chaque niveau pourrait constituer un compromis permettant d'obtenir de meilleurs résultats pour les instances les plus difficiles.

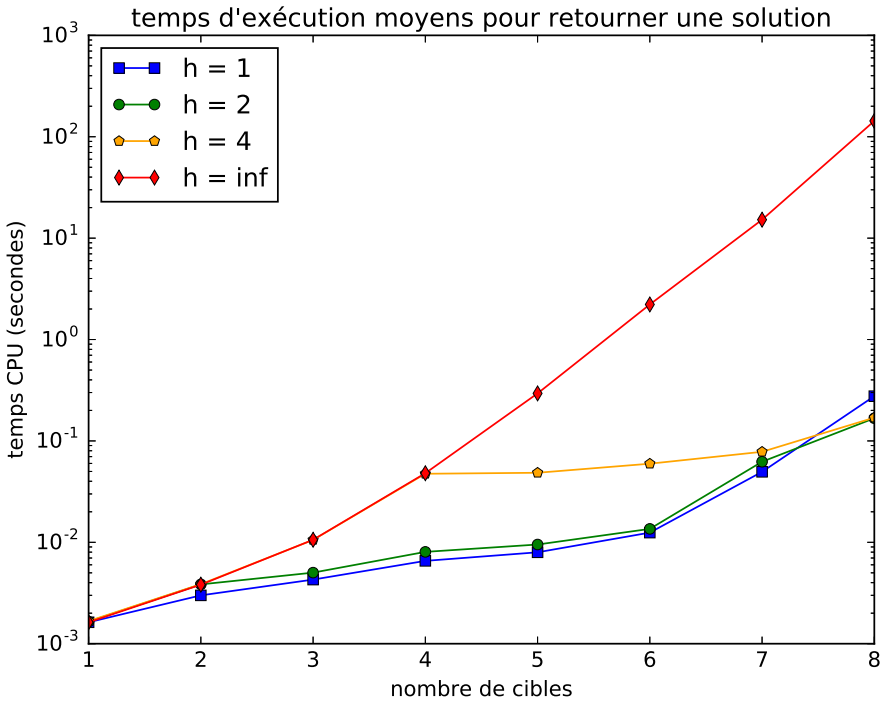


FIGURE 4.9 – Temps d'exécution pour les problèmes d'affectation de cibles
Moyennes des temps d'exécution (en temps CPU) obtenus lors de la résolution de problèmes d'affectation de cibles de tailles croissantes (la taille étant déterminée par le nombre de cible, en abscisse). Chaque courbe correspond à une valeur différente pour les horizons.

Mais observons ce dernier point de plus près. Pour cela, la figure 4.10 présente une répartition des différentes mesures obtenues sur la dernière série sous la forme d'un diagramme en boîte. On y observe que la majorité des mesures pour les deux cas qui utilise partiellement les effets abstraits ($h = 2$ et $h = 4$) dominent celles obtenues sans abstraction ($h = 1$). Mais on observe aussi qu'une poignée de mesure du cas $h = 1$ se place

à un niveau anormalement élevé (de l'ordre de la centaine de secondes), alors que dans les deux autres cas, les mesures les plus élevées sont de l'ordre de la dizaine de secondes. Ce phénomène explique donc le résultat observé au niveau des moyennes.

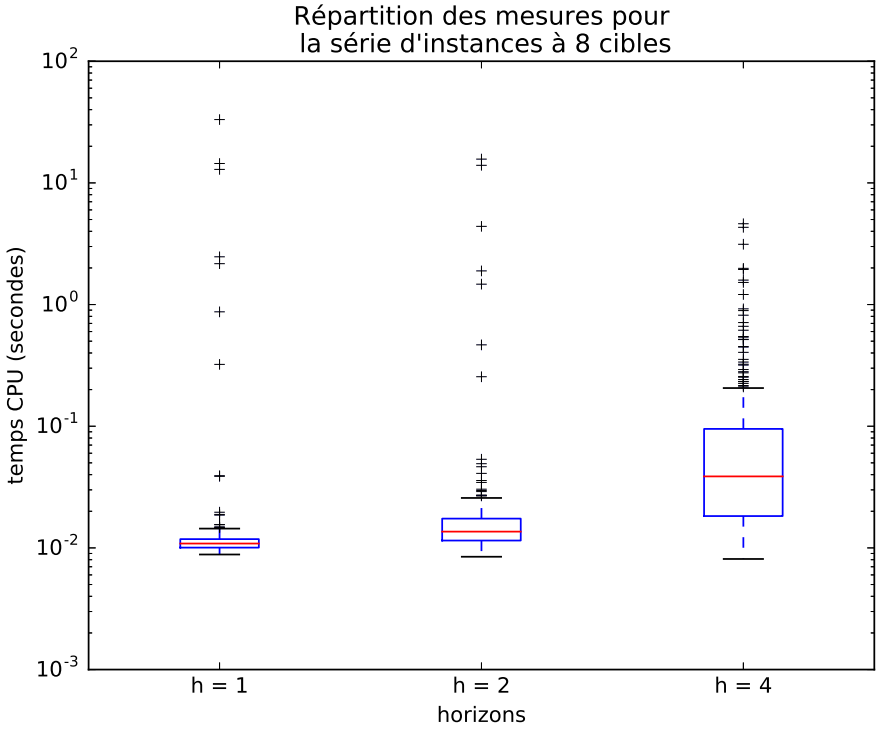


FIGURE 4.10 – Répartition des temps de calculs pour les problèmes d'affectation de cible
Diagramme en boîte représentant la répartition des temps de calcul obtenus pour résoudre les 500 instances dont le nombre de cibles est égale à 8. Chaque boîte représente une valeur d'horizon différente.

Il apparait donc qu'à ce point (en deçà de 9 cibles) l'abstraction provoque l'exploration d'un grand nombre de plans abstraits dont aucune concrétisation n'est exécutable. On peut donc dire que les effets abstraits modélisés sont trop « optimistes » par rapport aux effets des actions concrètes. Quand aux quelques instances pour lesquels ces effets abstraits apportent un avantage, on peut supposer qu'il s'agit de problèmes pour lesquels la majorité des plans non exécutables le sont à cause de leur dernières actions, ce qui ne permet pas non plus au planificateurs de type PHTN d'élaguer rapidement les branches de l'espace de recherche menant vers ces plans.

4.2.3 Évaluation du système pour un domaine de navigation hiérarchique

L'expérience précédente nous a enseigné que l'utilisation des effets abstraits est contreproductive si la qualité de l'abstraction n'est pas suffisante. Mais quel résultat obtiendrait-on pour un domaine où il serait possible de modéliser des effets abstraits plus précis ? Le domaine suivant, qui modélise un problème de navigation dans un graphe hiérarchique, a été conçu pour étudier cette question. À nouveau, cette sous-partie présente ce nouveau domaine, le protocole expérimental mis en place, ainsi qu'une analyse des résultats obtenus.

Description du domaine

Plutôt que de modéliser un problème correspondant à un cas d'application pour le combat d'infanterie, et dont il serait difficile d'évaluer à priori la qualité des effets abstraits, nous avons fait le choix d'évaluer notre système sur le modèle le plus général qu'il soit, à savoir un problème de recherche de chemin dans un graphe. En effet, tout problème de planification consiste en définitive à trouver un chemin (ce qui est la définition d'un plan exécutable) dans un graphe d'états. Afin de rendre ce problème pertinent pour une modélisation sous la forme d'un domaine de planification HTN avec hiérarchie d'abstractions, on regroupe les états dans des états abstraits, et on introduit une transition abstraite pour chaque paire d'états abstraits pour lesquels il existe au moins une transition depuis un état concret de l'un, vers un état concret de l'autre. On définit de plus un état courant, ainsi qu'un état but. Au niveau abstrait, l'état abstrait courant et simplement l'état abstrait qui contient l'état courant, et de même l'état but abstrait est l'état abstrait qui contient l'état but. La figure 4.11 propose une illustration d'un graphe hiérarchique correspondant à cette description.

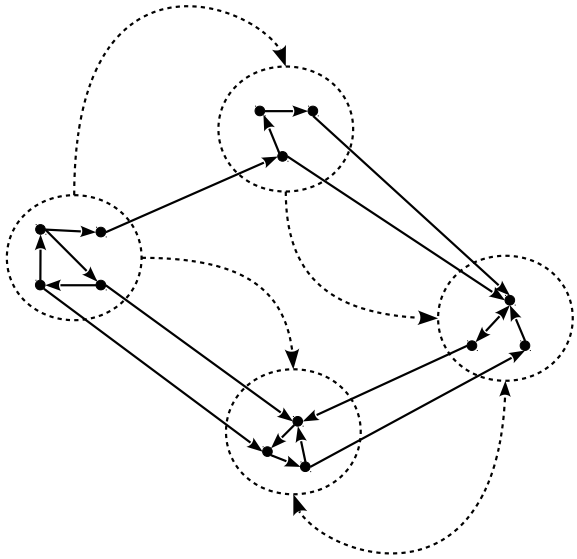


FIGURE 4.11 – Graphe hiérarchique
Graphe hiérarchique à deux niveaux. Les éléments en trait plein sont associés au niveau concret tandis que les éléments en pointillés sont associés au niveau abstrait. Les états sont représentés par des disques et les transitions entre états sont représentées par les arcs.

Le domaine modélisé décrit un type d'action qui permet de se déplacer sur le graphe concret en suivant une transition. Le même type d'action est disponible au niveau abstrait, afin d'accomplir la recherche de chemin à ce niveau. Les autres tâches non primitives du domaine permettent de programmer un algorithme simple de parcours de graphe en avant aux deux niveaux. Ce domaine est décrit en HTDL en annexe C.3.

Protocole expérimental

Ce domaine est évalué dans les mêmes conditions matérielles que le domaine précédent, et avec le même système de planification. Le lecteur est donc invité à se référer à la sous-partie précédente pour obtenir les détails de cette configuration. À nouveau, cette expérience compare quatre profils différents d'ABPHTN, où chaque profil correspond à une valeur différente pour les horizons : $h = 1$, $h = 2$, $h = 4$ et $h = \infty$. Pour rappel, le profil $h = 1$ est utilisé pour représenter les planificateurs de type PHTN, puisque cette

valeur d'horizons implémente une stratégie de décomposition des tâches identique à celle de ces planificateurs.

Pour obtenir des instances de problèmes à résoudre, un générateur aléatoire à été implémenté. Celui-ci prend quatre paramètres en entrée : le nombre d'états abstraits, le nombre d'états concrets par états abstraits, la proportion de transitions dans le graphe abstrait, et la proportion de transitions dans le graphe concret. Les instances sont alors générées de la façon suivante : une proportion de transitions est sélectionnée aléatoirement parmi l'ensemble de toutes les transitions pouvant être définies pour le graphe abstrait, puis le même procédé est appliqué au niveau concret, mais uniquement parmi l'ensemble des transitions entre états d'un même état abstrait, ou entre états appartenant à des états abstraits pour lesquels une transition abstraite a été sélectionnée.

Une première expérience est effectuée sur des séries d'instances avec un nombre d'états abstraits croissant, 4 états concrets par états abstraits, 50 % de transitions au niveau abstrait et seulement 12.5 % de transitions au niveau concret. La proportion réduite de transitions sélectionnées au niveau concret permet de générer des problèmes pour lesquels la probabilité d'obtenir un plan concret exécutable correspondant à une action abstraite applicable est faible. On obtient donc des instances similaires à celles du domaine précédent, pour lesquels les effets abstraits ne sont pas précis. Chaque série d'instances contient 500 problèmes résolubles, et celles-ci sont générées pour un nombre d'états abstraits variant entre 1 et 12.

La seconde expérience s'effectue dans les mêmes conditions, mais avec une proportion de transitions sélectionnées au niveau concret valant 50 %. Cette valeur permet d'augmenter la précision de l'abstraction, et donc d'observer les performances des planificateurs qui exploitent les effets abstraits dans ce contexte plus favorable. Comme ces instances sont plus difficiles à résoudre (à cause du plus grand nombre de transitions au niveau concret), les séries ont été générées pour un nombres d'état abstrait ne variant qu'entre 1 et 8.

Analyse des résultats

Les résultats de la première expérience sont représentés sous la forme de courbes dans la figure 4.12, où chaque point représente la moyenne des temps de calcul obtenus sur une série correspondant à un nombre d'états abstraits donné. On peut voir que le comportement des différents planificateurs est cohérent avec les observations effectuées pour le domaine précédent. En effet, le planificateur qui émule les planificateurs de type PHTN (avec $h = 1$) est celui qui obtient les meilleurs temps de calcul moyens. Les autres profils sont d'autant moins performants que leur valeur d'horizons est élevée (et donc qu'ils exploitent davantage les effets abstraits). Par exemple, pour des graphes contenant 12 états abstraits, on observe que le profil $h = \infty$ obtient des temps de calcul 100 fois plus longs que le profil $h = 1$, alors que le profil $h = 4$ obtient des temps de calcul presque 10 fois plus longs, et que la courbe du profil $h = 2$ s'établit légèrement au dessus de celle du profil $h = 1$. Ces résultats confirment que l'utilisation des effets abstraits, lorsque leur précision est faible, ce fait au détriment des performances par rapport à une simple approche de type PHTN.

En ce qui concerne la seconde expérience, celle où les instances de problèmes ont été générées de façon à améliorer la précisions des effets abstraits, on obtient des résultats opposés. En effet, si on s'intéresse dans un premier temps aux deux profils $h = 1$ et $h = \infty$, au obtient une situation inversée, où le planificateur qui correspond à $h = \infty$ obtient de meilleures performances que le planificateur qui correspond à $h = 1$. Ainsi, le temps de calcul moyen du premier est presque 10 fois inférieur à celui du second pour les séries d'instances au delà de 5 états abstraits, et près de 100 fois inférieur à partir de 8 états abstraits. En ce qui concerne les deux autres profils $h = 2$ et $h = 4$, les résultats obtenus

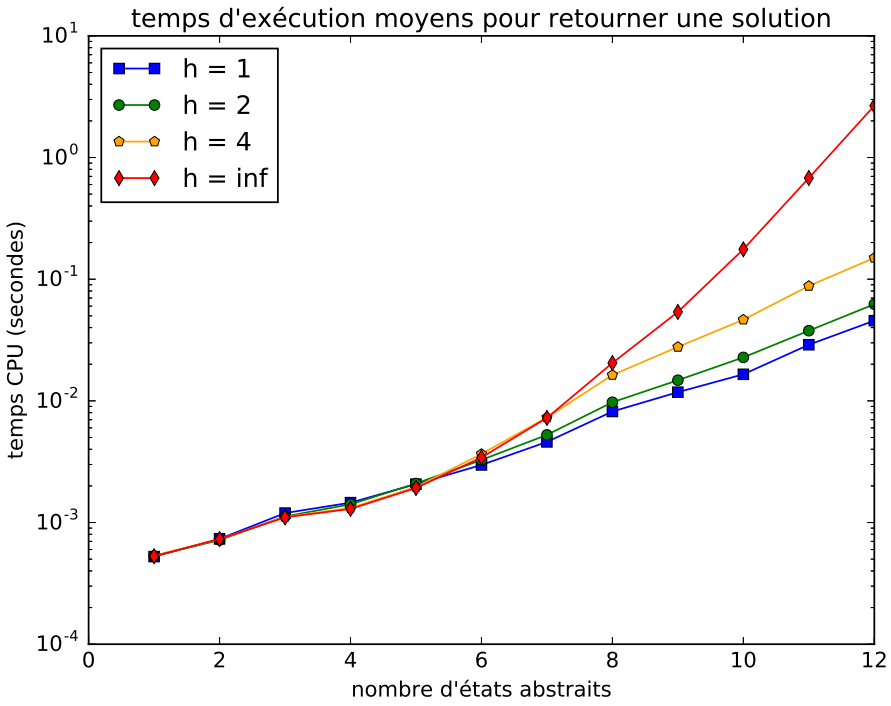


FIGURE 4.12 – Temps d'exécution avec abstraction de mauvaise qualité
Moyennes des temps d'exécution (en temps CPU) obtenus lors de la résolution de problèmes de recherche de chemin dans des graphes hiérarchiques de tailles croissantes (la taille étant déterminée par le nombre d'états abstraits, en abscisse), pour lesquels les effets abstraits sont peu précis. Chaque courbe correspond à une valeur différente pour les horizons.

ne se distinguent pas aussi clairement que lors de la première expérience. Ainsi, le cas $h = 2$ se comporte sensiblement comme le cas $h = 1$. En ce qui concerne le cas $h = 4$, on remarque qu'en dessous de 6 états abstraits, on obtient les mêmes performances que le cas $h = \infty$, ce qui est attendu puisque en deçà de cette valeur, la taille des plans abstraits ne peut pas dépasser 4. Au delà, on observe dans un premier temps une dégradation des performances de $h = 4$, dont les temps moyens deviennent comparables à ceux des cas $h = 1$ et $h = 2$. Ce phénomène semble confirmer la pertinence de l'utilisation des effets abstraits sur un horizon long (plus long que 4) dans ce contexte. On observe tout de même qu'à 8 états abstraits, le profil $h = 4$ se démarque des cas $h = 1$ et $h = 2$, avec un temps d'exécution moyen environ 10 fois inférieur aux leurs, tout en restant 10 fois plus élevé que le temps d'exécution moyen du cas $h = \infty$. Ainsi, l'utilisation des effets abstraits, même sur un horizon court, semble tout de même apporter un avantage pour des instances plus complexes.

Ces expériences nous montrent donc que, en accord avec nos hypothèses initiales, la modélisation d'effets abstraits ainsi que leur exploitation par un planificateur de type ABPHTN, permet d'améliorer les temps d'exécution obtenus lors de la résolution de problèmes de planification. Toutefois, cette amélioration n'est possible que si la structure des problèmes y est adaptée, avec une probabilité élevée d'obtenir des plans concrets exécutables à partir des actions abstraites applicables.

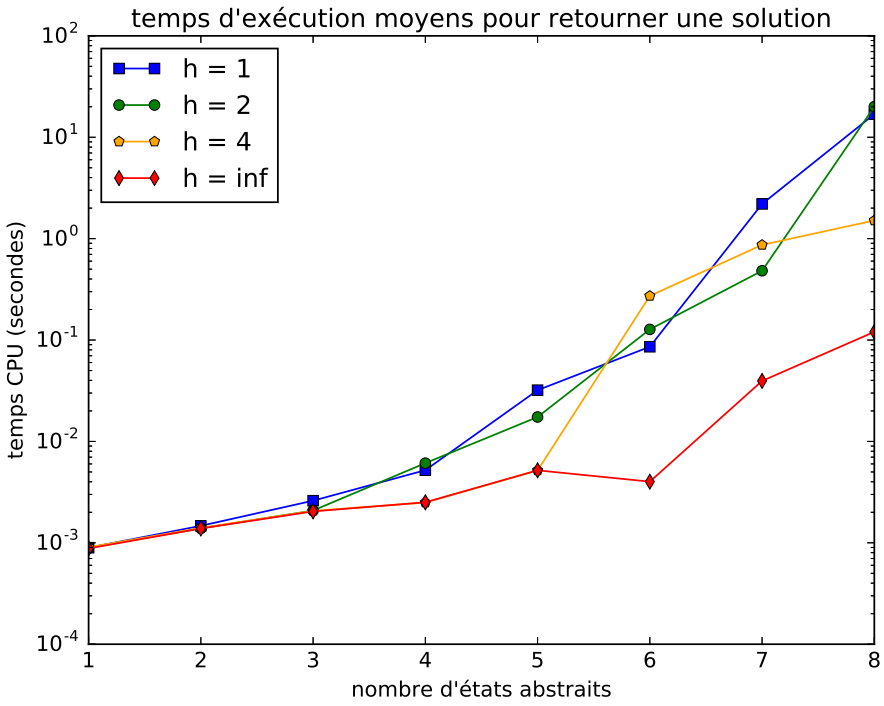


FIGURE 4.13 – Temps d'exécution avec abstraction de bonne qualité
Moyennes des temps d'exécution (en temps CPU) obtenus lors de la résolution de problèmes de recherche de chemin dans des graphes hiérarchiques de tailles croissantes (la taille étant déterminée par le nombre d'états abstraits, en abscisse), pour lesquels on a amélioré la précision des effets abstraits en augmentant le nombre de transitions au niveau concret. Chaque courbe correspond à une valeur différente pour les horizons.

4.3 Seconde approche : planification HTN avec abstraction

Nous avons vu que l'utilisation d'effets abstraits permet d'améliorer les temps de calculs d'un planificateur sous certaines conditions. Afin de modéliser ces effets abstraits, nous avons combiné la planification HTN avec le principe de la hiérarchie d'abstractions (qui correspond en fait à une séquence d'abstractions) qui avait déjà été appliquée en planification classique. Toutefois, ce principe de hiérarchie d'abstractions s'avère limité du point de vue de la modélisation. En effet, il impose une unique séquence d'abstractions pour l'ensemble du domaine, et n'apporte pas la flexibilité nécessaire pour modéliser des domaines complexes, représentatifs de cas d'application réels. Par exemple, la hiérarchie d'abstractions qui repose sur les différents échelons de la section d'infanterie n'est pas adaptée pour toutes les missions. Ainsi, le regroupement des soldats en trinôme n'est généralement pas respecté lors des missions en combat urbain. Mais encore, le groupe chargé d'appuyer la conquête d'une position peut tantôt être composé des véhicules de la section si le terrain le permet, tantôt être formé en rassemblant les trinômes les plus lourdement armés, ou être complété par des tireurs de précision, etc.

Le sujet de cette partie concerne donc la description d'un système alternatif pour la modélisation des effets abstraits. Ce système ne repose plus sur un ensemble d'espaces d'états où chaque espace correspond à un niveau d'abstraction, mais sur un unique espace composé d'états pour lesquels la valeur de vérité de certaines formules logiques peut être inconnue. Le langage logique utilisé pour décrire l'environnement devient donc un langage logique à trois valeurs de vérité, et les fonctions d'abstraction sont modélisées sous la forme d'effets dans les actions abstraites. Ces effets permettent ainsi de modéliser des fonctions

d'abstraction propres à chaque tâche composée dans un domaine HTN, et s'affranchissent ainsi du concept de hiérarchie d'abstractions. Comme ce système d'abstraction se caractérise par l'absence de hiérarchie d'abstractions explicite, on le désigne alors simplement par l'expression « planification HTN avec abstraction ».

La première sous-partie est consacrée à la description des états abstraits à partir d'un langage logique à trois valeurs de vérité. La seconde partie décrit le système de transitions défini par l'ensemble de ces états abstraits et des tâches composées du domaine HTN. Enfin, la troisième partie décrit un algorithme de planification HTN avec abstraction dans le contexte de la planification HTN totalement ordonnée, et analyse les principales propriétés de cet algorithme (terminaison, correction et complétude).

4.3.1 Espace d'états d'abstraits

Une extension naturelle au système de hiérarchie d'abstractions utilisé jusqu'ici consiste à modéliser une véritable hiérarchie d'espaces d'états, sous la forme d'un ordre partiel, au lieu d'une simple séquence. En effet, il n'est pas nécessaire d'abstraire chaque espace d'état d'une seule façon, puisque qu'en appliquant le principe des criticités, il existe autant d'espaces abstraits que de sous-ensembles de symboles pouvant être supprimés d'un langage logique. On obtient alors un treillis d'espaces d'états, dont l'élément minimal correspond à l'espace concret, et l'élément maximal correspond à l'espace dont le langage logique associé est vide, comme l'illustre le schéma de la figure 4.14.

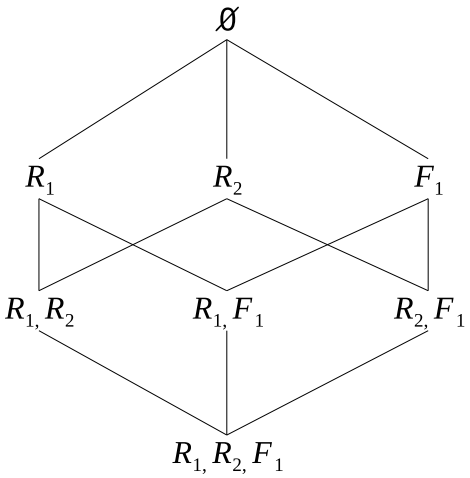


FIGURE 4.14 – Treillis d'abstractions
Exemple de hiérarchie d'abstractions pour un espace d'états décrit par un langage logique constitué de deux symboles de prédicat R_1 et R_2 et du symbole de fonction F_1. Chaque espace est représenté par le langage logique qui lui est associé.

Afin d'implémenter ce nouveau type de hiérarchie d'abstractions, il faut encore identifier un moyen d'encoder les fonctions d'abstraction. En effet le système des criticités, que nous avons emprunté à ABSTRIPS, n'est alors plus adapté. Une alternative consiste à spécifier, pour chaque action abstraite, les symboles définis dans le système de transition qui lui est associée. Il devient ainsi possible d'identifier les états suffisamment concrets pour que l'exécutabilité de l'action puisse être évaluée. Puisque ces opérateurs d'abstraction ont toujours pour effet de supprimer des symboles de prédicat ou de fonction du langage logique, il est encore possible de les identifier statiquement à partir du domaine de planification HTN pour chaque action abstraite, en comparant les effets de l'action avec les effets de ses décompositions.

Cependant, ce système souffre encore de deux défauts que l'on préfère éviter. Le premier concerne le niveau de granularité auquel interviennent les opérateurs d'abstraction.

Ceux-ci suppriment en effet l'intégralité de l'interprétation d'une relation ou d'une fonction, alors qu'on peut parfois souhaiter n'en abstraire qu'une sous-partie. Par exemple, si une action abstraite a pour effet de déplacer un groupe, tout en ignorant les positions atteintes par les soldats, on peut souhaiter que seules les positions des membres du groupe soient abstraites, et non pas celles de l'ensemble de tous les soldats modélisés dans l'environnement. Le second défaut concerne l'exécution des actions abstraites. Tant qu'un état appartient à un espace plus concret que celui de l'action, il suffit d'appliquer une suite d'opérateurs d'abstraction pour le transformer en état abstrait au même niveau que cette action. En revanche, si ce n'est pas le cas (soit parce que l'état est plus abstrait, soit parce qu'il appartient à une autre chaine de la hiérarchie d'abstraction), il doit alors être concrétisé. Or, plusieurs états concrets correspondent à un même état abstrait, ce qui impose de trouver parmi tous ces états un exemple pour lequel l'action est exécutable. Pour ne pas perdre de temps à effectuer ces calculs, on souhaite évaluer une exécutabilité « potentielle » de l'action abstraite par rapport à n'importe quel état abstrait, en ignorant les littéraux abstraits dans l'évaluation des formules logiques.

Afin de résoudre ces deux problèmes, on souhaite implémenter un système qui permet de marquer chaque élément de l'interprétation d'un prédicat ou d'une fonction comme étant « abstrait », et qui permet aussi d'évaluer des formules logiques pour des états dont la valeur de certains littéraux est inconnue. Les formules logiques utilisées pour définir les préconditions et les conditions des effets n'étant pas restreintes aux conjonctions de littéraux, il n'est pas possible de se contenter d'ignorer les littéraux inconnus, comme le fait ABSTRIPS avec les criticités. En effet, une conjonction est fausse dès qu'un de ces éléments est faux. Il est donc raisonnable de la considérer comme étant satisfaite tant que ces éléments sont vrais ou inconnus. Mais dans le cas d'une disjonction, celle-ci est vraie dès qu'un de ces éléments l'est. En conséquence, un système qui ignorerait les éléments inconnus d'une disjonction pourrait évaluer celle-ci comme étant fausse, quand bien même au moins un des éléments inconnus s'avèrerait être vrai. Pour résoudre ce problème, il faudrait disposer d'un moyen de raisonner sur des états où une partie de l'information est inconnue, et distinguer les formules logiques vraies et fausses de celles dont la valeur de vérité est inconnue.

La logique trivalente forte de Kleene [7] répond à cette problématique. Il s'agit d'une logique à trois valeurs : vrai ($\top$ ou 1), faux ($\bot$ ou 0) et indéterminé (I ou $\frac{1}{2}$). Cette logique généralise la logique naturelle lorsque le sens de la troisième valeur est associé à l'indéfini ou l'inconnu, quand on ne peut pas dire si une formule est vraie ou fausse. Cette troisième valeur représente donc bien le concept d'abstraction. Comme le représente les deux treillis de la figure 4.15, elle contient moins d'information par rapport à vrai et faux (elle masque donc de l'information), mais se situe entre ces deux valeurs en terme de vérité. Elle permet ainsi de raisonner sur des formules dont on ne peut pas dire si elles sont vraies, sans avoir à les qualifier de fausses, ce qui correspond à notre besoin d'évaluer une formule comme étant « potentiellement » vraie.

Les interprétations d'un langage du premier ordre pour la logique trivalente forte de Kleene correspondent donc à la définition d'un état abstrait que l'on souhaite obtenir, puisque ces états permettent d'interpréter les formules comme étant vraies, fausses ou inconnues. À la différence des hiérarchies d'abstractions, les différents degrés d'abstraction sont maintenant contenus dans le même espace d'états, que l'on appelle espace d'états abstraits et que l'on note S_3 (l'indice « $_3$ » faisant référence aux trois valeurs de vérité). Pour le reste de cette partie, on considère un langage logique du premier ordre appartenant à la logique trivalente forte de Kleene que l'on note $\mathcal{L}$. Ce langage utilise les mêmes symboles que le langage du premier ordre classique utilisé jusqu'ici, auxquels on ajoute deux nouveaux symboles : le symbole représentant la valeur logique « indéterminé » I, ainsi qu'un symbole représentant un objet du domaine de discours indéterminé que l'on

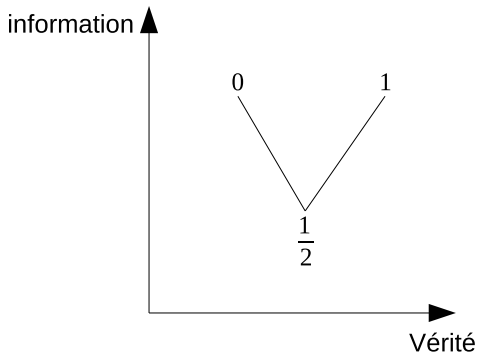


FIGURE 4.15 – Double treillis des valeurs de vérité
Treillis des trois valeurs de la logique forte de Kleene. L'abscisse représente le niveau de vérité de chaque valeur et l'ordonnée représente le niveau d'information.

note I_{obj}, et que l'on présente un peu plus loin. Voyons maintenant comment les états abstraits sont définis dans ce nouveau langage. Dans le cadre de $\mathcal{L}$, l'interprétation dans un état s d'un symbole de prédicat R d'arité n n'est plus défini par une relation sur $dom(s)^n$, mais par trois relations :

1. L'extension $s(R_{ext})$, qui est l'ensemble des tuples $(o_1, ..., o_n) \in dom(s)^n$ pour lesquels $R(o_1, ..., o_n)$ est vrai.
2. La marge $s(R_{marge})$, qui est l'ensemble des tuples $(o_1, ..., o_n) \in dom(s)^n$ pour lesquels $R(o_1, ..., o_n)$ est inconnu.
3. La contre-extension $s(R_{cext})$, qui est l'ensemble des tuples $(o_1, ..., o_n) \in dom(s)^n$ pour lesquels $R(o_1, ..., o_n)$ est faux.

Ces trois relations sont mutuellement exclusives et complémentaires dans $dom(s)^n$, c'est à dire que :

$$\begin{aligned} s(R_{ext}) \cap s(R_{cext}) &= \emptyset \\ s(R_{ext}) \cap s(R_{marge}) &= \emptyset \\ s(R_{cext}) \cap s(R_{marge}) &= \emptyset \\ s(R_{ext}) \cup s(R_{marge}) \cup s(R_{cext}) &= dom(s)^n \end{aligned}$$

Pour cette raison, on ne définit pas explicitement la contre-extension, et on considère que l'interprétation du symbole de relation R est uniquement déterminée par l'extension et la marge :

$$s(R) = (s(R_{ext}), s(R_{marge}))$$

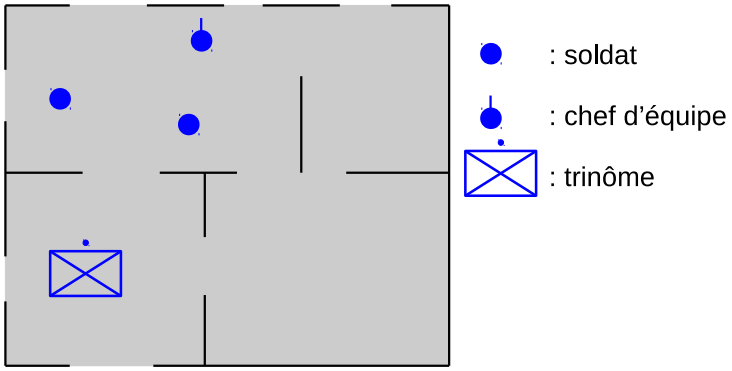
On utilise cependant la notation $s(R_{cext})$ pour simplifier l'expression de $dom(s) \setminus (s(R_{ext}) \cup s(R_{marge}))$.

En ce qui concerne les symboles de fonction, on utilise le nouveau symbole I_{obj} afin de représenter les termes fonctionnels dont la valeur est abstraite. I_{obj} est une constante associée à une valeur qui représente un objet du domaine du discours indéterminé. Cette valeur ne fait pas partie du domaine du discours, mais toute comparaison entre elle et un objet du domaine du discours est évaluée comme étant indéterminée. La définition des fonctions n'est donc pas modifiée, en dehors de l'introduction de I_{obj}. L'interprétation dans un état $s \in S_3$ d'un symbole de fonction F d'arité n vérifie donc :

$$s(F) \in dom(s)^n \rightarrow dom(s) \cup \{\mathrm{I}_{obj}\}$$

Un exemple d'état abstrait est illustré en figure 4.16. Cet exemple représente deux trinômes d'infanterie $fireteam1$ et $fireteam2$, chacun composé de trois soldats, dont

un chef de trinôme, localisé dans un bâtiment. L'état des soldats membre du trinôme $fireteam2$ et parfaitement connu, tandis que seules les informations au niveau du trinôme sont connues pour $fireteam1$. En conséquence, l'identité du chef du trinôme $fireteam1$ est inconnue (elle est donc associée à I_{obj}), et il en est de même pour les positions exactes occupées par les membres de $fireteam1$ à l'intérieur de la pièce occupée par le trinôme.



$$
\begin{aligned}
s(InRoom) &= \{(position1,\ room1),\ (position2,\ room1),\\
&\quad (position3,\ room1),\ (position4,\ room2),\\
&\quad (position5,\ room2),\ (position6,\ room2)\}\\
s(InFireteam) &= \{(soldier1,\ fireteam1),\ (soldier2,\ fireteam1),\\
&\quad (soldier3,\ fireteam1),\ (soldier4,\ fireteam2),\\
&\quad (soldier5,\ fireteam2),\ (soldier6,\ fireteam2)\}\\
s(FireteamLeader) &= \{(fireteam1,\ \mathrm{I}_{obj}),\ (fireteam2,\ soldier4)\}\\
s(FireteamAt) &= (\{(fireteam,\ room1),\ (fireteam2,\ room2)\},\ \{\})\\
s(SoldierAt) &= (\{(soldier4,\ position4),\ (soldier5,\ position5),\\
&\quad (soldier6,\ position6)\},\\
&\quad \{(soldier1,\ position1),\ (soldier1,\ position2),\\
&\quad (soldier1,\ position3),\ (soldier2,\ position1),\\
&\quad (soldier2,\ position2),\ (soldier2,\ position3),\\
&\quad (soldier3,\ position1),\ (soldier3,\ position2),\\
&\quad (soldier3,\ position3)\})
\end{aligned}
$$

FIGURE 4.16 – État abstrait en logique trivaluée
Description partielle d'un état abstrait correspondant à une structure d'interprétation pour un langage logique du premier ordre trivalué. $InRoom$, $InFireteam$ et $FireteamLeader$ sont des symboles de fonction tandis que $FireteamAt$ et $SoldierAt$ sont des symboles de prédicat.

À cause de l'introduction de la valeur I_{obj}, l'interprétation des termes fonctionnels dans un état s devient plus complexe. Considérons par exemple un terme de la forme $F(t_1, ..., t_n)$, où F est un symbole de fonction d'arité n et où les $t_1, ..., t_n$ sont des termes clos de $\mathcal{L}$. Si tous les termes sont interprétés dans s par des objets du discours, il suffit alors de retourner l'unique valeur v (qui peut être soit un objet du discours, soit I_{obj}) pour laquelle $(s(t_1), ..., s(t_n), v) \in s(F)$. Toutefois, il est aussi possible que l'un des termes t_i soit interprété par I_{obj} dans s. Dans ce cas, il n'existe pas de tuple correspondant dans $s(F)$, puisque ce dernier est un élément de $dom(s)^n \times (dom(s) \cup \{\mathrm{I}_{obj}\})$. Les termes valant I_{obj} jouent alors le rôle d'éléments de substitution pouvant être occupés par n'importe quel objet. Ainsi, si pour toutes les substitutions possibles de ces termes, la fonction retourne la même valeur v, alors le résultat de la fonction n'est pas ambigüe et v correspond bien à l'interprétation de $F(t_1, ..., t_n)$ dans s. Si tel n'est pas le cas, on considère alors que

l'interprétation de $F(t_1, ..., t_n)$ dans s est indéterminée, c'est-à-dire qu'elle correspond à la valeur I_{obj}. Formellement, pour tout $o \in dom(s)$ on a donc :

$$s(F(t_1, ..., t_n)) = o \iff O^s_{F(t_1,...,t_n)} \times \{o\} \subseteq s(F)$$

où $O^s_{F(t_1,...,t_n)} = \{(o_1, ..., o_n) \in dom(s)^n \mid \forall i \in [\![1, n]\!] : s(t_i) = o_i \vee s(t_i) = \mathrm{I}_{obj}\}$,

$$\text{et } s(F(t_1, ..., t_n)) = \mathrm{I}_{obj} \text{ sinon}$$

Le même principe s'applique pour l'interprétation des phrases de $\mathcal{L}$. L'opérateur d'évaluation des phrases de $\mathcal{L}$ dans un état s est noté $[\![\cdot]\!]^3_s \in PHRASE(\mathcal{L}) \to \{0, \frac{1}{2}, 1\}$. L'exposant « ^{3} » fait référence aux trois valeurs de vérité et permet de distinguer cet opérateur de $[\![\cdot]\!]^3_s$, qui est l'opérateur d'évaluation des phrases pour un langage logique classique à deux valeurs de vérité. L'opérateur $[\![\cdot]\!]^3_s$ associe chaque phrase de $\mathcal{L}$ avec la valeur 1 si elle est satisfaite dans s, 0 si elle ne l'est pas, ou $\frac{1}{2}$ si la satisfiabilité de cette phrase est indéterminée. On définit $[\![\cdot]\!]^3_s$ de la façon suivante :

1. $[\![\bot]\!]^3_s = 0$
2. $[\![\top]\!]^3_s = 1$
3. $[\![\mathrm{I}]\!]^3_s = \frac{1}{2}$
4. $[\![t_1 = t_2]\!]^3_s =$
   — $\frac{1}{2}$ si $s(t_1) = \mathrm{I}_{obj}$ ou $s(t_2) = \mathrm{I}_{obj}$
   — 1 sinon si $s(t_1) = s(t_2)$
   — 0 sinon
5. $[\![R(t_1, ..., t_n)]\!]^3_s =$
   — 1 si et seulement si $O^s_{R(t_1,...,t_n)} \subseteq s(R_{ext})$.
   — 0 si et seulement si $O^s_{R(t_1,...,t_n)} \subseteq s(R_{cext})$.
   — $\frac{1}{2}$ sinon
   où $O^s_{R(t_1,...,t_n)} = \{(o_1, ..., o_n) \in dom(s)^n \mid \forall i \in [\![1, n]\!] : s(t_i) = o_i \vee s(t_i) = \mathrm{I}_{obj}\}$.
6. $[\![\neg\varphi]\!]^3_s = 1 - [\![\varphi]\!]^3_s$
7. $[\![\varphi \wedge \psi]\!]^3_s = \min([\![\varphi]\!]^3_s, [\![\psi]\!]^3_s)$
8. $[\![\varphi \vee \psi]\!]^3_s = \max([\![\varphi]\!]^3_s, [\![\psi]\!]^3_s)$
9. $[\![\varphi \implies \psi]\!]^3_s = \max(1 - [\![\varphi]\!]^3_s, [\![\psi]\!]^3_s)$
10. $[\![\varphi \iff \psi]\!]^3_s = \min(\max(1 - [\![\varphi]\!]^3_s, [\![\psi]\!]^3_s), \max(1 - [\![\psi]\!]^3_s, [\![\varphi]\!]^3_s))$
11. $[\![\forall x : \varphi]\!]^3_s = \min\{[\![\varphi[o/x]]\!]^3_s \mid o \in dom(s)\}$
12. $[\![\exists x : \varphi]\!]^3_s = \max\{[\![\varphi[o/x]]\!]^3_s \mid o \in dom(s)\}$

où R représente un symbole de relation, les $t_1, t_2, ..., t_n$ représentent des termes clos de $\mathcal{L}$, φ et ψ représentent des phrases de $\mathcal{L}$, et $\varphi[t/x]$ est la formule logique obtenue en substituant les occurrences de la variable x dans la formule logique φ par le terme t.

Cette logique à trois valeurs doit nous permettre d'identifier les formules logiques fausses, de celles pouvant potentiellement être satisfaites, c'est-à-dire celles dont la valeur de vérité est soit indéterminée, soit vraie. Pour cela, on définit un nouvel opérateur pour exprimer qu'un état satisfait potentiellement une formule logique, que l'on note $\models_3$. À nouveau, l'indice « $_3$ » fait référence aux trois valeurs de vérité et permet de différencier cet opérateur et l'opérateur $\models$ qui exprime qu'un état satisfait une formule logique (et qui était déjà défini pour la logique à deux valeurs).

Définition 40 (Satisfiabilité)**.** *Une interprétation (un état) s satisfait potentiellement une phrase $\varphi \in PHRASE(\mathcal{L})$, noté $s \models_3 \varphi$, si et seulement si $[\![\varphi]\!]^3_s \geq \frac{1}{2}$.*
On dit aussi qu'un état s satisfait une phrase $\varphi \in PHRASE(\mathcal{L})$, noté $s \models \varphi$, si et seulement si $[\![\varphi]\!]^3_s = 1$.

Remarque 1. *D'après la définition précédente, on a* $s \models \varphi \implies s \models_3 \varphi$ *pour tout état* s *et phrase* φ*.*

Parmi l'ensemble des états abstraits, il existe des états pour lesquels toute formule est évaluée comme étant fausse ou vraie. Ces états correspondent aux états décrits par un langage du premier ordre classique contenant les mêmes symboles, c'est-à-dire aux états utilisés d'ordinaire en planification, et qui correspondent aux états concrets pour les hiérarchies d'abstraction. En conséquence, on les appelle des « états concrets ».

Définition 41 (État concret)**.** *Soit un état* $s \in S_3$*. On dit que* s *est un état concret si et seulement pour tout symbole de prédicat* R *d'arité* n *:*

$$s(R_{marge}) = \emptyset$$

et pour tout symbole de fonction F *d'arité* n *:*

$$\{(o_1, ..., o_n, v) \in s(F) \mid v = \mathrm{I}_{obj}\} = \emptyset$$

On note alors S *l'ensemble des états concrets.*

Proposition 13. *Pour tout état concret* $s \in S$ *et phrase* $\varphi \in PHRASE(\mathcal{L})$*, on a :*

$$[\![\varphi]\!]^3_s \neq \frac{1}{2}$$

Démonstration. Soit un état $s \in S$.
Dans un premier temps, montrons que pour tout terme $t \in TERME_c(\mathcal{L})$, $s(t) \neq \mathrm{I}_{obj}$. On procède par récurrence sur la structure des termes.

1. Cas de base : $F()$
   Soit F un symbole de fonction d'arité 0.
   Soit $o \in dom(s) \cup \{\mathrm{I}_{obj}\}$ tel que $s(F) = \{o\}$.
   D'après la définition 41, on en déduit que $o \neq \mathrm{I}_{obj}$ et donc que $s(F()) \neq \mathrm{I}_{obj}$.
2. Récurrence : $F(t_1, ..., t_n)$
   Soient F un symbole de fonction d'arité n et $(t_1, ...t_n) \in TERME_c(\mathcal{L})^n$.
   En tant qu'hypothèse de récurrence, on suppose que pour tout $i \in [\![1, n]\!]$, $s(t_i) \neq \mathrm{I}_{obj}$.
   Comme les fonctions sont complètement définies, il existe un $o \in dom(s) \cup \{\mathrm{I}_{obj}\}$ tel que $(s(t_1), ..., s(t_2), o) \in s(F)$.
   Or d'après la définition 41, $o \neq \mathrm{I}_{obj}$.
   Comme $s(F(t_1, ..., t_n)) = o$, on en déduit donc que $s(F(t_1, ..., t_n)) \neq \mathrm{I}_{obj}$.

Ainsi on a $s(t) \neq \mathrm{I}_{obj}$ pour tout terme $t \in TERME_c(\mathcal{L})$.
Montrons maintenant que pour toute phrase $\varphi \in PHRASE(\mathcal{L})$, $[\![\varphi]\!]^3_s \neq \frac{1}{2}$.
Comme pour les termes, on procède par récurrence sur la structure des phrases de $\mathcal{L}$.

1. Cas de base :
   (a) $t_1 = t_2$:
   Soit $(t_1, t_2) \in TERME_c(\mathcal{L})^2$.
   $s(t_1) \neq \mathrm{I}_{obj}$ et $s(t_2) \neq \mathrm{I}_{obj}$, donc $[\![t_1 = t_2]\!]^3_s \neq \frac{1}{2}$.
   (b) $R(t_1, ..., t_n)$: Soient $R \in PRED(\mathcal{L})$ d'arité n et $(t_1, ..., t_n) \in TERME_c(\mathcal{L})^n$.
   On sait que pour tout $i \in [\![1, n]\!]$ on a $s(t_i) \neq \mathrm{I}_{obj}$, donc $O^s_{R(t_1,...,t_n)} = \{(s(t_1), ..., s(t_n))\}$.
   Or d'après la définition 41, on a $s(R_{marge}) = \emptyset$, donc $(s(t_1), ..., s(t_n)) \notin s(R_{marge})$ et donc $O^s_{R(t_1,...,t_n)} \subseteq s(R_{ext})$ ou $O^s_{R(t_1,...,t_n)} \subseteq s(R_{cext})$.
   On en déduit donc que $[\![R(t_1, ..., t_2)]\!]^3_s \neq \frac{1}{2}$.

2. Récursivité :

(a) $\neg\varphi$:
Soit $\varphi \in PHRASE(\mathcal{L})$ telle que $[\![\varphi]\!]^3_s \neq \frac{1}{2}$.
On a :

$$[\![\neg\varphi]\!]^3_s = 1 - [\![\varphi]\!]^3_s$$

or $[\![\varphi]\!]^3_s \in \{0, 1\}$ donc $[\![\neg\varphi]\!]^3_s \in \{0, 1\}$ et donc $[\![\neg\varphi]\!]^3_s \neq \frac{1}{2}$.

(b) $\varphi \wedge \phi$, $\varphi \vee \phi$, $\varphi \implies \phi$ et $\varphi \iff \phi$:
Soient deux phrases $\varphi, \psi \in PHRASE(\mathcal{L})$.
On a $[\![\varphi]\!]^3_s \neq \frac{1}{2}$ et $[\![\psi]\!]^3_s \neq \frac{1}{2}$ donc :

$$\begin{aligned}
[\![\varphi \wedge \psi]\!]^3_s &= \min([\![\varphi]\!]^3_s, [\![\psi]\!]^3_s) \\
&\neq \frac{1}{2} \\
[\![\varphi \vee \psi]\!]^3_s &= \max([\![\varphi]\!]^3_s, [\![\psi]\!]^3_s) \\
&\neq \frac{1}{2} \\
[\![\varphi \implies \psi]\!]^3_s &= \max(1 - [\![\varphi]\!]^3_s, [\![\psi]\!]^3_s) \\
&\neq \frac{1}{2} \\
[\![\varphi \iff \psi]\!]^3_s &= \min(\max(1 - [\![\varphi]\!]^3_s, [\![\psi]\!]^3_s), \max(1 - [\![\psi]\!]^3_s, [\![\varphi]\!]^3_s)) \\
&\neq \frac{1}{2}
\end{aligned}$$

(c) $\exists x : \varphi$ et $\forall x : \varphi$:
Soit φ une formule de $\mathcal{L}$ contenant une unique variable libre x.
On suppose que pour tout $o \in dom(s)$: $[\![\varphi[o/x]]\!]^3_s \neq \frac{1}{2}$.
On a alors :

$$\begin{aligned}
[\![\forall x : \varphi]\!]^3_s &= \min\{[\![\varphi[o/x]]\!]^3_s \mid o \in dom(s)\} \\
&\neq \frac{1}{2} \\
[\![\exists x : \varphi]\!]^3_s &= \max\{[\![\varphi[o/x]]\!]^3_s \mid o \in dom(s)\} \\
&\neq \frac{1}{2}
\end{aligned}$$

Ainsi pour toute phrase φ de $\mathcal{L}$, on a $[\![\varphi]\!]^3_s \neq \frac{1}{2}$. □

En l'absence d'information abstraite pour les états concrets, on peut remarquer que la satisfiabilité et la satisfiabilité potentielle deviennent deux concepts équivalents :

Proposition 14. *Soient un état concret $s \in S$ et une phrase $\varphi \in PHRASE(\mathcal{L})$. On a alors :*

$$s \models \varphi \iff s \models_3 \varphi$$

Démonstration. □

D'après la définition 40, on a $s \models \varphi \implies s \models_3 \varphi$. Il suffit donc de montrer que $s \models_3 \varphi \implies s \models \varphi$.
Si $s \models_3 \varphi$ alors $[\![\varphi]\!]^3_s = 1$ ou $\frac{1}{2}$. Comme s est un état concret, on en déduit d'après la proposition 13 que $[\![\varphi]\!]^3_s \neq \frac{1}{2}$. Par conséquent, on en déduit que $[\![\varphi]\!]^3_s = 1$ et donc, d'après la définition 40, que $s \models \varphi$.

Comme tous les états abstraits appartiennent désormais au même espace, le lien entre un état et un état abstrait n'est plus identifié par une fonction d'abstraction entre deux espaces d'états, mais par une relation entre état abstraits. On dit alors qu'un état est une abstraction d'un autre état lorsque le premier s'obtient à partir du second en en diminuant la quantité d'information. Cette relation forme un ordre partiel sur les états abstraits, et admet un unique élément maximal, qui est l'état où toute information est abstraite (c'est-à-dire que les extensions et contre-extension de tout symbole de prédicat sont vides, et que I_{obj} est l'unique valeur retournée par une fonction). Contrairement au treillis de la figure 4.14, cet ordre partiel n'admet pas d'élément minimal unique, car tout état concret est un élément minimal. Pour retrouver le treillis de la figure 4.14, il faut définir un ensemble de classes regroupant les états dont l'information abstraite est identique, et définir un ordre partiel sur ces classes. La classe des états concrets est alors l'unique élément minimal, tandis que la classe qui ne contient que l'état dont toute l'information est abstraite est l'unique élément maximal.

Définition 42 (Ordre d'abstraction)**.** *Soient deux états $s_1, s_2 \in S_3$. On dit que s_1 est une concrétisation de s_2 (ou que s_2 est une abstraction de s_1), noté $s_1 \sqsubseteq s_2$, si et seulement si, pour tout symbole de prédicat R d'arité n :*

$$\begin{aligned} s_2(R_{ext}) &\subseteq s_1(R_{ext}) \\ s_2(R_{cext}) &\subseteq s_1(R_{cext}) \\ s_1(R_{marge}) &\subseteq s_2(R_{marge}) \end{aligned}$$

et pour tout symbole de fonction F d'arité n :

$$\begin{aligned} s_2(F) \cap dom(s_1)^{n+1} &\subseteq s_2(F) \cap dom(s_2)^{n+1} \\ s_1(F) \cap (dom(s_1)^n \times \{\mathrm{I}_{obj}\}) &\subseteq s_2(F) \cap (dom(s_2)^n \times \{\mathrm{I}_{obj}\}) \end{aligned}$$

En conséquence, si un état est une abstraction d'un autre état, toute phrase est évaluée dans le premier avec la même valeur que dans le second, ou devient indéterminée :

Proposition 15. *Pour toute paire d'états $(s_1, s_2) \in S_3^2$ tel que $s_1 \sqsubseteq s_2$ et toute phrase $\varphi \in PHRASE(\mathcal{L})$:*

$$[\![\varphi]\!]^3_{s_2} = [\![\varphi]\!]^3_{s_1} \vee [\![\varphi]\!]^3_{s_2} = \frac{1}{2}$$

Démonstration. Soit $(s_1, s_2) \in S_3^2$ tel que $s_1 \sqsubseteq s_2$.
Dans un premier temps, montrons que pour tout terme $t \in TERME_c(\mathcal{L})$, $s_2(t) = s_1(t)$ ou $s_2(t) = \mathrm{I}_{obj}$. Pour cela, on procède par récurrence sur la structure des termes.

1. Cas de base : $F()$
   Soit F un symbole de fonction d'arité 0.
   Supposons que $s_2(F()) \neq \mathrm{I}_{obj}$. Il existe alors un $o \in dom(s)$ tel que $s_2(F) = \{o\}$.
   Or, d'après la définition 42, on a $o \in s_2(F) \implies o \in s_1(F)$.
   On en déduit donc que $s_1(F()) = o$, et donc que $s_2(F()) = s_1(F())$. On a donc bien $s_2(F()) = s_1(F()) \vee s_2(F()) = \mathrm{I}_{obj}$.
2. Récursivité : $F(t_1, ..., t_n)$
   Soient F un symbole de fonction d'arité n et $(t_1, ..., t_n) \in TERME_c(\mathcal{L})^n$.
   En tant qu'hypothèse de récurrence, on suppose que pour tout $i \in [\![1, n]\!]$, on a $s_2(t_i) = s_1(t_i)$ ou $s_2(t_i) = \mathrm{I}_{obj}$.
   On en déduit alors :
   $$O^{s_1}_{F(t_1,...,t_n)} \subseteq O^{s_2}_{F(t_1,...,t_n)}$$
   On suppose que $s_2(F(t_1, ..., t_n)) \neq \mathrm{I}_{obj}$.
   Il existe donc un $o \in dom(s_1)$ tel que $s_2(F(t_1, ..., t_n)) = o$.

On en déduit donc que $O^{s_2}_{F(t_1,...,t_n)} \times \{o\} \subseteq s_2(F)$, et donc que $O^{s_1}_{F(t_1,...,t_n)} \times \{o\} \subseteq s_2(F)$.
Soit $(o_1, ..., o_n) \in O^{s_1}_{F(t_1,...,t_n)}$, on a alors $(o_1, ..., o_n, o) \in s_2(F)$.
D'après la définition 42, on en déduit que $(o_1, ..., o_n, o) \in s_1(F)$.
On en conclut donc que $O^{s_1}_{F(t_1,...,t_n)} \times \{o\} \subseteq s_1(F)$, et donc $s_1(F(t_1, ..., t_n)) = o$.
Ainsi, on a $s_2(F(t_1, ..., t_n)) = s_1(F(t_1, ..., t_n))$ ou $s_1(F(t_1, ..., t_n)) = \mathrm{I}_{obj}$.

Ainsi pour tout terme $t \in TERME_c(\mathcal{L})$, on a $s_2(t) = s_1(t)$ ou $s_2(t) = \mathrm{I}_{obj}$.

Montrons maintenant que pour toute phrase $\varphi \in PHRASE(\mathcal{L})$, on a $[\![\varphi]\!]^3_{s_2} = [\![\varphi]\!]^3_{s_1}$ ou $[\![\varphi]\!]^3_{s_2} = \frac{1}{2}$.
Comme pour les termes, on procède par récurrence sur la structure des phrases de $\mathcal{L}$.

1. Cas de base :
   (a) $t_1 = t_2$:
   Soit $(t_1, t_2) \in TERME_c(\mathcal{L})^2$.
   $s_2(t_1) = s_1(t_1)$ ou $s_2(t_1) = \mathrm{I}_{obj}$ et $s_2(t_2) = s_1(t_2)$ ou $s_2(t_2) = \mathrm{I}_{obj}$ donc $[\![t_1 = t_2]\!]^3_{s_2} = [\![t_1 = t_2]\!]^3_{s_2}$ ou $[\![t_1 = t_2]\!]^3_{s_2} = \frac{1}{2}$.
   (b) $R(t_1, ..., t_n)$:
   Soit $R \in PRED(\mathcal{L})$ d'arité n, et $(t_1, ..., t_n) \in TERME_c(\mathcal{L})^n$.

      i. Si $[\![R(t_1, ..., t_2)]\!]^3_{s_1} = 1$:
      Supposons que $[\![R(t_1, ..., t_2)]\!]^3_{s_2} = 0$.
      Comme pour tout $i \in [\![1, n]\!]$ on a $s_2(t_i) = s_1(t_i)$ ou $s_2(t_i) = \mathrm{I}_{obj}$, on en déduit que :
      $$O^{s_1}_{R(t_1,...,t_n)} \subseteq O^{s_2}_{R(t_1,...,t_n)}$$
      Comme $[\![R(t_1, ..., t_2)]\!]^3_{s_1} = 1$, on en déduit que $O^{s_1}_{R(t_1,...,t_n)} \subseteq s_1(R_{ext})$.
      De plus par hypothèse on a $[\![R(t_1, ..., t_2)]\!]^3_{s_2} = 0$, ce qui implique que $O^{s_2}_{R(t_1,...,t_n)} \subseteq s_2(R_{cext})$.
      On en déduit donc que $O^{s_1}_{R(t_1,...,t_n)} \subseteq s_2(R_{cext})$.
      D'après la définition 42, on en déduit donc que $O^{s_2}_{R(t_1,...,t_n)} \subseteq s_1(R_{cext})$.
      Comme $s_1(R_{ext}) \cap s_1(R_{cext}) = \emptyset$, on obtient donc une contradiction. On en déduit donc que $[\![R(t_1, ..., t_2)]\!]^3_{s_2} \in \{\frac{1}{2}, 1\}$, et donc que $[\![R(t_1, ..., t_2)]\!]^3_{s_2} = [\![R(t_1, ..., t_2)]\!]^3_{s_1}$ ou que $[\![R(t_1, ..., t_2)]\!]^3_{s_2} = \frac{1}{2}$.
      ii. Si $[\![R(t_1, ..., t_2)]\!]^3_{s_1} = 0$:
      La démonstration est analogue au cas précédent et est donc omise.
      iii. Si $[\![R(t_1, ..., t_2)]\!]^3_{s_1} = \frac{1}{2}$:
      Comme pour tout $i \in [\![1, n]\!]$ on a $s_2(t_i) = s_1(t_i)$ ou $s_2(t_i) = \mathrm{I}_{obj}$, on en déduit que :
      $$O^{s_1}_{R(t_1,...,t_n)} \subseteq O^{s_2}_{R(t_1,...,t_n)}$$
      A. Supposons que $[\![R(t_1, ..., t_2)]\!]^3_{s_2} = 1$.
      On a alors $O^{s_2}_{R(t_1,...,t_n)} \subseteq s_2(R_{ext})$, et donc $O^{s_1}_{R(t_1,...,t_n)} \subseteq s_2(R_{ext})$. D'après la définition 42, on en déduit donc que $O^{s_1}_{R(t_1,...,t_n)} \subseteq s_1(R_{ext})$, ce qui est équivalent à $[\![R(t_1, ..., t_2)]\!]^3_{s_1} = 1$, qui est une contradiction.
      B. Supposons que $[\![R(t_1, ..., t_2)]\!]^3_{s_2} = 0$.
      Par une démonstration analogue au cas précédent, on obtient aussi une contradiction.
      On en déduit donc que $[\![R(t_1, ..., t_2)]\!]^3_{s_2} = \frac{1}{2}$.

2. Récursivité :

(a) $\neg\varphi$:
Soit $\varphi \in PHRASE(\mathcal{L})$ telle que $[\![\varphi]\!]^3_{s_2} = [\![\varphi]\!]^3_{s_1}$ ou $[\![\varphi]\!]^3_{s_2} = \frac{1}{2}$.
On suppose que $[\![\neg\varphi]\!]^3_{s_2} \neq \frac{1}{2}$.
On a alors :

$$\begin{aligned}
[\![\neg\varphi]\!]^3_{s_2} \neq \frac{1}{2} &\implies 1 - [\![\varphi]\!]^3_{s_2} \neq \frac{1}{2} \\
&\implies [\![\varphi]\!]^3_{s_2} \neq \frac{1}{2} \\
&\implies [\![\varphi]\!]^3_{s_2} = [\![\varphi]\!]^3_{s_1}
\end{aligned}$$

(b) $\varphi\Box\psi$ (où $\Box$ correspond à $\wedge$, $\vee$, $\implies$ ou $\iff$) :
Soient deux phrases $\varphi, \psi \in PHRASE(\mathcal{L})$ telle que $[\![\varphi]\!]^3_{s_2} = [\![\varphi]\!]^3_{s_1} \vee [\![\varphi]\!]^3_{s_2} = \frac{1}{2}$ et $[\![\psi]\!]^3_{s_2} = [\![\psi]\!]^3_{s_1} \vee [\![\psi]\!]^3_{s_2} = \frac{1}{2}$.

i. Si $[\![\varphi\Box\psi]\!]^3_{s_2} = 1$:

$$\begin{aligned}
[\![\varphi \wedge \psi]\!]^3_{s_2} = 1 &\implies \min([\![\varphi]\!]^3_{s_2}, [\![\psi]\!]^3_{s_2}) = 1 \\
&\implies [\![\varphi]\!]^3_{s_2} = 1 \wedge [\![\psi]\!]^3_{s_2} = 1 \\
&\implies [\![\varphi]\!]^3_{s_1} = 1 \wedge [\![\psi]\!]^3_{s_1} = 1 \\
&\implies \min([\![\varphi]\!]^3_{s_1}, [\![\psi]\!]^3_{s_1}) = 1 \\
&\implies [\![\varphi \wedge \psi]\!]^3_{s_1} = 1
\end{aligned}$$

$$\begin{aligned}
[\![\varphi \vee \psi]\!]^3_{s_2} = 1 &\implies \max([\![\varphi]\!]^3_{s_2}, [\![\psi]\!]^3_{s_2}) = 1 \\
&\implies [\![\varphi]\!]^3_{s_2} = 1 \vee [\![\psi]\!]^3_{s_2} = 1 \\
&\implies [\![\varphi]\!]^3_{s_1} = 1 \vee [\![\psi]\!]^3_{s_1} = 1 \\
&\implies \max([\![\varphi]\!]^3_{s_1}, [\![\psi]\!]^3_{s_1}) = 1 \\
&\implies [\![\varphi \vee \psi]\!]^3_{s_1} = 1
\end{aligned}$$

$$\begin{aligned}
[\![\varphi \implies \psi]\!]^3_{s_2} = 1 &\implies \max(1 - [\![\varphi]\!]^3_{s_2}, [\![\psi]\!]^3_{s_2}) \\
&\implies 1 - [\![\varphi]\!]^3_{s_2} = 1 \vee [\![\psi]\!]^3_{s_2} = 1 \\
&\implies [\![\varphi]\!]^3_{s_2} = 0 \vee [\![\psi]\!]^3_{s_2} = 1 \\
&\implies [\![\varphi]\!]^3_{s_1} = 0 \vee [\![\psi]\!]^3_{s_1} = 1 \\
&\implies 1 - [\![\varphi]\!]^3_{s_1} = 1 \vee [\![\psi]\!]^3_{s_1} = 1 \\
&\implies \max(1 - [\![\varphi]\!]^3_{s_1}, [\![\psi]\!]^3_{s_1}) \\
&\implies [\![\varphi \implies \psi]\!]^3_{s_1} = 1
\end{aligned}$$

$$\begin{aligned}
[\![\varphi \iff \psi]\!]^3_{s_2} = 1 &\implies \min(\max(1 - [\![\varphi]\!]^3_{s_2}, [\![\psi]\!]^3_{s_2}), \max([\![\psi]\!]^3_{s_2}, [\![\varphi]\!]^3_{s_2})) = 1 \\
&\implies \max(1 - [\![\varphi]\!]^3_{s_2}, [\![\psi]\!]^3_{s_2}) = 1 \wedge \max(1 - [\![\psi]\!]^3_{s_2}, [\![\varphi]\!]^3_{s_2}) = 1 \\
&\implies ([\![\varphi]\!]^3_{s_2} = 0 \vee [\![\psi]\!]^3_{s_2} = 1) \wedge ([\![\psi]\!]^3_{s_2} = 0 \vee [\![\varphi]\!]^3_{s_2} = 1) \\
&\implies ([\![\varphi]\!]^3_{s_1} = 0 \vee [\![\psi]\!]^3_{s_1} = 1) \wedge ([\![\psi]\!]^3_{s_1} = 0 \vee [\![\varphi]\!]^3_{s_1} = 1) \\
&\implies \max(1 - [\![\varphi]\!]^3_{s_1}, [\![\psi]\!]^3_{s_1}) = 1 \wedge \max(1 - [\![\psi]\!]^3_{s_1}, [\![\varphi]\!]^3_{s_1}) = 1 \\
&\implies \min(\max(1 - [\![\varphi]\!]^3_{s_1}, [\![\psi]\!]^3_{s_1}), \max([\![\psi]\!]^3_{s_1}, [\![\varphi]\!]^3_{s_1})) = 1 \\
&\implies [\![\varphi \iff \psi]\!]^3_{s_1} = 1
\end{aligned}$$

ii. Si $[\![\varphi \Box \psi]\!]^3_{s_2} = 0$:
Par une démonstration analogue au cas précédent, on en déduit que $[\![\varphi \Box \psi]\!]^3_{s_1} = 0$.

Ainsi on a $[\![\varphi \Box \psi]\!]^3_{s_2} = [\![\varphi \Box \psi]\!]^3_{s_1}$ ou $[\![\varphi \Box \psi]\!]^3_{s_2} = \frac{1}{2}$.

(c) $\forall x : \varphi$:
Soit φ une formule de $\mathcal{L}$ contenant une unique variable libre x.
On suppose que pour tout $o \in dom(s)$: $[\![\varphi[o/x]]\!]^3_{s_2} = [\![\varphi[o/x]]\!]^3_{s_1}$ ou $[\![\varphi[o/x]]\!]^3_{s_2} = \frac{1}{2}$.
Supposons que $[\![\forall x : \varphi]\!]^3_{s_2} \neq \frac{1}{2}$.

i. Si $[\![\forall x : \varphi]\!]^3_{s_2} \neq 0$:
Il existe alors $o \in dom(s)$ tel que $[\![\varphi[o/x]]\!]^3_{s_2} = 0$.
D'après l'hypothèse de récurrence, on en déduit alors que $[\![\varphi[o/x]]\!]^3_{s_1} = 0$, et donc que $[\![\forall x : \varphi]\!]^3_{s_1} = 0$.

ii. Si $[\![\forall x : \varphi]\!]^3_{s_2} \neq 1$:
Alors pour tout $o \in dom(s)$ on a $[\![\varphi[o/x]]\!]^3_{s_2} = 1$.
D'après l'hypothèse de récurrence, on en déduit alors que pour tout $o \in dom(s)$, $[\![\varphi[o/x]]\!]^3_{s_1} = 1$, et donc que $[\![\forall x : \varphi]\!]^3_{s_1} = 1$.

On en déduit donc que $[\![\forall x : \varphi]\!]^3_{s_2} = [\![\forall x : \varphi]\!]^3_{s_1}$ ou que $[\![\forall x : \varphi]\!]^3_{s_2} = \frac{1}{2}$.

(d) $\exists x : \varphi$:
La démonstration est analogue au cas précédent, et est donc omise.

Ainsi pour toute phrase φ de $\mathcal{L}$, on a $[\![\varphi]\!]^3_{s_2} = [\![\varphi]\!]^3_{s_1}$ ou $[\![\varphi]\!]^3_{s_2} = \frac{1}{2}$. □

Enfin, on en déduit directement que toute phrase potentiellement satisfaite par un état est aussi potentiellement satisfaite dans toutes les abstractions de cet état :

Corollaire 1. *Soit une paire d'états $(s_1, s_2) \in S_3^2$ telle que $s_1 \sqsubseteq s_2$ et une phrase $\varphi \in PHRASE(\mathcal{L})$. On a alors :*

$$s_1 \models_3 \varphi \implies s_2 \models_3 \varphi$$

Démonstration. $s_1 \models_3 \varphi$ est équivalent à $[\![\varphi]\!]^3_{s_1} \geq \frac{1}{2}$.
Comme d'après la proposition 15, on a $[\![\varphi]\!]^3_{s_2} = [\![\varphi]\!]^3_{s_1}$ ou $[\![\varphi]\!]^3_{s_2} = \frac{1}{2}$, on en déduit que $[\![\varphi]\!]^3_{s_2} \geq \frac{1}{2}$, et donc que $s_2 \models_3 \varphi$. □

4.3.2 Actions abstraites

Avec ce système d'états abstraits, toute tâche composée peut être associée à son propre niveau d'abstraction. En effet, il devient possible d'indiquer les éléments dont la valeur est indéterminée après l'exécution de n'importe quelle tâche. Pour spécifier la transformation d'un élément d'un état en élément indéterminé, on peut employer le même moyen que pour tout autre type de transformation effectué par une action : les effets. Ainsi toute tâche composée peut désormais se modéliser comme une action, en y ajoutant une catégorie supplémentaire d'effets pour modéliser les opérations d'abstraction. D'une part, en ce qui concerne les relations, on ajoute une troisième catégorie d'effets en plus des effets « d'ajout » et de « suppression » avec une catégorie d'effets « d'abstraction ». D'autre part, en ce qui concerne les fonctions, on conserve toujours une seule catégorie, à savoir les effets « d'affectation », mais on y autorise l'utilisation de la constante I_{obj} pour indiquer les valeurs qui deviennent indéterminées. Ces effets sont toujours modélisés par des formules logiques dans le domaine de planification. De plus, chaque tâche composée est aussi associée à un terme numérique (entier positif) qui représente son niveau dans la hiérarchie des tâches. Ce niveau remplace le niveau d'abstraction introduit avec les hiérarchies

d'abstractions, et est utilisé pour modéliser l'ordre d'abstraction entre les tâches. Ainsi, en plus des méthodes de décomposition, chaque tâche composée est désormais modélisée avec les mêmes éléments que les actions, à l'exception des coûts, et en y ajoutant les effets abstraits et le niveau. On désigne alors ce type de tâche composée par l'expression « action abstraite », et on note A_3 l'ensemble de ces tâches. Le sous-ensemble des actions abstraites qui ne sont associées à aucune méthode de décomposition correspond alors à l'ensemble des tâches primitives, ou « actions concrètes », toujours noté A.

Définition 43 (Description d'une action abstraite)**.** *Une action abstraite $a \in A_3$ est décrite par l'ensemble des expressions de $\mathcal{L}$ suivantes :*

- *Une condition d'exécution (aussi appelée préconditions) π^a qui définit l'ensemble des états auxquels a est applicable : $\forall s \in S_3, s \models_3 \pi^a$ si et seulement si a est potentiellement applicable à s.*
- *Des conditions d'ajout $\alpha^a_R(x_1, ..., x_n)$, de suppression $\delta^a_R(x_1, ..., x_n)$ et d'abstraction $\iota^a_R(x_1, ..., x_n)$ pour chaque symbole de relation R d'arité $n \in \mathbb{N}$ où $x_1, ..., x_n$ sont des variables libres de $\mathcal{L}$.*
- *Une condition d'affectation $\mu^a_F(x_1, ..., x_n, y)$ pour chaque symbole de fonction F d'arité $n \in \mathbb{N}$ où $x_1, ..., x_n, y$ sont des variables libres de $\mathcal{L}$.*
- *Un niveau ν^a tel que pour tout $s \in S_3$, $s(\nu^a) \in \mathbb{N}$ et $\nu^a(s) = 0$ si $a \in A$.*

À noter que les tâches primitives ne peuvent pas avoir d'effets d'abstraction. En conséquence, toutes les conditions d'abstraction de ces tâches sont équivalentes à $\bot$, et le symbole I_{obj} ne peut pas apparaître dans leurs conditions d'affectation.

La figure 45 apporte un exemple de description d'une action abstraite dans le langage HTDL. L'exemple contient une tâche abstraite `VehicleAreaTravel(v, a1, a2)` dont toutes les décompositions sont formées d'une unique tâche primitive, qui est une instance de `VehicleTravel(v, p1, p2, a1, a2)`. Ce court domaine décrit le déplacement d'un véhicule (par exemple, un véhicule de transport de troupe ou de combat d'infanterie) à travers des positions qui sont regroupées par zone. L'action abstraite fait alors abstraction de la position occupée par le véhicule, et ne s'intéresse qu'à la zone dans laquelle il se situe. La consommation du véhicule en carburant est aussi modélisée par une fonction, ce qui permet à cet exemple d'illustrer l'abstraction au niveau des conditions d'abstraction et des conditions d'affectation. Enfin, la tâche `VehicleTravel(v, p1, p2, a1, a2)` étant primitive, son niveau n'est pas modélisé explicitement, puisqu'il est par défaut affecté à la valeur 0.

Dans le langage de modélisation HTDL, les formules associées aux effets suivent les mêmes règles que celles définies auparavant pour les actions concrètes (voir la table 2.1.2). On obtient donc les mêmes types d'effets : atomique, conditionnel et universellement quantifié. L'exécution d'une action abstraite potentiellement applicable tient désormais compte de ces effets abstraits. Comme pour la planification d'actions classique (avec ADL), cette transformation s'exprime par les ensembles de substitutions qui satisfont les conditions d'ajout, de suppression, d'affectation ainsi que les conditions d'abstractions. À ces substitutions qui satisfont directement les conditions associées aux effets s'ajoutent aussi des substitutions « indirectes », qui satisfont potentiellement ces conditions mais ne les satisfont pas (leur évaluation dans l'état courant est indéterminée). Bien qu'elles ne proviennent pas explicitement des conditions d'abstraction, ces substitutions indirectes y sont assimilées dans la mesure où leur appartenance définitive aux effets d'ajout (ou de suppression) est indéterminée. Ainsi, même une action concrète peut abstraire un état si certaines de ces conditions d'effet dépendent d'éléments qui sont indéterminés dans cet état.

L'interprétation dans l'état courant de chaque relation est alors transformée de la façon suivante : on ajoute les substitutions qui satisfont les conditions d'ajout à l'extension, et on

```
task VehicleAreaTravel(v, a1, a2):
    precond:
        Vehicle(v) and Area(a1) and Area(a2)
        VehicleAt(v, a1)
    add:
        VehicleAtArea(v, a2)
    delete:
        VehicleAt(v, p) forall p if Position(p) and InArea(p) == a1
        VehicleAtArea(v, a1)
    abstract:
        VehicleAt(v, p) forall p if Position(p) and InArea(p) == a2
    update:
        FuelLevel(v) = Indeterminate
    method p1, p2:
        precond:
            Position(p1) and Position(p2)
            VehicleAt(p1) and InArea(p2) == a2
        subtasks:
            VehicleTravel(v, p1, p2, a1, a2)
    level: 1

task VehicleTravel(v, p1, p2, a1, a2):
    precond:
        Vehicle(v) and Position(p1) and Position(p2)
        Area(a1) and Area(a2) and InArea(p1) == a1 and InArea(p2) == a2
        VehicleAt(v, p1)
        FuelLevel(v) > FuelConsumption(v) * Distance(p1, p2)
    add:
        VehicleAt(v, p2)
        VehicleAtArea(v, a2)
    delete:
        VehicleAt(v, p1)
        VehicleAtArea(v, a1)
    update:
        FuelLevel(v) = FuelLevel(v) - FuelConsumption(v) * Distance(p1, p2)
```

FIGURE 4.17 – Action abstraite
Exemple d'action abstraite en HTDL. Dans `VehicleAreaTravel` le champ `abstract:` permet de modéliser l'abstraction de la position exacte du véhicule dans la zone de destination, et la constante `Indeterminate` permet d'abstraire la valeur de la fonction `FuelLevel`.

en supprime les substitutions qui satisfont potentiellement les conditions de suppression et d'abstraction, sans oublier les substitutions indirectes des conditions d'ajout. En ce qui concerne la marge, on supprime les substitutions qui satisfont les conditions d'ajout et de suppression, puis on ajoute toutes les autres, à savoir les substitutions qui satisfont potentiellement les conditions d'abstraction, et celles pour lesquelles l'évaluation des formules d'ajout et de suppression est indéterminée.

Du côté des fonctions, on supprime d'abord toutes les substitutions qui satisfont potentiellement leurs conditions d'affectation, on ajoute celles pour lesquelles la condition d'affectation est satisfaite, et on ajoute celles pour lesquelles cette formule est indéterminée

en remplaçant par I_{obj} la valeur retournée par la fonction.

L'ensemble des états abstraits ainsi que l'ensemble des actions abstraites définissent ainsi un système de transition déterministe que l'on qualifie « d'abstrait » (pour le distinguer du système de transition « classique »). On note alors $\mathcal{T}_3 \in S_3 \times A_3 \to S_3$ la fonction de transition qui est associée à ce système.

Définition 44 (Progression d'un état)**.** *Soient une action $a \in A_3$ et deux états $s, s' \in S_3$ tels que $s \models_3 \pi^a$ et $\mathcal{T}_3(s,a) = s'$. Alors on peut calculer s' de la façon suivante :*

— *$dom(s') = dom(s)$.*

— *pour tout symbole de relation R d'arité $n \in \mathbb{N}$, on a :*

$$s'(R_{ext}) = (s(R_{ext}) \setminus (D_R \cup I^3_R \cup A^3_R \cup D^3_R)) \cup A_R$$
$$s'(R_{marge}) = (s(R_{marge}) \setminus (D_R \cup A_R)) \cup I^3_R \cup D^3_R \cup A^3_R$$

où :

$$A_R = \{(o_1,...,o_n) \in dom(s)^n \mid s \models \alpha^a_R(x_1,...,x_n)[o_1/x_1,...,o_n/x_n]\}$$
$$D_R = \{(o_1,...,o_n) \in dom(s)^n \mid s \models \delta^a_R(x_1,...,x_n)[o_1/x_1,...,o_n/x_n]\}$$
$$I^3_R = \{(o_1,...,o_n) \in dom(s)^n \mid s \models_3 \iota^a_R(x_1,...,x_n)[o_1/x_1,...,o_n/x_n]\}$$
$$A^3_R = \{(o_1,...,o_n) \in dom(s)^n \mid [\![\alpha^a_R(x_1,...,x_n)[o_1/x_1,...,o_n/x_n]]\!]^3_s = \frac{1}{2}\}$$
$$D^3_R = \{(o_1,...,o_n) \in dom(s)^n \mid [\![\delta^a_R(x_1,...,x_n)[o_1/x_1,...,o_n/x_n]]\!]^3_s = \frac{1}{2}\}$$

— *pour tout symbole de fonction F d'arité $n \in \mathbb{N}$, on a :*

$$s'(F) = (s(F) \setminus D_F) \cup A_F \cup I^3_F$$

où :

$$A_F = \{(o_1,...,o_n,v) \in dom(s)^{n+1} \mid s \models \mu^a_F(x_1,...,x_n,y)[o_1/x_1,...,o_n/x_n,v/y]\}$$
$$D_F = \{(o_1,...,o_n,v) \in dom(s)^n \times \{\mathrm{I}_{obj}\} \mid s \models_3 (\exists y : \mu^a_F(x_1,...,x_n,y))[o_1/x_1,...,o_n/x_n]\}$$
$$I^3_F = \{(o_1,...,o_n,v) \in dom(s)^n \times \{\mathrm{I}_{obj}\} \mid [\![(\exists y : \mu^a_F(x_1,...,x_n,y))[o_1/x_1,...,o_n/x_n]]\!]^3_s\}$$

Comme pour ADL, on impose pour tout symbole de prédicat R et tout état s, que $A_R \cap D_R = \emptyset$. Pour les actions abstraites, on impose en plus que $A_R \cap I^3_R = \emptyset$ et que $D_R \cap I^3_R = \emptyset$, afin de pouvoir écrire indifféremment :

$$s'(R_{ext}) = (s(R_{ext}) \setminus (D_R \cup I^3_R \cup A^3_R \cup D^3_R)) \cup A_R$$
$$s'(R_{marge}) = (s(R_{marge}) \setminus (D_R \cup A_R)) \cup I^3_R \cup I^3_R \cup D^3_R \cup A^3_R$$

ou bien :

$$s'(R_{ext}) = (s(R_{ext}) \cup A_R) \setminus (D_R \cup I^3_R \cup A^3_R \cup D^3_R)$$
$$s'(R_{marge}) = (s(R_{marge}) \cup I^3_R \cup D^3_R \cup A^3_R) \setminus (D_R \cup A_R))$$

Une conséquence directe de cette contrainte et que la contre-extension d'un symbole de relation s'exprime alors par une formule analogue à celle de l'extension :

Proposition 16. *Pour tout symbole de prédicat R, tout état $s \in S_3$, et toute action abstraite a, on a :*

$$\mathcal{T}_3(s,a)(R_{cext}) = (s(R_{cext}) \setminus (A_R \cup I^3_R \cup A^3_R \cup D^3_R)) \cup D_R$$

(voir définition 44 pour les définitions de A_R, D_R, I_R, A^3_R, D^3_R et I^3_R)

Démonstration. Soient un symbole de prédicat R d'arité n, une action $a \in A_3$ et un état $s \in S_3$.
On pose $s' = \mathcal{T}_3(s, a)$.
Comme $s'(R_{cext})$, $s'(R_{ext}$ et $s'(R_{marge})$ sont complémentaires dans $dom(s)^n$, on a :

$$s'(R_{cext}) = dom(s)^n \setminus (s'(R_{ext} \cup s'(R_{marge})))$$

D'après la définition 44, on en déduit alors :

$$\begin{aligned}
s'(R_{cext}) &= dom(s)^n \setminus (s'(R_{ext} \cup s'(R_{marge}))) \\
&= dom(s)^n \setminus (((s(R_{ext}) \setminus (D_R \cup I_R^3 \cup A_R^3 \cup D_R^3)) \cup A_R) \\
&\quad \cup ((s(R_{marge}) \setminus (D_R \cup A_R)) \cup I_R^3 \cup D_R^3 \cup A_R^3)) \\
&= dom(s)^n \setminus (((s(R_{ext}) \setminus D_R) \cup A_R) \cup ((s(R_{marge}) \setminus D_R) \cup I_R^3 \cup D_R^3 \cup A_R^3)) \\
&= dom(s)^n \setminus (((s(R_{ext}) \cup s(R_{marge})) \setminus D_R) \cup A_R \cup I_R^3 \cup D_R^3 \cup A_R^3) \\
&= dom(s)^n \setminus ((s(R_{cext}) \setminus D_R) \cup A_R \cup I_R^3 \cup D_R^3 \cup A_R^3) \\
&= (dom(s)^n \setminus ((s(R_{cext}) \setminus D_R))) \setminus (A_R \cup I_R^3 \cup D_R^3 \cup A_R^3)
\end{aligned}$$

Or, comme $D_R \subseteq dom(s)^n$, on peut en déduire que :

$$(dom(s)^n \setminus ((s(R_{cext}) \setminus D_R)) = (dom(s)^n \setminus s(R_{cext})) \cup D_R)$$

De plus, on a aussi $D_R \cap (A_R \cup I_R^3 \cup D_R^3 \cup A_R^3) = \emptyset$, ce qui nous permet d'en déduire que :

$$\begin{aligned}
s'(R_{cext}) &= (dom(s)^n \setminus ((s(R_{cext}) \setminus D_R))) \setminus (A_R \cup I_R^3 \cup D_R^3 \cup A_R^3) \\
&= (s'(R_{cext}) \cup D_R) \setminus (A_R \cup I_R^3 \cup D_R^3 \cup A_R^3) \\
&= (s'(R_{cext}) \setminus (A_R \cup I_R^3 \cup D_R^3 \cup A_R^3)) \setminus D_R
\end{aligned}$$

□

Afin d'illustrer la progression des états par les actions abstraites, un exemple est proposé par la figure 4.18. Cet exemple est associé à l'action abstraite $VehicleAreaTravel$ déjà décrite à la figure 4.17. L'application des effets abstraits y est illustrée à travers une condition d'abstraction et une condition d'affectation. Un effet abstrait « indirect » est aussi exposé : en effet la position initiale du véhicule est indéterminée (soit $position1$, soit $position2$). En conséquence, ces deux positions ne sont pas supprimées de la marge par l'application de la condition de suppression qui est associée au prédicat $VehicleAt$.

Avant d'achever cette sous-partie, intéressons nous aux propriétés des actions abstraites. En premier lieu, considérons l'exécutabilité d'un plan composé d'actions abstraites. La définition de l'exécutabilité pour les actions concrètes n'est pas modifiée et correspond à celle introduite dans le cadre de la planification d'actions classique. Mais par analogie avec cette définition, on peut en plus définir une « exécutabilité potentielle », déterminée par l'application des actions abstraites dans le système de transition abstrait :

Définition 45 (Séquence d'actions abstraites potentiellement exécutable)**.** *Soient $\sigma \in A_3^*$ une séquence d'actions abstraites et un état abstrait $s \in S_3$. σ est potentiellement exécutable en s si et seulement si l'une des deux propositions suivantes est satisfaite :*

1. *$\sigma = \epsilon$.*
2. *Il existe $a \in A_3$ et $\sigma' \in A^*$ telles que $\sigma = a \circ \sigma'$, $s \models_3 \pi^a$ et σ' est potentiellement exécutable en $\mathcal{T}(s, a)$.*

Cette définition s'applique aux séquences d'actions abstraites, mais peut aussi être étendue aux réseaux de tâches dont les éléments sont des actions abstraites :

$s(InArea) \ =$
$\quad \{(position1,\ area1),\ (position2,\ area1),\ (position3,\ area1),$
$\quad\ (position4,\ area2),\ (position5,\ area2)\}$
$s(VehicleAtArea) \ =$
$\quad (\{(vhl,\ area1)\},\ \{\})$
$s(VehicleAt) \ =$
$\quad (\{\},\ \{(vhl,\ position1),\ (vhl,\ position2)\})$
$s(FuelLevel) \ =$
$\quad \{(vhl,\ 50)\}$

$\mathcal{T}_3(s,\ VehicleAreaTravel(vhl,\ area1,\ area2))(InArea) \ =$
$\quad \{(position1,\ area1),\ (position2,\ area1),\ (position3,\ area1),$
$\quad\ (position4,\ area2),\ (position5,\ area2)\}$
$\mathcal{T}_3(s,\ VehicleAreaTravel(vhl,\ area1,\ area2))(VehicleAtArea) \ =$
$\quad (\{(vhl,\ area2)\},\ \{\})$
$\mathcal{T}_3(s,\ VehicleAreaTravel(vhl,\ area1,\ area2))(VehicleAt) \ =$
$\quad (\{\},\ \{(vhl,\ position3),\ (vhl,\ position4),\ (vhl,\ position5)\})$
$\mathcal{T}_3(s,\ VehicleAreaTravel(vhl,\ area1,\ area2))(FuelLevel) \ =$
$\quad (\{(vhl,\ \mathrm{I}_{obj})\}$

FIGURE 4.18 – Progression d'état par une action abstraite
Progression d'un état s par l'action abstraite $VehicleAreaTravel(vhl, area1, area2)$.
L'état obtenu est noté $\mathcal{T}_3(s, VehicleAreaTravel(vhl, area1, area2))$.

Définition 46 (Réseau d'actions abstraites potentiellement exécutable)**.** *Soient $\rho \in R_{A_3}$ un réseau d'actions abstraites et un état abstrait $s \in S_3$. ρ est potentiellement exécutable en s si et seulement si il existe une linéarisation de ρ potentiellement exécutable en s.*

La propriété suivante est une propriété essentielle des actions abstraites. En effet, elle est une base qui nous permettra plus tard de prouver la complétude de l'algorithme que nous présenterons. Cette propriété est la conservation de l'ordre d'abstraction entre deux états par l'application à chacun d'eux d'une même action abstraite :

Proposition 17 (Monotonicité des actions abstraites)**.** *Soient deux états $s_1, s_2 \in S_3$ est une action abstraite $a \in A_3$ applicable en s_1 et en s_2. On a alors :*

$$s_1 \sqsubseteq s_2 \implies \mathcal{T}_3(s_1, a) \sqsubseteq \mathcal{T}_3(s_2, a)$$

Démonstration. On note $s'_1 = \mathcal{T}_3(s_1, a)$ et $s'_2 = \mathcal{T}_3(s_2, a)$.
On suppose que $s_1 \sqsubseteq s_2$, et on considère que $dom(s_1) = dom(s_2)$, ce qui est toujours le cas en planification, puisque les actions ne créent pas d'objet. Les ensembles $dom(s_1)$, $dom(s_2)$, $dom(s'_1)$ et $dom(s'_2)$ sont donc identiques.
Soit un symbole R d'arité n.
On note :

$$A_{R,1} = \{(o_1, ..., o_n) \in dom(s_1)^n \mid s_1 \models \alpha^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]\}$$
$$D_{R,1} = \{(o_1, ..., o_n) \in dom(s_1)^n \mid s_1 \models \delta^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]\}$$
$$I^3_{R,1} = \{(o_1, ..., o_n) \in dom(s_1)^n \mid s_1 \models_3 \iota^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]\}$$
$$A^3_{R,1} = \{(o_1, ..., o_n) \in dom(s_1)^n \mid [\![\alpha^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]]\!]^3_{s_1} = \frac{1}{2}\}$$
$$D^3_{R,1} = \{(o_1, ..., o_n) \in dom(s_1)^n \mid [\![\delta^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]]\!]^3_{s_1} = \frac{1}{2}\}$$

et :

$$A_{R,2} = \{(o_1, ..., o_n) \in dom(s_2)^n \mid s_2 \models \alpha^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]\}$$
$$D_{R,2} = \{(o_1, ..., o_n) \in dom(s_2)^n \mid s_2 \models \delta^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]\}$$
$$I^3_{R,2} = \{(o_1, ..., o_n) \in dom(s_2)^n \mid s_2 \models_3 \iota^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]\}$$
$$A^3_{R,2} = \{(o_1, ..., o_n) \in dom(s_2)^n \mid [\![\alpha^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]]\!]^3_{s_2} = \frac{1}{2}\}$$
$$D^3_{R,2} = \{(o_1, ..., o_n) \in dom(s_2)^n \mid [\![\delta^a_R(x_1, ..., x_n)[o_1/x_1, ..., o_n/x_n]]\!]^3_{s_2} = \frac{1}{2}\}$$

Comme $s_1 \sqsubseteq s_2$, on peut en déduire d'après le corollaire 1, que :

$$A_{R,2} \subseteq A_{R,1} \tag{4.5}$$
$$D_{R,2} \subseteq D_{R,1} \tag{4.6}$$
$$I^3_{R,1} \subseteq I^3_{R,2} \tag{4.7}$$
$$A^3_{R,1} \subseteq A^3_{R,2} \tag{4.8}$$
$$D^3_{R,1} \subseteq D^3_{R,2} \tag{4.9}$$

1. Montrons que $s'_1(R_{marge}) \subseteq s'_2(R_{marge})$.
   Soit $(o_1, ..., o_n) \in s'_1(R_{marge})$. D'après la définition 44, on a :

   $$s'_1(R_{marge}) = (s_1(R_{marge}) \setminus (D_{R,1} \cup A_{R,1})) \cup I^3_{R,1} \cup D^3_{R,1} \cup A^3_{R,1}$$

   On en déduit donc que $(o_1, ..., o_n) \in (s_1(R_{marge}) \setminus (D_{R,1} \cup A_{R,1}))$ ou que $(o_1, ..., o_n) \in I^3_{R,1} \cup D^3_{R,1} \cup A^3_{R,1}$.

   (a) Supposons que $(o_1, ..., o_n) \in s_1(R_{marge}) \setminus (D_{R,1} \cup A_{R,1})$.
   On en déduit donc que :

   $$(o_1, ..., o_n) \in s_1(R_{marge})$$
   $$(o_1, ..., o_n) \notin D_{R,1} \cup A_{R,1}$$

   Comme $s_1 \sqsubseteq s_2$, on en déduit d'après la définition 42 que $(o_1, ..., o_n) \in s_2(R_{marge})$. De plus, on a $D_{R,2} \cup A_{R,2} \subseteq D_{R,1} \cup A_{R,1}$ d'après 4.5 et 4.6, on en déduit donc que $(o_1, ..., o_n) \notin D_{R,2} \cup A_{R,2}$.
   On en déduit ainsi que $(o_1, ..., o_n) \in (s_2(R_{marge}) \setminus (D_{R,2} \cup A_{R,2}))$ et donc, d'après la définition 44, on en déduit que $(o_1, ..., o_n) \in s'_2(R_{marge})$.

   (b) Supposons que $(o_1, ..., o_n) \in I^3_{R,1} \cup D^3_{R,1} \cup A^3_{R,1}$.
   D'après 4.5, 4.6 et 4.7, on en déduit alors que $(o_1, ..., o_n) \in I^3_{R,2} \cup D^3_{R,2} \cup A^3_{R,2}$.
   D'après la définition 44, on en déduit donc que $(o_1, ..., o_n) \in s'_2(R_{marge})$.

   On a donc bien $s'_1(R_{marge}) \subseteq s'_2(R_{marge})$.

2. Montrons que $s'_2(R_{ext}) \subseteq s'_1(R_{ext})$.
   Soit $(o_1, ..., o_n) \in s'_2(R_{ext})$. D'après la définition 44, on a :

   $$s'_2(R_{ext}) = (s_2(R_{ext}) \setminus (D_{R,2} \cup I^3_{R,2} \cup A^3_{R,2} \cup D^3_{R,2})) \cup A_{R,2}$$

   On en déduit donc que $(o_1, ..., o_n) \in (s_2(R_{ext}) \setminus (D_{R,2} \cup I^3_{R,2} \cup A^3_{R,2} \cup D^3_{R,2}))$ ou que $(o_1, ..., o_n) \in A_{R,2}$.

(a) Supposons que $(o_1, ..., o_n) \in (s_2(R_{ext}) \setminus (D_{R,2} \cup I^3_{R,2} \cup A^3_{R,2} \cup D^3_{R,2}))$.
On en déduit que :

$$(o_1, ..., o_n) \in s_2(R_{ext})$$
$$(o_1, ..., o_n) \notin D_{R,2} \cup I^3_{R,2} \cup A^3_{R,2} \cup D^3_{R,2}$$

Comme $s_1 \sqsubseteq s_2$, on en déduit d'après la définition 42 que $(o_1, ..., o_n) \in s_1(R_{ext})$.
De plus comme $s_1 \sqsubseteq s_2$, on peut en déduire d'après le corollaire 1 que $D_{R,1} \subseteq D_{R,2} \cup D^3_{R,2}$. D'après les équations 4.7, 4.8 et 4.9, on peut donc en déduire que $D_{R,1} \cup I^3_{R,1} \cup A^3_{R,1} \cup D^3_{R,1} \subseteq D_{R,2} \cup I^3_{R,2} \cup A^3_{R,2} \cup D^3_{R,2}$.
On en déduit donc que $(o_1, ..., o_n) \notin D_{R,1} \cup I^3_{R,1} \cup A^3_{R,1} \cup D^3_{R,1}$.
On en déduit alors que $(o_1, ..., o_n) \in (s_1(R_{ext}) \setminus (D_{R,1} \cup I^3_{R,1} \cup A^3_{R,1} \cup D^3_{R,1}))$ et donc, d'après la définition 44, on en déduit que $(o_1, ..., o_n) \in s'_1(R_{marge})$.

(b) Supposons que $(o_1, ..., o_n) \in A_{R,1}$.
D'après l'équation 4.5, on a $A_{R,1} \subseteq A_{R,2}$, on en déduit donc que $(o_1, ..., o_n) \in A_{R,2}$.
D'après la définition 44, on en déduit alors que $(o_1, ..., o_n) \in s'_1(R_{ext})$.

On a donc bien $s'_2(R_{ext}) \subseteq s'_1(R_{ext})$.

3. Montrons que $s'_2(R_{cext}) \subseteq s'_1(R_{cext})$.
D'après la proposition 16, on a :

$$s'_2(R_{cext}) = (s_2(R_{cext}) \setminus (A_R \cup I^3_R \cup A^3_R \cup D^3_R)) \cup D_R$$

À partir de ce point, la démonstration devient analogue au cas précédent, et est donc ignorée.

Soit un symbole de fonction F d'arité n.
On note alors :

$$A_{F,1} = \{(o_1, ..., o_n, v) \in dom(s_1)^{n+1} \mid s \models \mu^a_F(x_1, ..., x_n, y)[o_1/x_1, ..., o_n/x_n, v/y]\}$$
$$D_{F,1} = \{(o_1, ..., o_n, v) \in dom(s_1)^n \times (dom(s_1) \cup \{\mathrm{I}_{obj}\}) \mid s_1 \models_3 (\exists y : \mu^a_F(x_1, ..., x_n, y))[o_1/x_1, ..., o_n/x_n]\}$$
$$I^3_{F,1} = \{(o_1, ..., o_n, v) \in dom(s_1)^n \times \{\mathrm{I}_{obj}\} \mid [\![(\exists y : \mu^a_F(x_1, ..., x_n, y))[o_1/x_1, ..., o_n/x_n]]\!]^3_{s_1}\}$$

et :

$$A_{F,2} = \{(o_1, ..., o_n, v) \in dom(s_2)^{n+1} \mid s_2 \models \mu^a_F(x_1, ..., x_n, y)[o_1/x_1, ..., o_n/x_n, v/y]\}$$
$$D_{F,2} = \{(o_1, ..., o_n, v) \in dom(s_2)^n \times (dom(s_2) \cup \{\mathrm{I}_{obj}\}) \mid s_2 \models_3 (\exists y : \mu^a_F(x_1, ..., x_n, y))[o_1/x_1, ..., o_n/x_n]\}$$
$$I^3_{F,2} = \{(o_1, ..., o_n, v) \in dom(s_2)^n \times \{\mathrm{I}_{obj}\} \mid [\![(\exists y : \mu^a_F(x_1, ..., x_n, y))[o_1/x_1, ..., o_n/x_n]]\!]^3_{s_2}\}$$

D'après le corollaire 1, on peut alors en déduire que :

$$A_{F,2} \subseteq A_{F,1} \tag{4.10}$$
$$D_{F,1} \subseteq D_{F,2} \tag{4.11}$$
$$A_{F,1} \cup I^3_{F,1} \subseteq A_{F,2} \cup I^3_{F,2} \tag{4.12}$$

1. Montrons que $s'_2(F) \cap dom(s'_2)^{n+1} \subseteq s'_1(F) \cap dom(s'_1)^{n+1}$:
Soit $(o_1, ..., o_{n+1}) \in dom(s'_2)^{n+1}$ tel que $(o_1, ..., o_{n+1}) \in s'_2(F)$.
D'après la définition 44, on en déduit que $(o_1, ..., o_{n+1}) \in (s_2(F) \setminus D_{F,2})$ ou que $(o_1, ..., o_{n+1}) \in A_{F,2} \cup I^3_{F,2}$.

(a) Supposons que $(o_1, ..., o_{n+1}) \in s_2(F) \setminus D_{F,2}$.
On en déduit donc que :

$$(o_1, ..., o_{n+1}) \in s_2(F)$$
$$(o_1, ..., o_{n+1}) \notin D_{F,2}$$

Comme $s_1 \sqsubseteq s_2$, on en déduit d'après la définition 42, que $(o_1, ..., o_{n+1}) \in s_1(F)$.
On déduit aussi de l'équation 4.11 que $(o_1, ..., o_{n+1}) \notin D_{F,1}$.
On en déduit donc que $(o_1, ..., o_{n+1}) \in s_1(F) \setminus D_{F,1}$, et donc, d'après la définition 44, que $(o_1, ..., o_{n+1}) \in s'_1(F)$.

(b) Sinon, on a $(o_1, ..., o_{n+1}) \in A_{F,2} \cup I^3_{F,2}$.
Comme $o_{n+1} \in dom(s'_2)$, on en déduit que $(o_1, ..., o_{n+1}) \in A_{F,2}$.
D'après l'équation 4.10, on en déduit donc que $(o_1, ..., o_{n+1}) \in A_{F,1}$, et donc que $(o_1, ..., o_{n+1}) \in A_{F,1} \cup I^3_{F,1}$.
D'après la définition 44, on en déduit alors que $(o_1, ..., o_{n+1}) \in s'_1(F)$.

On a donc bien $s'_2(F) \cap dom(s'_2)^{n+1} \subseteq s'_1(F) \cap dom(s'_1)^{n+1}$.

2. Montrons que $s'_1(F) \cap (dom(s'_1)^n \times \{\mathrm{I}_{obj}\}) \subseteq s'_2(F) \cap (dom(s'_1)^n \times \{\mathrm{I}_{obj}\})$.
Soit $(o_1, ..., o_n) \in dom(s'_1)^n$ tel que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in s'_1(F)$.
D'après la définition 44, on en déduit que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in (s_1(F) \setminus D_{F,1})$ ou que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in A_{F,1} \cup I^3_{F,1}$.

(a) Supposons que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in s_1(F) \setminus D_{F,1}$.
On a alors :

$$(o_1, ..., o_n, \mathrm{I}_{obj}) \in s_1(F)$$
$$(o_1, ..., o_n, \mathrm{I}_{obj}) \notin D_{F,1}$$

Comme $s_1 \sqsubseteq s_2$, on en déduit d'après la définition 42, que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in s_2(F)$.

i. Supposons que $(o_1, ..., o_n, \mathrm{I}_{obj}) \notin D_{F,2}$:
Dans ce cas, on a alors $(o_1, ..., o_n, \mathrm{I}_{obj}) \in s_2(F) \setminus D_{F,2}$ et d'après la définition 44, on en déduit donc que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in s'_2(F)$.

ii. Supposons que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in D_{F,2}$:
Dans ce cas, on a alors :

$$s_2 \models_3 (\exists y : \mu^a_F(x_1, ..., x_n, y))[o_1/x_1, ..., o_n/x_n])$$

Ce qui implique que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in I^3_{F,2}$ ou qu'il existe $o \in dom(s_2)$ tel que $(o_1, ..., o_n, o) \in A_{F,2}$.
Toutefois, s'il existe $o \in dom(s_2)$ tel que $(o_1, ..., o_n, o) \in A_{F,2}$, alors on a aussi $(o_1, ..., o_n, o) \in A_{F,1}$ d'après l'équation 4.10. On obtient alors $(o_1, ..., o_n, o) \in s'_1(F)$, ce qui est incompatible avec l'hypothèse de départ ($(o_1, ..., o_n, \mathrm{I}_{obj}) \in s'_1(F)$).
On en déduit donc que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in I^3_{F,2}$, ce qui nous permet de déduire, d'après la définition 44, que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in s'_2(F)$.

(b) Sinon, on a $(o_1, ..., o_n, \mathrm{I}_{obj}) \in A_{F,1} \cup I^3_{F,1}$.
D'après l'équation 4.12, on a alors $(o_1, ..., o_n, \mathrm{I}_{obj}) \in A_{F,2} \cup I^3_{F,2}$.
D'après la définition 44, on en déduit donc que $(o_1, ..., o_n, \mathrm{I}_{obj}) \in s'_2(F)$.

On a donc bien $s'_1(F) \cap (dom(s'_1)^n \times \{\mathrm{I}_{obj}\}) \subseteq s'_2(F) \cap (dom(s'_1)^n \times \{\mathrm{I}_{obj}\})$.

D'après la définition 42, on en déduit donc que $s'_1 \sqsubseteq s'_2$. □

Une conséquence de cette propriété est la suivante : si une action abstraite est potentiellement exécutable sur un état, alors on peut en déduire qu'elle est exécutable dans toute abstraction de cet état :

Proposition 18. *Soit une séquence d'actions abstraites $\sigma \in A_3^*$ et deux états abstraits $s_1, s_2 \in S_3$ tels que $s_1 \sqsubseteq s_2$.*
Si σ est potentiellement exécutable en s_1, alors σ est potentiellement exécutable en s_2.

Démonstration. La démonstration se fait par récurrence structurelle sur σ.

1. Si $\sigma = \epsilon$.
   D'après la définition 45, ϵ est potentiellement exécutable en s_2.
2. Soit $\sigma \in A^*$ et $a \in A$ tels que $a \circ \sigma$ est potentiellement exécutable en s_1, et tel que pour toute paire d'états $(s'_1, s'_2) \in S_3^2$, si σ est potentiellement exécutable en s'_1, alors σ est potentiellement exécutable en s'_2.
   Supposons que $a \circ \sigma$ est potentiellement exécutable en s_1.
   D'après la définition 45 on a donc $s_1 \models_3 \pi^a$ et σ est potentiellement exécutable en $\mathcal{T}(s_1, a)$.
   Comme $s_1 \sqsubseteq s_2$, on en déduit d'après le corollaire 1 que $s_2 \models_3 \pi^a$ et d'après la proposition 17 que $\mathcal{T}(s_1, a) \sqsubseteq \mathcal{T}(s_2, a)$.
   Par hypothèse, on en déduit donc que σ est potentiellement exécutable en $\mathcal{T}(s_2, a)$.
   D'après la définition 45, on en déduit finalement que $a \circ \sigma$ est potentiellement exécutable en s_2.

□

Enfin, on définit à nouveau une propriété d'action abstraite « cohérente ». Cette propriété est similaire à celle définie pour les hiérarchies d'abstractions, à la différence qu'elle s'établit désormais entre une action abstraite et les sous-tâches de ses méthodes. Pour rappel, dans le cas des hiérarchies d'abstractions, cette propriété était établie entre une action abstraite et ces décompositions au niveau d'abstraction inférieur. On dit qu'une action abstraite est cohérente si pour un état initial donné, l'état obtenu en l'exécutant est une abstraction de tout état obtenu en exécutant ses décompositions. Cette cohérence concerne aussi le niveau attribué à chaque action abstraite. En effet, le bon sens veut que le niveau d'une action abstraite soit supérieur à celui de ses sous-tâches, ou éventuellement qui leur soit égal en cas de décomposition cyclique.

Définition 47 (Action abstraite cohérente)**.** *Soit une action abstraite $a \in A_3 \setminus A$. On dit que a est cohérente si et seulement si pour tout état $s \in S_3$ et pour toute méthode $m \in M$ telle que $c_m = a$:*

1. *Si ρ_m est potentiellement exécutable en s, alors a est potentiellement applicable en s, et pour toute linéarisation $\sigma \in A^*$ de ρ_m, on a :*
$$\mathcal{T}_3(s, \sigma) \sqsubseteq \mathcal{T}_3(s, a)$$
2. *Pour toute sous-tâche $t \in \{t_n \mid n \in N_{\rho_m}\}$, on a :*
$$\nu^a(s) \geq \nu^t(s)$$

4.3.3 Planification

Dans cette dernière sous-partie, nous introduisons dans un premier temps la définition d'un domaine et d'un problème de planification HTN avec abstraction. Nous décrivons ensuite le principe sur lequel reposent les stratégies de décompositions des tâches employés par l'algorithme du planificateur. Enfin, nous présentons cet algorithme et discutons de ces propriétés.

Domaine et problème de planification HTN avec abstraction

Un domaine de planification HTN avec abstraction n'est ni plus ni moins qu'un domaine de planification HTN étendu pour prendre en compte le système de transition d'états abstraits. On considère aussi que toutes les actions abstraites modélisées dans un domaine de planification HTN avec abstraction sont cohérentes (voire définition 47). Il appartient alors au modélisateur de s'assurer que les actions abstraites du domaine satisfont cette propriété.

Définition 48 (Domaine de planification HTN avec abstraction)**.** *Un domaine de planification HTN avec abstraction est un quadruplet $D_3 = (S_3, A_3, M, \mathcal{T}_3, \mathcal{C})$ où :*

1. *S_3 est un ensemble fini d'états abstraits.*
2. *A_3 est un ensemble fini d'actions abstraites cohérentes.*
3. *$M \subseteq (A_3 \setminus A) \times R_{A_3}$ est un ensemble fini de méthodes de décomposition associées aux éléments de $A_3 \setminus A$.*
4. *$\mathcal{T}_3 \in S_3 \times A_3 \to S_3$ est une fonction de transition abstraite.*
5. *$\mathcal{C} \in S \times A \to \mathbb{R}^+$ est une fonction de coût.*

Un problème de planification HTN avec abstraction est identique à un problème de planification HTN classique, si ce n'est qu'il repose sur la modélisation d'un domaine de planification HTN avec abstraction.

Définition 49 (Problème de planification HTN avec abstraction)**.** *Un problème de planification HTN avec abstraction est un triplet $D_3 = (D_3, s_0, \rho_0)$ où :*

1. *D_3 est domaine de planification HTN avec abstraction.*
2. *$s_0 \in S$ est un état initial.*
3. *$\rho_0 \in R_{A_3}$ est un réseau de tâches initial.*

L'ensemble des solutions d'un problème de planification HTN avec abstraction est donc identique à l'ensemble des solutions d'un problème de planification HTN (voir la définition 21, ainsi que la définition 22 pour la définition des solutions optimales).

Stratégies de décomposition des tâches

Le système de transition abstrait permet d'évaluer l'exécutabilité d'un réseau de tâches sans avoir à se soucier de l'ordre des tâches par rapport à leur niveau d'abstraction. En conséquence, n'importe quelle stratégie de décomposition des tâches peut être implémentée et exploitée dans un algorithme de planification HTN avec abstraction. Cependant, nous souhaitons conserver un mécanisme similaire à celui que nous avons employé pour les hiérarchies d'abstractions, et qui consiste à choisir la tâche associée au plus haut niveau d'abstraction dans les limites d'un horizon donnée.

Pour cela, on définit pour chaque réseau de tâches un sous-ensemble de nœuds « accessibles » par rapport à un horizon. Un nœud est considéré comme étant accessible s'il existe une chaine dans l'ordre partiel du réseau de tâches (en ignorant les nœuds qui correspondent à des tâches primitives) pour lequel ce nœud occupe un rang (une position) inférieur à la valeur de l'horizon. Pour chaque nœud associé à une tâche composée, on définit donc un rang dans le réseau de tâches :

Définition 50 (Rang d'un nœud décomposable dans un réseau de tâches)**.** *Soit $\rho \in R_{A_3}$ un réseau d'actions abstraites. Pour tout nœud $n \in N_\rho$ tel que $t_n \notin A$, on définit le rang de n, noté r_n, par :*

$$r_n = 1 + \max(\{0\} \cup \{r(n') \mid n' \in N_\rho \wedge t_{n'} \notin A \wedge n' \prec_\rho n\})$$

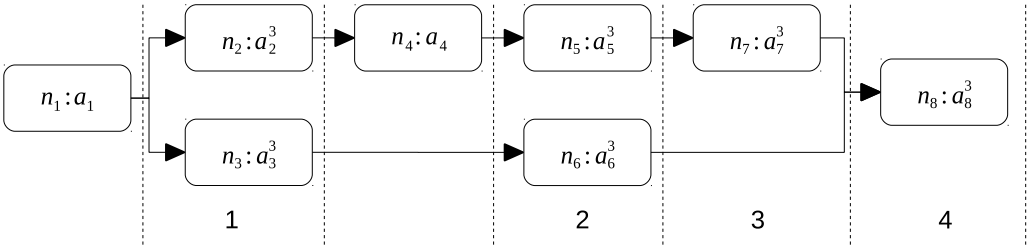


FIGURE 4.19 – Rang d'un nœud composé dans un réseau de tâches
Un réseau d'actions abstraites où les nœuds sont regroupés en fonction de la valeur de leur rang (indiqué en dessous des nœuds). Les tâches a_1 et a_4 sont des actions concrètes, tandis que les tâches a_2^3, a_3^3, a_5^3, a_6^3, a_7^3 et a_8^3 sont des actions abstraites décomposables.

Afin de visualiser l'attribution des rangs aux nœuds d'un réseaux d'actions abstraites, un exemple est illustré par la figure 4.19.

L'ensemble des nœuds accessibles d'un réseau de tâches est donc constitué de l'ensemble des nœuds décomposables dont le rang est inférieur ou égal à la valeur de l'horizon :

Définition 51 (Nœuds accessibles d'un réseau de tâches)**.** *Soient ρ un réseau de tâches totalement ordonné et non primitif, ainsi qu'un entier $h \in \mathbb{N}^*$. On appelle ensemble des nœuds accessibles de ρ pour un horizon h, noté N_ρ^h l'ensemble défini par :*

$$N_\rho^h = \{n \in N_\rho \mid t_n \notin A \wedge r_n \leq h\}$$

La sélection d'une tâche à décomposer est ensuite effectuée en choisissant un nœud associé à une tâche avec le niveau le plus élevé parmi tous les nœuds accessibles. Ainsi, si la valeur de l'horizon est nulle, on retrouve à nouveau la stratégie de décomposition utilisée par SHOP, qui consiste à choisir les premières tâches non primitives dans un réseau de tâches. Si au contraire la valeur de l'horizon est infinie, on obtient la stratégie de décomposition qui sélectionne toujours en priorité les tâches les plus abstraites.

ABDHTN

Comme mentionné plus haut, le principe des effets abstraits permet de s'affranchir de toute contrainte sur l'ordre de décomposition des tâches. En conséquence, l'algorithme de résolution peut être fondé sur un cadre général, et parcourir l'espace des décompositions plutôt que d'être restreint à l'espace des progressions. L'algorithme que nous présentons ici est donc fondé sur l'algorithme DHTN, et est en conséquence nommé ABDHTN. Par rapport à DHTN, ABDHTN ajoute l'évaluation de l'exécutabilité potentielle de chaque réseau de tâches parcouru, ce qui lui permet de supprimer les réseaux non exécutables de son espace de recherche. De plus, cet algorithme implémente l'utilisation d'un horizon pour déterminer la stratégie de décomposition des tâches, selon le principe décrit plus haut. L'algorithme 12 fournit une description d'ABDHTN en pseudo code.

Dans la suite de cette partie, on considère un domaine de planification HTN avec abstraction D_3 ainsi qu'un entier strictement positif h pour la valeur de l'horizon.

ABDHTN étant une extension de DHTN, il termine dans les mêmes conditions que celui-ci, c'est à dire quand l'espace de décomposition du réseau de tâches initial est fini. Pour rappel, il a été démontré que cet espace est fini si et seulement si le réseau de tâches initial, ainsi que les réseaux de tâches des méthodes de décomposition, vérifie une propriété appelée « $\leq_1$-stratification » [3].

Théorème 6 (Terminaison de ABDHTN)**.** *Soit un problème de planification HTN avec abstraction $P_3 = (D_3, s_0, \rho_0)$.*
Si $\{\rho \in R_{A_3} \mid \mathcal{D}^*(\rho_0, \rho)\}$ est fini, alors la résolution de P_3 par ABDHTN termine.

Algorithm 12 Algorithme ABDHTN

entrées : un problème de planification HTN avec abstraction $P_3 = (D_3, s_0, \rho_0)$ et un horizon $h \in \mathbb{N}^*$.

sorties : un plan solution de P_3 ou é*chec*

$Ouvert \leftarrow \{\rho_0\}$ **si** ρ_0 est potentiellement exécutable en s **sinon** $\emptyset$
$Fermé \leftarrow \emptyset$
tant que $Ouvert \neq \emptyset$ **faire**
  $\rho \leftarrow$ extraire un élément de $Ouvert$
  $Fermé \leftarrow Fermé \cup \{\rho\}$
  si ρ est primitif **alors**
    retourner une linéarisation de ρ potentiellement exécutable en s_0
  sinon
    $n \leftarrow$ choisir un élément de $\{n \in N_\rho^h \mid \forall n' \in N_\rho^h : \nu^{t_{n'}} \leq \nu^{t_n}\}$
    pour toute méthode $m \in M$ telle que $c_m = t_n$ **faire**
      $\rho' \leftarrow \mathcal{R}(\rho, n, m)$
      si $\rho' \notin Fermé$ **et** ρ' est potentiellement exécutable en s_0 **alors**
        $Ouvert \leftarrow Ouvert \cup \{\rho'\}$
      fin si
    fin pour
  fin si
fin tant que
retourner é*chec*

Démonstration. Si $\{\rho \in R_{A_3} \mid \mathcal{D}^*(\rho_0, \rho)\}$ est fini, alors ABDHTN ne peut visiter qu'un nombre fini de réseaux de tâches.
Comme l'ensemble $Fermé$ permet de s'assurer que chaque réseau de tâches n'est visité qu'une seule fois, on en déduit qu'ABDHTN visite un nombre fini d'éléments, et donc qu'il termine. □

Intéressons nous maintenant à la correction d'ABDHTN. Pour rappel, ABDHTN est correct si et seulement si chaque séquence de tâches primitives retournée appartient à l'espace de décomposition du réseau de tâches initial et est exécutable dans l'état initial. D'après l'algorithme 12, la première condition est bien vérifiée, mais la seconde n'est pas évidente. En effet l'algorithme garantit seulement que le plan retourné est potentiellement exécutable dans l'état initial. Il nous faut donc nous assurer que toute séquence d'actions concrètes potentiellement exécutable dans un état concret y est en fait exécutable. Cette propriété est démontrée dans le lemme suivant :

Lemme 5. *Soit une séquence d'actions $\sigma \in A^*$.*
Pour tout état concret $s \in S$, si σ est potentiellement exécutable en s, alors σ est exécutable en s.

Démonstration. On effectue une démonstration par récurrence structurelle sur σ.

1. Si $\sigma = \epsilon$.
   ϵ est exécutable en tout état $s \in S$ d'après la définition 3.
2. Si $\sigma = a \circ \sigma'$, où $a \in A$ et $\sigma \in A^*$ est une séquence d'actions telle que pour tout $s \in S$, si σ' est potentiellement exécutable en s, alors σ' est exécutable en s.
   Soit un état concret $s \in S$.
   On suppose que σ est potentiellement exécutable en s.
   D'après le définition 45, on en déduit donc que $s \models_3 \pi^a$ et que σ' est potentiellement exécutable en $\mathcal{T}_3(s, a)$.

On en déduit donc que σ' est exécutable en $\mathcal{T}_3(s, a)$.
s étant un état concret, on en déduit aussi d'après la proposition 14 que $s \models \pi^a$, et donc que a est applicable en s.
D'après la définition 21, on en déduit donc que σ est exécutable en s.

□

On peut alors en déduire que ABDHTN est un algorithme correct :

Théorème 7 (Correction d'ABDHTN)**.** *Soit un problème de planification HTN avec abstraction $P_3 = (D_3, s_0, \rho_0)$. Toute séquence de tâches retournée par ABDHTN lors de la résolution de P_3 est une solution de P_3.*

Démonstration. Soit σ une séquence de tâches retournée par ABDHTN.
D'après l'algorithme 12, σ est une séquence de tâches primitive, $\mathcal{D}^*(\rho_0, \varrho(\sigma))$ et σ est potentiellement exécutable en s_0).
s_0 étant un état concret, on en déduit d'après le lemme 5 que σ est exécutable en s_0.
D'après la définition 21, on en déduit donc que σ est une solution de P_3.
ABDHTN est donc bien un algorithme correct. □

On s'intéresse maintenant à la complétude d'ABDHTN. À cause des effets abstraits, ABDHTN est confronté au même problème qu'ABPHTN. Il est en effet possible qu'un plan partiellement ordonné ne soit pas potentiellement exécutable, alors que certaines de ses décompositions le sont (pour rappel, un exemple de situation où ce problème apparait est illustré en figure 4.5). La complétude d'un algorithme ne peut donc pas être garantie en général. Comme les réseaux de tâches totalement ordonnés ne sont pas affectés par ce problème, on se contente de démontrer la complétude d'ABDHTN pour des problèmes de planification HTN avec abstraction totalement ordonnés.

ABDHTN ne supprime un réseau de tâches de son espace de recherche que si celui-ci a déjà été visité, ou s'il n'est pas potentiellement exécutable. En conséquence, pour montrer qu'ABDHTN est complet, il nous faut montrer qu'un réseau de tâches totalement ordonné qui permet d'atteindre une solution est nécessairement potentiellement exécutable. Pour cela, on montre que les effets abstraits garantissent une forme de « upward solution property » (par analogie avec ABSTRIPS), ce qui signifie que si une décomposition d'un réseau de tâches est potentiellement exécutable, alors ce réseau de tâches est aussi potentiellement exécutable.

Lemme 6. *Soient un état abstrait $s \in S_3$, et deux réseaux de tâches totalement ordonnés $\rho, \rho' \in \varrho(A_3^*)$ tels que $\mathcal{D}(\rho, \rho')$. On a alors :*

$$\rho' \text{ potentiellement exécutable en } s \implies \rho \text{ potentiellement exécutable en } s$$

Démonstration. Comme ρ et ρ' sont deux réseaux de tâches totalement ordonnés tels que $\mathcal{D}(\rho, \rho')$, on en déduit d'après la proposition 5 qu'il existe $n \in N_\rho$ et $m \in M$ tel que $c_m = t_n$ et $\rho' = \mathcal{R}(\rho, n, m)$.
Soient $(a_1, ..., a_{|\rho|}) \in {A_3}^{|\rho|}$ tel que $\rho = \varrho(a_1 \circ ... \circ a_{|\rho|})$ et $(a_{m,1}, ..., a_{m,|\rho_m|}) \in {A_3}^{|\rho_m|}$ tel que $\rho_m = \varrho(a_{m,1} \circ ... \circ a_{m,|\rho|})$ et soit Λ la fonction de linéarisation associée à $a_1 \circ ... \circ a_{|\rho|}$.
D'après la proposition 6, on peut donc en déduire que :

$$\begin{aligned}
\rho' &= \mathcal{R}(\varrho(a_1 \circ ... \circ a_{\Lambda(n)-1}) * \varrho(a_{\Lambda(n)}) * \varrho(a_{\Lambda(n)+1} \circ ... \circ a_{|\rho|})) \\
&= \varrho(a_1 \circ ... \circ a_{\Lambda(n)-1}) * \mathcal{R}(\varrho(a_{\Lambda(n)}), n, m) * \varrho(a_{\Lambda(n)+1} \circ ... \circ a_{|\rho|})) \\
&= \varrho(a_1 \circ ... \circ a_{\Lambda(n)-1} \circ a_{m,1} \circ ... \circ a_{m,|\rho_m|} \circ (a_{\Lambda(n)+1} \circ ... \circ a_{|\rho|})
\end{aligned}$$

Comme ρ' est potentiellement exécutable en s, on en déduit d'après la définition 45 que :

— $a_1 \circ ... \circ a_{\Lambda(n)-1}$ est potentiellement exécutable en s
— $a_{m,1} \circ ... \circ a_{m,|\rho_m|}$ est potentiellement exécutable en $\mathcal{T}_3(s, a_1 \circ ... \circ a_{\Lambda(n)-1})$
— $a_{\Lambda(n)+1} \circ ... \circ a_{|\rho|}$ est potentiellement exécutable en $\mathcal{T}_3(s, a_1 \circ ... \circ a_{\Lambda(n)-1} \circ a_{m,1} \circ ... \circ a_{m,|\rho_m|})$

Comme $a_{\Lambda(n)}$ est une action abstraite cohérente, on en déduit d'après la définition 47 que $a_{\Lambda(n)}$ est applicable en $\mathcal{T}_3(s, a_1 \circ ... \circ a_{\Lambda(n)-1})$ et que :

$$\mathcal{T}_3(s', a_{m,1} \circ ... \circ a_{m,|\rho_m|}) \sqsubseteq \mathcal{T}_3(s', a)$$

où $s' = \mathcal{T}_3(s, a_1 \circ ... \circ a_{\Lambda(n)-1})$.
De plus d'après le proposition 18, on en déduit que $a_{\Lambda(n)+1} \circ ... \circ a_{|\rho|}$ est potentiellement exécutable en $\mathcal{T}_3(s', a)$, et donc que $a_1 \circ ... \circ a_{|\rho|}$ est potentiellement exécutable en s. D'après la définition 45, ρ est donc potentiellement exécutable en s. □

À partir de ce résultat, on peut finalement en déduire la complétude d'ABDHTN vis-à-vis des problèmes de planification HTN avec abstraction totalement ordonnés :

Théorème 8 (Complétude d'ABDHTN)**.** *Soit un problème de planification HTN avec abstraction totalement ordonné $P_3 = (D_3, s_0, \rho_0)$. Si P_3 est résoluble et qu'ABDHTN termine, alors ABDHTN retourne une solution.*

Démonstration. Supposons qu'ABDHTN termine sans retourner de solution.
Soit $\sigma \in A^*$ une solution de P_3 (une telle solution existe puisque P_3 est résoluble par hypothèse).
Comme ABDHTN ne retourne pas σ, on en déduit qu'il existe $\rho \in \varrho(A_3^*)$ tel que $\mathcal{D}^*(\rho_0, \rho)$ et $\mathcal{D}^*(\rho, \varrho(\sigma))$, et qui n'est pas potentiellement exécutable en s_0.
Or, $\varrho(\sigma)$ est potentiellement exécutable en s, et d'après le lemme 6, pour tout $(\rho_1, \rho_2) \in \varrho(A_3^*)$ tel que $\mathcal{D}(\rho_1, \rho_2)$, si ρ_2 est potentiellement exécutable en s, alors ρ_1 est potentiellement exécutable en s.
Par récurrence structurelle sur $\mathcal{D}^*$, on en déduit donc que ρ est potentiellement exécutable en s, ce qui est une contradiction.
On en conclut donc qu'ABDHTN retourne une solution. □

4.4 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la résolution de problèmes de planification HTN dont certaines tâches composées peuvent être assimilées à des actions abstraites.

En première partie, nous avons proposé une sémantique pour des effets de haut niveau reposant sur le principe d'une hiérarchie d'abstractions, et nous avons tâché de répondre à la question suivante : quels sont les avantages de cette sémantique de transitions d'états abstraits pour la planification HTN ? Nous y avons répondu en proposant un algorithme de planification HTN nommé ABPHTN. L'algorithme proposé exploite les effets abstraits afin d'implémenter des stratégies de décomposition variée, ce qui permet de contrôler l'exécution de plans abstraits à plusieurs niveaux, et ainsi d'anticiper l'identification dans l'espace de recherche de certains plans non exécutables.

Dans la seconde partie, nous avons traité la question des performances que l'on pouvait obtenir avec des stratégies de décompositions différentes. Afin de répondre à cette question, l'algorithme ABPHTN a fait l'objet d'expériences visant à évaluer la différence des temps de calcul obtenus pour différentes stratégies de décompositions des tâches. Les résultats ont permis d'établir empiriquement que l'amélioration du temps d'exécution est subordonnée à la qualité d'un système d'abstraction, c'est à dire à la propension des actions abstraites exécutables à ce décomposer en plans exécutables par rapport à un

état donné. Il a aussi été constaté qu'une abstraction de « mauvaise qualité » peut être contre-productive en terme de temps de calcul.

Enfin en troisième partie, nous avons proposé une forme alternative d'effets de haut niveau qui reposent aussi sur le concept d'abstraction, mais fondent désormais leur sémantique sur un espace de transition d'états abstrait décrit par un langage logique du premier ordre à trois valeurs de vérité. Tout en présentant ce nouveau système, nous avons tâché de répondre à la question suivante : en quoi ces effets qui reposent sur une logique à trois valeurs de vérité permettent-ils de résoudre des problèmes introduits par les hiérarchies d'abstractions ? Ainsi, nous avons montré que ces effets abstraits permettent d'obtenir les mêmes avantages que les hiérarchies d'abstractions, mais qu'ils apportent plus de flexibilité au modélisateur. En effet, celui-ci peut désormais modéliser n'importe quelle tâche composée sous la forme d'une action abstraite, et n'est plus obligé d'imposer une unique séquence d'abstractions par domaine pour modéliser des effets de haut niveau.

En ce qui concerne l'amélioration des temps de calcul grâce à l'introduction des effets abstraits, celle-ci n'a ici été envisagée que par rapport à la possibilité d'anticiper les plans non exécutables, afin de réduire la taille de l'espace de recherche. Or, il est possible d'explorer une autre caractéristique de la recherche de plans solutions, en s'intéressant à la qualité des plans afin d'améliorer la recherche de solutions optimales. En effet, en plus d'évaluer plus précisément l'exécutabilité d'un plan partiel, on peut aussi s'attendre à ce que le concept d'abstraction permette d'évaluer le coût d'un plan final, et donc soit une source d'heuristique pour guider des algorithmes en planification optimale.

Chapitre 5

Planification HTN optimale

Dans ce dernier chapitre, nous nous intéressons à la planification HTN optimale, c'est-à-dire à la recherche de plans solutions qui minimisent la somme des coûts de leurs actions. Dans l'application de la planification d'actions pour l'animation de personnages dans un environnement de simulation, il est pertinent d'obtenir des solutions de bonne qualité, sans quoi le comportement des personnages simulés pourrait apparaître incohérent. Toutefois, il n'est pas nécessaire que ces solutions soient optimales. En effet, un comportement n'apparaît incohérent que s'il est grossièrement sous optimale. On peut donc ici s'orienter vers la recherche de solutions presque optimales, ce qui permet aussi de rechercher un compromis entre le temps de calcul disponible, nécessairement court dans le cadre de la planification en ligne, et la qualité des solutions.

A la fin du chapitre 3, nous avions évalué la recherche de solutions optimales pour une modélisation HTN du domaine SimpleFPS. L'algorithme utilisé était une implémentation de SHOP combinée à une étape de séparation et d'évaluation permettant d'améliorer dans le temps la qualité des solutions. Il s'agit du système de planification optimale proposé par SHOP2 [75], et qui a l'avantage de fonctionner selon le principe des algorithmes « anytime », en retournant rapidement une première solution exécutable, puis en mettant à profit le temps restant pour l'améliorer (ce qui est adapté pour la planification en ligne). Seulement, nous avions vu que les performances de cet algorithme ne permettaient pas une amélioration significative de la qualité des solutions dans la fenêtre de temps disponible (de l'ordre de quelques millisecondes).

Si on s'intéresse à l'état de la littérature en relation à la planification HTN optimale, on trouve peu de travaux. En dehors de SHOP2, Sohrabi et al. proposent par exemple avec HTNPREF, puis HTNPLAN, une approche de la planification HTN fondée sur des algorithmes de type « meilleur d'abord », dont l'un est combiné à une étape de séparation et évaluation. Toutefois, la métrique que ces planificateurs cherchent à optimiser ne correspond pas aux coûts des actions. Autrement, on retrouve l'approche suivie par AHP, qui propose d'estimer pour chaque tâche et chaque état un coût optimiste et un coût pessimiste, et d'en dériver une heuristique admissible exploitable par une application de l'algorithme A* au cadre de planification hiérarchique d'AHP. Un système similaire est aussi employé pour le planificateur *PlannedAssault* [108] destiné à la planification de missions pour les jeux Arma/Arma2 et le simulateur VBS2. Pour *PlannedAssault*, des coûts optimistes (qui sous-estiment les coûts de leurs décompositions) sont aussi modélisés au niveau des tâches composées afin de calculer une heuristique utilisable par une implémentation de A*. Toutefois, ces deux dernières approches sont destinées à la recherche de solutions optimales, et ne sont pas adaptées à un cadre « anytime ».

Dans ce chapitre, nous poursuivons deux objectifs. Le premier consiste à étudier la combinaison des deux approches existantes en planification HTN optimale, c'est-à-dire le principe de séparation et évaluation introduit par SHOP2, qui permet d'obtenir un

algorithme de recherche « anytime », et une recherche en meilleur d'abord dont la fonction d'évaluation repose sur des estimations des coûts au niveau des tâches composées, comme le fait AHP. Le second objectif concerne l'évaluation des stratégies de décomposition des tâches. Toutefois, nous ne nous intéressons plus cette fois à leur influence vis à vis de l'exécutabilité d'un plan, mais vis à vis de l'estimation de son coût.

Pour mieux comprendre cette intention, le lecteur est invité à consulté le domaine HTN représenté en figure 5.1, et qui servira de fil conducteur tout au long de ce chapitre. Ce domaine modélise le déplacement d'une équipe de trois soldats (un trinôme) entre des points de passage regroupés en zones. Au niveau du trinôme, la tâche `FireteamTravel(f, a1, a2)` fait figure « d'action abstraite », permettant de déplacer un trinôme entre deux zones. Les décompositions de cette tâche permettent ensuite de déplacer chaque soldat de ce trinôme depuis la position qu'il occupe dans la zone de départ, vers une position (qu'il faut choisir) dans la zone d'arrivée. Le déplacement entre deux zones n'est possible que si celles-ci sont adjacentes. La navigation entre deux zones s'effectue donc dans un graphe de zones. Cette navigation est modélisée par la tâche récursive `AchieveFireteamAt(f, a)`, qui introduit un déplacement quelconque dans ses décompositions tant que le trinôme désigné n'a pas atteint la zone d'arrivée.

Si on considère une stratégie de décomposition des tâches qui sélectionne en priorité les tâches de plus haut niveau, on obtient alors un plan formé uniquement de tâches de type `FireteamTravel(f, a1, a2)` avant de décomposer l'une d'entre elles. Ce plan contient une tâche pour chaque déplacement entre zones, et peut permettre d'estimer la coût total du plan final. À l'opposé, une stratégie de décomposition des tâches qui sélectionne toujours la première tâche non primitive (comme dans SHOP et SHOP2), doit décomposer chaque tâche de type `FireteamTravel(f, a1, a2)` et donc multiplier les branches de son espace de recherche (en supposant qu'il y ait dix points de passage par zones, il y aurait alors 10^3 décompositions pour chacune de ces tâches) sans disposer d'un plan précis à un horizon plus lointain, et donc sans bénéficier d'une bonne estimation des coûts.

À partir de cet exemple, on constate donc que la décomposition des réseaux de tâches en suivant les niveaux hiérarchiques pourrait apporter un avantage en orientant rapidement un algorithme de recherche heuristique vers des solutions de bonne qualité, et donc pourrait potentiellement nous permettre d'effectuer de la planification en ligne presque optimale.

La principale question que l'on ce pose dans ce chapitre, et qui est traitée en première partie est donc : pourquoi la modélisation d'actions abstraites en planification HTN apporte un cadre adapté à la planification optimale ? Pour y répondre, nous décrivons comment ces actions peuvent être utilisées pour obtenir des estimations optimistes et pessimistes des coûts de leurs décompositions, et comment ces estimations peuvent être exploitées dans un planificateur qui combine un algorithme de recherche en meilleur d'abord avec un algorithme de séparation et évaluation.

Dans la seconde partie, on se pose ensuite la question suivante : ce cadre de planification HTN optimale, permet-il d'améliorer la qualité des solutions dans le contexte de la planification en ligne ? On effectue alors une évaluation empirique du système de planification, en comparant les temps d'exécution et la qualité des solutions retournées pour trois domaines différents, ainsi que pour différentes heuristiques et différentes stratégies de décomposition.

```
task AchieveFireteamAt(f, a):
    method:
        precond:
            InArea(SoldierAt(FireteamSoldier1(f)), a)
            InArea(SoldierAt(FireteamSoldier2(f)), a)
            InArea(SoldierAt(FireteamSoldier3(f)), a)
    method a1, a2:
        precond:
            Area(a1) and Area(a2) and a1 != a and Adjacent(a1, a2)
            InArea(SoldierAt(FireteamSoldier1(f)), a1)
            InArea(SoldierAt(FireteamSoldier2(f)), a1)
            InArea(SoldierAt(FireteamSoldier3(f)), a1)
        subtasks:
            FireteamTravel(f, a1, a2)
            AchieveFireteamAt(f, a)

task FireteamTravel(f, a1, a2):
    method p1, p2, p3:
        precond:
            Adjacent(a1, a2)
            InArea(SoldierAt(FireteamSoldier1(f)), a1)
            InArea(SoldierAt(FireteamSoldier2(f)), a1)
            InArea(SoldierAt(FireteamSoldier3(f)), a1)
            InArea(p1, a2) and InArea(p2, a2) and InArea(p3, a2)
        subtasks:
            n1 : AchieveSoldierNotCovering(FireteamSoldier1(f))
            n2 : SoldierTravel(FireteamSoldier1(f), p1)
            n3 : AchieveSoldierNotCovering(FireteamSoldier2(f))
            n4 : SoldierTravel(FireteamSoldier2(f), p2)
            n5 : AchieveSoldierNotCovering(FireteamSoldier3(f))
            n6 : SoldierTravel(FireteamSoldier3(f), p3)
        order:
            n1 < n2, n3 < n4, n5 < n6

task AchieveSoldierNotCovering(s):
    method:
        precond:
            not Covering(s)
    method:
        precond:
            Covering(s)
        subtasks:
            SoldierStopCovering(s)

task SoldierTravel(s, p):
    precond:
        Soldier(s) and Waypoint(p)
        not Covering(s)
    update:
        SoldierAt(s) = p
    cost: 1

task SoldierStopCovering(s):
    precond:
        Soldier(s) and Covering(s)
    delete:
        Covering(s)
    cost: 1
```

FIGURE 5.1 – Modélisation du déplacement d'une équipe de soldats
Modélisation en HTDL du déplacement d'une équipe composée de trois soldats à travers un graphe de zones.

5.1 Recherche heuristique pour planification HTN optimale

Cette partie présente un cadre général de planification HTN optimale. Elle présente tout d'abord l'introduction de coûts de haut niveau optimistes et pessimistes associés aux tâches composées, puis décrit un algorithme de planification HTN optimale adapté à la planification en ligne, et présente enfin différents types d'heuristiques pouvant être modélisées pour cet algorithme à partir des coûts de haut niveau.

5.1.1 Estimation du coût des tâches composées

Notre objectif est de mettre en place un système de planification HTN qui permet de minimiser les coûts des solutions. Afin de simplifier la présentation du cadre de planification, on choisit de se restreindre à des coûts constants, c'est-à-dire des coûts qui ne varient pas en fonction des états. Pour un domaine de planification HTN $D = (S, A, C, M, \mathcal{T}, \mathcal{C})$, la fonction de coût $\mathcal{C}$ est donc un élément de $A \to \mathbb{R}^+$. Cette hypothèse permet entre autre de simplifier la généralisation de notre système à la planification HTN non linéaire, puisque qu'il n'est pas toujours possible de définir exactement l'état qui précède chaque action dans un plan partiellement ordonné.

Afin d'établir un algorithme de recherche fondé sur une approche en meilleur d'abord (BFS, pour « Best First Search » en anglais), il faut disposer d'une fonction d'évaluation (généralement baptisée « f ») qui à chaque élément de l'espace de recherche, associe une estimation de la grandeur que l'on souhaite minimiser. Ici, les éléments sont les réseaux de tâches, et la grandeur que l'on souhaite minimiser est le coût du plan que l'on obtiendra en décomposant ce réseau de tâches. Une première composante qui peut être prise en compte dans le calcul de cette estimation est la somme des coûts des tâches primitives déjà présentes dans un réseau de tâches. On désigne cette composante sous le nom de « coût primitif » d'un réseau de tâches.

Définition 52 (Coût primitif d'un réseau de tâche)**.** *Soit un réseau de tâches $\rho \in R_{A \cup C}$. Le coût primitif de ρ, noté $g(\rho)$, est défini par :*

$$g(\rho) = \sum_{n \in N_\rho | t_n \in A} \mathcal{C}(t_n)$$

Or, à lui seul, ce coût primitif n'apporte pas une estimation satisfaisante. En effet, il ne prend pas du tout en compte le coût des actions qui seront introduites par les tâches composées. Comment peut on alors obtenir une estimation de ces coûts ? Une solution nous est proposée par AHP, qui introduit deux fonctions de coûts au niveau des tâches composées. La première de ces deux fonctions est dite « optimiste » car elle sous-estime le coût de n'importe quelle séquence d'actions que l'on peut obtenir à partir d'une tâche composée. Contrairement à AHP, on définit ici une fonction de coût optimiste indépendante par rapport aux états :

Définition 53 (Fonction de coût optimiste)**.** *Une fonction de coût optimiste $\mathcal{C}_{opt}$ et une fonction de $C \to \mathbb{R}^+$ telle que pour tout $c \in C$, et toute séquence $\sigma \in A^*$ telle que $\mathcal{D}^*(\varrho(c), \varrho(\sigma))$, on a :*

$$\mathcal{C}_{opt}(c) \leq \mathcal{C}(\sigma)$$

La deuxième fonction correspond à une estimation « pessimiste », car elle sur-estime le coût de n'importe quelle séquence d'actions que l'on peut obtenir en décomposant successivement une tâche composée. Pour ce cadre, on définit à nouveau une fonction indépendante des états :

Définition 54 (Fonction de coût pessimiste)**.** *Une fonction de coût pessimiste $\mathcal{C}_{pes}$ et une fonction de $C \to \mathbb{R}^+$ telle que pour tout $c \in C$, et toute séquence $\sigma \in A^*$ telle que $\mathcal{D}^*(\varrho(c), \varrho(\sigma))$, on a :*

$$\mathcal{C}_{pes}(c) \geq \mathcal{C}(\sigma)$$

En HTDL, on introduit simplement ces deux valeurs par des champs « `optimistic:` » et « `pessimistic:` » similaires au champ « `cost:` », mais réservés aux tâches composées. La figure 5.2 représente la tâche `FireteamTravel(f, a1, a2)`, extraite du domaine en HTDL décrit à la figure 5.1, à laquelle on a ajouté les deux champs correspondant aux coûts optimiste et pessimiste. Le coût optimiste de cette tâche est de 3, car dans le meilleur des cas, il suffit d'effectuer un déplacement pour chacun des trois soldats, chaque action de déplacement `SoldierTravel(s, p)` ayant un coût égal à 1, pour exécuter cette tâche composée. Quant au coût pessimiste, celui-ci vaut 6, car dans le pire des cas, chaque soldat peut avoir a effectuer en plus une action de type `StopCovering(s)`, dont le coût est aussi égale à 1.

```
task FireteamTravel(f, a1, a2):
    method p1, p2, p3:
        precond:
            Adjacent(a1, a2)
            InArea(SoldierAt(FireteamSoldier1(f)), a1)
            InArea(SoldierAt(FireteamSoldier2(f)), a1)
            InArea(SoldierAt(FireteamSoldier3(f)), a1)
            InArea(p1, a2) and InArea(p2, a2) and InArea(p3, a2)
        subtasks:
            n1 : AchieveSoldierNotCovering(FireteamSoldier1(f))
            n2 : SoldierTravel(FireteamSoldier1(f), p1)
            n3 : AchieveSoldierNotCovering(FireteamSoldier2(f))
            n4 : SoldierTravel(FireteamSoldier2(f), p2)
            n5 : AchieveSoldierNotCovering(FireteamSoldier3(f))
            n6 : SoldierTravel(FireteamSoldier3(f), p3)
        order:
            n1 < n2, n3 < n4, n5 < n6
    optimistic:  3
    pessimistic: 6
```

FIGURE 5.2 – Coûts optimiste et pessimiste
Modélisation en HTDL de la tâche composée `FireteamTravel(f, a1, a2)` avec un coût optimiste et un coût pessimiste.

La définition que nous venons de donner à un coup pessimiste n'est cependant pas tout à fait satisfaisante. En effet, il est possible que l'ensemble des coûts des séquences d'actions que l'on peut obtenir en décomposant une tâche composée ne soit pas borné, ce qui est souvent le cas avec les tâches récursives. Le coût pessimiste ne peut donc pas toujours être défini. En pratique, on peut contourner ce problème en choisissant une valeur suffisamment importante pour sur-estimer les séquences d'actions qui feront partie des solutions optimales. Une autre possibilité consiste à introduire un paramètre indiquant la profondeur d'appel d'une tâche récursive : la première fois que cette tâche est introduite dans un réseau de tâches, ce paramètre vaut 0, puis il est incrémenté de 1 à chaque appel récursif, comme on peut le voir dans l'exemple de la figure 5.3. On peut alors limiter la profondeur d'appel, ce qui a pour effet de limiter la taille des séquences d'actions pouvant être obtenues en décomposant cette tâche. Dans l'exemple de la figure 5.3, la profondeur

d'appel est limitée à 10. Si on suppose qu'aucun problème ne contient plus de 10 zones de navigation, alors cette limite n'impacte pas la complétude du domaine, puisque seules des séquences d'actions non optimales (car cycliques) sont supprimées par ce procédé.

```
task AchieveFireteamAt(f, a, k):
    method:
        precond:
            InArea(SoldierAt(FireteamSoldier1(f)), a)
            InArea(SoldierAt(FireteamSoldier2(f)), a)
            InArea(SoldierAt(FireteamSoldier3(f)), a)
    method a1, a2:
        precond:
            k < 10
            Area(a1) and Area(a2) and a1 != a and Adjacent(a1, a2)
            InArea(SoldierAt(FireteamSoldier1(f)), a1)
            InArea(SoldierAt(FireteamSoldier2(f)), a1)
            InArea(SoldierAt(FireteamSoldier3(f)), a1)
        subtasks:
            FireteamTravel(f, a1, a2)
            AchieveFireteamAt(f, a, k + 1)
    optimistic: 0
    pessimistic: (10 - k) * 6
```

FIGURE 5.3 – Coût pessimiste bornée
Ajout d'un paramètre k pour limiter la profondeur d'appel de la tâche récursive `AchieveFireteamAt(f, a, k)` et permettre ainsi de définir un coût pessimiste.

Il est possible de modéliser manuellement les coûts optimiste et pessimiste de chaque tâche dans un domaine de planification HTN. Cependant ce travail peut s'avérer très fastidieux pour un domaine contenant de nombreux niveaux de décomposition. En effet, plus une tâche se situe à un haut niveau dans la hiérarchie des tâches d'un domaine de planification HTN, et plus l'ensemble des séquences d'actions qui correspondent à son exécution peut être difficile à évaluer pour un opérateur humain. De plus, ce procédé nuit aussi à l'évolution et à la maintenabilité d'un domaine de planification, car toute modification d'une tâche existante ou introduction de nouveaux éléments dans le domaine remet en cause la définition d'une partie des coûts optimistes et pessimistes.

La situation idéale serait alors de disposer d'une procédure pour calculer automatiquement les coûts optimistes et pessimistes à partir des coûts des tâches primitives. Une telle procédure est décrite par l'algorithme 13. Cet algorithme prend en entrée un domaine de planification HTN, et retourne une fonction de coût optimiste ainsi qu'une fonction de coût pessimiste pour les tâches composées. Pour cela, tous les coûts optimistes sont initialisés avec une valeur de $+\infty$ et tous les coûts pessimistes sont initialisés avec une valeur de $-\infty$. Ensuite, à chaque itération de la boucle principale de l'algorithme, la valeur du coût optimiste de chaque tâche est mise à jour en prenant le minimum de la somme du coût primitif et des coûts optimistes de chaque décomposition. De même, la valeur du coût pessimiste d'une tâche est mise à jour en prenant le maximum de la somme du coût primitif et des coûts pessimistes pour chaque décomposition de cette tâche. Cette étape est répétée jusqu'à ce qu'un point fixe soit atteint, et les coûts optimistes et pessimistes obtenus pour chaque tâche sont alors retournés.

Cependant, cet algorithme ne fonctionne pas dans toutes les situations. En effet, s'il existe une tâche du domaine pour lequel l'ensemble des coûts des séquences d'actions que

Algorithm 13 Algorithme pour calculer les fonctions de coût optimiste et pessimiste

entrées : Un domaine de planification HTN $D = (S, A, C, M, \mathcal{T}, \mathcal{C})$.
sorties : Une fonction de coût optimiste $\mathcal{C}_{opt}$ et une fonction de coût pessimiste $\mathcal{C}_{pes}$.

pour tout $c \in C$ **faire**
  $\mathcal{C}_{opt}(c), \mathcal{C}_{pes}(c) \leftarrow (+\infty, -\infty)$
fin pour
faire
  $(\mathcal{C}'_{opt}, \mathcal{C}'_{pes}) \leftarrow (\mathcal{C}_{opt}, \mathcal{C}_{pes})$
  pour tout $c \in C$ **faire**
    $\mathcal{C}_{opt}(c) \leftarrow \min\{\sum_{n \in N_{\rho m} | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_{\rho m} | t_n \in C} \mathcal{C}'_{opt}(t_n) \mid m \in M : c_m = c\} \cup \{\mathcal{C}'_{opt}(c)\}$
    $\mathcal{C}_{pes}(c) \leftarrow \max\{\sum_{n \in N_{\rho m} | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_{\rho m} | t_n \in C} \mathcal{C}'_{pes}(t_n) \mid m \in M : c_m = c\} \cup \{\mathcal{C}'_{pes}(c)\}$
  fin pour
tant que $(\mathcal{C}'_{opt}, \mathcal{C}'_{pes}) \neq (\mathcal{C}_{opt}, \mathcal{C}_{pes})$
retourner $(\mathcal{C}_{opt}, \mathcal{C}_{pes})$

l'on peut obtenir en la décomposant n'est pas borné, alors le point fixe ne peut pas être atteint, et le coût pessimiste de cette tâche tendra alors vers l'infini. Dans ce cas, cet algorithme peut encore être exploité, en ajoutant en entrée des bornes supérieures pour ces tâches dont les coûts pessimistes sont théoriquement infinis. À noter que dans le cas où on est exclusivement intéressé par la fonction de coût optimiste, alors celle-ci peut être retournée dans toute les situations, puisque chaque coût optimiste est borné par 0.

On va maintenant démontrer que cet algorithme termine bien en temps fini. Pour cela, on commence par définir la notation suivante, qui concerne l'ensemble des coûts qui correspondent aux séquences d'actions accessibles en décomposant successivement une tâche. Ainsi, pour tout $c \in C$, on note :

$$V(c) = \mathcal{C}(\{\sigma \in A^* \mid \mathcal{D}^*(\varrho(c), \varrho(\sigma))\})$$

C'est donc cet ensemble $V(c)$ qui doit être borné pour chaque tâche composée c du domaine afin de garantir la terminaison de l'algorithme. On peut ensuite montrer que si cet ensemble est borné, alors il est aussi fini :

Lemme 7. *Soit une tâche $c \in C$.*
Si $V(c)$ est borné, alors $V(c)$ est un ensemble fini.

Démonstration. Soit $c \in C$ tel que $V(c)$ est borné. Il existe alors un nombre $B > 0$ tel que $\max V(c) < B$.
On note $\mathcal{C}_{min} = \min \mathcal{C}(A)$.

1. On suppose dans un premier temps que $\mathcal{C}_{min} > 0$.
   On a $A^* = \bigcup_{i \in [\![0, +\infty]\!]} A^i$, ce qui nous permet d'écrire :

$$V(c) = \bigcup_{i \in [\![0, +\infty]\!]} \mathcal{C}(\{\sigma \in A^i \mid \mathcal{D}^*(\varrho(c), \varrho(\sigma))\})$$

   Soit $i_0 \in [\![0, +\infty]\!]$ tel que $i_0 > \frac{B}{\mathcal{C}_{min}}$.
   Soient $i \geq i_0$ et $\sigma \in A^i$. On a alors :

$$\begin{aligned} \mathcal{C}(\sigma) &\geq i * \mathcal{C}_{min} \\ &> \frac{B}{\mathcal{C}_{min}} \times \mathcal{C}_{min} \\ &> B \end{aligned}$$

On en déduit donc que pour tout $i > 0$, $\{\sigma \in A^i \mid \mathcal{D}^*(\varrho(t), \varrho(\sigma))\} = \emptyset$.
On en déduit donc que :

$$V(t) = \bigcup_{i \in [\![0, i_0]\!]} \mathcal{C}(\{\sigma \in A^i \mid \mathcal{D}^*(\varrho(t), \varrho(\sigma))\})$$

Or pour tout $i \in [\![0, i_0]\!]$, A^i est un ensemble fini (de cardinal $|A|^i$). On en déduit donc que $\{\sigma \in A^i \mid \mathcal{D}^*(\varrho(t), \varrho(\sigma))\}$ est un ensemble fini, et donc que $\mathcal{C}(\{\sigma \in A^i \mid \mathcal{D}^*(\varrho(c), \varrho(\sigma))\})$ est un ensemble fini.
$V(c)$ est donc une réunion finie d'ensembles finis, ce qui implique que $V(c)$ est un ensemble fini.

2. Supposons maintenant que $\mathcal{C}_{min} = 0$.
On note alors $A' = \{a \in A \mid \mathcal{C}(a) > 0\}$.
Soit $\sigma \in A^*$. On note $\sigma' \in A'^*$ la séquence d'actions obtenue à partir de σ en en supprimant les actions de coûts nuls.
On a alors $\mathcal{C}(\sigma) = \mathcal{C}(\sigma')$, ce qui nous permet d'en déduire que :

$$V(c) \subseteq \mathcal{C}(\{\sigma \in A'^* \mid \mathcal{D}^*(\varrho(c), \varrho(\sigma))\})$$

D'après la démonstration précédente, on en déduit donc que $\mathcal{C}(\{\sigma \in A'^* \mid \mathcal{D}^*(\varrho(c), \varrho(\sigma))\})$ est un ensemble fini, et donc que $V(c)$ est un ensemble fini.

□

On note alors :

$$V = \bigcup_{c \in C} V(c)$$

et :

$$E = (V \cup \{+\infty\})^C \times (V \cup \{-\infty\})^C$$

E est l'ensemble des paires de fonctions dont la première associe les éléments de C aux éléments de l'ensemble V ou à $+\infty$, et la seconde associe les éléments de C aux éléments de V ou a $-\infty$. E est l'ensemble auquel appartiennent les fonctions de coût optimiste et pessimiste à chaque étape de l'algorithme (ce que nous montrerons plus tard). On peut montrer que E est un treillis. Si de plus on fait l'hypothèse que $V(c)$ est borné pour tout $c \in C$, alors on peut en déduire que V est fini, et donc que E est un treillis fini, ce qui implique qu'il s'agit d'un treillis complet.

Lemme 8. *Soient $\sqcup$ et $\sqcap$ les deux applications internes de E telles que pour tout $((f_{1,1}, f_{1,2}), (f_{2,1}, f_{2,2})) \in E^2$ et tout $c \in C$, on a :*

$$((f_{1,1}, f_{1,2}) \sqcup (f_{2,1}, f_{2,2}))(c) = (\min(f_{1,1}(c), f_{2,1}(c)), \max(f_{1,2}(c), f_{2,2}(c)))$$
$$((f_{1,1}, f_{1,2}) \sqcap (f_{2,1}, f_{2,2}))(c) = (\max(f_{1,1}(c), f_{2,1}(c)), \min(f_{1,2}(c), f_{2,2}(c)))$$

De plus, on note $\sqsubseteq$ la relation sur E définie telle que pour tout $(p_1, p_2) \in E^2$ on a :

$$p_1 \sqsubseteq p_2 \iff p_1 \sqcup p_2 = p_2$$

$(E, \sqsubseteq)$ forme alors un treillis complet.

Démonstration. Pour montrer que $(E, \sqsubseteq)$ est un treillis, il suffit de montrer que $\sqcup$ et $\sqcap$ sont associatives, commutatives et satisfont la loi d'absorption.

1. Associativité :
   Pour tout $(x, y, z) \in \mathbb{R}^3$ on a :

$$\begin{aligned}\min(x, \min(x, y)) &= \min(\min(x, y), z)\\ \max(x, \max(x, y)) &= \max(\max(x, y), z)\end{aligned}$$

   On peut alors en déduire que $\sqcup$ et $\sqcap$ sont associatives.
2. Commutativité :
   Pour tout $x, y \in \mathbb{R}^2$ on a :

$$\begin{aligned}\min(x, y) &= \min(y, x)\\ \max(x, y) &= \max(y, x)\end{aligned}$$

   On peut alors en déduire que $\sqcup$ et $\sqcap$ sont commutatives.
3. Loi d'absorption :
   Soit $((f_{1,1}, f_{1,2}), (f_{2,1}, f_{2,2})) \in E^2$.
   Montrons que :

$$(f_{1,1}, f_{1,2}) \sqcap ((f_{1,1}, f_{1,2}) \sqcup (f_{2,1}, f_{2,2})) = (f_{1,1}, f_{1,2}) \tag{5.1}$$
$$(f_{1,1}, f_{1,2}) \sqcup ((f_{1,1}, f_{1,2}) \sqcap (f_{2,1}, f_{2,2})) = (f_{1,1}, f_{1,2}) \tag{5.2}$$

   Montrons d'abords 5.1.
   Soit $c \in C$. On a alors :

$$\begin{aligned}&(f_{1,1}(c), f_{1,2}(c)) \sqcap ((f_{1,1}(c), f_{1,2}(c)) \sqcup (f_{2,1}(c), f_{2,2}(c))) =\\ &\quad(\min(f_{1,1}(c), \max(f_{1,1}(c), f_{2,1}(c))), \max(f_{1,2}(c), \min(f_{1,2}(c), f_{2,2}(c))))\end{aligned}$$

   On a alors d'une part :
   (a) si $f_{1,1}(c) < f_{2,1}(c)$:

$$\begin{aligned}\min(f_{1,1}(c), \max(f_{1,1}(c), f_{2,1}(c))) &= \min(f_{1,1}(c), f_{2,1}(c))\\ &= f_{1,1}(c)\end{aligned}$$

   (b) sinon :

$$\begin{aligned}\min(f_{1,1}(c), \max(f_{1,1}(c), f_{2,1}(c))) &= \min(f_{1,1}(c), f_{1,1}(c))\\ &= f_{1,1}(c)\end{aligned}$$

   Et d'autre part :
   (a) si $f_{1,2}(c) < f_{2,2}(c)$:

$$\begin{aligned}\max(f_{1,2}(c), \min(f_{1,2}(c), f_{2,2}(c))) &= \max(f_{1,2}(c), f_{1,2}(c))\\ &= f_{1,2}(c)\end{aligned}$$

   (b) sinon :

$$\begin{aligned}\max(f_{1,2}(c), \min(f_{1,2}(c), f_{2,2}(c))) &= \max(f_{1,2}(c), f_{2,2}(c))\\ &= f_{1,2}(c)\end{aligned}$$

5.1 est donc bien satisfaite.
La démonstration pour 5.2 étant analogue, celle-ci est omise.

$(E, \sqsubseteq)$ forme donc bien un treillis.
De plus, d'après le lemme 7, pour tout $c \in C$, $V(c)$ est un ensemble fini, donc V est un ensemble fini et donc E est un ensemble fini.
Or, un treillis fini admet toujours une borne inférieure et une borne supérieure, donc E est bien un treillis complet. □

De plus, si on modélise les opérations effectuées par la boucle principale de l'algorithme 13 par une fonction, on peut montrer que cette fonction est croissante sur E. Comme E est un treillis complet, on en déduit alors que cette fonction atteint un point fixe (d'après le théorème de Tarsky [97]). Comme E est un ensemble fini, ce point fixe est donc atteint en un nombre fini d'étapes, ce qui garanti donc que l'algorithme termine en temps fini.

Théorème 9. *Si pour tout $c \in C$, $\mathcal{C}(\{\sigma \in A^* \mid \mathcal{D}^*(\varrho(c), \varrho(\sigma))\})$ est borné, alors l'algorithme 13 termine.*

Démonstration. Ce théorème est une conséquence du théorème du point fixe de Tarsky [97], énoncé comme suit :
« Soient $(E, \sqsubseteq)$ un treillis complet et u une application croissante sur E. Alors l'ensemble des points fixes de u est un treillis complet. »
Afin d'appliquer ce théorème, nous allons montrer que la boucle de l'algorithme 13 peut être vu comme l'application d'une fonction croissante sur les éléments de E.
On note $B = maxV$ et $F = ([0, B] \cup \{+\infty\})^C \times ([0, B] \cup \{-\infty\})^C$.
Soit $u \in F \to F$ l'application qui pour tout $(f_1, f_2) \in F^2$ retourne une paire de fonctions $(f_1', f_2') \in F^2$ telle que pour tout $c \in C$ on a :

$$f_1'(c) = \min\{\sum_{n \in N_{\rho m} | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_{\rho m} | t_n \in C} f_1(t_n) \mid m \in M : c_m = c\} \cup \{f_1(c)\}$$
$$f_2'(c) = \max\{\sum_{n \in N_{\rho m} | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_{\rho m} | t_n \in C} f_2(t_n) \mid m \in M : c_m = c\} \cup \{f_2(c)\}$$

On note aussi $\mathcal{C}_{opt}^i$ et $\mathcal{C}_{pes}^i$ les valeurs de $\mathcal{C}_{opt}$ et de $\mathcal{C}_{pes}$ obtenues à la i-ième itération de l'algorithme 13.
D'après la description de l'algorithme, on a pour tout $i \geq 0$:

$$(\mathcal{C}_{opt}^{i+1}, \mathcal{C}_{pes}^{i+1}) = u((\mathcal{C}_{opt}^i, \mathcal{C}_{pes}^i))$$

Et de plus, l'algorithme termine à la première itération $i_0 > 0$ telle que $(\mathcal{C}_{opt}^{i_0}, \mathcal{C}_{pes}^{i_0}) = u((\mathcal{C}_{opt}^{i_0}, \mathcal{C}_{pes}^{i_0}))$.

1. Montrons que u est une application croissante sur F :
   Soit $(f_1, f_2) \in F^2$ et $(f_1', f_2') \in F^2$ telles que $u((f_1, f_2)) = (f_1', f_2')$. Montrons que $(f_1, f_2) \sqsubseteq u((f_1, f_2))$, c'est-à-dire que $(f_1, f_2) \sqcup (f_1', f_2') = (f_1', f_2')$.
   Soit $c \in C$. On a alors :

$$((f_1, f_2) \sqcup (f_1', f_2'))(c) = (\min(f_1(c), f_1'(c)), \max(f_2(c), f_2'(c)))$$

Or on a d'une part :

$$\begin{aligned}\min(f_1(c), f_1'(c)) &= \min(f_1(c), \\ &\quad \min(\{\sum_{n\in N_{\rho_m}|t_n\in A} \mathcal{C}(t_n) + \sum_{n\in N_{\rho_m}|t_n\in C} f_1(t_n) \mid m \in M : c_m = c\} \\ &\quad \cup \{f_1(c)\})) \\ &= \min(\{\sum_{n\in N_{\rho_m}|t_n\in A} \mathcal{C}(t_n) + \sum_{n\in N_{\rho_m}|t_n\in C} f_1(t_n) \mid m \in M : c_m = c\} \\ &\quad \cup \{f_1(c)\}) \\ &= f_1'(c)\end{aligned}$$

Et d'autre part :

$$\begin{aligned}\min(f_2(c), f_2'(c)) &= \max(f_2(c), \\ &\quad \max(\{\sum_{n\in N_{\rho_m}|t_n\in A} \mathcal{C}(t_n) + \sum_{n\in N_{\rho_m}|t_n\in C} f_2(t_n) \mid m \in M : c_m = c\} \\ &\quad \cup \{f_2(c)\})) \\ &= \min(\{\sum_{n\in N_{\rho_m}|t_n\in A} \mathcal{C}(t_n) + \sum_{n\in N_{\rho_m}|t_n\in C} f_2(t_n) \mid m \in M : c_m = c\} \\ &\quad \cup \{f_2(c)\}) \\ &= f_2'(c)\end{aligned}$$

On a donc bien pour tout $c \in C$:

$$((f_1, f_2) \sqcup (f_1', f_2'))(c) = (f_1'(c), f_2'(c))$$

On en déduit donc que u est une application croissante F.

2. Montrons que pour tout $i \geq 0$, $(\mathcal{C}^i_{opt}, \mathcal{C}^i_{pes}) \in E$:
   Pour cela, on montre que pour tout $i \geq 0$, et tout $c \in C$, on a $\mathcal{C}^i_{opt}(c) \in V(c) \cup \{+\infty\}$ et $\mathcal{C}^i_{pes}(c) \in V(c) \cup \{-\infty\}$:
   La démonstration se fait par récurrence sur i :
   (a) $i = 0$: D'après l'algorithme 13, on a initialement $\mathcal{C}^0_{opt}(c) = +\infty$ et $\mathcal{C}^0_{pes}(c) = -\infty$ pour tout $c \in C$.
   On a donc bien $\mathcal{C}^0_{opt}(c) \in V(c) \cup \{+\infty\}$ et $\mathcal{C}^0_{pes}(c) \in V(c) \cup \{-\infty\}$ pour tout $c \in C$.
   (b) Soit $i \geq 0$. On suppose que quelque soit $c \in C$, $\mathcal{C}^i_{opt}(c) \in V(c) \cup \{+\infty\}$ et $\mathcal{C}^i_{pes}(c) \in V(c) \cup \{-\infty\}$.
   Soit $c \in C$, on a alors :

$$\mathcal{C}^{i+1}_{opt}(c) = \min\{\sum_{n\in N_{\rho_m}|t_n\in A} \mathcal{C}(t_n) + \sum_{n\in N_{\rho_m}|t_n\in C} \mathcal{C}^i_{opt}(t_n) \mid m \in M : c_m = c\} \cup \{\mathcal{C}^i_{opt}(c)\}$$

   Ainsi si $\mathcal{C}^{i+1}_{opt}(c) = \mathcal{C}^i_{opt}(c)$, on a alors $\mathcal{C}^{i+1}_{opt}(c) \in V \cup \{+\infty\}$.
   Sinon, il existe alors un $m \in M$ tel que $c_m = c$ et :

$$\mathcal{C}^{i+1}_{opt}(c) = \sum_{n\in N_{\rho_m}|t_n\in A} \mathcal{C}(t_n) + \sum_{n\in N_{\rho_m}|t_n\in C} \mathcal{C}^i_{opt}(t_n)$$

   Ainsi s'il existe un $n \in N_{\rho_m}$ tel que $\mathcal{C}^i_{opt}(t_n) = +\infty$, alors $\mathcal{C}^{i+1}_{opt}(c) = +\infty$.
   Sinon, on a $\forall n \in N_{\rho_m} : \mathcal{C}^i_{opt}(t_n) \in V(t_n)$, et on peut en déduire que $\mathcal{C}^{i+1}_{opt}(c) \in V(c)$.
   Pour tout $c \in C$, on a donc bien $\mathcal{C}^i_{opt}(c) \in V(c) \cup \{+\infty\}$.
   La démonstration pour $\mathcal{C}^{i+1}_{pes}$ est analogue et est donc ignorée.

On en déduit alors que pour tout $i \geq 0$, on a $(\mathcal{C}^i_{opt}, \mathcal{C}^i_{pes}) \in E$.

Ainsi pour tout $i \geq 0$, $(\mathcal{C}^i_{opt}, \mathcal{C}^i_{pes}) \in E$, qui est un treillis complet d'après le lemme 8.
De plus, f est croissante sur F. Or $E \subseteq F$, donc u est aussi une fonction croissante sur E.
D'après le théorème de Tarsky, on en déduit donc que l'ensemble des points fixes de f sur E est un treillis complet. Or $\emptyset$ n'est pas un treillis complet (un treillis complet admet au moins une borne inférieure et une borne supérieure) donc il existe un point fixe de f sur E.
De plus, on sait d'après le lemmme 7 que E est un ensemble fini, ce qui nous permet d'en déduire que le point fixe est atteint en un nombre fini d'étapes.
On en déduit donc que l'algorithme 13 termine. □

On a donc montré que l'algorithme 13 termine si $V(c)$ est borné pour tout $c \in C$. Il ne reste donc plus qu'à montrer que dans ces conditions, les fonctions de coût optimiste et pessimiste retournées satisfont bien les définitions 53 et 54.

Théorème 10. *Si l'algorithme 13 termine, alors il retourne une paire de fonctions de coût optimiste et pessimiste.*

Démonstration. Soit $i_0 > 0$ la plus petite itération pour laquelle $(\mathcal{C}^{i_0}_{opt}, \mathcal{C}^{i_0}_{pes}) = (\mathcal{C}^{i_0-1}_{opt}, \mathcal{C}^{i_0-1}_{pes})$.
Montrons que pour tout $c \in C$, on a :

$$\mathcal{C}^{i_0}_{opt}(c) \leq \min V(c)$$
$$\mathcal{C}^{i_0}_{pes}(c) \geq \max V(c)$$

Pour tout $c \in C$ et tout $k > 1$, on note $D_k(c)$ l'ensemble défini par :

$$D_k(c) = \{\sigma \in A^* \mid \exists (r_1, ..., r_k) \in R^*_{A \cup C} : r_1 = \varrho(c) \wedge r_k = \varrho(\sigma) \wedge \forall i \in [\![1, k-1]\!] : \mathcal{D}(r_i, r_{i+1})\}$$

Pour tout $k > 1$, on note alors $P(k)$ la proposition suivante :

$$\forall c \in C : \mathcal{C}^{i_0}_{ops}(c) \leq \min \mathcal{C}(D_k(c)) \wedge$$
$$\mathcal{C}^{i_0}_{pes}(c) \geq \max \mathcal{C}(D_k(c))$$

Montrons par récurrence forte que $P(k)$ est vraie pour tout $k > 1$:

1. $P(2)$:
   Soient $c \in C$ et $\sigma \in D_2(c)$.
   On a alors $\mathcal{D}(\varrho(c), \varrho(\sigma))$ donc il existe $m \in M$ telle que $c_m = c$ et $\rho_m = \varrho(\sigma)$.
   On remarque que $\{n \in N_{\rho_m} \mid t_n \in C\} = \emptyset$, car ρ_m est un réseau de tâches primitif.
   D'après l'algorithme 13, on a alors :

$$\begin{aligned} \mathcal{C}^{i_0}_{ops}(c) &\leq \sum_{n \in N_{\varrho(\sigma)} | t_n \in A} \mathcal{C}(t_n) \\ &\leq \mathcal{C}(\sigma) \\ \mathcal{C}^{i_0}_{pes}(c) &\geq \sum_{n \in N_{\varrho(\sigma)} | t_n \in A} \mathcal{C}(t_n) \\ &\leq \mathcal{C}(\sigma) \end{aligned}$$

On en déduit donc que $\mathcal{C}^{i_0}_{ops}(c) \leq \min \mathcal{C}(D_2(c)) \wedge \mathcal{C}^{i_0}_{pes}(c) \geq \max \mathcal{C}(D_2(c))$.

2. $\forall k_0 > 1 : ((\forall k \leq k_0 : P(k)) \implies P(k_0+1))$:
   Soit $k_0 > 1$, on suppose que $P(k)$ est vraie pour tout entier positif $k \leq k_0$.
   Soient $c \in C$ et $\sigma \in D_{k_0+1}(c)$.
   Il existe alors $m \in M$ telle que $c_m = c$, et telle que

$$\mathcal{C}_{ops}^{i_0}(c) \leq \sum_{n \in N_{\rho m} | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_{\rho m} | t_n \in C} \mathcal{C}_{ops}^{i_0-1}(t_n)$$
$$\mathcal{C}_{pes}^{i_0}(c) \geq \sum_{n \in N_{\rho m} | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_{\rho m} | t_n \in C} \mathcal{C}_{pes}^{i_0-1}(t_n)$$

   De plus pour tout $n \in N_{\rho_m}$ tel que $t_n \in C$, il existe $\sigma_n \in \bigcup_{k \in [\![0,k_0]\!]} D_k(t_n)$ tels que :

$$\mathcal{C}(\sigma) = \sum_{n \in N_{\rho m} | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_{\rho m} | t_n \in C} \mathcal{C}(\sigma_n)$$

   Comme $P(k)$ est vraie pour tout $k \in [\![2, k_0]\!]$, et que $\mathcal{C}_{opt}^{i_0} = \mathcal{C}_{opt}^{i_0-1}$ et $\mathcal{C}_{pes}^{i_0} = \mathcal{C}_{pes}^{i_0-1}$, on en déduit alors que pour tout $n \in N_{\rho_m}$ tel que $t_n \in C$, on a :

$$\mathcal{C}_{ops}^{i_0-1}(t_n) \leq \mathcal{C}(\sigma_n)$$
$$\mathcal{C}_{pes}^{i_0-1}(t_n) \geq \mathcal{C}(\sigma_n)$$

   On en déduit donc que :

$$\begin{aligned}
\mathcal{C}_{ops}^{i_0}(c) &\leq \sum_{n \in N_{\rho m} | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_{\rho m} | t_n \in C} \mathcal{C}(\sigma_n) \\
&\leq \mathcal{C}(\sigma) \\
\mathcal{C}_{pes}^{i_0}(c) &\geq \sum_{n \in N_{\rho m} | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_{\rho m} | t_n \in C} \mathcal{C}(\sigma_n) \\
&\geq \mathcal{C}(\sigma)
\end{aligned}$$

   Et donc que $\mathcal{C}_{ops}^{i_0}(c) \leq \min \mathcal{C}(D_{k_0+1}(c))$ et $\mathcal{C}_{pes}^{i_0}(c) \geq \max \mathcal{C}(D_{k_0+1}(c))$.

Ainsi $P(2)$ est vraie et pour tout $k_0 > 1$, si $P(k)$ est vraie pour tout $k \in [\![2, k_0]\!]$, alors $P(k_0+1)$ est vraie.
Par récurrence forte, on en déduit donc que $P(k)$ est vraie pour tout $k > 1$.
On en déduit donc que pour tout $c \in C$ on a :

$$\mathcal{C}_{opt}^{i_0}(c) \leq \min \bigcup_{k>1} \mathcal{C}(D_k(c))$$
$$\mathcal{C}_{pes}^{i_0}(c) \geq \max \bigcup_{k>1} \mathcal{C}(D_k(c))$$

Or, pour tout $c \in C$, on a $V(c) = \bigcup_{k>1} \mathcal{C}(D_k(c))$.
On en déduit donc que pour tout $c \in C$:

$$\mathcal{C}_{opt}^{i_0}(c) \leq \min V(c)$$
$$\mathcal{C}_{pes}^{i_0}(c) \geq \max V(c)$$

D'après les définitions 53 et 54, on en déduit que l'algorithme 13 retourne bien une paire de fonctions de coût optimiste et pessimiste. □

5.1.2 Algorithme de planification

Avant d'approfondir le calcul des fonctions d'estimation à partir des coûts optimistes et pessimistes, intéressons nous à l'algorithme qui les utilise. Cet algorithme de planification est très fortement inspiré d'Anytime-A* [33], un algorithme qui recherche les solutions en suivant le principe d'A*, mais en utilisant une fonction heuristique non admissible. L'optimalité de la première solution atteinte n'est donc pas garantie. Cependant, la recherche n'est pas interrompue, et le coût de cette solution définit alors une borne supérieure permettant d'élaguer certaines solutions partielles dans l'espace de recherche restant, selon le principe des algorithmes de recherche par séparation et évaluation. À chaque fois qu'une nouvelle solution est atteinte, son coût est utilisé pour mettre à jour la valeur de la borne supérieure, jusqu'à obtention d'une solution optimale. Ainsi, cet algorithme retourne une suite de solutions de coûts décroissants, convergeant vers une solution optimale.

Toutefois dans notre cas, les nœuds de l'espace de recherche correspondent à des réseaux de tâches et la fonction d'estimation est constituée à partir du coût primitif et d'une heuristique fondée sur les coûts optimistes ou pessimistes. Or pour A*, la fonction d'estimation est fondée sur les coûts liés aux transitions dans l'espace de recherche (coût des transitions entre le nœud initial et le nœud courant, et estimation du coût pour atteindre un nœud terminal), ce qui n'est donc pas le cas ici. En conséquence, notre algorithme ne correspond pas exactement à la définition de A*. Il est donc affilié à la classe plus générale des algorithmes de type BFS, et est en conséquence désigné sous le nom d'Anytime-BFS-HTN. En dehors de cette différence mineure, on retrouve bien le fonctionnement d'Anytime-A*, décrit plus haut, pour Anytime-BFS-HTN, comme l'atteste la description de cette procédure par l'algorithme 14.

Anytime-BFS-HTN parcourt l'espace des décompositions des réseaux de tâches, et est donc un algorithme de type DHTN. Trois leviers sont disponibles pour contrôler la recherche de solutions. Le premier est la stratégie de décomposition des tâches, qui se manifeste à la ligne 18. L'implémentation de cette stratégie est laissée totalement libre dans la description d'Anytime-BFS-HTN. Toutefois, celle-ci peut être combinée avec le système des effets abstraits décrit au chapitre précédent, et donc utiliser par exemple des stratégies de décomposition des tâches fondées sur des horizons. Le second levier de contrôle est la fonction d'évaluation liée à l'étape de séparation et évaluation. Celle-ci est notée f_1, et se calcule à partir de la somme du coût primitif d'un réseau de tâches (g) est d'une heuristique h_1 « admissible », c'est-à-dire qui doit sous-estimer le coût des solutions accessibles à partir d'un réseau de tâches donné. Le dernier levier est la fonction d'évaluation f_2, utilisée pour la recherche de type BFS. f_2 est calculée comme f_1, à la différence que sa composante heuristique h_2 ne doit pas nécessairement être admissible (il est même recommandé qu'elle ne le soit pas, sans quoi la première solution retournée est déjà optimale et l'étape de séparation et évaluation devient inutile).

Enfin, cet algorithme calcule aussi une sur-estimation de l'erreur entre le coût de la dernière solution retournée et celui de la solution optimale. Le coût de la solution optimale ne pouvant être connu avant que l'espace de recherche ne soit entièrement exploré, cette estimation se base sur la plus petite valeur de f_1 dans l'espace de recherche restant à explorer. Comme cette grandeur minimise le coût de la solution optimale, on obtient un taux d'erreur plus important que celui qu'on obtiendrait si le coût de la solution optimale était connu. L'erreur s'exprime en pourcentage : ainsi une erreur de 0 % indique que la solution optimale a été découverte, tandis qu'une erreur de 100 % indique que la solution actuelle est au plus deux fois plus coûteuse que la solution optimale. Ainsi, plus Anytime-BFS-HTN progresse dans la recherche d'une solution optimale, et plus la valeur de cette erreur progresse vers 0.

En ce qui concerne les propriétés de terminaison d'Anytime-BFS-HTN, on retrouve

Algorithm 14 Anytime-BFS-HTN

entrées : Un problème de planification $P = (D, s_0, \rho_0)$.
sorties : Une solution de P.

1: $solution \leftarrow$ *échec*
2: $borne \leftarrow +\infty$
3: $f_1(\rho_0) \leftarrow g(\rho_0) + h_1(\rho_0)$
4: $f_2(\rho_0) \leftarrow g(\rho_0) + h_2(\rho_0)$
5: $Ouvert \leftarrow \{\rho_0\}$
6: $Fermé \leftarrow \emptyset$
7: **tant que** $Ouvert \neq \emptyset$ **et non** *interrompu* **faire**
8: $\quad \rho \leftarrow$ choisir un élément dans $Ouvert$ minimisant f_2
9: $\quad Ouvert \leftarrow Ouvert \setminus \{\rho\}$
10: $\quad Fermé \leftarrow Fermé \cup \{\rho\}$
11: $\quad$ **si** ρ est primitif **alors**
12: $\quad\quad$ **si** ρ est exécutable en s_0 **alors**
13: $\quad\quad\quad \sigma \leftarrow$ une linéarisation de ρ exécutable en s_0
14: $\quad\quad\quad solution \leftarrow \sigma$
15: $\quad\quad\quad borne \leftarrow \mathcal{C}(\sigma)$
16: $\quad\quad$ **fin si**
17: $\quad$ **sinon**
18: $\quad\quad n \leftarrow$ choisir un élément dans N_ρ
19: $\quad\quad$ **pour** toute méthode $m \in M$ telle que $c_m = t_n$ **faire**
20: $\quad\quad\quad \rho' \leftarrow \mathcal{R}(\rho, n, m)$
21: $\quad\quad\quad f_1(\rho') \leftarrow g(\rho') + h_1(\rho')$
22: $\quad\quad\quad f_2(\rho') \leftarrow g(\rho') + h_2(\rho')$
23: $\quad\quad\quad$ **si** $f_1(\rho') < borne$ **et** $\rho' \notin Fermé$ **alors**
24: $\quad\quad\quad\quad Ouvert \leftarrow Ouvert \cup \{\rho'\}$
25: $\quad\quad\quad$ **fin si**
26: $\quad\quad$ **fin pour**
27: $\quad$ **fin si**
28: **fin tant que**
29: **si** $Ouvert = \emptyset$ **alors**
30: $\quad erreur \leftarrow 0$
31: **sinon**
32: $\quad f_e \leftarrow \min\{f_1(\rho) \mid \rho \in Ouvert\}$
33: $\quad erreur \leftarrow 100 \times (borne - f_e)/f_e$
34: **fin si**
35: **retourner** *solution*

les mêmes conditions que pour n'importe quel algorithme de type DHTN. L'espace de décomposition du réseau de tâches initial ne peut donc être fini qu'à la condition que le problème de planification HTN posé soit « $\leq_1$-stratifiable » [3].

Théorème 11 (Terminaison d'Anytime-BFS-HTN)**.** *Soit un problème de planification HTN $P = (D, s_0, \rho_0)$.*
Si $\{\rho \in R_{A \cup C} \mid \mathcal{D}^*(\rho_0, \rho)\}$ est fini, alors la résolution de P par Anytime-BFS-HTN termine.

Démonstration. Si $\{\rho \in R_{A \cup C} \mid \mathcal{D}^*(\rho_0, \rho)\}$ est fini, alors Anytime-BFS-HTN ne peut visiter qu'un nombre fini de réseaux de tâches.
Comme l'ensemble $Fermé$ permet de s'assurer que chaque réseau de tâches n'est visité

qu'une seule fois, on en déduit qu'Anytime-BFS-HTN visite un nombre fini d'éléments, et donc qu'il termine. □

La correction d'Anytime-BFS-HTN est garantie en toute circonstance, étant donné que l'algorithme ne retourne que des solutions exécutables obtenues par réductions successives du réseau de tâches initial.

Théorème 12 (Correction d'Anytime-BFS-HTN)**.** *Soit un problème de planification HTN $P = (D, s_0, \rho_0)$. Toute séquence de tâches retournée par Anytime-BFS-HTN lors de la résolution de P est une solution de P.*

Démonstration. Soit σ une séquence de tâches retournée par Anytime-BFS-HTN.
D'après l'algorithme 14, toute solution retournée par Anytime-BFS-HTN est une séquence de tâches primitives vérifiant $\mathcal{D}^*(\rho_0, \varrho(\sigma))$. De plus, et toujours d'après l'algorithme 14, σ est exécutable en s_0.
D'après la définition 21, on en déduit donc que σ est une solution de P.
Anytime-BFS-HTN est donc bien un algorithme correct. □

En ce qui concerne la complétude, celle-ci aussi est garantie, dans la mesure où le système d'évaluation et séparation ne peut pas élaguer de solutions partielles tant qu'au moins une solution n'a pas été retournée.

Théorème 13 (Complétude d'Anytime-BFS-HTN)**.** *Soit un problème de planification HTN $P = (D, s_0, \rho_0)$.*
Si P est résoluble et que Anytime-BFS-HTN termine, alors Anytime-BFS-HTN retourne au moins une solution en résolvant P.

Démonstration. Supposons que P admette au moins une solution σ, linéarisation d'un réseau de tâches exploré ρ.
Comme l'espace de recherche est fini, si aucune solution n'est retournée par Anytime-BFS-HTN, on en déduit alors que l'un des parent ρ' de ρ a été élagué, ce qui ne peut arriver que si $f_1(\rho') \geq borne$. Or, si aucune solution n'est retournée, on a $borne = +\infty$, et donc on a nécessairement $f_1(\rho') < borne$, ce qui implique une contradiction.
On en déduit donc qu'Anytime-BFS-HTN retourne une solution, et donc qu'il s'agit d'un algorithme complet. □

Enfin, on peut montrer qu'Anytime-BFS-HTN est bien optimal, dans le sens où il finit bien par retourner une solution optimale, si la fonction heuristique h_1 est admissible. En effet si tel est bien le cas, aucune solution partielle menant à une solution optimale ne peut être élaguée par l'étape de séparation et évaluation avant que la solution optimale n'ait été obtenue.

Théorème 14 (Optimalité d'Anytime-BFS-HTN)**.** *Soit un problème de planification HTN $P = (D, s_0, \rho_0)$.*
Si P est résoluble, qu'Anytime-BFS-HTN termine et que h_1 est admissible, alors la dernière solution retournée est optimale.

Démonstration. Soient σ^* une solution optimale de P et n le nombre de solutions retournées par Anytime-BFS-HTN.
On note alors $c_1, ..., c_n$ la suite des coûts des solutions retournées.
Supposons que la dernière solution ne soit pas optimale.
D'après l'algorithme 14, on a alors :

$$c_1 > ... > c_n > \mathcal{C}(\sigma^*) \tag{5.3}$$

Comme σ^* n'a pas été retournée, alors il existe un réseau de tâche ρ tel que $\mathcal{D}^*(\rho, \varrho(\sigma))$ et qui a été élagué par Anytime-BFS-HTN.
Pour que ρ soit élagué, il doit alors existé un $i \in [\![1, n]\!]$ tel que $f_1(\rho) \geq c_i$.
Or, d'après 5.3, on peut alors en déduire que $f_1(\rho) > \mathcal{C}(\sigma^*)$, ce qui est en contradiction avec l'admissibilité de h_1.
On en déduit donc que la dernière solution retournée par Anytime-BFS-HTN est optimale.
□

5.1.3 Heuristiques

Jusqu'ici, nous avons introduit les fonctions de coûts optimistes et pessimistes sur lesquelles repose la définition des fonctions d'estimations, et nous avons décrit l'usage de ces fonctions d'estimation dans l'algorithme de planification Anytime-BFS-HTN. Il nous reste donc encore à présenter les différentes méthodes suivant lesquelles ces fonctions d'estimation sont dérivées des coûts optimistes et pessimistes, et de quelles façons ces méthodes influences la recherche de solutions pour Anytime-BFS-HTN.

Nous allons commencer par traiter le cas de la fonction d'évaluation f_1, qui sert à évaluer une borne inférieure pour le coût des solutions atteignables lors de l'étape de séparation et évaluation. Cette fonction d'évaluation doit sous-estimer le coût des séquences d'actions accessibles en réduisant le réseau de tâches considéré. En conséquence, une implémentation satisfaisante de la fonction heuristique h_1 consiste à effectuer la somme des coûts optimistes pour chacune des tâches composées de ce réseau de tâches. Ainsi pour tout réseau de tâches ρ, on obtient :

$$h_1(\rho) = \sum_{n \in N_\rho | t_n \in C} \mathcal{C}_{opt}(t_n)$$

Si la même définition est utilisée pour h_2, la recherche de solution de la composante « BFS » d'Anytime-BFS-HTN est alors admissible, et la première solution retournée est optimale :

Théorème 15. *Soit $\mathcal{C}_{opt}$ une fonction de coût optimiste. Si pour tout réseau de tâches ρ, $h_2(\rho) = \sum_{n \in N_\rho | t_n \in C} \mathcal{C}_{opt}(t_n)$, alors la première solution retournée par Anytime-BFS-HTN est optimale.*

Démonstration. Soient un problème de planification HTN $P = (D, \rho_0, s_0)$ et σ la première solution retournée par Anytime-BFS-HTN lors de la résolution de σ.
On note aussi ρ le réseau de tâches exploré par Anytime-BFS-HTN, et dont σ est une linéarisation (on a alors $f_2(\rho) = \mathcal{C}(\sigma)$).
D'après l'algorithme 14, on a alors :

$$f_2(\rho) = \min f_2(Ouvert)$$

Supposons que σ ne soit pas une solution optimale.
Dans ce cas, il existe $\rho' \in Ouvert$, tel que $f_2(\rho') >= f_2(\rho)$, et $\sigma' \in A^*$ tel que $\mathcal{D}^*(\rho', \varrho(\sigma'))$ et $\mathcal{C}(\sigma) > \mathcal{C}(\sigma')$.
Comme $h_2(\rho') = \sum_{n \in N_{\rho'} | t_n \in C} \mathcal{C}_{opt}(t_n)$ et que $\mathcal{C}_{opt}$ est une fonction de coût optimiste, on peut en déduire que :

$$f_2(\rho') \leq \mathcal{C}(\sigma') \tag{5.4}$$

On en déduit donc que $f_2(\rho') < \mathcal{C}(\sigma)$, et donc que $f_2(\rho') < f_2(\rho)$, ce qui est en contradiction avec $f_2(\rho) = \min f_2(Ouvert)$).
La première solution retournée par Anytime-BFS-HTN est donc optimale. □

Le planificateur obtenu n'est donc pas « anytime », puisqu'il faut attendre que la solution optimale soit découverte, sans pouvoir bénéficier de solutions sous-optimales obtenues dans un délai plus court. On a donc intérêt à rendre cette heuristique non admissible. Un procédé classique pour rendre une heuristique non admissible avec A* consiste à la pondérer par un facteur multiplicatif $\omega > 1$ [86]. Pour tout réseau de tâches ρ, on obtient alors :

$$h_2(\rho) = \sum_{n \in N_\rho | t_n \in C} \omega \times \mathcal{C}_{opt}(t_n)$$

En provoquant une surestimation du coût des réductions d'un réseau de tâches, cette pondération à normalement pour effet de favoriser l'exploration en profondeur de l'espace de recherche, ce qui permet d'atteindre des solutions plus rapidement. Toutefois, en fonction de la stratégie de décomposition des tâches implémentée dans Anytime-BFS-HTN, l'impact de cette pondération peut être très limité. En effet, si la stratégie utilisée donne la priorité aux tâches les plus élevées dans la hiérarchie du domaine HTN, alors les tâches primitives auront tendance à être introduites à une grande profondeur dans l'arbre de recherche. Le coût primitif demeure alors nul sur une partie importante de la recherche, et la pondération, qui s'applique uniformément sur chaque coût optimiste, ne modifie donc pas l'ordre d'exploration des réseaux de tâches. La recherche est alors identique à une recherche admissible tant qu'une profondeur suffisante n'a pas été atteinte.

Pour contrer ce phénomène, une solution consiste à utiliser un principe similaire au poids dynamique [87], où le poids devient une fonction décroissante de la profondeur tendant vers 1. En planification HTN, on peut atteindre un résultat similaire en utilisant un poids différent ω_c pour chaque tâche différente $c \in C$, et en attribuant à une tâche un poids d'autant plus élevé que celle-ci ce situe à un niveau élevé dans la hiérarchie. Ainsi, chaque décomposition d'une tâche composée entraine une diminution de la pondération, et permet à nouveau de favoriser l'exploration de l'espace de recherche en profondeur. Pour tout réseau de tâches ρ, on considère alors :

$$h_2(\rho) = \sum_{n \in N_\rho | t_n \in C} \omega_{t_n} \times \mathcal{C}_{opt}(t_n)$$

Mais plutôt que de pondérer les coûts optimistes, une méthode plus directe pour obtenir une surestimation du coût des réductions d'un réseau de tâches consiste à considérer les coûts pessimistes. Ainsi, pour tout réseau de tâches ρ, on considère désormais :

$$h_2(\rho) = \sum_{n \in N_\rho | t_n \in C} \mathcal{C}_{pes}(t_n)$$

On peut remarquer que si tous les coûts pessimistes surestiment strictement toutes les décompositions des tâches auxquelles ils correspondent, alors le meilleur élément, selon f_2, est toujours obtenu parmi les successeurs du réseau de tâches courant. La recherche devient alors une recherche en profondeur.

Définition 55 (Fonction de coût pessimiste strictement monotone)**.** *Une fonction de coût pessimiste $\mathcal{C}_{pes}$ est dite strictement monotone si et seulement si pour tout $c \in C$ et tout réseau de tâches $\rho \in R_{A \cup C}$ tel que $\mathcal{D}(\varrho(t), \rho)$, on a :*

$$\mathcal{C}_{pes}(c) > \sum_{n \in N_\rho | t_n \in A} \mathcal{C}(t_n) + \sum_{n \in N_\rho | t_n \in C} \mathcal{C}_{pes}(t_n)$$

Théorème 16. *Soit $\mathcal{C}_{pes}$ une fonction de coût pessimiste strictement monotone. Si pour tout réseau de tâches ρ, $h_2(\rho) = \sum_{n \in N_\rho | t_n \in C} \mathcal{C}_{pes}(t_n)$, alors Anytime-BFS-HTN explore l'espace de recherche en profondeur d'abord.*

Démonstration. Anytime-BFS-HTN explore l'espace de recherche en profondeur d'abord si et seulement si le réseau de tâches visité est toujours un de ceux dont la profondeur dans l'espace de recherche est la plus élevée.
Notons $p(\rho)$ la profondeur d'un réseau de tâches dans l'espace de recherche. Cette profondeur est attribuée de la façon suivante :

1. $p(\rho_0) = 0$, uniquement pour le premier réseau de tâches exploré.
2. À chaque étape, si ρ et le réseau de tâches exploré et que ρ' est un de ses successeurs, on a $p(\rho') = p(\rho) + 1$.

Pour que le parcourt de l'espace de recherche se fasse en profondeur d'abord, il faut donc qu'a chaque étape, on ait :

$$\forall \rho \in Ouvert : f_2(\rho) = \min f_2(Ouvert) \implies p(\rho) = \max\{p(Ouvert)\}$$

On note i le nombre d'itérations exécutées par Anytime-BFS-HTN, et $Ouvert_i$ la valeur de l'ensemble $Ouvert$ à la i-ième itération.
Montrons par récurrence que la propriété suivante, notée $P(i)$, est vraie pour tout $i \in \mathbb{N}$:

$$\forall (\rho_1, \rho_2) \in Ouvert_i^2 : f_2(\rho_1) \leq f_2(\rho_2) \implies p(\rho_1) \geq p(\rho_2)$$

Démontrons cette propriété par récurrence sur le nombre d'étapes exécutées.

1. $P(0)$:
   Si $i = 0$, on a alors $Ouvert_0 = \{\rho_0\}$.
   Pour tout $(\rho_1, \rho_2) \in Ouvert_i^2$, on a alors $\rho_1 = \rho_0$ et $\rho_2 = \rho_0$.
   On en déduit donc que $f_2(\rho_1) \leq f_2(\rho_2)$ et que $p(\rho_1) \leq p(\rho_2)$.
   $P(0)$ est donc vraie.
2. $\forall i \in \mathbb{N} : (P(i) \implies P(i+1))$:
   Soit $i \in \mathbb{N}$. On suppose que $P(i)$ est vraie.
   Soit $\rho \in Ouvert_i$ le réseau de tâches exploré à la i-ième itération.
   On note n le nœud de ρ choisie par la stratégie de décomposition, et on pose $Succ(\rho) = \{\mathcal{R}(\rho, n, m) \mid m \in M \wedge t_n = c_m\}$.

   (a) Si $Succ(\rho) = \emptyset$:
   On a alors $Ouvert_{i+1} = Ouvert_i \setminus \{\rho\}$.
   Soit $(\rho_1, \rho_2) \in Ouvert_{i+1}^2$ telle que $f_2(\rho_1) \leq f_2(\rho_2)$.
   On a alors $(\rho_1, \rho_2) \in Ouvert_i^2$, et donc d'après $P(i)$, on en déduit que $p(\rho_1) \geq p(\rho_2)$.
   $P(i+1)$ est donc vraie.

   (b) Sinon :
   On a alors $Ouvert_{i+1} = (Ouvert_i \setminus \{\rho\}) \cup Succ(\rho)$.
   Soit $(\rho_1, \rho_2) \in Ouvert_{i+1}^2$ telle que $f_2(\rho_1) \leq f_2(\rho_2)$.

      i. Si $(\rho_1, \rho_2) \in Ouvert_i^2 \setminus \{\rho\}$:
      D'après $P(i)$, on en déduit que $p(\rho_1) \geq p(\rho_2)$.

      ii. Sinon si $\rho_1 \in Ouvert_i$ et $\rho_2 \in Succ(\rho)$:
      Soit $m \in M$ telle que $\rho_2 = \mathcal{R}(\rho, n, m)$.
      Comme $\mathcal{C}_{pes}$ est strictement monotone, et on en déduit d'après la définition 55,

que :

$$\mathcal{C}_{pes}(t_n) > \sum_{n' \in N_{\rho_m}} \mathcal{C}(t_{n'}) \iff$$

$$g(\rho) + \sum_{n' \in N_\rho | t_{n'} \in C} \mathcal{C}_{pes}(t_{n'}) > g(\rho_2) + \sum_{n' \in (N_{\rho_2} | t_{n'} \in C} \mathcal{C}_{pes}(t_{n'}) \iff$$

$$f_2(\rho) > f_2(\rho_2)$$

ρ étant le réseau de tâches visité à la i-ième itération, on a aussi $f_2(\rho) = \min f_2(Ouvert_i)$.
On en déduit donc que $f_2(\rho) \leq f_2(\rho_1)$, et donc que $f_2(\rho_1) > f_2(\rho_2)$, ce qui contredit l'hypothèse que $f_2(\rho_1) \leq f_2(\rho_2)$.
Ce cas n'est donc pas possible.

iii. Sinon si $(\rho_1, \rho_2) \in Succ(\rho)$:
D'après la définition de p, on a alors $p(\rho_1) = p(\rho) + 1$ et $p(\rho_2) = p(\rho) + 1$, donc on a bien $p(\rho_1) \geq p(\rho_2)$.

iv. Sinon :
On a alors $\rho_1 \in Succ(\rho)$ et $\rho_2 \in Ouvert_i$.
D'après $P(i)$, on a $p(\rho) = \max p(Ouvert_i)$, donc $p(\rho) \geq p(\rho_2)$.
Or, $p(\rho_1) = p(\rho) + 1$. On en déduit donc que $p(\rho_1) \geq p(\rho_2) + 1$, et donc que $p(\rho_1) \geq p(\rho_2)$.

Ainsi, on en déduit par récurrence que pour tout $i \in \mathbb{N}$, $P(i)$ est vraie, ce qui nous permet d'en déduire que pour tout $i \in \mathbb{N}$, on a :

$$\forall \rho \in Ouvert_i : f_2(\rho) = \min f_2(Ouvert_i) \implies p(\rho) = \max p(Ouvert_i)$$

On en déduit donc qu'Anytime-BFS-HTN explore l'espace de recherche en profondeur d'abord. □

Ce dernier constat nous incite à proposer un dernier type de formulation pour h_2, sous la forme d'une combinaison linaire entre les coûts optimistes, et des coûts pessimistes strictement monotones. On utilise pour cela un poids linaire $\lambda_c \in [0, 1]$ pour chaque tâche $c \in C$. Ainsi il devient possible, à l'échelle de chaque tâche, de favoriser une recherche soit admissible (par exemple pour les tâches avec un nombre réduit de décompositions), soit en profondeur (pour les tâches dont la décomposition engendre de nombreux successeurs). Pour chaque réseau de tâches ρ, on peut donc désormais formuler h_2 de la façon suivante :

$$h_2(\rho) = \sum_{n \in N_\rho | t_n \in C} (1 - \lambda_{tn}) \times \mathcal{C}_{opt}(t_n) + \lambda_{tn} \times \mathcal{C}_{pes}(t_n)$$

5.2 Expériences

Cette partie décrit une suite d'expériences dont l'objectif est d'évaluer les performances du système de planification par rapport à différentes heuristiques et différentes stratégies de décomposition des tâches. Trois domaines extraits de la littérature en planification d'actions ont été utilisés : *Robotbox*Robotbox, *Logistics* et *SimpleFPS*.

5.2.1 Robotbox

Description du domaine

La première expérience est effectuée sur le domaine de planification *Robotbox*Robotbox. Ce domaine de planification est apparu dans le contexte de la planification par hiérarchie

d'abstractions [6], et modélise une situation où un robot déplace des boîtes à travers un ensemble de pièces. Un exemple de situation est illustré par la figure 5.4. L'environnement est composé d'un graphe de pièces, reliées entre elles par des portes, et plusieurs boîtes sont disposées dans différentes pièces.

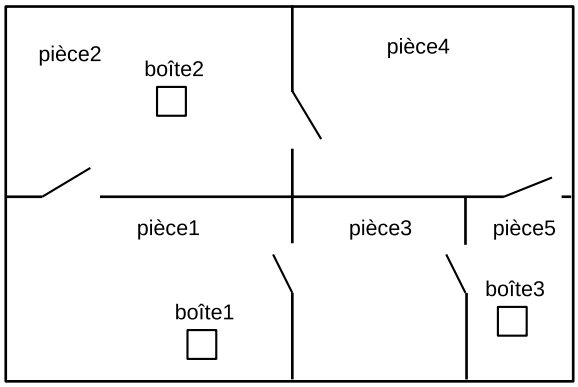


FIGURE 5.4 – Exemple de situation pour le domaine *Robotbox*.

L'objectif est de déplacer chaque boîte depuis une salle de départ vers une salle d'arrivée. Les déplacements ne peuvent s'effectuer qu'entre deux pièces reliées par une porte ouverte, et sont de deux natures. Un premier déplacement consiste à tirer une caisse avec le robot (modélisé par une action `PullThruDoor`), et nécessite que la caisse soit au préalable attachée au robot (via l'action `AttachBox`). Le second déplacement s'effectue en portant la caisse (action `CarryThruDoor`), et nécessite que la caisse soit chargée dans le robot (avec l'action `LoadBox`). Le franchissement d'une porte n'étant possible que si celle-ci est ouverte, une action `OpenDoor` permet en plus d'ouvrir les portes fermées, à condition qu'elles ne soient pas verrouillées. Le robot lui-même n'est pas représenté dans le domaine, et ses déplacements à l'intérieur de l'environnement ne sont donc pas planifiés.

L'intérêt de ce domaine réside dans l'existence des deux types de déplacement, et de la possibilité offerte d'introduire de l'abstraction dans la modélisation du domaine. Il est en effet possible de modéliser une tâche `MoveThruDoor` faisant office d'action abstraite, associée à deux méthodes de décomposition permettant de la concrétiser soit par `PullThruDoor`, soit par `CarryThruDoor`. Il devient alors possible de résoudre chaque problème de navigation (pour déplacer une boîte entre deux pièces) sans avoir à décomposer la tâche `MoveThruDoor`. En attribuant des coûts égaux à 1 aux deux actions `PullThruDoor` et `CarryThruDoor`, on peut obtenir une estimation optimiste précise du coût final d'un plan non primitif composé de tâches `MoveThruDoor`. Pour faire un parallèle avec la planification classique, cela revient à résoudre un problème abstrait dont le facteur de branchement est divisé par deux par rapport au problème initial, puis de se servir du coût de la solution abstraite pour obtenir une heuristique.

Une description en HTDL du domaine est disponible en annexe D.1. Les tâches composés y sont décrites avec les effets abstraits présentés dans la partie 4.3 du chapitre précédent. Le tableau 5.1 fournit aussi un résumé des différentes tâches, avec leurs coûts optimistes et pessimistes, ainsi que les niveaux qui leur sont attribués afin d'implémenter la stratégie de décomposition des tâches. On peut aussi voir sur ce tableau que ce domaine contient une tâche récursive `AchieveAt` dont la profondeur de récursion a été limité à 5, dans la mesure où les instances générées pour les expériences contiennent un nombre de pièces inférieur ou égal à 6. Enfin, il est à noter que les coûts pessimistes sont strictement monotones. Pour cela, chaque coût pessimiste a été calculé en ajoutant une valeur $\epsilon = 0.001$ à la somme des coûts pessimistes maximale parmi les décompositions de la tâche correspondante.

Compte tenu de la distribution des coûts au niveau des tâches primitives, un réseau

tâche primitive	coût
CarryThruDoor(b, d, r1, r2)	1
PullThruDoor(b, d, r1, r2)	1
LoadBox(b)	1
AttachBox(b)	1
OpenDoor(d)	1

tâche composée	niveau	$\mathcal{C}_{ops}$	$\mathcal{C}_{pes}$	λ_c
AchieveLoaded(b)	1	0	1.001	1
AchieveAttached(b)	1	0	1.001	1
MoveThruDoor(b, d, r1, r2)	2	1	2.002	1
AchieveOpen(d)	3	0	1.001	1
AchieveAt(b, r, k)	4	0	$3.004 \times (6 - k)$	0.5

TABLE 5.1 – Résumé des différentes tâches pour le domaine *Robotbox*.

de tâches pour lequel toutes les tâches de type `AchieveAt` et `AchieveOpen` ont été décomposées permet d'obtenir un coût optimiste qui reflète très fidèlement le coût final de ses réductions. En effet, la seule différence entre ce coût optimiste et le coût de n'importe quel plan issu de ce réseau de tâches est égale au coûts des actions servant à charger ou attacher les boîtes avant de les déplacer. Or pour une solution optimale, il n'est nécessaire d'exécuter une telle action qu'une seule fois par boîte. En conséquence, cette différence est égale au nombre de boîtes, et est donc fixe pour un problème donné. Ainsi, à cet état de décomposition, le réseau de tâche avec le meilleur coût optimiste est aussi le réseau de tâches dont les décompositions ont le meilleur coût. De plus, et toujours à cet état de décomposition, l'existence d'une solution est aussi garantie, puisque le problème de navigation est résolu et que l'incertitude sur l'existence d'un itinéraire (à cause des portes verrouillées) est levée. En conséquence, on fait face à un cas où l'obtention d'un réseau de tâches de coût optimiste minimal permet d'ignorer les autres éléments de l'espace de recherche, et de focaliser la recherche sur ses successeurs.

À partir de ce raisonnement, on peut supposer que la meilleure configuration pour un planificateur consiste à utiliser une stratégie de décomposition donnant priorité aux tâches ayant les plus hauts niveaux, afin de ne pas décomposer les tâches `MoveThruDoor` trop tôt, et d'utiliser une heuristique qui aura pour effet de ne pas trop altérer l'ordre d'exploration des réseaux de tâches à un haut niveau, afin d'obtenir un élément avec un coût optimiste garantissant approximativement un coût optimal parmi ses décompositions, puis à explorer l'espace de recherche davantage en profondeur pour concrétiser les tâches non primitives restantes. Ainsi la première solution retournée, sans être optimale, nécessite un temps de calcul réduit tout en étant déjà de bonne qualité.

Expérience

Le système est évalué à partir d'une implémentation en Python d'un planificateur combinant la procédure Anytime-BFS-HTN avec la procédure ABDHTN, mais ne maintient pas d'ensemble *Fermé* pour vérifier qu'un réseau de tâches a déjà été exploré (cette vérification n'est pas utile compte tenu de la structure du domaine). Ce planificateur est exécuté sur une machine Linux dont le processeur est cadencé à 2.20 GHz (chaque problème est résolu sur un seul cœur), et qui dispose de 32 Go de mémoire vive. Chaque problème a été résolu avec un temps limite de 180 secondes. Ce temps peut paraître important vis à vis de notre objectif de planification en ligne. Ce choix a été effectué afin de permettre l'observation des différentes configurations sur le long terme, et aussi parce

que le planificateur utilisé est un prototype en Python dont les performances ne sont pas optimales par rapport à une implémentation en C++.

Le générateur d'instance aléatoire utilisé pour ces expériences produit des problèmes où l'environnement est composé de 6 pièces. Chaque pièce contient une moyenne de 3 portes, chaque porte a une probabilité de 66 % de ne pas être verrouillée et chaque porte non verrouillée a une probabilité de 50 % d'être ouverte. Seule le nombre de boîtes à déplacer a été considéré comme élément variable entre les différentes séries d'instances. Afin de ne pas introduire de confusion entre l'impact des effets abstraits et l'impact des fonctions d'estimation sur les performances de la recherche, les instances ont été générée de façon à ce que tous les réseaux de tâches introduits dans l'espace de recherche soient potentiellement exécutables. Pour cela, pour chaque boîte, la pièce initiale ainsi que la pièce à atteindre ont été choisies au sein de la même composante connexe du graphe de pièces (par rapport aux portes non verrouillées). Trois séries d'instances aléatoires ont été générées pour cette expérience, avec un nombre de boîtes à déplacer variant de 1 à 3, et chaque série est composée de 100 instances.

L'utilisation des effets abstraits permet d'implémenter différentes stratégies de décomposition des tâches, ce qui nous permet d'évaluer une stratégie pour laquelle $h = +\infty$, qui favorise la décomposition des tâches aux plus niveaux, et une stratégie avec $h = 1$, qui décompose toujours une des premières tâches non-primitives.

Pour servir de référence, le banc de test contient une implémentation (toujours en Python) d'un planificateur de type PHTN, qui n'utilise pas d'effets abstraits, et qui est donc limité à la stratégie de décomposition qui décompose toujours une des premières tâches non primitives. Une heuristique à pondération uniforme $\omega = 1$ sert aussi de référence pour évaluer l'écart séparant les performances des différentes heuristiques par rapport à une recherche admissible. Un poids uniforme $\omega = 1.5$ a été sélectionné afin de mesurer le gain de performance tout en garantissant que la qualité de la première solution ne soit pas trop mauvaise (au plus 50 % du coût de la solution optimale). Un poids $\omega = 5$ permet ensuite d'évaluer les performances du planificateur avec un poids qui sacrifie la garantie d'obtenir une solution de bonne qualité afin d'obtenir un temps d'exécution plus court. Aucune heuristique avec des poids distribués de type (ω_c) n'a été introduite, dans la mesure où seule la tâche `MoveThruDoor` a un coût optimiste non nul parmi les tâches non primitives. Enfin une heuristique à poids linéaires distribués de type (λ_c) a été modélisée avec pour objectif de maximiser les performances du planificateur en favorisant une recherche presque admissible à haut niveau, avec un poids de 0.5 pour les tâches de types `AchieveAt`, et une exploration en profondeur ensuite, avec des poids de 1 pour toutes les autres tâches. Enfin, une heuristique avec un poids linéaire uniforme $\lambda = 1$ permet d'évaluer les performances d'un algorithme qui explore toujours en profondeur d'abord tout en bénéficiant des deux fonctions d'évaluation.

Résultats

Les résultats sont présentés de façon exhaustive dans la table 5.2. Pour chacune des trois séries, le pourcentage d'instances résolues est indiqué. Pour rappel, chaque instance générée est résoluble, ce qui implique qu'une instance non résolue l'est à cause des performances du planificateur concerné. Ensuite, le tableau indique le nombre de nœuds visités pendant la recherche, le temps CPU obtenu, le coût de la solution et l'erreur à deux instants différents : au moment où la première solution est retournée, et au moment où la meilleure solution est retournée. L'erreur obtenue au bout du temps limite est aussi indiquée, ce qui permet d'évaluer l'optimalité de la meilleure solution retournée.

série	planificateur	% résolus	première solution				meilleure solution				dernière erreur
			nœuds visités	temps CPU	coût	erreur	nœuds visités	temps CPU	coût	erreur	
Robotbox 1 boîte	PHTN	100.0	23	0.047	4.43	inf	74	0.110	1.80	inf	0.00
	$h = 1, \omega = 1$	100.0	37	0.073	1.80	0.00	37	0.073	1.80	0.00	0.00
	$h = +\infty, \omega = 1$	100.0	25	0.084	1.80	0.00	25	0.084	1.80	0.00	0.00
	$h = 1, \omega = 1.5$	100.0	22	0.042	1.80	0.00	22	0.042	1.80	0.00	0.00
	$h = +\infty, \omega = 1.5$	100.0	12	0.035	1.80	6.10	12	0.035	1.80	6.10	0.00
	$h = 1, \omega = 5$	100.0	22	0.041	1.80	5.00	22	0.041	1.80	5.00	0.00
	$h = +\infty, \omega = 5$	100.0	8	0.024	1.80	14.23	8	0.024	1.80	14.23	0.00
	$h = 1, \lambda = 1$	100.0	16	0.034	2.70	197.50	41	0.073	1.80	108.00	0.00
	$h = +\infty, \lambda = 1$	100.0	10	0.031	2.70	162.50	50	0.118	1.80	70.25	0.00
	$h = 1, (\lambda_c)$	100.0	16	0.033	2.28	49.25	32	0.061	1.80	32.50	0.00
	$h = +\infty, (\lambda_c)$	100.0	6	0.020	1.80	46.12	6	0.020	1.80	46.12	0.00
Robotbox 2 boîtes	PHTN	100.0	57	0.092	9.18	inf	598	0.717	3.99	inf	0.00
	$h = 1, \omega = 1$	100.0	386	1.095	3.99	0.00	386	1.095	3.99	0.00	0.00
	$h = +\infty, \omega = 1$	100.0	268	1.002	3.99	0.00	268	1.002	3.99	0.00	0.00
	$h = 1, \omega = 1.5$	100.0	289	0.768	3.99	0.00	289	0.768	3.99	0.00	0.00
	$h = +\infty, \omega = 1.5$	100.0	74	0.271	3.99	20.13	74	0.271	3.99	20.13	0.00
	$h = 1, \omega = 5$	100.0	275	0.787	3.99	7.96	275	0.787	3.99	7.96	0.00
	$h = +\infty, \omega = 5$	100.0	36	0.170	4.00	36.95	38	0.178	3.99	36.48	0.00
	$h = 1, \lambda = 1$	100.0	40	0.115	6.25	532.50	224	0.602	3.99	291.00	0.00
	$h = +\infty, \lambda = 1$	100.0	23	0.101	6.25	465.50	2829	8.030	3.99	208.75	0.00
	$h = 1, (\lambda_c)$	100.0	44	0.132	5.01	203.76	120	0.322	3.99	153.31	0.00
	$h = +\infty, (\lambda_c)$	100.0	15	0.054	4.03	154.73	18	0.065	3.99	150.39	0.00
Robotbox 3 boîtes	PHTN	100.0	66	0.121	12.43	inf	2303	3.047	5.09	inf	0.00
	$h = 1, \omega = 1$	100.0	1340	4.988	5.09	0.00	1340	4.988	5.09	0.00	0.00
	$h = +\infty, \omega = 1$	93.0	1904	8.522	4.65	0.00	1904	8.522	4.65	0.00	0.00
	$h = 1, \omega = 1.5$	100.0	931	3.895	5.09	0.00	931	3.895	5.09	0.00	0.00
	$h = +\infty, \omega = 1.5$	100.0	198	0.996	5.09	22.07	198	0.996	5.09	22.07	0.09
	$h = 1, \omega = 5$	100.0	869	3.677	5.09	7.84	869	3.677	5.09	7.84	0.00
	$h = +\infty, \omega = 5$	100.0	65	0.445	5.09	43.02	65	0.445	5.09	43.02	0.09
	$h = 1, \lambda = 1$	100.0	39	0.149	7.57	608.00	629	2.233	5.09	343.50	0.00
	$h = +\infty, \lambda = 1$	100.0	27	0.120	7.57	567.83	3641	11.712	6.00	405.75	195.50
	$h = 1, (\lambda_c)$	100.0	39	0.150	6.23	275.17	237	0.862	5.09	212.38	0.00
	$h = +\infty, (\lambda_c)$	100.0	16	0.082	5.09	230.10	16	0.082	5.09	230.10	0.09

TABLE 5.2 – Tableau des résultats pour *Robotbox*

Ce tableau est en partie illustré par les figures 5.5 et 5.6. Ces figures apportent une représentation graphique des données en plaçant chaque planificateur sur un graphe coût/temps CPU pour la première solution retournée et pour la meilleure solution retournée, dans le cadre de la dernière série (avec 3 boîtes à déplacer).

En ce qui concerne la première solution retournée, on peut voir que les planificateurs qui utilisent des heuristiques pondérées de type ω retournent des solutions optimales (d'après le tableau), mais avec des temps de calcul élevés, incompatibles avec la planification en ligne. Ce résultat illustre l'inadéquation de ce type d'heuristique en planification HTN, le poids n'ayant pas d'influence tant que la décomposition des tâches composées n'introduit pas de tâche primitive. À noter que le point $h = +\infty, \omega = 1$ n'est pas représentatif du coût optimal, puisque ce planificateur ne résout pas l'intégralité de la série.

On obtient un comportement totalement différent avec l'heuristique $\lambda = 1$ ainsi que pour PHTN, qui apportent les plus mauvaises solutions (en particulier PHTN) mais avec des temps d'exécution inférieurs aux heuristiques pondérés de type ω.

Le meilleur des deux mondes est apporté avec le planificateur $h = +\infty, (\lambda_c)$, qui comme prévu, retourne une première solution de bonne qualité avec un temps d'exécution réduit. Ici, ce planificateur domine tous les autres dans la mesure où la solution retournée est optimale, et que le temps d'exécution est le plus court. Si on s'intéresse aux deux autres séries, le tableau nous indique que cette situation se répète pour la première, et est presque atteinte pour la deuxième, où la différence de coût entre la solution retournée et la solution optimale est négligeable.

Enfin de façon générale, on constate que la stratégie de décomposition $h = +\infty$ apporte les meilleures performances, en dominant presque systématiquement la seconde stratégie de décomposition au niveau de chaque heuristique, à l'exception du cas $h = +\infty, \omega = 1$.

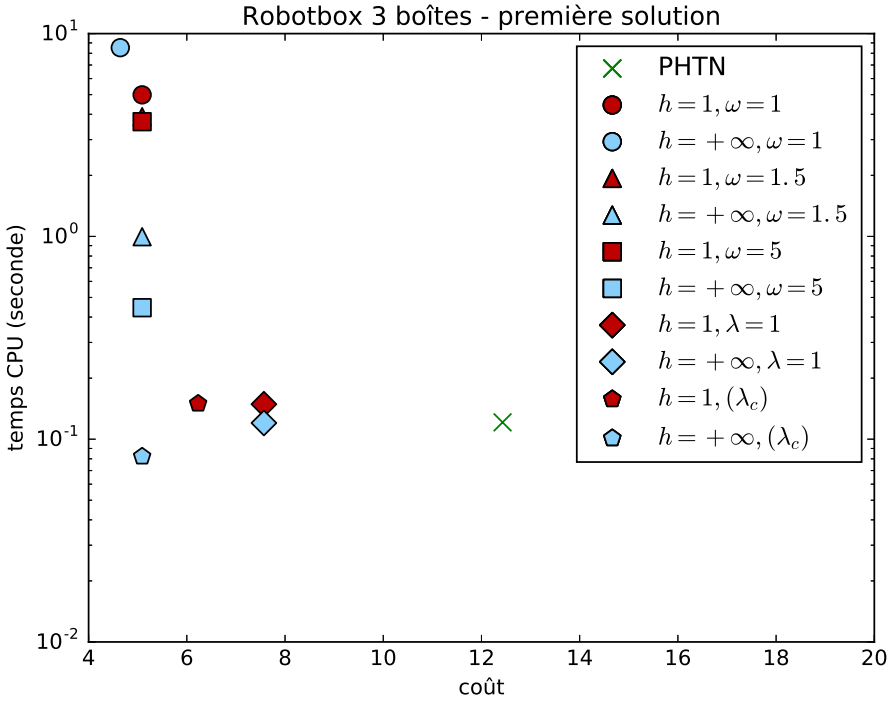


FIGURE 5.5 – Résultat de Robotbox pour la première solution retournée
Représentation des performances de chaque planificateur sur un graphe coût/temps CPU pour la première solution retournée.

En ce qui concerne la meilleure solution retournée dans le temps imparti, on

peut voir que les résultats des planificateurs qui n'ont pas découvert la solution optimale du premier coup sont variés. Les approches les plus performantes correspondent aux heuristiques (λ_c) ainsi qu'au point $\omega = 1.5$ et $\omega = 5$ lorsque la stratégie de décomposition correspond à $h = +\infty$. On remarque aussi que l'heuristique $\lambda = 1$ et PHTN demeurent compétitifs avec les heuristiques $\omega = 1$ et $\omega = 1.5$ pour retourner une solution optimale.

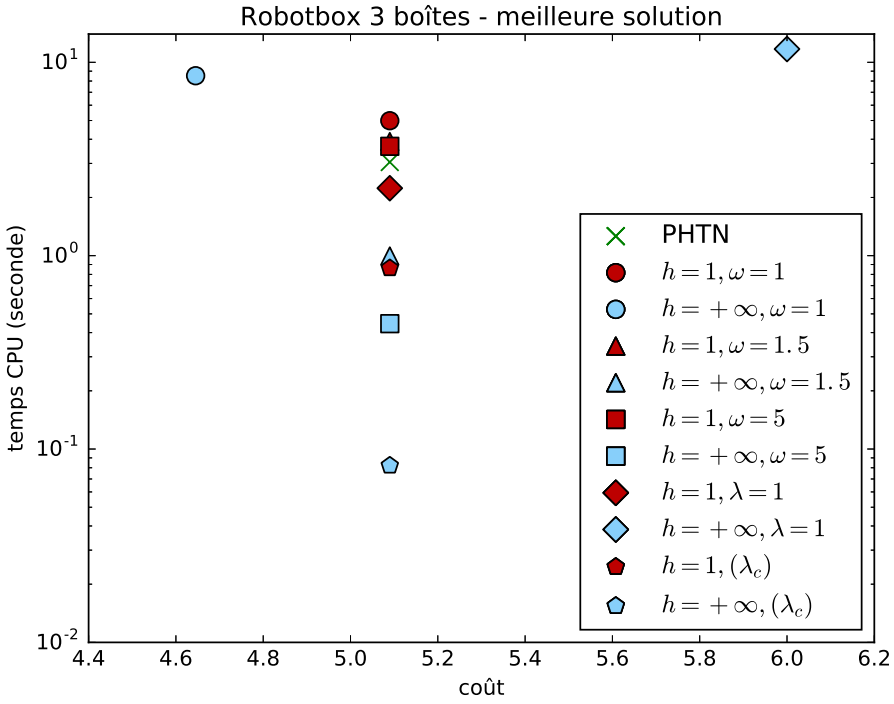


FIGURE 5.6 – Résultat de Robotbox pour la meilleure solution retournée
Représentation des performances de chaque planificateur sur un graphe coût/temps CPU pour la meilleure solution retournée.

Ce premier exemple confirme donc nos pronostiques en ce qui concerne l'utilité de la stratégie de décomposition $h = +\infty$ combinées avec l'heuristique (λ_c), lorsque celle-ci est configurée pour exploiter les niveaux de hiérarchies dans le domaine. L'avantage est flagrant, mais le domaine utilisé est aussi particulièrement simple et favorable. Quand est-il avec un domaine plus complexe ?

5.2.2 Logistics

Description du domaine

Le second domaine est une version HTN de *Logistics*Robotbox, un domaine classique pour les bancs de test PDDL. Ce domaine modélise des problèmes où plusieurs paquets doivent être transportés par camions ou avions vers plusieurs destinations appartenant à différentes villes. Au sein d'une même ville, les destinations sont reliées entre elles par un réseau de routes que les camions peuvent emprunter. De plus, certaines destinations font aussi office d'aéroports, et un réseau de voies aériennes permet ainsi de relier les villes. La figure 5.7 représente un réseau de routes et de voies aériennes reliant un ensemble de destinations pour un exemple de situation de *Logistics*.

La liste des tâches modélisées dans le domaine HTN est fournie par le tableau 5.3. Le coût du déplacement d'un avion a été choisi de façon à dominer le coût des autres actions : ainsi l'action `Fly` est associée à un coût de 10, alors que le coût des autres actions est

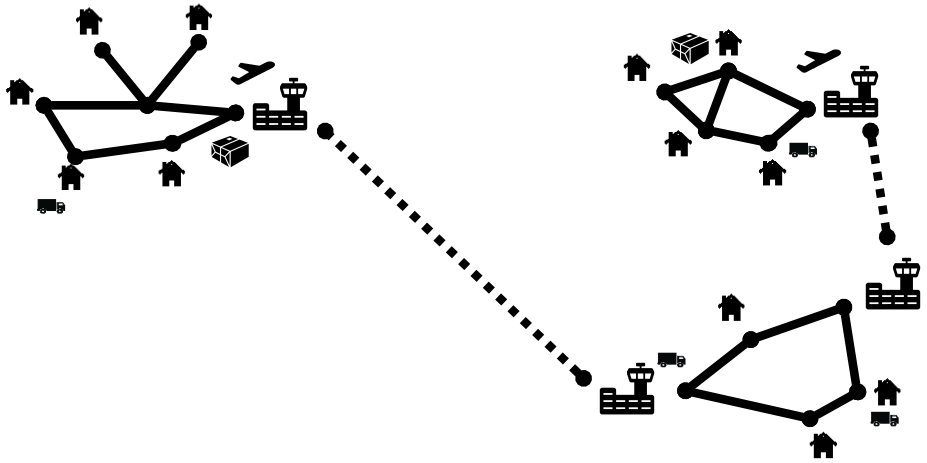


FIGURE 5.7 – Exemple de situation pour le domaine *Logistics*.

inférieur à 2. De cette façon, un plan abstrait qui décrit le déplacement des paquets au niveau des villes, sans entrer dans les détails de leurs déplacements à l'intérieur des villes, permet à nouveau d'obtenir une estimation précise du coût de ses décompositions primitives.

Le domaine contient deux tâches récursives `AchieveTruckAt` et `AchieveAirplaneAtCity` qui permettent respectivement de planifier les déplacements des camions à l'intérieur des villes et les déplacements des avions entre les villes. Afin d'obtenir un espace de décomposition fini, et permettre de définir des coûts pessimistes, leur profondeur d'appel est limitée à 5. En conséquence, les problèmes sont modélisés avec au plus 5 villes et 5 destinations par villes. La description complète du domaine est disponible en annexe D.2.

Expérience

Les conditions expérimentales sont identiques à l'expérience précédente, effectuée sur le domaine *Robotbox*Robotbox, et le lecteur est donc invité à se référer à la partie 5.2.1 pour prendre connaissance des détails vis à vis du système de planification employé et de la configuration matérielle.

Pour ce domaine, les 3 séries de 100 instances qui ont été générées aléatoirement varient par rapport au nombre de paquets à délivrer, compris entre 1 et 3. Chaque instance est générée avec 5 villes, 5 destinations par villes (dont une est sélectionnée au hasard pour être un aéroport), 7 camions (avec au moins un par ville) et 2 avions. Le degré moyen pour chaque réseau de routes, ainsi que pour le réseau de voies aériennes correspond à 1.5. À nouveau, chaque instance est générée de façon à ce que tout problème soit résoluble, et que tout réseau de tâches soit potentiellement exécutable.

Le planificateur est évalué pour les deux stratégies de décomposition $h = 1$ et $h = +\infty$, combinées avec cinq heuristiques différentes (sans compter PHTN et la recherche admissible pour laquelle $\omega = 1$. Parmi ces cinq heuristiques, deux correspondent à des poids uniformes $\omega = 4$ et $\omega = 10$, et le poids linéaire uniforme $\lambda = 1$ est aussi inclus. Cette fois, une heuristique à pondération distribuée (ω_c) a aussi été modélisée, puisque plusieurs tâches composées ayant un coût optimiste non nul sont désormais disponibles. Cette heuristique est modélisée de façon à favoriser l'exploration de l'espace de recherche en profondeur, afin de retourner une première solution rapidement. Enfin, une heuristique à pondération linéaire distribuée (λ_c) est aussi modélisée. Cette heuristique associe des poids intermédiaires (0.5 et 0.3) aux tâches récursives qui planifient la navigation entre les villes et dans les villes, afin de favoriser l'exploration de l'espace de recherche en profondeur tout en limitant la croissance de l'estimation des coûts des réseaux de tâches

tâche primitive	coût
Load(c, p, l, c)	1
Unload(c, p, l, c)	1
Drive(t, l1, l2, c)	2
Fly(a, l1, l2, c1, c2)	10

tâche composée	niveau	$\mathcal{C}_{ops}$	$\mathcal{C}_{pes}$	ω_c	λ_c
AchieveTruckAt(t, l, c, k)	1	0	$2.001 \times (5-k)$	0	0.5
Pickup(t, p, l, c)	2	0	11.006	12	1
Deliver(t, p, l, c)	2	1	11.006	12	1
AchieveInCityAt(p, c, l)	3	0	22.013	0	1
LoadAtCity(a, p, c)	4	1	23.014	24	1
UnloadAtCity(a, p, c)	4	1	1.001	2	1
FlyToCity(a, c1, c2)	4	10	10.001	1.5	1
AchieveAirplaneAtCity(a, c, k)	5	0	$10.002 \times (5-k)$	0	0.3
PickupAtCity(a, p, c)	6	1	74.024	80	0.3
DeliverToCity(a, p, c)	6	1	51.012	80	0.3
AchieveAtCity(p, c)	7	0	125.037	0	0.3
AchieveAt(p, l)	8	0	147.051	0	0.3

TABLE 5.3 – Résumé des différentes tâches pour le domaine *Logistics*.

obtenus. Toutes les autres tâches ont un poids de 1, car leur décomposition n'a qu'une influence limitée sur la qualité des solutions obtenues. Les heuristiques (ω_c) et (λ_c) sont toutes les deux décrites dans le tableau 5.3.

Résultats

À nouveau, les résultats complets sont présentés dans le tableau 5.4, et les figures 5.8 et 5.9 apportent une présentation graphique pour une série. Ici, c'est la seconde série qui est illustrée, la troisième n'étant pas suffisamment fiable à cause des mauvaises performances des planificateurs à heuristiques pondérées, plusieurs d'entre eux ne parvenant pas à y résoudre la moitié des instances.

D'après le tableau, il n'est pas tout à fait possible d'identifier catégoriquement la valeur de la solution optimale. Toutefois, plusieurs planificateurs retournent un plan final avec un coût de 67.24 et ce coût obtient par ailleurs une marge d'erreur inférieure à 0.5 % pour l'un d'entre eux. On peut donc raisonnablement assimiler ce coût à la solution optimale.

série	planificateur	% résolus	première solution				meilleure solution				dernière erreur
			nœuds visités	temps CPU	coût	erreur	nœuds visités	temps CPU	coût	erreur	
Logistics 1 package	PHTN	100.0	42	0.135	56.48	inf	256	0.861	35.14	inf	0.00
	$h = 1, \omega = 1$	100.0	220	1.913	35.14	0.00	220	1.913	35.14	0.00	0.00
	$h = +\infty, \omega = 1$	100.0	127	1.202	35.14	0.00	127	1.202	35.14	0.00	0.00
	$h = 1, \omega = 4$	100.0	171	1.183	35.16	17.74	172	1.184	35.14	15.74	0.00
	$h = +\infty, \omega = 4$	100.0	79	0.528	35.20	22.45	79	0.529	35.14	20.23	0.00
	$h = 1, \omega = 10$	100.0	130	0.881	35.42	50.42	131	0.886	35.14	35.38	0.00
	$h = +\infty, \omega = 10$	100.0	65	0.431	35.66	43.96	75	0.482	35.14	25.41	0.00
	$h = 1, (\omega_c)$	100.0	39	0.262	40.16	1887.92	90	0.539	35.14	1169.05	0.00
	$h = +\infty, (\omega_c)$	100.0	56	0.352	40.16	1886.00	95	0.555	35.14	1098.64	0.00
	$h = 1, \lambda = 1$	100.0	41	0.271	56.76	2715.17	183	0.963	35.14	1173.08	0.00
	$h = +\infty, \lambda = 1$	100.0	38	0.257	56.76	2715.17	173	0.941	35.14	1136.78	0.00
	$h = 1, (\lambda_c)$	100.0	63	0.622	35.52	390.88	67	0.647	35.14	368.18	0.00
	$h = +\infty, (\lambda_c)$	100.0	38	0.371	35.66	158.18	45	0.414	35.14	141.34	0.00
Logistics 2 packages	PHTN	100.0	83	0.325	107.58	inf	3573	15.958	68.32	inf	inf
	$h = 1, \omega = 1$	93.0	2176	22.356	64.84	0.00	2176	22.356	64.84	0.00	0.00
	$h = +\infty, \omega = 1$	99.0	1091	14.628	66.91	0.00	1091	14.628	66.91	0.00	0.00
	$h = 1, \omega = 4$	96.0	2709	23.531	65.75	19.01	2709	23.531	65.75	19.01	0.24
	$h = +\infty, \omega = 4$	99.0	430	5.779	67.01	43.83	435	5.825	66.91	43.18	0.00
	$h = 1, \omega = 10$	96.0	2344	19.372	66.08	32.75	2347	19.385	65.75	31.84	0.09
	$h = +\infty, \omega = 10$	100.0	305	4.347	68.32	53.40	393	5.085	67.24	49.75	0.35
	$h = 1, (\omega_c)$	100.0	1324	8.709	73.36	567.19	1409	9.367	67.24	433.09	2.68
	$h = +\infty, (\omega_c)$	100.0	242	2.907	75.92	2219.00	628	7.014	67.44	1587.58	29.00
	$h = 1, \lambda = 1$	100.0	81	0.744	105.88	4983.83	2348	19.772	68.72	2603.78	363.29
	$h = +\infty, \lambda = 1$	100.0	75	1.070	105.88	5168.50	1568	17.909	67.44	2602.06	59.00
	$h = 1, (\lambda_c)$	100.0	554	4.728	67.62	492.45	558	4.761	67.24	486.98	5.10
	$h = +\infty, (\lambda_c)$	100.0	100	1.696	68.42	406.54	174	2.541	67.24	394.45	0.56
Logistics 3 packages	PHTN	100.0	133	0.745	171.66	inf	13058	68.977	121.52	inf	inf
	$h = 1, \omega = 1$	38.0	4510	53.798	81.21	0.00	4510	53.798	81.21	0.00	0.00
	$h = +\infty, \omega = 1$	58.0	3264	56.432	91.24	0.00	3264	56.432	91.24	0.00	0.00
	$h = 1, \omega = 4$	45.0	5679	45.040	89.38	14.66	5679	45.040	89.38	14.66	0.69
	$h = +\infty, \omega = 4$	96.0	2954	40.944	111.54	48.43	3073	42.514	111.10	47.90	8.89
	$h = 1, \omega = 10$	48.0	6105	47.478	91.25	26.90	6108	47.487	90.75	25.86	1.84
	$h = +\infty, \omega = 10$	98.0	1725	27.153	114.78	58.72	2747	37.647	112.71	48.76	9.61
	$h = 1, (\omega_c)$	60.0	6206	40.227	103.33	160.10	6354	41.186	95.37	112.01	14.14
	$h = +\infty, (\omega_c)$	98.0	1224	15.613	128.65	2523.56	3744	44.806	116.39	2113.94	910.79
	$h = 1, \lambda = 1$	100.0	132	1.255	172.80	8015.83	9070	66.666	122.82	5449.98	3871.89
	$h = +\infty, \lambda = 1$	100.0	120	1.880	172.80	8540.00	6839	76.012	127.64	5907.36	3687.33
	$h = 1, (\lambda_c)$	84.0	2180	23.682	104.00	534.22	2195	23.782	103.50	532.53	115.75
	$h = +\infty, (\lambda_c)$	100.0	324	9.354	115.52	436.16	1133	21.551	113.02	387.96	96.13

TABLE 5.4 – Tableau des résultats pour *Logistics*

En ce qui concerne la première solution retournée, la figure 5.8 nous présente des résultats similaires à l'expérience précédente. Les heuristiques pondérées uniformes (ω) retourne une première solution quasiment optimale, mais exige un temps de calcul important (leur temps de calcul est par ailleurs sous-estimé sur la figure, puisque la majorité des planificateurs correspondant ne parviennent pas à résoudre l'intégralité de la série). PHTN ainsi que les planificateurs avec $\lambda = 1$ apportent encore des résultats opposés, avec ici les meilleurs temps de calcul, mais les plus mauvais coûts (au moins 50 % plus élevés que la solution optimale). Quant à l'heuristique (λ_c), si combinée avec la stratégie de décomposition $h = +\infty$, elle constitue à nouveau le meilleur compromis entre qualité de solution et temps d'exécution. Les performances de l'heuristique (ω_c) ne sont pas aussi favorables, toutefois cette heuristique permet tout de même d'améliorer le temps d'exécution par rapport aux heuristiques pondérées uniformes, en sacrifiant la qualité de la solution dans des proportions qui demeurent acceptables (de l'ordre de 10 %) étant donné qu'on ne recherche pas une solution absolument optimale.

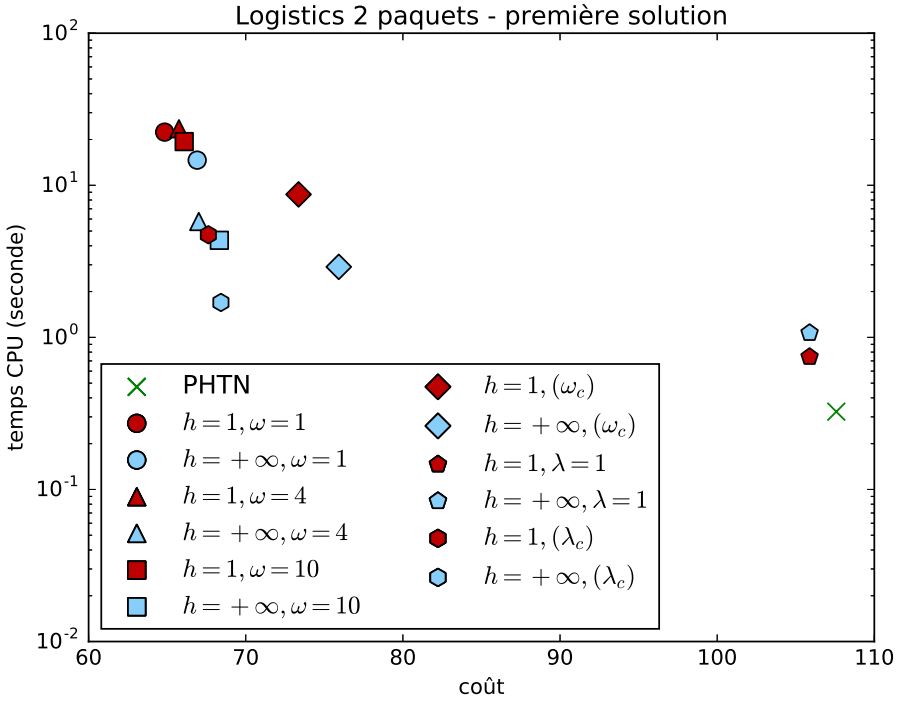


FIGURE 5.8 – Résultat de Logistics pour la première solution retournée
Représentation des performances de chaque planificateur sur un graphe coût/temps CPU pour la meilleure solution retournée.

En ce qui concerne la meilleure solution retournée, tous les planificateurs retournent ici des solutions presque optimales, avec une marge inférieure à 5 %. Le meilleur temps d'exécution est à nouveau assuré par le planificateur avec $h = +\infty, (\lambda_c)$. On remarque aussi que l'heuristique (ω_c) obtient un temps de calcul inférieur à l'heuristique $\omega = 10$ pour la stratégie de décomposition $h = +\infty$, ce qui peut s'expliquer par la propension qu'à la première à sacrifier la précision de la fonction d'estimation pour explorer l'espace de recherche en profondeur. PHTN et le planificateur $h = 1, \lambda = 1$ retournent les moins bonnes solutions, avec des temps de calcul élevés. Enfin, on peut remarquer que pour les trois séries, la stratégie de décomposition $h = +\infty$ domine presque systématiquement la stratégie $h = 1$ au niveau de chaque heuristique.

Ces résultats confirment globalement la tendance obtenue avec l'expérience précédente. Bien que la complexité des problèmes à résoudre a augmenté le temps de calcul moyen (les

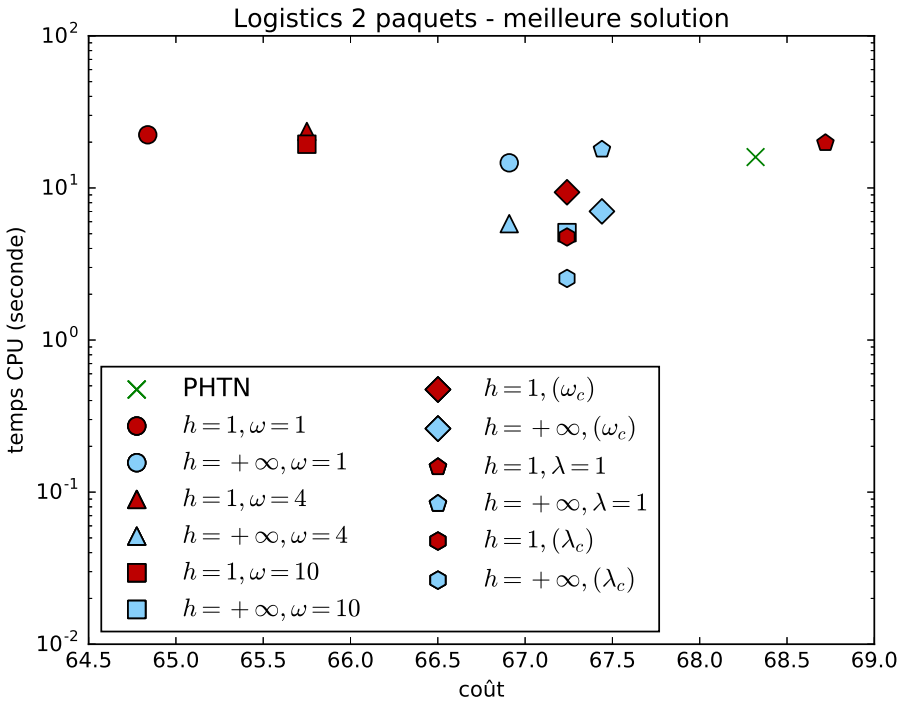


FIGURE 5.9 – Résultat de Logistics pour la meilleure solution retournée
Représentation des performances de chaque planificateur sur un graphe coût/temps CPU pour la meilleure solution retournée.

temps de calcul sont désormais de l'ordre de la seconde), le planificateur avec $h = +\infty, (\lambda_c)$ demeure la meilleur approche afin d'obtenir rapidement une solution de bonne qualité.

5.2.3 SimpleFPS

Description du domaine

Le dernier domaine évalué est *SimpleFPS*, qui a déjà fait l'objet d'une expérience au chapitre 3. Le lecteur est donc invité à consulter la partie 3.3.1 pour prendre connaissance de la description de ce domaine.

La version HTN de *SimpleFPS* utilisée ici présente néanmoins des différences par rapport à celle modélisée au chapitre 3. Tout d'abord, les coûts des tâches primitives sont attribués différemment. Le but ici est de favoriser l'utilisation des actions qui permettent au PNJ d'agir en prenant le moins de risques possible. Pour cela l'action `AttackMelee`, qui modélise une attaque au corps-à-corps et qui est le mode d'action le plus dangereux pour le PNJ, est associée à un coup élevé de 15. Les attaques à distances bénéficient quant à elles de coûts plus faibles, et ces coûts sont encore réduits si l'attaque s'effectue dans l'obscurité ou à partir d'un point de couverture. Ainsi l'action `SneakKillFromCover`, qui modélise une attaque à distance dans l'obscurité et depuis un point de couverture, est associée à un coût de 2. Les coûts des déplacements sont aussi modélisés avec des valeurs faibles (inférieures à 1), car on considère qu'il est préférable pour le PNJ de se déplacer afin d'obtenir des munitions ou une arme appropriée, ou de se rendre à un point de couverture, que d'attaquer directement au corps-à-corps à cause d'un manque d'équipement.

La hiérarchie des tâches composées est aussi différente par rapport à celle du chapitre 3. En effet on cherche ici à obtenir des tâches composées qui apportent une vision abstraite du plan avec une bonne estimation du coût final. Pour cela, on introduit plusieurs tâches composées qui correspondent à des actions pour lesquels ont a fait abstraction de l'objet

particulier utilisé. Par exemple, la tâche composée `AttackRanged(p, a)`, d'arité 2, est une abstraction de l'action `AttackRanged(g, p, a)`, d'arité 3. L'abstraction se fait en ignorant à ce niveau l'identité de l'arme à feu employée (représentée par le paramètre `g`). Ainsi, il est possible de choisir ce mode d'action à un haut niveau en limitant l'explosion combinatoire introduite par le choix de l'objet, et de bénéficier d'une estimation précise du coût du plan final, puisque le coût de l'attaque peut alors être pris en compte par la fonction d'estimation.

Un résumé des tâches avec les coûts qui leur sont associés est disponible dans le tableau 5.5. La description intégrale du domaine en HTDL est aussi disponible en annexe D.3.

Expériences

Une fois de plus, le système de planification ainsi que la configuration matérielle sont les mêmes que pour les deux expériences précédentes, et le lecteur est invité à en prendre connaissance via la partie 5.2.1.

Trois séries de 100 instances sont encore une fois générées aléatoirement. Chaque problème est composé d'un graphe de 5 zones avec un degré moyen de 1.5. La probabilité qu'un point de passage soit bloqué est de 30 %, la probabilité qu'une pièce soit éclairée est de 50 % et le PNJ a une probabilité égale à 60 % d'être blessé (dans ce cas, il doit initialement planifier pour se soigner). Le nombre de points d'intérêt est fixe pour chaque série, avec une répartition aléatoire entre les armes, les munitions, les kits de soins, les outils (pour débloquer les points de passage) et les positions de couverture. Afin que chaque problème soit résoluble, le PNJ dispose au moins d'une arme de corps-à-corps dans son inventaire. Les tâches et les méthodes sont aussi à nouveau modélisées de façon à ce que tous les réseaux de tâches explorés soient potentiellement exécutables. Les trois séries sont associées à des nombres de points de passage croissant égaux à 20, 40 et 80.

En plus de PHTN et de la recherche admissible, chacune des deux stratégies de décomposition $h = 1$ et $h = +\infty$ est combinée avec une heuristique à pondération uniforme $\omega = 5$, une heuristique à pondération linéaire $\lambda = 1$, une heuristique à pondérations distribuées (ω_c) et une heuristique à pondérations linéaires (λ_c). Ces deux dernières heuristiques sont décrites en détail dans le tableau 5.5. Leur modélisation suit toujours les règles utilisées pour les deux domaines précédents. Avec (ω_c), les poids sont réglés de façon à surestimer les coûts des décompositions à un haut niveau, et se réduisent à la mesure que le niveau des tâches auxquelles ils sont associés diminue. Pour (λ_c) on cherche à favoriser une recherche admissible pour les tâches aux plus hauts niveaux, puis on règle chaque pondération de façon à explorer l'espace de recherche en profondeur pour les tâches inférieures.

Résultats

Comme pour les deux expériences précédentes, les résultats complets sont présentés dans un tableau (le tableau 5.6) et une série particulière, en l'occurrence la troisième, est illustrée par une figure représentant les performances de chaque planificateur pour retourner une première solution (figure 5.10), ainsi que par une seconde figure représentant ces performances pour la meilleure solution retournée (figure 5.10).

tâche primitive	coût
MoveToPoint(pt, a)	1
MoveBetweenPoints(pt1, pt2, a)	1
Move(a1, a2, w)	0.5
TakeCover(cp, a)	1
Uncover	1
MakeAccessible(a1, a2, w, t)	1
PlaceInInventory(i, a)	2
Reload(g, am)	3
TurnLightOn(a, cb)	2
TurnLightOff(a, cb)	2
UseMedikit(m)	5
AttackMelee(k, p, a)	15
AttackRanged(g, p, a)	10
AttackRangedFromCover(g, p, a)	7
SneakKill(g, p, a)	5
SneakKillFromCover(g, p, a)	2

tâche composée	niveau	$\mathcal{C}_{ops}$	$\mathcal{C}_{pes}$	ω_c	λ_c
AchieveCloseTo(pt, a)	1	0	1.001	1	1
TakeCover(a)	2	1	2.002	5	1
TurnLightOn(a)	2	2	3.002	5	1
TurnLightOff(a)	2	2	3.002	5	1
UseMedikit	2	5	5.001	5	1
ReloadGun	2	3	3.001	5	1
ReloadGunWithNightvision	2	3	3.001	5	1
MakeAccessible(a1, a2, w)	2	1	1.001	5	1
AttackMelee(p, a)	2	15	16.002	5	1
AttackRanged(p, a)	2	10	10.001	5	1
AttackRangedFromCover(p, a)	2	7	7.001	5	1
SneakKill(p, a)	2	5	5.001	5	1
SneakKillFromCover(p, a)	2	2	2.001	5	1
PlaceMedikitInInventory(a)	2	2	3.002	5	1
PlaceAmmoInInventory(a)	2	2	3.002	5	1
PlaceKnifeInInventory(a)	2	2	3.002	5	1
PlaceGunInInventory(a)	2	2	3.002	5	1
PlaceToolInInventory(a)	2	2	3.002	5	1
PlaceGunWithNightvisionInInventory(a)	2	2	3.002	5	1
PlaceLoadedGunInInventory(a)	2	2	3.002	5	1
PlaceLoadedGunWithNightvisionInInventory(a)	2	2	3.002	5	1
AchieveUnFromCover	3	0	1.001	1	1
AchieveFromCover(a)	3	0	2.003	1	1
AchieveLighted(a)	4	0	4.004	1	1
AchieveDark(a)	4	0	4.004	1	1
Navigate(a1, a2, k)	5	0.5	$(4-k) \times 1.502 + 1.502$	10	0.2
AchieveAt(a)	6	0	7.01	1	0.2
AchieveHoldingMedikit	7	0	14.017	1	0.2
AchieveHoldingKnife	7	0	14.017	1	0.2
AchieveHoldingAmmo	7	0	14.017	1	0.2
AchieveHoldingGun	7	0	14.017	1	0.2
AchieveHoldingNvGun	7	0	14.017	1	0.2
AchieveHoldingLoadedGun	7	0	31.036	1	0.2
AchieveHoldingLoadedNvGun	7	0	31.036	1	0.2
AchieveFullHealth	8	0	19.019	1	0.2
AchieveWounded(p)	8	0	41.034	1	0.2

TABLE 5.5 – Résumé des différentes tâches pour le domaine *SimpleFPS*.

série	planificateur	% résolus	première solution				meilleure solution				dernière erreur
			nœuds visités	temps CPU	coût	erreur	nœuds visités	temps CPU	coût	erreur	
SimpleFPS 20 poi	PHTN	100.0	37	0.220	33.53	inf	1739	11.101	20.79	inf	inf
	$h = 1, \omega = 1$	99.0	341	36.411	20.77	0.00	341	36.411	20.77	0.00	0.00
	$h = +\infty, \omega = 1$	81.0	609	46.653	19.25	0.00	609	46.653	19.25	0.00	0.00
	$h = 1, \omega = 5$	100.0	109	9.697	21.79	61.93	211	18.856	20.83	44.25	0.57
	$h = +\infty, \omega = 5$	95.0	181	16.164	21.01	42.47	300	23.043	20.38	34.38	3.87
	$h = 1, (\omega_c)$	100.0	100	8.613	21.79	63.10	199	17.286	20.83	45.28	0.57
	$h = +\infty, (\omega_c)$	100.0	147	15.007	21.54	46.41	280	23.570	20.88	37.99	5.71
	$h = 1, \lambda = 1$	100.0	37	2.466	33.38	487.89	353	23.685	20.82	127.94	3.05
	$h = +\infty, \lambda = 1$	100.0	37	3.194	33.38	441.72	780	45.606	21.27	105.87	24.19
	$h = 1, (\lambda_c)$	100.0	76	6.938	21.52	60.73	107	9.229	20.79	53.12	0.18
	$h = +\infty, (\lambda_c)$	100.0	47	5.613	21.64	66.88	81	8.032	20.79	58.43	5.58
SimpleFPS 40 poi	PHTN	100.0	32	0.280	30.70	inf	2242	24.246	17.17	inf	inf
	$h = 1, \omega = 1$	89.0	271	52.241	16.26	0.00	271	52.241	16.26	0.00	0.00
	$h = +\infty, \omega = 1$	76.0	317	35.116	15.26	0.00	317	35.116	15.26	0.00	0.00
	$h = 1, \omega = 5$	100.0	117	17.876	17.86	74.44	153	23.205	17.14	62.02	6.22
	$h = +\infty, \omega = 5$	97.0	152	25.660	17.27	45.73	177	28.346	16.76	39.72	5.94
	$h = 1, (\omega_c)$	100.0	101	16.587	17.89	76.85	135	21.159	17.14	64.29	6.22
	$h = +\infty, (\omega_c)$	98.0	106	18.675	17.30	48.22	129	21.072	16.85	43.37	6.14
	$h = 1, \lambda = 1$	100.0	32	4.583	30.07	556.20	328	41.460	17.26	120.98	31.59
	$h = +\infty, \lambda = 1$	100.0	31	5.042	30.07	540.13	451	42.546	18.66	125.47	53.37
	$h = 1, (\lambda_c)$	100.0	59	11.572	17.94	85.38	95	16.258	17.00	70.03	3.40
	$h = +\infty, (\lambda_c)$	100.0	37	9.790	18.14	85.54	61	12.351	17.01	69.40	8.84
SimpleFPS 80 poi	PHTN	100.0	25	0.320	27.04	inf	1130	16.161	14.89	inf	inf
	$h = 1, \omega = 1$	67.0	196	58.619	13.36	0.00	196	58.619	13.36	0.00	0.00
	$h = +\infty, \omega = 1$	79.0	299	55.678	13.75	0.00	299	55.678	13.75	0.00	0.00
	$h = 1, \omega = 5$	96.0	106	35.962	15.12	70.05	127	42.881	14.61	58.03	16.21
	$h = +\infty, \omega = 5$	93.0	93	28.410	14.96	49.92	142	34.462	14.22	39.70	3.52
	$h = 1, (\omega_c)$	99.0	90	30.093	15.06	70.73	111	36.357	14.66	62.33	17.74
	$h = +\infty, (\omega_c)$	95.0	84	27.391	14.94	50.67	118	31.276	14.41	43.82	5.26
	$h = 1, \lambda = 1$	100.0	25	8.678	26.53	529.51	226	57.941	16.17	128.20	60.57
	$h = +\infty, \lambda = 1$	100.0	23	7.504	26.53	526.49	360	56.027	16.21	109.08	45.78
	$h = 1, (\lambda_c)$	97.0	40	20.472	15.04	96.32	61	24.706	14.51	86.67	13.49
	$h = +\infty, (\lambda_c)$	100.0	27	14.305	15.45	92.56	60	18.728	14.78	80.07	11.31

TABLE 5.6 – Tableau des résultats pour *SimpleFPS*.

En ce qui concerne la première solution retournée, ni les heuristiques à pondération uniforme ω, ni l'heuristique à pondérations distribuées (ω_c) ne parviennent à résoudre l'intégralité de la série. Ces mauvaises performances sont de plus confirmées par les deux autres séries d'instances. D'un autre côté, PHTN et les planificateurs avec l'heuristique $\lambda = 1$ parviennent à résoudre l'intégralité de la série, mais ne parviennent toujours pas à retourner une solution de bonne qualité. D'après le tableau, la solution optimale vaut environ 14.78, avec une marge d'erreur de 11.31 %, alors que ces planificateurs retournent des solutions dont le coût vaut au moins 26.53, soit une erreur supérieure de 80 % par rapport à la solution optimale. L'heuristique (λ_c) ne parvient pas non plus à résoudre l'intégralité de la série pour la stratégie de décomposition $h = 1$. En revanche, lorsqu'elle est combinée à la stratégie de décomposition $h = +\infty$, elle retourne une première solution avec un coût égal à 15.45, soit une erreur réelle inférieure à 15 % par rapport à la solution optimale (en étant pessimiste). Les planificateurs avec $\lambda = 1$ sont toutefois presque deux fois plus rapides. D'autre part PHTN apporte le meilleur temps d'exécution, près de 20 fois inférieur aux temps d'exécution obtenus avec $\lambda = 1$. Pourtant le tableau nous indique que PHTN, les planificateurs avec $\lambda = 1$, ainsi que le planificateur avec $h = +\infty$ et (λ_c) explorent approximativement le même nombre de nœuds dans leurs espaces de recherche, variant entre 23 et 27. On pourrait alors s'attendre à ce que ces planificateurs obtiennent aussi des performances équivalentes en terme de temps d'exécution. Cependant le traitement des effets abstraits, qui nous permet ici d'implémenter la stratégie de décomposition $h = +\infty$, nécessite un surplus de temps de calcul qui explique la différence de temps d'exécution entre PHTN (qui n'implémente pas d'effet abstrait) et les autres planificateurs. Cependant le prototype du système de planification dispose d'une implémentation non optimale du traitement de ces effets abstraits, qui par exemple évalue plusieurs fois une même formule logique associée à plusieurs effets conditionnels. Or dans cette version de *SimpleFPS*, les tâches aux plus hauts niveaux, telles que `AchieveHoldingLoadedGun` ou `AchieveFullHealth` sont modélisées avec des effets abstraits complexes. Ainsi, une implémentation plus efficace du traitement des effets abstraits, ou une application de ce système de planification HTN optimale à la planification non linéaire (pour laquelle les effets abstraits ne seraient plus nécessaires), pourrait permettre de réduire ces écarts de temps d'exécution.

En ce qui concerne la meilleure solution retournée, l'heuristique $\lambda = 1$ retourne des solutions aux coûts acceptables, seulement 10 % plus élevés que le coût de la meilleure solution retournée, mais requiert toujours un temps de calcul important. PHTN ainsi que le planificateur avec $h = +\infty$ et (λ_c) sont au coude à coude en terme de performance, avec des temps d'exécution comparables (mais légèrement en faveur de PHTN), et des solutions de coûts similaires. Cependant cette fois, le tableau nous indique que ces temps d'exécution similaires correspondent en fait à un espace de recherche exploré beaucoup plus important pour PHTN (avec plus de 1000 nœuds explorés), alors que le second planificateur n'en explore que 60.

Ainsi la stratégie de décomposition $h = +\infty$ couplée avec une heuristique à pondérations linéaires nous permet à nouveau d'obtenir un bon compromis entre temps d'exécution et coût pour la première solution retournée. Toutefois, la complexité du domaine au niveau des effets abstraits entraîne une augmentation du temps de calcul par nœud qui diminue considérablement l'intérêt de ce planificateur comparé à PHTN, qui même avec une exploration moins bien informée de son espace de recherche, demeure compétitif grâce à un temps d'exécution par nœud significativement plus court.

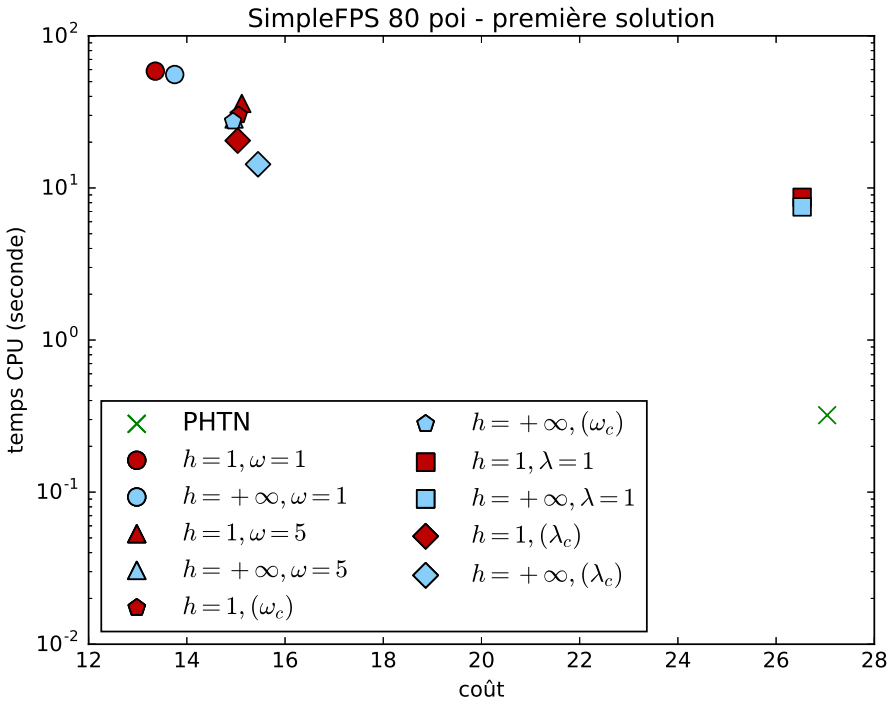


FIGURE 5.10 – Résultat de SimpleFPS pour la première solution retournée
Représentation des performances de chaque planificateur sur un graphe coût/temps CPU pour la première solution retournée.

5.3 Conclusions

Dans ce dernier chapitre, nous avons eu pour ambition de montrer pourquoi l'utilisation d'actions abstraites en planification HTN apporte un cadre adapté à la planification HTN optimale. Nous avons proposé de calculer des estimations optimistes et pessimistes des coûts de chaque tâche composée, et nous avons montré que lorsqu'une tâche correspond à une action abstraite, c'est-à-dire lorsqu'elle correspond à un ensemble de séquences d'actions primitives similaires, dont elle ne fait qu'abstraire des détails, on obtient des estimations optimistes et pessimistes proches. Il devient alors possible d'exploiter ces estimations précises dans un algorithme de planification HTN « anytime », en formulant une fonction d'évaluation pour l'étape de séparation et évaluation mieux informée que celle utilisée par SHOP2, et en implémentant une recherche heuristique non admissible au lieu d'une recherche en profondeur non informée.

Nous avons ensuite abordé la question des performances, et avons traité la question suivante : est-ce que le cadre de planification HTN optimale proposé permet d'améliorer la qualité des solutions dans le contexte de la planification en ligne ? Pour y répondre, nous avons évalué ce système sur trois domaines de planification HTN qui contiennent des actions abstraites. Nous avons ainsi identifié que l'un des profils proposé s'est avéré particulièrement adapté à la planification en ligne, en étant capable de retourner rapidement une première solution presque optimale. Ce profil est obtenu en combinant la formulation d'heuristiques à partir d'une pondération linéaire distribuée qui permet d'exploiter aisément les actions abstraites contenues dans les trois domaines, avec une stratégie de décomposition des tâches qui donne priorité aux tâches appartenant aux plus hauts niveaux de la hiérarchie.

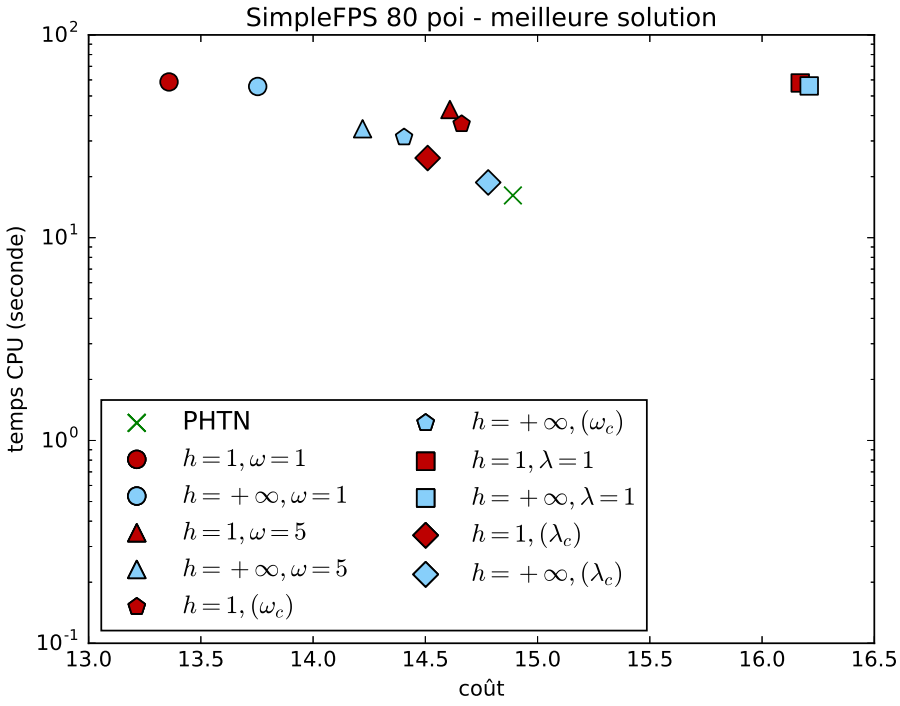


FIGURE 5.11 – Résultat de SimpleFPS pour la meilleure solution retournée
Représentation des performances de chaque planificateur sur un graphe coût/temps CPU pour la meilleure solution retournée.

Chapitre 6

Conclusions

6.1 Contributions

Dans cette thèse, notre objectif a été de mettre au point un composant de planification d'actions automatique, capable d'animer une section d'infanterie dans un environnement de simulation temps réel. Nous avons fait le constat que la prise de décision dans une section d'infanterie suit la structure hiérarchique de la chaine de commandement, c.-à-d. que chaque niveau de décision établit un plan d'actions cohérent avec son niveau de représentation, et seulement ensuite, décompose les actions du plan en un ensemble d'objectifs à atteindre pour ses subordonnés. À partir de ce constat, nous nous sommes orientés vers la planification HTN qui, sur la base de la décomposition de plans hiérarchiques sous la forme de réseaux de tâches, propose un modèle de planification d'actions adapté à la modélisation d'une chaine de commandement. Nous avons toutefois remarqué que la planification HTN ne disposait pas d'éléments permettant d'évaluer les plans autrement qu'au niveau primitif, par rapport à leur faisabilité, ou à leur qualité. Nous avons alors introduit le concept « d'actions abstraites » en planification HTN . Ces actions abstraites correspondent à des tâches composées qui, comme les actions primitives, sont associées à des effets et des coûts. Ces effets et ces coûts de haut niveau permettent alors d'évaluer l'exécutabilité ou le coût total à différents niveaux de représentation pour les plans hiérarchiques. Nous avons alors naturellement formulé la problématique suivante : est-ce que la modélisation d'actions abstraites en planification HTN permet d'améliorer l'efficacité de la recherche de solutions pour des problèmes fortement hiérarchisés ? Afin d'y répondre, nous avons développé les contributions suivantes à travers nos travaux :

1. Nous avons proposé une architecture de planification réactive, conçue pour la planification en ligne. Afin d'adapter les plans à l'évolution dynamique de l'environnement de simulation, cette architecture dispose d'un composant de perception de l'environnement qui maintient à jour la représentation de l'environnement, et d'un composant de contrôle de l'exécution qui déclenche des évènements de re-planification lorsqu'un plan n'est plus adapté à la situation courante. Mais encore, nous avons proposé de séparer la planification de mission de la planification des actions tactiques, la première étant dédiée au long terme alors que le second prend un charge une horizon temporelle plus courte. Cette séparation permet de re-planifier avec davantage de flexibilité, en ne modifiant le plan de mission que lorsque cela s'avère nécessaire.

   Pour obtenir un planificateur tactique capable de traiter en ligne des requêtes de planification pour près de 40 agents, nous avons proposé une implémentation fondée sur SHOP, combinée à un procédé de compilation du domaine de planification. Ce procédé permet de générer un planificateur optimisé pour un domaine donné, où les éléments de planification sont transcrits en structures de donnée statiques ou sous forme procédurale en C++. Ainsi, nous avons montré que le planificateur obtenu

est capable de produire des plans d'actions à l'échelle de la milliseconde. Toutefois, nous avons aussi montré que les performances du planificateur ne sont pas encore suffisantes pour retourner des solutions de bonnes qualité dans un délai compatible avec la planification en ligne.

2. Nous avons alors proposé d'améliorer l'efficacité du planificateur en combinant la planification HTN avec la planification par abstraction, via les effets des actions abstraites. Ces effets permettent d'implémenter un planificateur HTN capable de progresser les états à différents niveaux d'abstraction, ce qui permet d'évaluer l'exécutabilité des plans non primitifs et d'élaguer de l'espace de recherche ceux qui n'aboutissent à aucune solution. Afin de définir ces effets, nous avons proposé une sémantique qui repose sur le principe des hiérarchies d'abstractions, et qui consiste à définir une suite d'espaces d'états abstraits pour lesquels certaines tâches composées correspondent à des actions exécutables.

   À partir de cette sémantique, nous avons montré empiriquement que ce système permet de réduire les temps de calcul si les abstractions utilisées sont de bonne qualité, mais qu'il peut aussi être contre-productif dans le cas contraire.

   Nous avons aussi proposé une sémantique alternative qui consiste à décrire l'espace des états dans un langage logique à trois valeurs de vérité, où la troisième valeur permet d'interpréter les formules logiques indéterminées par rapport à un état. Les tâches composées correspondent alors à des actions déterministes dans cet espace d'états abstraits. Cette sémantique apporte plus de flexibilité au modélisateur, qui ainsi n'est plus contraint de modéliser les abstractions sous la forme d'une simple séquence d'espaces d'états. Mais elle est aussi plus lourde à rédiger, chaque tâche devant désormais implémenter ces propres opérateurs d'abstraction sous la forme d'effets.

3. Notre dernière contribution est un cadre de planification en ligne visant à obtenir des solutions presque optimales. Ce cadre repose sur la définition de coûts optimistes et pessimistes au niveau des tâches composées, analogues à ceux introduits par AHP. Afin d'effectuer de la planification en ligne, nous avons proposé un algorithme de planification HTN « anytime », qui combine une recherche heuristique non admissible avec une étape de séparation et évaluation , et retourne une succession de solutions de coûts décroissants.

   Plusieurs formulations d'heuristiques non admissibles ont aussi été proposées. Elles reposent sur la définition de poids différents pour chaque tâches : soit des poids « classiques », qui multiplient le coût optimiste d'une tâche par une constante supérieure à 1, soit des poids qui effectuent une combinaison linéaire entre le coût optimiste et le coût pessimiste de chaque tâche. Nous avons alors montré qu'il est possible de configurer ces heuristiques pour reproduire le principe de la chaine de commandement pour le guidage de la recherche, en favorisant une recherche admissible à un niveau donné, puis une recherche en profondeur entre deux niveaux. Pour trois domaines disposant d'une structure hiérarchique similaire à celle d'une chaine de commandement, nous avons montré empiriquement que les heuristiques à pondérations linéaires, lorsqu'elles sont associées à une stratégie de décomposition des tâches qui suit les différents niveaux, permettent de retourner rapidement des solutions de bonne qualité.

À la vue de ces contributions, nous pouvons apporter la réponse suivante à notre problématique : la modélisation d'actions abstraites permet bien d'améliorer l'efficacité de la planification HTN, pour des problèmes fortement hiérarchisés. Cette amélioration apparait principalement du point de vue de la planification optimale, où les actions abstraites apportent des éléments permettant de guider efficacement la recherche de

solutions presque optimales. Toutefois, si on se contente de solutions exécutables, en ignorant les coûts, alors le gain d'efficacité ne s'applique qu'à une catégorie plus restreinte de problèmes de planification.

6.2 Perspectives

Afin d'approfondir les travaux décrits dans ce manuscrit, nous proposons les pistes de recherche suivantes :

1. En ce qui concerne le système de planification SHPE, nous pensons qu'il existe encore des mécanismes à implémenter afin de réduire davantage le temps d'exécution du planificateur. Tout d'abord, nous avons fait le choix d'implémenter l'expansion des réseaux de tâches en générant l'ensemble de leurs réductions. Comme SHPE effectue une recherche en profondeur, de nombreuses réductions ne sont jamais visitées. Or, nous avons pu constater que la construction et l'insertion de ces éléments occupaient une part significative du temps d'exécution du planificateur, car il arrive parfois que des tâches composées soient associées à plusieurs dizaines, voire centaines, de méthodes (une fois leur paramètres instanciés). Ainsi un gain de temps d'exécution additionnel pourrait être obtenu en ne générant qu'une seule réduction à la fois, et en ne générant les réductions alternatives qu'en cas de retour arrière. Ce mécanisme aurait de plus l'avantage de réduire la complexité en espace du planificateur, qui ne nécessiterait plus qu'une quantité de mémoire linéaire par rapport à la profondeur de l'espace de recherche. Un second mécanisme à implémenter concerne l'allocation de la mémoire dynamique. Bien que le procédé de compilation permette d'encoder un maximum d'éléments de planification sous une forme statique (structure de données ou procédurale), il reste nécessaire d'instancier dynamiquement un certain nombre de ces éléments. Il se trouve que le langage C++ donne accès à l'implémentation des allocateurs mémoire, ce qui permet de remplacer l'allocateur standard par des allocateurs spécialisés si besoin. Or, l'exploration de l'espace de recherche en profondeur qu'effectue SHPE semble se prêter à l'utilisation d'une structure de pile pour gérer l'ordre des opérations d'allocation et de désallocation sur le tas. Ainsi, à chaque fois qu'un nouveau nœud est généré dans l'espace de recherche, la mémoire qu'il occupe peut être alloué au sommet de la pile, et cette mémoire peut ensuite être désallouer à partir du moment où toutes les branches de ce nœuds ont été exploré. Ainsi, les opérations d'allocation et de désallocation deviennent réalisables avec une complexité en $O(1)$, ce qui pourrait encore améliorer le temps d'exécution du planificateur.
2. Dans cette thèse, nous avons étudié empiriquement l'utilisation des effets abstraits pour évaluer l'exécutabilité des réseaux de tâches non primitifs. En identifiant les éléments qui ne mènent vers aucune solution, et en les élaguant de l'espace de recherche, nous comptions améliorer les performances de la recherche de solutions. Or, nos expériences ont révélé que ce principe fonctionnait comme nous l'attendions pour certains problèmes, mais aussi qu'il pouvait dégrader les performances de la recherche pour d'autres problèmes. Une explication « intuitive » à ce phénomène veut que si l'abstraction est de mauvaise qualité, alors peu de plans abstraits non exécutables en définitive pourront être identifier, et qu'il devient alors plus favorable de décomposer directement les actions abstraites afin d'obtenir au plus vite des tâches primitives plus informatives. Cependant, il serait très utile de disposer d'une analyse théorique de la complexité, analogue à celle effectuée par Fahiem Bacchus et Qiang Yang pour l'application de l'abstraction à la planification classique [6]. Cette analyse pourrait prendre en compte des paramètres tels que le

nombre de niveaux de décomposition, le nombre de méthodes par tâches, et le nombre de tâches par décomposition en moyenne pour un domaine, ainsi que la probabilité d'obtenir un plan exécutable à partir d'une action abstraite elle-même exécutable. En fonction de ces paramètres, on pourrait alors obtenir un ensemble de classes de complexité pour la planification HTN avec abstraction, et identifier ainsi les types de problèmes pour lesquels l'utilisation de l'abstraction est pertinente.

3. Nous avons aussi orienté nos travaux exclusivement sur des systèmes HTN où l'évaluation de l'exécution se fait en « progressant » les états, c'est-à-dire en exécutant chaque action sur l'état courant afin de calculer l'état suivant. En conséquence, nous n'avons exploité l'utilisation des effets abstraits que pour ce contexte. Or, il existe d'autres paradigmes de planification d'actions, comme la planification non linéaire, qui procède généralement par régression des buts. En ce qui concerne les effets abstraits reposant sur la définition d'une hiérarchie d'abstractions, leur application à la planification non-linéaire devrait être directe. En effet, il suffit à priori d'utiliser un planificateur HTN non-linéaire, tel que NONLIN, à chaque niveau d'abstraction. Mais en ce qui concerne les effets qui reposent sur la logique trivaluée, leur application dans un contexte de planification non-linéaire n'est pas aussi évidente. Une possibilité à explorer consiste à maintenir pour chaque plan partiel une structure de lien causaux potentiels. Ces liens causaux potentiels permettent de conserver une trace des causalités entre les effets de type « `abstract` » et les sous-buts. Ainsi, si aucun lien causal potentiel (ou lien causal « classique ») ne peut être établit pour au moins un sous-but, le plan peut être élagué de l'espace de recherche.

4. Pour cette thèse, nous nous sommes concentré sur les algorithmes de planification d'actions, et nous avons étudié des exemples où les différents niveaux d'abstraction sont modélisés par un opérateur humain. Cependant, pour une utilisation de ces algorithmes dans des environnements de simulations complexes, il semble difficilement réalisable de détailler manuellement les niveaux d'abstraction pour chaque scénario. À titre d'exemple, on peut modéliser un domaine de combat d'infanterie en associant les actions abstraites des groupes de combats à des zones plutôt qu'à des points précis dans l'environnement, afin que le niveau de représentation de ces actions soit cohérent avec le niveau du groupe. Comment peut-on alors obtenir ces zones pour un scénario donné ? Une solution est de compléter notre système de planification avec un programme d'analyse du terrain et de la situation tactique, capable d'identifier les éléments abstraits pour chaque situation. C'est par exemple ce que fait STRADA [58], un système de décision conçu pour les jeux de stratégie en temps réel. La prise de décision dans STRADA ne repose pas sur de la planification d'actions, mais sur de l'apprentissage par renforcement. Toutefois, une approche de l'abstraction reposant sur la structure hiérarchique des unités militaires y est aussi employée pour réduire la complexité des problèmes de décision. STRADA est doté d'un algorithme d'analyse du terrain capable de générer des abstractions topologiques adaptées aux différents niveaux de représentation dans la hiérarchie d'une unité militaire. La prise en compte de ce type d'algorithme dans notre architecture de planification constitue donc une piste de recherche supplémentaire dans l'objectif de rendre notre système de planification HTN par abstraction applicable à la simulation temps réel.

Annexe A

Définition formelle de HTDL

Les règles qui suivent constituent une définition formelle du langage HTDL . Celles-ci sont rédigées dans le langage *Extended Backus-Naur Form* (EBNF) tel que décrit par la norme ISO/IEC 14977 [1], à l'exception des symboles terminaux « NEWLINE », « INDENT » et « DEDENT » qui ne peuvent pas être définis sous la forme de chaines de caractères. De plus, la grammaire de HTDL n'est pas indépendante du contexte. La grammaire présenté ici est une grammaire indépendante du contexte, et ne permet d'analyser HTDL qu'à la suite d'une phase de pré-traitement permettant de générer les trois marqueurs « NEWLINE », « INDENT » et « DEDENT » et de supprimer les commentaires.

```
domain = domain element, {domain element};

domain element = primitive task | compound task | axiom;

primitive task =
    "task", name, ["(", variable list, ")"], ":", NEWLINE,
    INDENT, primitive task body, DEDENT;

compound task =
    "task", name, ["(", variable list, ")",], ":", NEWLINE,
    INDENT, compound task body, DEDENT;

primitive task body =
    [preconditions],
    [add effects],
    [delete effects],
    [update effects],
    [cost];

compound task body = method, {method};

preconditions =
    "precond", ":", NEWLINE,
    INDENT, formula statement, {formula statement}, DEDENT;

add effects =
    "add", ":", NEWLINE,
    INDENT, atom effect statement, {atom effect statement}, DEDENT;
```

```
delete effects =
    "delete", ":", NEWLINE, INDENT,
    atom effect statement, {atom effect statement}, DEDENT;

update effects =
    "update", ":", NEWLINE,
    INDENT, function effect statement, {function effect statement}, DEDENT;

cost = "cost", ":", NEWLINE INDENT term NEWLINE DEDENT;

formula statement =
    formula, NEWLINE
  | "forall", variable list, "if", formula":", NEWLINE,
    INDENT, formula statement, {formula statement}, DEDENT
  | "exists", variable list, ":", NEWLINE,
    INDENT, formula statement, {formula statement}, DEDENT;

atom effect statement =
    atom effect, NEWLINE
  | "if", formula, ":", NEWLINE,
    INDENT, atom effect statement, {atom effect statement}, DEDENT
  | "forall", variable list, "if", formula":", NEWLINE,
    INDENT, atom effect statement, {atom effect statement}, DEDENT;

function effect statement =
    function effect, NEWLINE
  | "if", formula, ":", NEWLINE,
    INDENT, function effect statement, {function effect statement}, DEDENT
  | "forall", variable list, "if", formula":", NEWLINE,
    INDENT, function effect statement, {function effect statement}, DEDENT;

atom effect =
    atom
  | atom, "if", formula
  | atom, "forall", variable list, "if", formula;

function effect =
    assignment
  | assignment, "if", formula
  | assignment, "forall", variable list, "if", formula;

assignment = function "=" term;

method =
    "method", [variable list], ":", NEWLINE,
    INDENT, method body, DEDENT;

method body = [preconditions], [subtasks], [orders];

subtasks =
```

```
    "subtasks", ":", NEWLINE,
    INDENT, subtask list, NEWLINE, {subtask list, NEWLINE}, DEDENT;

subtask list =
    (subtask | variable, ":", subtask),
    {",", (subtask | variable, ":", subtask)};

subtask = name, ["(", term list, ")"];

order =
    "order", ":", NEWLINE,
    INDENT, order list, NEWLINE, {order list, NEWLINE}, DEDENT;

order list = order, {",", order};

order = variable, "<", variable;

formula =
    atom
  | conjunction
  | disjunction
  | negation
  | implication
  | equivalence
  | universally quantified
  | existentially quantified
  | "(", formula, ")";

atom = name, ["(", term list, ")"] | term, "boolean operator", term;

conjunction = formula, "and", formula;

disjunction = formula, "or", formula;

negation = "not", formula;

implication = formula, "imply", formula;

equivalence = formula, "iff", formula;

universally quantified = "forall", variable list, "if", formula, ":", formula;

existantially quantified = "exists", variable list, ":", formula;

term = function | variable | number | "(", term, ")";

function = name, ["(", term list, ")"] | term, numeric operator, term;

term list = term, {",", term};

variable list = variable, {",", variable};
```

```
boolean operator = "==" | "!=" | "<" | ">", "<=" | ">=";

numeric operator = "+" | "-" | "*" | "/" | "**";

name = upper case letter, {alpha numeric character};

variable = lower case letter, {alpha numeric character | "_"}, {"'"};

number = ["-"], digit, {digit}, [".", digit, {digit}];

alpha numeric character = lower case letter | upper case letter | digit;

upper case letter = "A" | ... | "Z";

lower case letter = "a" | ... | "z";

digit = "0" | ... | "9";
```

Annexe B

SimpleFPS pour JSHOP2

```
(defdomain simplefps (
    ; definition of operators, as they have been defined by Stavros Vassos in
    ; his PDDL domain
    (:operator (!moving-to-patrol ?fromarea ?toarea ?waypoint)
        ((connected ?fromarea ?toarea ?waypoint)
         (npc-at ?fromarea)
         (waypoint ?waypoint)
         (open ?waypoint)
         (npc-uncovered)
         (npc-unaware)
         (npc-close-to ?waypoint))
        ((npc-at ?fromarea)
         (npc-close-to ?waypoint))
        ((npc-at ?toarea)
         (npc-not-close-to-point)))

    (:operator (!moving-to-take-position ?fromarea ?toarea ?waypoint)
        ((connected ?fromarea ?toarea ?waypoint)
         (npc-at ?fromarea)
         (waypoint ?waypoint)
         (open ?waypoint)
         (npc-aware)
         (npc-uncovered)
         (npc-close-to ?waypoint))
        ((npc-at ?fromarea)
         (npc-close-to ?waypoint))
        ((npc-at ?toarea)
         (npc-not-close-to-point)))

    (:operator (!move-away-from-point ?area ?point)
        ((npc-at ?area)
         (point-of-interest ?point ?area)
         (npc-close-to ?point))
        ((npc-close-to ?point))
        ((npc-not-close-to-point)))

    (:operator (!move-to-point ?area ?point)
        ((npc-at ?area)
```

```
     (point-of-interest ?point ?area)
     (npc-not-close-to-point))
    ((npc-not-close-to-point))
    ((npc-close-to ?point)))

(:operator (!move-to-point-from-point ?area ?point ?previouspoint)
    ((npc-at ?area)
     (point-of-interest ?point ?area)
     (point-of-interest ?previouspoint ?area)
     (npc-close-to ?previouspoint))
    ((npc-close-to ?previouspoint)
     (npc-not-close-to-point))
    ((npc-close-to ?point)))

(:operator (!make-accessible ?area1 ?area2 ?waypoint ?item)
    ((npc-holding ?item)
     (keycard ?item ?waypoint)
     (npc-at ?area1)
     (connected ?area1 ?area2 ?waypoint)
     (waypoint ?waypoint)
     (closed ?waypoint)
     (npc-close-to ?waypoint))
    ((closed ?waypoint)
     (npc-holding ?item))
    ((open ?waypoint)))

(:operator (!take-cover ?area ?point)
    ((npc-at ?area)
     (point-of-interest ?point ?area)
     (cover-point ?point)
     (npc-close-to ?point)
     (npc-uncovered))
    ((npc-uncovered))
    ((npc-covered)))

(:operator (!uncover ?point)
    ((npc-covered)
     (npc-close-to ?point))
    ((npc-covered)
     (npc-close-to ?point)
     (npc-not-close-to-point))
    ((npc-uncovered)))

(:operator (!place-in-inventory ?area ?item)
    ((npc-at ?area)
     (point-of-interest ?item ?area)
     (npc-close-to ?item)
     (item ?item)
     (npc-uncovered)
     (lighted ?area))
    ((point-of-interest ?item ?area)
```

```
     (npc-close-to ?item))
    ((npc-holding ?item)
     (npc-not-close-to-point)))

(:operator (!reload ?gun ?item)
    ((npc-holding ?gun)
     (npc-holding ?item)
     (gun ?gun)
     (unloaded ?gun)
     (ammo ?item ?gun))
    ((unloaded ?gun)
     (npc-holding ?item))
    ((loaded ?gun)))

(:operator (!attack-melee ?area ?knife ?player)
    ((npc-aware)
     (npc-at ?area)
     (point-of-interest ?player ?area)
     (lighted ?area)
     (npc-holding ?knife)
     (knife ?knife)
     (npc-close-to ?player)
     (player ?player)
     (npc-uncovered))
    ()
    ((player-wounded)))

(:operator (!attack-ranged ?npcarea ?gun ?player)
    ((lighted ?npcarea)
     (npc-aware)
     (npc-at ?npcarea)
     (point-of-interest ?player ?npcarea)
     (player ?player)
     (gun ?gun)
     (loaded ?gun)
     (npc-holding ?gun))
    ((loaded ?gun))
    ((player-wounded)
     (unloaded ?gun)))

(:operator (!use-medikit ?item)
    ((item ?item)
     (npc-holding ?item)
     (medikit ?item)
     (npc-injured))
    ((npc-injured)
     (npc-holding ?item))
    ((npc-full-health)))

(:operator (!make-contact ?npcarea ?player)
    ((lighted ?npcarea)
```

```
     (npc-at ?npcarea)
     (point-of-interest ?player ?npcarea)
     (player ?player)
     (npc-unaware))
    ((npc-unaware))
    ((npc-aware)))

(:operator (!sneak-kill ?npcarea ?gun ?player)
    ((npc-at ?npcarea)
     (point-of-interest ?player ?npcarea)
     (player ?player)
     (npc-holding ?gun)
     (has-nightvision ?gun)
     (loaded ?gun)
     (dark ?npcarea))
    ((npc-unaware)
     (loaded ?gun))
    ((npc-aware)
     (player-wounded)
     (unloaded ?gun)))

(:operator (!turn-on-lights ?area ?point)
    ((npc-uncovered)
     (npc-at ?area)
     (dark ?area)
     (point-of-interest ?point ?area)
     (npc-close-to ?point)
     (control-box ?point))
    ((dark ?area))
    ((lighted ?area)))

(:operator (!turn-off-lights ?area ?point)
    ((npc-uncovered)
     (npc-at ?area)
     (lighted ?area)
     (npc-close-to ?point)
     (control-box ?point))
    ((lighted ?area)
     (npc-close-to ?point))
    ((dark ?area)))

; definition of some internal operators, for bookkeeping during
; navigation (as JSHOP2 algorithm nether checks that a subproblem has
; already been met before
(:operator (!!explore ?area)
    ()
    ()
    ((explored ?area)))

(:operator (!!unexplore ?area)
    ()
```

```
    ((explored ?area))
    ())

; definition of the hierarchical task network structure

(:method (behave)
    ((player ?player) (point-of-interest ?player ?area)
     (cover-point ?coverpoint) (point-of-interest ?coverpoint ?area))
    ((restore-full-health)
     (take-cover-and-wound-player ?player ?area ?coverpoint))
    ((player ?player) (point-of-interest ?player ?area))
    ((restore-full-health) (wound-player ?player ?area)))

(:method (moving ?fromarea ?toarea ?waypoint)
    ((npc-unaware))
    ((uncover) (!moving-to-patrol ?fromarea ?toarea ?waypoint))
    ()
    ((uncover) (!moving-to-take-position ?fromarea ?toarea ?waypoint)))

(:method (move-to-point ?area ?point)
    ; NPC is in the area and is close to a point
    ((npc-at ?area) (npc-close-to ?previouspoint))
    ((!move-to-point-from-point ?area ?point ?previouspoint))
    ; NPC is in the area but is not close to any point
    ((npc-at ?area))
    ((!move-to-point ?area ?point)))

; first move-to-area method, when npc is already in the area
(:method (move-to-area ?toarea)
    ((npc-at ?toarea))
    ())

; second move-to-area method, when npc is not in the area, and we
; try to reach the area from an open door
(:method (move-to-area ?toarea)
    (:sort-by ?d < ((not (npc-at ?toarea))
                    (npc-at ?npc-area)
                    (connected ?fromarea ?toarea ?waypoint)
                    (not (explored ?fromarea))
                    (open ?waypoint)
                    (distance ?fromarea ?npc-area ?d)))
    ((!!explore ?toarea)
     (move-to-area ?fromarea)
     (move-to-point ?fromarea ?waypoint)
     (moving ?fromarea ?toarea ?waypoint)
     (!!unexplore ?toarea)))

; last move-to-area method, when npc is not in the area, and we try
; to reach the area from a closed door
(:method (move-to-area ?toarea)
    (:sort-by ?d < ((not (npc-at ?toarea))
```

```
                    (npc-at ?npc-area)
                    (connected ?fromarea ?toarea ?waypoint)
                    (not (explored ?fromarea))
                    (closed ?waypoint)
                    (keycard ?keycard ?waypoint)
                    (distance ?fromarea ?npc-area ?d)))
    ((!!explore ?toarea)
     (get-item ?keycard)
     (move-to-area ?fromarea)
     (move-to-point ?fromarea ?waypoint)
     (!make-accessible ?fromarea ?toarea ?waypoint ?keycard)
     (moving ?fromarea ?toarea ?waypoint)
     (!!unexplore ?toarea)))

(:method (uncover)
    ; if the NPC is already uncovered, do nothing
    ((npc-uncovered))
    ()
    ; else leave cover
    ((npc-close-to ?point))
    ((!uncover ?point)))

; get a loaded gun
(:method (get-loaded-gun)
    (:first ((gun ?gun) (npc-holding ?gun) (loaded ?gun)))
    ()
    ((gun ?gun) (npc-holding ?gun))
    ((reload ?gun)))

(:method (get-loaded-gun)
    (:sort-by ?d < ((npc-at ?npc-area)
                    (gun ?gun)
                    (point-of-interest ?gun ?gun-area)
                    (loaded ?gun)
                    (distance ?npc-area ?gun-area ?d)))
    ((get-item ?gun-area ?gun)))

(:method (get-loaded-gun)
    (:sort-by ?d < ((npc-at ?npc-area)
                    (gun ?gun)
                    (point-of-interest ?gun ?gun-area)
                    (not (loaded ?gun))
                    (distance ?npc-area ?gun-area ?d)))
    ((get-item ?gun-area ?gun) (reload ?gun)))

; get a loaded gun with night vision
(:method (get-loaded-gun-with-nightvision)
    (:first ((gun ?gun) (has-nightvision ?gun)
             (npc-holding ?gun) (loaded ?gun)))
    ()
    ((gun ?gun) (npc-holding ?gun))
```

```
    ((reload ?gun)))

(:method (get-loaded-gun-with-nightvision)
    (:sort-by ?d < ((npc-at ?npc-area)
                    (gun ?gun)
                    (has-nightvision ?gun)
                    (point-of-interest ?gun ?gun-area)
                    (loaded ?gun)
                    (distance ?npc-area ?gun-area ?d)))
    ((get-item ?gun-area ?gun)))

(:method (get-loaded-gun-with-nightvision)
    (:sort-by ?d < ((npc-at ?npc-area)
                    (gun ?gun)
                    (has-nightvision ?gun)
                    (point-of-interest ?gun ?gun-area)
                    (not (loaded ?gun))
                    (distance ?npc-area ?gun-area ?d)))
    ((get-item ?gun-area ?gun) (reload ?gun)))

(:method (reload ?gun)
    ((ammo ?ammo ?gun) (npc-holding ?ammo))
    ((!reload ?gun ?ammo))
    (:sort-by ?d < ((npc-area ?npc-area) (ammo ?ammo ?gun)
                    (point-of-interest ?ammo ?ammo-area)
                    (distance ?npc-area ?gun-area ?d)))
    ((get-item ?ammo-area ?ammo) (!reload ?gun ?ammo)))

; get a knife
(:method (get-knife)
    (:first ((knife ?knife) (npc-holding ?knife)))
    ()
    (:sort-by ?d < ((npc-at ?npc-area)
                    (knife ?knife)
                    (point-of-interest ?knife ?area)
                    (distance ?npc-area ?area ?d)))
    ((get-item ?area ?knife)))

(:method (attack-ranged ?area ?player)
    ((npc-holding ?gun) (loaded ?gun))
    ((!attack-ranged ?area ?gun ?player)))

(:method (sneak-kill ?area ?player)
    ((npc-holding ?gun) (loaded ?gun) (has-nightvision ?gun))
    ((!sneak-kill ?area ?gun ?player)))

(:method (attack-melee ?area ?player)
    ((knife ?knife) (npc-holding ?knife))
    ((!attack-ranged ?area ?gun ?player)))

(:method (wound-player ?player ?area )
```

```
    ()
    ((get-loaded-gun)
     (move-to-area ?area)
     (turn-on-lights ?area)
     (make-contact ?player)
     (attack-ranged ?area ?player)))

(:method (wound-player ?player ?area )
    ()
    ((get-loaded-gun-with-nightvision)
     (move-to-area ?area)
     (turn-off-lights ?area)
     (sneak-kill ?area ?player)))

(:method (wound-player ?player ?area )
    ()
    ((get-knife)
     (move-to-area ?area)
     (turn-on-lights ?area)
     (make-contact ?player)
     (move-to-point ?area ?player)
     (attack-melee ?area ?player)))

(:method (take-cover-and-wound-player ?player ?area ?cover-point)
    ()
    ((get-loaded-gun)
     (move-to-area ?area)
     (turn-on-lights ?area)
     (make-contact ?player)
     (move-to-point ?area ?cover-point)
     (!take-cover ?area ?cover-point)
     (attack-ranged ?area ?player)))

(:method (take-cover-and-wound-player ?player ?area ?cover-point)
    ()
    ((get-loaded-gun-with-nightvision)
     (move-to-area ?area)
     (turn-off-lights ?area)
     (move-to-point ?area ?cover-point)
     (!take-cover ?area ?cover-point)
     (sneak-kill ?area ?player)))

(:method (restore-full-health)
    ; if the NPC already has full health, do nothing
    ((npc-full-health))
    ()
    ; if the NPC have a medikit, use it
    (:first ((medikit ?medikit) (npc-holding ?medikit)))
    ((!use-medikit ?medikit))
    ; else get a medikit and use it
    (:sort-by ?d < ((npc-at ?npc-area)
```

```
                    (medikit ?medikit)
                    (point-of-interest ?medikit ?area)
                    (distance ?npc-area ?area ?d)))
    ((get-item ?area ?medikit) (!use-medikit ?medikit)))

(:method (get-item ?item)
    ((npc-holding ?item))
    ()
    ((point-of-interest ?item ?area))
    ((get-item ?area ?item)))

(:method (get-item ?area ?item)
    ()
    ((move-to-area ?area)
     (turn-on-lights ?area)
     (move-to-point ?area ?item)
     (!place-in-inventory ?area ?item)))

(:method (make-contact ?player)
    ((npc-aware))
    ()
    ((point-of-interest ?player ?area))
    ((move-to-area ?area)
     (turn-on-lights ?area)
     (!make-contact ?area ?player)))

(:method (turn-on-lights ?area)
    ; if area is already lighted, do nothing
    ((lighted ?area))
    ()
    ; else find the control box and turn the light on
    ((point-of-interest ?control-box ?area) (control-box ?control-box))
    ((move-to-area ?area)
     (move-to-point ?area ?control-box)
     (!turn-on-lights ?area ?control-box)))

(:method (turn-off-lights ?area)
    ; if area is already dark, do nothing
    ((dark ?area))
    ()
    ; else find the control box and turn the light off
    ((point-of-interest ?control-box ?area) (control-box ?control-box))
    ((move-to-area ?area)
     (move-to-point ?area ?control-box)
     (!turn-off-lights ?area ?control-box)))))
```

Annexe C

Domaines de planification HTDL avec hiérarchie d'abstractions

C.1 Reconnaisance de bâtiment

```
# axiomes pour définir les symboles abstraits

SectionClear(section) forall section if \
    Section(section)
    forall room if InSection(room) == section:
        RoomClear(room)

SquadAt(squad, section) forall squad, section if \
    Squad(squad) and Section(section)
    forall fireteam if InSquad(fireteam) == squad:
        InSection(FireteamAt(fireteam)) == section

SquadHolding(squad) forall squad if \
    Squad(squad)
    exists fireteam:
        InSquad(fireteam) == squad and FireteamHolding(fireteam)

FireteamAt(fireteam) = room forall fireteam, room if \
    Fireteam(fireteam) and Room(room)
    forall soldier if InFireteam(soldier) == fireteam
        InRoom(SoldierAt(soldier)) == room

FireteamHolding(fireteam) forall fireteam if \
    Fireteam(fireteam)
    exists soldier:
        InFireteam(soldier) == fireteam and SoldierCovering(soldier)

# tâches

task SoldierStartCovering(soldier):
    precond:
        Soldier(soldier) and Coverpoint(SoldierAt(soldier))
        not in SoldierCovering(soldier)
    add:
```

```
        SoldierCovering(soldier)

task SoldierStopCovering(soldier):
    precond:
        Soldier(soldier)
        SoldierCovering(soldier)
    delete:
        SoldierCovering(soldier)

task SoldierMove(soldier, waypoint):
    precond:
        Soldier(soldier) and Wayoint(waypoint)
        InRoom(SoldierAt(soldier)) == InRoom(waypoint) or \
            WaypointAccessible(SoldierAt(soldier), waypoint)
        not SoldierCovering(soldier)
    update:
        SoldierAtPoint(soldier) = waypoint

task AchieveSoldiersInFireteamAt(fireteam, waypoint):
    method:
        precond:
            forall soldier if InFireteam(soldier) == fireteam:
                SoldierAt(soldier) == waypoint
    method soldier:
        precond:
            InFireteam(soldier) == fireteam and SoldierAt(soldier) != waypoint
        subtasks:
            n1 : SoldierMove(soldier, waypoint)
            n2 : AchieveSoldiersInFireteamAt(fireteam, waypoint)
        order:
            n1 < n2

task AchieveSoldiersInFireteamHolding(fireteam):
    method:
        precond:
            forall wpt1, wpt2 if WaypointAccessible(wpt1, wpt2) and \
                    InRoom(wpt1) == FiretealAt(fireteam) and \
                        not RoomClear(InRoom(wpt2))
                exists soldier:
                    InFireteam(soldier) == fireteam
                    SoldierCovering(soldier)
                    FieldOfView(SoldierAt(soldier), wpt1)
     method soldier:
        precond:
            InFireteam(soldier) == fireteam and not SoldierCovering(soldier)
            Coverpoint(coverpoint)
            InRoom(coverpoint) == FireteamAt(fireteam)
        subtasks:
            n1 : SoldierMove(soldier, coverpoint)
            n2 : SoldierStartCovering(soldier)
            n3 : AchieveSoldiersInFireteamHolding(fireteam))
```

```
        order:
            n1 < n2
            n2 < n3

task AchieveSoldiersInFireteamNotHolding(fireteam):
    method:
        precond:
    method:
        precond:
            InFireteam(soldier) == fireteam and SoldierCovering(soldier)
        subtasks:
            n1 : SoldierStopCovering(soldier)
            n2 : AchieveSoldiersInFireteamNotHolding(fireteam)
        order:
            n1 < n2

task ReportRoomClear(fireteam, room):
    precond:
        not RoomClear(room) and FireteamAt(fireteam) == room:
    add:
        RoomClear(room)

task ExploreRoom(room):
    precond:
        Room(room)
    add:
        RoomExplored(room)
    level: 0...1

task UnexploreRoom(room):
     precond:
         Room(room) and RoomExplored(room)
     delete:
         RoomExplored(room)
    level: 0...1


task FireteamMove(fireteam, room):
    precond:
        Fireteam(fireteam) and Room(room) and Room(room2)
        RoomAccessible(FireteamAtRoom(fireteam), room)
        RoomClear(room) and not FireteamHolding(fireteam)
    update:
        FireteamAtRoom(fireteam) = room
    method waypoint, waypoint2:
        precond:
            WaypointAccessible(waypoint1, waypoint2)
            InRoom(waypoint1) == FireteamAt(fireteam)
            InRoom(waypoint2) == room
        subtasks:
            n1 : AchieveSoldiersInFireteamAt(fireteam, waypoint1)
```

```
            n2 : AchieveSoldiersInFireteamAt(fireteam, waypoint2)
        order:
            n1 < n2
    level: 1

task FireteamSecureRoom(fireteam, room):
    precond:
        Fireteam(fireteam) and Room(room)
        RoomAccessible(FireteamAtRoom(fireteam), room)
        not RoomClear(room)
        not FireteamHolding(fireteam) or forall room' if \
                RoomAccessible(FireteamHolding(fireteam), room') and \
                    room != room': RoomClear(room')
    add:
        FireteamHolding(fireteam) if (
            exists room': RoomAccessible(room, room') and not RoomClear(room'))
        RoomClear(room2)
    delete:
        FireteamHolding(fireteam) if (
            forall room' if RoomAccessible(room, room'): RoomClear(room'))
    update:
        FireteamAtRoom(fireteam) = room
    method waypoint1, waypoint2:
        precond:
            WaypointAccessible(waypoint1, waypoint2)
            InRoom(waypoint1) == FireteamAt(fireteam)
            InRoom(waypoint2) == room
        subtasks:
            n1 : AchieveSoldiersInFireteamNotHolding(fireteam)
            n2 : AchieveSoldiersInFireteamAt(fireteam, waypoint1)
            n3 : AchieveSoldiersInFireteamAt(fireteam, waypoint2)
            n4 : AchieveSoldiersInFireteamHolding(fireteam)
            n5 : ReportRoomClear(fireteam, room)
        order:
            n1 < n2
            n2 < n3
            n3 < n4
            n4 < n5
    level: 1

task FireteamStopHolding(fireteam):
    precond:
        FireteamHolding(fireteam)
        forall room if RoomAccessible(FireteamAt(fireteam), room):
            RoomClear(room)
    delete:
        FireteamHolding(fireteam)
    method:
        subtasks:
            AchieveSoldiersInFireteamNotHolding(fireteam)
    level: 1
```

```
task AchieveFireteamNotHolding(fireteam):
    method:
        precond:
            not FireteamHolding(fireteam)
    method:
        precond:
            FireteamHolding(fireteam)
        subtasks:
            FireteamStopHolding(fireteam)
    level: 1

task AchieveFireteamAt(fireteam, room):
    method:
        precond:
            FireteamAt(fireteam) == room
    method room':
        precond:
            FireteamAt(fireteam) != room
            RoomAccessible(FireteamAt(fireteam), room')
            RoomClear(room') and not RoomExplored(room')
        subtasks:
            n1 : ExploreRoom(room')
            n2 : FireteamMove(fireteam, room')
            n3 : AchieveFireteamAt(fireteam, room)
            n4 : UnexploreRoom(room')
        order:
            n1 < n2
            n2 < n3
            n3 < n4
    level: 1

task AchieveSectionClear(squad, section):
    method:
        precond:
            SectionClear(section)
    method fireteam, room:
        precond:
            InSquad(fireteam) == squad
            FireteamHolding(fireteam)
            InSection(room) == section
            not RoomClear(room)
            RoomAccessible(FireteamAt(fireteam), room)
        subtasks:
            n1 : FireteamClearRoom(fireteam, room)
            n2 : AchieveSectionClear(squad, section)
        order:
            n1 < n2
    method fireteam, room1, room2:
        precond:
            InSquad(fireteam) == squad
```

```
            FireteamHolding(fireteam)
            InSection(room1) == section
            InSection(room2) == section
            RoomClear(room1)
            not RoomClear(room2)
            RoomAccessible(room1, room2)
        subtasks:
            n1 : AchieveFireteamNotHolding(fireteam)
            n2 : AchieveFireteamAt(fireteam, room1)
            n3 : FireteamClearRoom(fireteam, room2)
            n4 : AchieveSectionClear(squad, section)
        order:
            n1 < n2
            n2 < n3
            n3 < n4
    level: 1

task AchieveFireteamsInSquadAtSection(squad, section):
    method:
        precond:
            SquadAt(squad) == section
    method fireteam, room:
        precond:
            InSquad(fireteam) == squad
            InSection(FireteamAt(fireteam)) != section
            InSection(room) == section
            Roomclear(room)
        subtasks:
            n1 : AchieveFireteamAt(fireteam, room)
            n2 : AchieveFireteamsInSquadAtSection(squad, section)
        ordre:
            n1 < n2
    level: 1

task SquadClearSection(squad, section):
    precond:
        not SectionClear(section)
        SectionAccessible(SquadAt(squad), section)
        not SquadHolding(squad) or forall section' if \
                SectionAccessible(SquadAt(squad), section') and \
                    section != section': SectionClear(section')
    add:
        SectionClear(section)
        SquadHolding(squad)
    update:
        SquadAt(squad) = section
    method fireteam, room:
        precond:
            InSquad(fireteam) == squad and FireteamHolding(fireteam)
            InSection(room) == section and not in RoomClear(room)
            RoomAccessible(FireteamAt(fireteam), room)
```

```
        subtasks:
            n1 : FireteamClearRoom(fireteam, room)
            n2 : AchieveSectionClear(squad, section)
            n3 : AchieveFireteamsInSquadAtSection(squad, section)
        order:
            n1 < n2
            n2 < n3
    method fireteam, room1, roo2:
        precond:
            InSquad(fireteam) == squad
            RoomAccessible(room1, room2)
            InSection(room1) == SquadAt(squad)
            InSection(room2) == section
        subtasks:
            n1 : AchieveFireteamAt(fireteam, room1)
            n2 : FireteamClearRoom(fireteam, room2)
            n3 : AchieveSectionClear(squad, section)
            n4 : AchieveFireteamsInSquadAtSection(squad, section)
        order:
            n1 < n2
            n2 < n3
            n3 < n4
    level: 2
```

C.2 Affectation de cibles

```
# axiomes pour définir les symboles abstraits

PlatoonFieldOfFire(platoon, target) forall platoon, target if \
    Platoon(platoon) and Target(target)
    exists squad:
        Squad(squad) and InPlatoon(squad, platoon)
        SquadFieldOfFire(squad, target)

PlatoonDistance(platoon, target) = min(
    SquadDistance(squad, target) forall squad if InPlatoon(squad, platoon)
) forall platoon, target if Platoon(platoon) and Target(target)

PlatoonAmmoCount(platoon, weapon_type) = sum(
    SquadAmmoCount(squad, weapon_type) forall squad if InPlatoon(squad)
) forall platoon, weapon_type if Platoon(platoon) and WeaponType(weapon_type)

SquadFieldOfFire(squad, target) forall squad, target if \
    Squad(squad) and Target(target)
    exists fireteam:
        Fireteam(fireteam) and InSquad(fireteam, squad)
        FireteamFieldOfFire(fireteam, target)

SquadDistance(squad, target) = min(
    FireteamDistance(fireteam, target)
    forall fireteam if InSquad(fireteam, squad)
```

```
) forall squad, target if Squad(squad) and Target(target)

SquadAmmoCount(squad, weapon_type) = sum(
    FireteamAmmoCount(fireteam, weapon_type) forall fireteam if \
        InSquad(fireteam)) forall squad, weapon_type if \
            Squad(squad) and WeaponType(weapon_type)

# tâches

task AchieveNeutralized(target):
    method:
        precond:
            Neutralized(target)
    method platoon, weapon_type:
        precond:
            not Neutralized(target)
            Platoon(platoon) and WeaponType(weapon_type)
            PlatoonDistance(platoon, target) <= Range(weapon_type)
            PlatoonFieldOfFire(platoon, target)
            not Armored(target) or AntiArmored(weapon_type)
            PlatoonAmmoCount(platoon, weapon_type) * \
                Effectiveness(weapon_type) >= Capacity(target)
        subtasks:
            PlatoonNeutralize(platoon, weapon_type, target)
    level: 2

task PlatoonNeutralize(platoon, weapon_type, target):
    precond:
        Platoon(platoon) and WeaponType(weapon_type) and Target(target)
        not Neutralized(target)
        PlatoonDistance(platoon, target) <= Range(weapon_type)
        PlatoonFieldOfFire(platoon, target)
        not Armored(target) or AntiArmored(weapon_type)
        PlatoonAmmoCount(platoon, weapon_type) *
            Effectiveness(weapon_type) >= Capacity(target)
    add:
        Neutralized(target)
    update:
        PlatoonAmmoCount(platoon, weapon_type) -= \
            ceil(Capacity(target) / Effectiveness(weapon_type))
    method squad:
        precond:
            Squad(squad) and InPlatoon(squad) == platoon:
        subtasks:
            SquadNeutralize(squad, weapon_type, target)
    level: 2

task SquadNeutralize(squad, weapon_type, target):
    precond:
        Squad(squad) and WeaponType(weapon_type) and Target(target)
        not Neutralized(target)
```

```
        SquadDistance(squad, target) <= Range(weapon_type)
        SquadFieldOfFire(squad, target)
        not Armored(target) or AntiArmored(weapon_type)
        SquadAmmoCount(squad, weapon_type) * Effectiveness(weapon_type) >= \
            Capacity(target)
    add:
        Neutralized(target)
    update:
        SquadAmmoCount(squad, weapon_type) -= \
            ceil(Capacity(target) / Effectiveness(weapon_type))
    method fireteam:
        precond:
            Fireteam(fireteam) and InSquad(fireteam) == squad:
        subtasks:
            FireteamNeutralize(fireteam, weapon_type, target)
    level: 1

task FireteamNeutralize(fireteam, weapon_type, target):
    precond:
        Fireteam(fireteam) and WeaponType(weapon_type) and Target(target)
           not Neutralized(target)
           FireteamDistance(fireteam, target) <= Range(weapon_type)
           FireteamFieldOfFire(fireteam, target)
           not Armored(target) or AntiArmored(weapon_type)
           FireteamAmmoCount(fireteam, weapon_type) * \
               Effectiveness(weapon_type) >= Capacity(target)
    add:
        Neutralized(target)
    update:
        FireteamAmmoCount(fireteam, weapon_type) -= \
            ceil(Capacity(target) / Effectiveness(weapon_type))
    cost: ceil(Capacity(target) / Effectiveness(weapon_type)) * \
        Effectiveness(weapon_type)
```

C.3 Navigation dans un graphe hiérarchique

```
# axiomes pour définir les symboles abstraits

AbsVisited(abs_state) forall abs_state if \
    AbsState(abs_state)
    exists state:
        Visited(state) and AbstractTo(state) = abs_state

AbsTransition(abs_state1, abs_state2) forall abs_state1, abs_state2 if \
    AbsState(abs_state1) and AbsState(abs_state2)
    exists state1, state2:
        Transition(state1, state2)
        AbstractTo(state1) == abs_state1 and AbstractTo(state2) == abs_state2

AbsCurrent = AbstractTo(Current)
```

```
AbsGoal = AbstractTo(Goal)

# tâches

task AchieveGoal:
    method:
        subtasks:
            n1 : AchieveAbsState(AbsGoal)
            n2 : IsGoal
        order:
            n1 < n2
    level: 1

task AchieveAbsState(abs_state):
    method:
        precond:
            AbsCurrent == abs_state
    method abs_state':
        precond:
            AbsTransition(AbsCurrent, abs_state')
            not AbsVisited(abs_state')
        subtasks:
            n1 : AbsAction(abs_state')
            n2 : AchieveAbsState(abs_state)
        order:
            n1 < n2
    level: 1

task AbsAction(abs_state):
    precond:
        AbsTransition(AbsCurrent, abs_state)
        not AbsVisited(abs_state)
    add:
        AbsVisited(AbsCurrent)
    update:
        AbsCurrent = abs_state
    method state1, state2:
        precond:
            Transition(state1, state2) and not Visited(state2)
            AbstractTo(state1) == AbsCurrent
            AbstractTo(state2) == abs_state
        subtasks:
            n1 : AchieveState(state1)
            n2 : Action(state2)
        order:
            n1 < n2
    level: 1

task IsGoal:
    precond:
        AbsCurrent == AbsGoal:
```

```
    method:
        subtasks:
            AchieveState(Goal)
    level: 1

task AchieveState(state):
    method:
        precond:
            Current == state
    method state':
        precond:
            Transition(Current, state')
            not Visited(state')
        subtasks:
            n1 : Action(state')
            n2 : AchieveState(state)
        order:
            n1 < n2
    level: 1

task Action(state):
    precond:
        Transition(Current, state)
        not Visited(state)
    add:
        Visited(Current)
    update:
        Current = state
```

Annexe D

Domaine de planification HTN avec abstraction est coûts de haut niveau

D.1 *Robotbox*

```
# CarryThruDoorCost = 1
# PullThruDoorCost  = 1
# OpenDoorCost      = 1
# LoadBoxCost       = 1
# AttachBoxCost     = 1
# Epsilon           = 0.001

task CarryThruDoor(box, door, room1, room2):
    precond:
        Box(box) and Door(door) and Room(room1) and Room(room2)
        Connects(door, room1, room2)
        AtRoom(box, room1) and Open(door) and Loaded(box)
    add:
        AtRoom(box, room2)
    delete:
        AtRoom(box, room1)
    cost:
        CarryThruDoorCost

task PullThruDoor(box, door, room1, room2):
    precond:
        Box(box) and Door(door) and Room(room1) and Room(room2)
        Connects(door, room1, room2)
        AtRoom(box, room1) and Open(door) and Attached(box)
    add:
        AtRoom(box, room2)
    delete:
        AtRoom(box, room1)
    cost:
        PullThruDoorCost

task OpenDoor(door):
    precond:
        Door(door) and Openable(door) and not Open(door)
```

```
    add:
        Open(door)
    cost:
        OpenDoorCost

task AttachBox(box):
    precond:
        Box(box) and not Attached(box)
    add:
        Attached(box)
    cost:
        AttachBoxCost

task LoadBox(box):
    precond:
        Box(box) and not Loaded(box)
    add:
        Loaded(box)
    cost:
        LoadBoxCost

task AchieveAttached(box):
    precond:
        Box(box)
    add:
        Attached(box)
    method:
        precond:
            Attached(box)
    method:
        precond:
            not Attached(box) and not Loaded(box)
        subtasks:
            AttachBox(box)
    optimistic:
        0
    pessimistic:
        AttachBoxCost + Epsilon
    level:
        1

task AchieveLoaded(box):
    precond:
        Box(box)
    add:
        Loaded(box)
    method:
        precond:
            Loaded(box)
    method:
        precond:
```

```
            not Attached(box) and not Loaded(box)
        subtasks:
            LoadBox(box)
    optimistic:
        0
    pessimistic:
        LoadBoxCost + Epsilon
    level:
        1

task MoveThruDoor(box, door, room1, room2):
    precond:
        Box(box) and Door(door) and Room(room1) and Room(room2)
        Connects(door, room1, room2)
        AtRoom(box, room1) and Open(door)
    add:
        AtRoom(box, room2)
    delete:
        AtRoom(box, room1)
    abstract:
        Attached(box)
        Loaded(box)
    method:
        subtasks:
            n1 : AchieveLoaded(box)
            n2 : CarryThruDoor(box, door, room1, room2)
        order:
            n1 < n2
    method:
        subtasks:
            n1 : AchieveAttached(box)
            n2 : PullThruDoor(box, door, room1, room2)
        order:
            n1 < n2
    optimistic:
        Min(PullThruDoorCost, CarryThruDoorCost)
    pessimistic:
        Max(
            AttachBoxCost + Epsilon + PullThruDoorCost,
            LoadBoxCost + Epsilon + CarryThruDoorCost
        ) + Epsilon
    level:
        2

task AchieveOpen(door):
    precond:
        Door(door) and Openable(door)
    add:
        Open(door)
    method:
        precond:
```

```
            Open(door)
    method:
        precond:
            not Open(door)
        subtasks:
            OpenDoor(door)
    optimistic:
        0
    pessimistic:
        OpenDoorCost + Epsilon
    level:
        3

task AchieveAtRoom(box, room, depth):
    precond:
        Box(box) and Room(room)
    add:
        AtRoom(box, room)
    delete:
        AtRoom(box, room) forall room' \
            if Room(room') and room' != room and AtRoom(box, room')
    abstract:
        Attached(box)
        Loaded(box)
        Open(door) forall door if Openable(door) and not Open(door)
    method:
        precond:
            AtRoom(box, room)
    method door, room1, room2:
        precond:
            Connects(door, room1, room2)
            AtRoom(box, room1) and room1 != room
            Openable(door)
            depth < MaxDistance
        subtasks:
            n1 : AchieveOpen(door)
            n2 : MoveThruDoor(box, door, room1, room2)
            n3 : AchieveAtRoom(box, room, depth + 1)
        order:
            n1 < n2
            n2 < n3
    optimistic:
        0
    pessimistic:
        (MaxDistance - depth) * (
            OpenDoorCost + Epsilon +
            Max(
                AttachBoxCost + Epsilon + PullThruDoorCost,
                LoadBoxCost + Epsilon + CarryThruDoorCost) + Epsilon +
            Epsilon)
    level:
```

```
        4
```

D.2 logistics

```
# LoadCost   = 1
# UnloadCost = 1
# DriveCost  = 2
# FlyCost    = 10
# Epsilon    = 0.001

# Derived predicate :

AdjacentCity(city1, city2) forall city1, city2, airport1, airport2 \
    if City(city1) and City(city2) and \
       Airport(airport1) and Airport(airport2) and \
       InCity(airport1, city1) and InCity(airport2, city2) and \
       Connected(airport1, airport2)

# Tasks definitions:

task Load(carrier, package, location, city):
    precond:
        Package(package) and Location(location) and Carrier(carrier)
        At(carrier, location) and At(package, location)
        InCity(location, city)
    add:
        Inside(package, carrier)
    delete:
        At(package, location)
        AtCity(package, city)
    cost:
        LoadCost

task Unload(carrier, package, location, city):
    precond:
        Package(package) and Location(location) and Carrier(carrier)
        At(carrier, location) and Inside(package, carrier)
        InCity(location, city)
    add:
        At(package, location)
        AtCity(package, city)
    delete:
        Inside(package, carrier)
    cost:
        UnloadCost

task Drive(truck, location1, location2, city):
    precond:
        Truck(truck) and Location(location1) and Location(location1)
        City(city) and InCity(location1, city) and InCity(location2, city)
        Adjacent(location1, location2)
```

```
        At(truck, location1)
    add:
        At(truck, location2)
    delete:
        At(truck, location1)
    cost:
        DriveCost

task Fly(airplane, airport1, airport2, city1, city2):
    precond:
        Airplane(airplane) and Airport(airport1) and Airport(airport2)
        Connected(airport1, airport2) and At(airplane, airport1)
        InCity(airport1, city1) and InCity(airport2, city2)
    add:
        At(airplane, airport2)
        AtCity(airplane, city2)
    delete:
        At(airplane, airport1)
        AtCity(airplane, city1)
    cost:
        FlyCost

task AchieveTruckAt(truck, location, city, depth):
    precond:
        Truck(truck) and Location(location) and City(city)
        InCity(location, city) and AtCity(truck, city)
    add:
        At(truck, location)
    delete:
        At(truck, location') forall location' \
            if At(truck, location') and (location' != location)
    method:
        precond:
            At(truck, location)
    method location1, location2:
        precond:
            At(truck, location1) and (location1 != location)
            Adjacent(location1, location2)
            InCity(location1, city) and InCity(location2, city)
            depth < MaxLocationDistance
        subtasks:
            Drive(truck, location1, location2, city)
            AchieveTruckAt(truck, location, city, depth + 1)
    optimistic:
        0
    pessimistic:
        (MaxLocationDistance - depth) * (DriveCost + Epsilon)
    level:
        1

task Pickup(truck, package, location, city):
```

```
    precond:
        Truck(truck) and Location(location) and Package(package) and City(city)
        InCity(location, city)
        AtCity(truck, city) and At(package, location)
    add:
        At(truck, location)
        Inside(package, truck)
    delete:
        At(package, location)
        At(truck, location') forall location' \
            if At(truck, location') and (location' != location)
    method:
        subtasks:
            AchieveTruckAt(truck, location, city, 0)
            Load(truck, package, location, city)
    optimistic:
        LoadCost
    pessimistic:
        MaxLocationDistance * (DriveCost + Epsilon) + \
        LoadCost + Epsilon
    level:
        2

task Deliver(truck, package, location, city):
    precond:
        Truck(truck) and Location(location) and Package(package) and City(city)
        InCity(location, city) and AtCity(truck, city)
        Inside(package, truck)
    add:
        At(truck, location)
        At(package, location)
    delete:
        Inside(package, truck)
        At(truck, location') forall location' \
            if At(truck, location') and (location' != location)
    method:
        subtasks:
            AchieveTruckAt(truck, location, city, 0)
            Unload(truck, package, location, city)
    optimistic:
        UnloadCost
    pessimistic:
        MaxLocationDistance * (DriveCost + Epsilon) + \
        UnloadCost + Epsilon
    level:
        2

task AchieveInCityAt(package, city, location):
    precond:
        Package(package) and City(city) and Location(location)
        InCity(location, city) and AtCity(package, city)
```

```
    add:
        At(package, location)
    delete:
        At(package, location') forall location' \
            if At(package, location') and (location' != location)
    abstract:
        At(truck, location') forall truck, location' \
            if Truck(truck) and AtCity(truck, city) and InCity(location', city)
    method:
        precond:
            At(package, location)
    method location', truck:
        precond:
            At(package, location') and (location' != location)
            Truck(truck) and AtCity(truck, city)
        subtasks:
            Pickup(truck, package, location', city)
            Deliver(truck, package, location, city)
    optimistic:
        0
    pessimistic:
        2 * MaxLocationDistance * (DriveCost + Epsilon) + \
        LoadCost + UnloadCost + 3 * Epsilon
    level:
        3

task LoadAtCity(airplane, package, city):
    precond:
        Package(package) and City(city) and Airplane(airplane)
        AtCity(airplane, city) and AtCity(package, city)
    add:
        Inside(package, airplane)
    delete:
        AtCity(package, city)
        At(package, location) forall location if At(package, location)
    abstract:
        At(truck, location) forall truck, location \
            if Truck(truck) and AtCity(truck, city) and InCity(location, city)
    method airport:
        precond:
            At(airplane, airport)
        subtasks:
            AchieveInCityAt(package, city, airport)
            Load(airplane, package, airport, city)
    optimistic:
        LoadCost
    pessimistic:
        2 * MaxLocationDistance * (DriveCost + Epsilon) + \
        2 * LoadCost + UnloadCost + 4 * Epsilon
    level:
        4
```

```
task UnloadAtCity(airplane, package, city):
    precond:
        Package(package) and City(city) and Airplane(airplane)
        Inside(package, airplane) and AtCity(airplane, city)
    add:
        AtCity(package, city)
        At(package, airport) forall airport if At(airplane, airport)
    delete:
        Inside(package, airplane)
    method airport:
        precond:
            At(airplane, airport)
        subtasks:
            Unload(airplane, package, airport, city)
    optimistic:
        UnloadCost
    pessimistic:
        UnloadCost + Epsilon
    level:
        4

task FlyToCity(airplane, city1, city2):
    precond:
        Airplane(airplane) and City(city1) and City(city2)
        AdjacentCity(city1, city2) and AtCity(airplane, city1)
    add:
        AtCity(airplane, city2)
    delete:
        AtCity(airplane, city1)
        At(airplane, airport) forall airport if At(airplane, airport)
    abstract:
        At(airplane, airport) forall airport \
            if Airport(airport) and InCity(airport, city2)
    method airport1, airport2:
        precond:
            At(airplane, airport1) and Airport(airport2)
            InCity(airport2, city2)
        subtasks:
            Fly(airplane, airport1, airport2, city1, city2)
    optimistic:
        FlyCost
    pessimistic:
        FlyCost + Epsilon
    level:
        4

task AchieveAirplaneAtCity(airplane, city, depth):
    precond:
        Airplane(airplane) and City(city)
    add:
```

```
        AtCity(airplane, city)
    delete:
        At(airplane, airport) forall airport \
            if At(airplane, airport) and not InCity(airport, city)
        AtCity(airplane, city') forall city' \
            if AtCity(airplane, city') and city' != city
    abstract:
        At(airplane, airport) forall airport \
            if Airport(airport) and InCity(airport, city)
    method:
        precond:
            AtCity(airplane, city)
    method city1, city2:
        precond:
            AtCity(airplane, city1) and city1 != city
            AdjacentCity(city1, city2)
            depth < MaxCityDistance
        subtasks:
            FlyToCity(airplane, city1, city2)
            AchieveAirplaneAtCity(airplane, city, depth + 1)
    optimistic:
        0
    pessimistic:
        (MaxCityDistance - depth) * (FlyCost + 2 * Epsilon)
    level:
        5

task PickupAtCity(airplane, package, city):
    precond:
        Airplane(airplane) and Package(package) and City(city)
        AtCity(package, city)
    add:
        AtCity(airplane, city)
        Inside(package, airplane)
    delete:
        AtCity(package, city)
        At(package, location) forall location \
            if At(package, location) and InCity(location, city)
        At(airplane, airport) forall airport \
            if At(airplane, airport) and not InCity(airport, city)
        AtCity(airplane, city') forall city' \
            if AtCity(airplane, city') and city' != city
    abstract:
        At(truck, location) forall truck, location \
            if AtCity(truck, city) and InCity(location, city)
    method:
        subtasks:
            AchieveAirplaneAtCity(airplane, city, 0)
            LoadAtCity(airplane, package, city)
    optimistic:
        LoadCost
```

```
    pessimistic:
        MaxCityDistance * (FlyCost + 2 * Epsilon) + \
        2 * MaxLocationDistance * (DriveCost + Epsilon) + \
        2 * LoadCost + UnloadCost + 5 * Epsilon
    level:
        6

task DeliverToCity(airplane, package, city):
    precond:
        Airplane(airplane) and Package(package) and City(city)
        Inside(package, airplane)
    add:
        AtCity(airplane, city)
        AtCity(package, city)
    delete:
        Inside(package, airplane)
        At(airplane, airport) forall airport \
            if At(airplane, airport) and not InCity(airport, city)
        AtCity(airplane, city') forall city' \
            if AtCity(airplane, city') and city' != city
    abstract:
        At(airplane, airport) forall airport \
            if Airport(airport) and InCity(airport, city)
        At(package, airport) forall airport \
            if Airport(airport) and InCity(airport, city)
    method:
        subtasks:
            AchieveAirplaneAtCity(airplane, city, 0)
            UnloadAtCity(airplane, package, city)
    optimistic:
        UnloadCost
    pessimistic:
        MaxCityDistance * (FlyCost + 2 * Epsilon) + \
        UnloadCost + 2 * Epsilon
    level:
        6

task AchieveAtCity(package, city):
    add:
        AtCity(package, city)
    delete:
        At(package, location) forall location \
            if At(package, location) and not InCity(location, city)
        AtCity(package, city') forall city' \
            if AtCity(package, city') and city' != city
    abstract:
        At(airport, city) forall airport \
            if not AtCity(package, city) and InCity(airport, city)
        At(truck, location) forall truck, city', location \
            if AtCity(package, city') and (city' != city) and \
               AtCity(truck, city') and InCity(location, city')
```

```
        At(airplane, airport) forall airplane, airport \
            if Airplane(airplane) and Airport(airport)
        AtCity(airplane, city') forall airplane, city' \
            if Airplane(airplane) and City(city')
    method:
        precond:
            AtCity(package, city)
    method airplane, city':
        precond:
            Airplane(airplane) and AtCity(package, city') and city' != city
        subtasks:
            PickupAtCity(airplane, package, city')
            DeliverToCity(airplane, package, city)
    optimistic:
        0
    pessimistic:
        2 * MaxCityDistance * (FlyCost + 2 * Epsilon) + \
        2 * MaxLocationDistance * (DriveCost + Epsilon) + \
        2 * LoadCost +  2 * UnloadCost + 8 * Epsilon
    level:
        7

task AchieveAt(package, location):
    add:
        At(package, location)
    delete:
        At(package, location') forall location' if \
            At(package, location') and location' != location
    abstract:
        At(truck, location') forall truck, city, location' \
            if Truck(truck) and AtCity(truck, city) and InCity(location', city)
        At(airplane, airport) forall airplane, airport \
            if Airplane(airplane) and Airport(airport)
        AtCity(airplane, city) forall airplane, city \
            if Airplane(airplane) and City(city)
    method city:
        precond:
            InCity(location, city)
        subtasks:
            AchieveAtCity(package, city)
            AchieveInCityAt(package, city, location)
    optimistic:
        0
    pessimistic:
        2 * MaxCityDistance * (FlyCost + 2 * Epsilon) + \
        4 * MaxLocationDistance * (DriveCost + Epsilon) + \
        3 * LoadCost +  3 * UnloadCost + 12 * Epsilon
    level:
        8
```

D.3 SimpleFPS

```
# Cost constants
# MoveToPointCost           = 1
# MoveCost                  = 0.5
# TakeCoverCost             = 1
# UncoverCost               = 1
# MakeAccessibleCost        = 1
# PlaceInInventoryCost      = 2
# ReloadCost                = 3
# TurnLightOnCost           = 2
# TurnLightOffCost          = 2
# UseMedikitCost            = 5
# AttackMeleeCost           = 15
# AttackRangedCost          = 10
# AttackRangedFromCoverCost = 7
# SneakKillCost             = 5
# SneakKillFromCoverCost    = 2
# Epsilon                   = 0.001

# Derivation rules.

AtMedikit(area) forall area \
    if Area(area) and \
        exists medikit:
            Medikit(medikit) and PointOfInterest(medikit, area)

AtKnife(area) forall area \
    if Area(area) and \
        exists knife:
            Knife(knife) and PointOfInterest(knife, area)

AtAmmo(area) forall area \
    if Area(area) and \
        exists ammo:
            Ammo(ammo) and PointOfInterest(ammo, area)

AtGun(area) forall area \
    if Area(area) and \
        exists gun:
            Gun(gun) and PointOfInterest(gun, area)

AtGunWithNightvision(area) forall area \
    if Area(area) and \
        exists gun:
            HasNightvision(gun) and PointOfInterest(gun, area)

AtLoadedGun(area) forall area \
    if Area(area) and \
        exists gun:
            Loaded(gun) and PointOfInterest(gun, area)
```

```
AtLoadedGunWithNightvision(area) forall area \
    if Area(area) and \
        exists gun:
            HasNightvision(gun) and Loaded(gun) and PointOfInterest(gun, area)

AtTool(area) forall area \
    if Area(area) and \
        exists tool:
            Tool(tool) and PointOfInterest(tool, area)

AtCoverpoint(area) forall area \
    if Area(area) and \
        exists coverpoint:
            Coverpoint(coverpoint) and PointOfInterest(coverpoint, area)

HoldingMedikit if exists medikit: Medikit(medikit) and Holding(medikit)

HoldingKnife if exists knife: Knife(knife) and Holding(knife)

HoldingAmmo if exists ammo: Ammo(ammo) and Holding(ammo)

HoldingGun if exists gun: Gun(gun) and Holding(gun)

HoldingGunWithNightvision if exists gun: HasNightvision(gun) and Holding(gun)

HoldingLoadedGun if exists gun: Loaded(gun) and Holding(gun)

HoldingLoadedGunWithNightvision \
    if exists gun: HasNightvision(gun) and Loaded(gun) and Holding(gun)

HoldingTool if exists tool: Tool(tool) and Holding(tool)

# Primitive tasks.

task MoveToPoint(point, area):
    precond:
        Area(area) and PointOfInterest(point, area)
        NotCloseToPoint and not Covered
    add:
        CloseTo(point)
    delete:
        NotCloseToPoint
    cost:
        MoveToPointCost

task MoveToPointFromPoint(point1, point2, area):
    precond:
        Area(area)
        PointOfInterest(point1, area) and PointOfInterest(point2, area)
        CloseTo(point1) and not Covered
```

```
    add:
        CloseTo(point2)
    delete:
        CloseTo(point1)
    cost:
        MoveToPointCost

task Move(area1, area2, waypoint):
    precond:
        Area(area1) and Area(area2) and Waypoint(waypoint)
        Open(waypoint) and  Connected(area1, area2, waypoint)
        (not Covered) and CloseTo(waypoint)
    add:
        At(area2)
        NotCloseToPoint
    delete:
        CloseTo(waypoint)
        At(area1)
    cost:
        MoveCost

task TakeCover(coverpoint, area):
    precond:
        Coverpoint(coverpoint) and Area(area)
        PointOfInterest(coverpoint, area) and CloseTo(coverpoint)
        not Covered
    add:
        Covered
    cost:
        TakeCoverCost

task Uncover:
    precond:
        Covered
    delete:
        Covered
    cost:
        UncoverCost

task MakeAccessible(area1, area2, waypoint, tool):
    precond:
        Waypoint(waypoint) and Tool(tool) and Holding(tool)
        Connected(area1, area2, waypoint) and not Open(waypoint)
        CloseTo(waypoint)
    add:
        Open(waypoint)
    cost:
        MakeAccessibleCost

task PlaceInInventory(item, area):
    precond:
```

```
        Item(item) and Area(area)
        PointOfInterest(item, area) and CloseTo(item)
        (not Covered) and Lighted(area)
    add:
        Holding(item)
        NotCloseToPoint
    delete:
        PointOfInterest(item, area)
        CloseTo(item)
    cost:
        PlaceInInventoryCost

task Reload(gun, ammo):
    precond:
        Gun(gun) and Ammo(ammo)
        not Loaded(gun) and Holding(ammo) and Holding(gun)
    add:
        Loaded(gun)
    delete:
        Holding(ammo)
    cost:
        ReloadCost

task TurnLightOn(area, controlbox):
    precond:
        Area(area) and Controlbox(controlbox)
        CloseTo(controlbox)
        PointOfInterest(controlbox, area) and not Lighted(area)
    add:
        Lighted(area)
    cost:
        TurnLightOnCost

task TurnLightOff(area, controlbox):
    precond:
        Area(area) and Controlbox(controlbox)
        CloseTo(controlbox)
        PointOfInterest(controlbox, area) and Lighted(area)
    delete:
        Lighted(area)
    cost:
        TurnLightOffCost

task UseMedikit(medikit):
    precond:
        Medikit(medikit) and Holding(medikit) and Injured
    delete:
        Injured
        Holding(medikit)
    cost:
        UseMedikitCost
```

```
task AttackMelee(knife, player, area):
    precond:
        At(area) and PointOfInterest(player, area) and Lighted(area)
        Holding(knife) and Knife(knife)
        CloseTo(player) and Player(player)
        not Covered
    add:
        Wounded(player)
    cost:
        AttackMeleeCost

task AttackRanged(gun, player, area):
    precond:
        Lighted(area) and At(area)
        PointOfInterest(player, area) and Player(player)
        Gun(gun) and Loaded(gun) and Holding(gun)
        not Covered
    add:
        Wounded(player)
    delete:
        Loaded(gun)
    cost:
        AttackRangedCost

task AttackRangedFromCover(gun, player, area):
    precond:
        Lighted(area) and At(area)
        PointOfInterest(player, area) and Player(player)
        Gun(gun) and Loaded(gun) and Holding(gun)
        Covered
    add:
        Wounded(player)
    delete:
        Loaded(gun)
    cost:
        AttackRangedFromCoverCost

task SneakKill(gun, player, area):
    precond:
        At(area) and PointOfInterest(player, area)
        Player(player) and Holding(gun)
        HasNightvision(gun) and Loaded(gun)
        not Lighted(area) and (not Covered)
    add:
        Wounded(player)
    delete:
        Loaded(gun)
    cost:
        SneakKillCost
```

```
task SneakKillFromCover(gun, player, area):
    precond:
        At(area) and PointOfInterest(player, area)
        Player(player) and Holding(gun)
        HasNightvision(gun) and Loaded(gun)
        not Lighted(area) and Covered
    add:
        Wounded(player)
    delete:
        Loaded(gun)
    cost:
        SneakKillFromCoverCost

# High-level actions:

task AchieveCloseTo(point, area):
    precond:
        At(area) and not Covered
    add:
        CloseTo(point)
    delete:
        NotCloseToPoint
        CloseTo(point') forall point' if CloseTo(point') and point' != point
    method:
        precond:
            CloseTo(point)
    method:
        precond:
            NotCloseToPoint
        subtasks:
            MoveToPoint(point, area)
    method point':
        precond:
            CloseTo(point') and point' != point
        subtasks:
            MoveToPointFromPoint(point', point, area)
    optimistic:
        0
    pessimistic:
        MoveToPointCost + Epsilon
    level:
        1

task TakeCover(area):
    precond:
        At(area) and AtCoverpoint(area) and not Covered
    add:
        Covered
    delete:
        NotCloseToPoint
        CloseTo(point) forall point \
```

```
            if PointOfInterest(point, area) and not Coverpoint(point)
    abstract:
        CloseTo(coverpoint) forall coverpoint \
            if PointOfInterest(coverpoint, area) and Coverpoint(coverpoint)
    method coverpoint:
        precond:
            PointOfInterest(coverpoint, area) and Coverpoint(coverpoint)
        subtasks:
            AchieveCloseTo(coverpoint, area)
            TakeCover(coverpoint, area)
    optimistic:
        TakeCoverCost
    pessimistic:
        MoveToPointCost + TakeCoverCost + 2 * Epsilon
    level:
        2

task TurnLightOn(area):
    precond:
        Area(area) and not Lighted(area)
    add:
        Lighted(area)
    delete:
        NotCloseToPoint
        CloseTo(controlbox) forall controlbox \
            if PointOfInterest(controlbox, area) and not Controlbox(controlbox)
    abstract:
        CloseTo(controlbox) forall controlbox \
            if PointOfInterest(controlbox, area) and Controlbox(controlbox)
    method controlbox:
        precond:
            Controlbox(controlbox) and PointOfInterest(controlbox, area)
        subtasks:
            AchieveCloseTo(controlbox, area)
            TurnLightOn(area, controlbox)
    optimistic:
        TurnLightOnCost
    pessimistic:
        MoveToPointCost + TurnLightOnCost + 2 * Epsilon
    level:
        2

task TurnLightOff(area):
    precond:
        Area(area) and Lighted(area)
    delete:
        Lighted(area)
        NotCloseToPoint
        CloseTo(controlbox) forall controlbox \
            if PointOfInterest(controlbox, area) and not Controlbox(controlbox)
    abstract:
```

```
        CloseTo(controlbox) forall controlbox \
            if PointOfInterest(controlbox, area) and Controlbox(controlbox)
    method controlbox:
        precond:
            Controlbox(controlbox) and PointOfInterest(controlbox, area)
        subtasks:
            AchieveCloseTo(controlbox, area)
            TurnLightOff(area, controlbox)
    optimistic:
        TurnLightOffCost
    pessimistic:
        MoveToPointCost + TurnLightOffCost  + 2 * Epsilon
    level:
        2

task UseMedikit:
    precond:
        HoldingMedikit and Injured
    delete:
        Injured
    abstract:
     HoldingMedikit
    method medikit:
        precond:
            Medikit(medikit) and Holding(medikit)
        subtasks:
            UseMedikit(medikit)
    optimistic:
        UseMedikitCost
    pessimistic:
        UseMedikitCost + Epsilon
    level:
        2

task ReloadGun:
    precond:
        HoldingGun and HoldingAmmo
    add:
        HoldingLoadedGun
    abstract:
        Holding(ammo) forall ammo if Ammo(ammo) and Holding(ammo)
        Loaded(gun) forall gun if Gun(gun) and Holding(gun)
    method gun, ammo:
        precond:
            Gun(gun) and Ammo(ammo)
            Holding(gun) and Holding(ammo)
        subtasks:
            Reload(gun, ammo)
    optimistic:
        ReloadCost
    pessimistic:
```

```
        ReloadCost + Epsilon
    level:
        2

task ReloadGunWithNightvision:
    precond:
        HoldingGunWithNightvision and HoldingAmmo
    add:
        HoldingLoadedGunWithNightvision
    abstract:
        Holding(ammo) forall ammo if Ammo(ammo) and Holding(ammo)
        Loaded(gun) forall gun if HasNightvision(gun) and Holding(gun)
    method gun, ammo:
        precond:
            HasNightvision(gun) and Ammo(ammo)
            Holding(gun) and Holding(ammo)
        subtasks:
            Reload(gun, ammo)
    optimistic:
        ReloadCost
    pessimistic:
        ReloadCost + Epsilon
    level:
        2

task MakeAccessible(area1, area2, waypoint):
    precond:
        HoldingTool and Connected(area1, area2, waypoint)
        not Open(waypoint)
        CloseTo(waypoint)
    add:
        Open(waypoint)
    method tool:
        precond:
            Tool(tool) and Holding(tool)
        subtasks:
            MakeAccessible(area1, area2, waypoint, tool)
    optimistic:
        MakeAccessibleCost
    pessimistic:
        MakeAccessibleCost + Epsilon
    level:
        2

task AttackMelee(player, area):
    precond:
        Player(player) and At(area) and PointOfInterest(player, area)
        HoldingKnife and Lighted(area) and not Covered
    add:
        Wounded(player)
        CloseTo(player)
```

```
    delete:
        NotCloseToPoint
    abstract:
        CloseTo(poi) forall poi if PointOfInterest(poi, area) and poi != player
    method knife:
        precond:
            Knife(knife) and Holding(knife)
        subtasks:
            AchieveCloseTo(player, area)
            AttackMelee(knife, player, area)
    optimistic:
        AttackMeleeCost
    pessimistic:
        MoveToPointCost + AttackMeleeCost + 2 * Epsilon
    level:
        2

task AttackRanged(player, area):
    precond:
        Player(player) and At(area) and PointOfInterest(player, area)
        HoldingLoadedGun and Lighted(area) and not Covered
    add:
        Wounded(player)
    abstract:
        Loaded(gun) forall gun if Gun(gun) and Holding(gun)
    method gun:
        precond:
            Gun(gun) and Holding(gun) and Loaded(gun)
        subtasks:
            AttackRanged(gun, player, area)
    optimistic:
        AttackRangedCost
    pessimistic:
        AttackRangedCost + Epsilon
    level:
        2

task AttackRangedFromCover(player, area):
    precond:
        Player(player) and At(area) and PointOfInterest(player, area)
        HoldingLoadedGun and Lighted(area) and Covered
    add:
        Wounded(player)
    abstract:
        Loaded(gun) forall gun if Gun(gun) and Holding(gun)
    method gun:
        precond:
            Gun(gun) and Holding(gun) and Loaded(gun)
        subtasks:
            AttackRangedFromCover(gun, player, area)
    optimistic:
```

```
        AttackRangedFromCoverCost
    pessimistic:
        AttackRangedFromCoverCost + Epsilon
    level:
        2

task SneakKill(player, area):
    precond:
        Player(player) and At(area) and PointOfInterest(player, area)
        HoldingLoadedGunWithNightvision and not Lighted(area) and not Covered
    add:
        Wounded(player)
    abstract:
        Loaded(gun) forall gun if HasNightvision(gun) and Holding(gun)
    method gun:
        precond:
            HasNightvision(gun) and Holding(gun) and Loaded(gun)
        subtasks:
            SneakKill(gun, player, area)
    optimistic:
        SneakKillCost
    pessimistic:
        SneakKillCost + Epsilon
    level:
        2

task SneakKillFromCover(player, area):
    precond:
        Player(player) and At(area) and PointOfInterest(player, area)
        HoldingLoadedGunWithNightvision and not Lighted(area) and Covered
    add:
        Wounded(player)
    abstract:
        Loaded(gun) forall gun if HasNightvision(gun) and Holding(gun)
    method gun:
        precond:
            HasNightvision(gun) and Holding(gun) and Loaded(gun)
        subtasks:
            SneakKillFromCover(gun, player, area)
    optimistic:
        SneakKillFromCoverCost
    pessimistic:
        SneakKillFromCoverCost + Epsilon
    level:
        2

task PlaceMedikitInInventory(area):
    precond:
        AtMedikit(area) and At(area) and not Covered
    add:
        HoldingMedikit
```

```
    abstract:
        PointOfInterest(medikit, area) forall medikit \
            if Medikit(medikit) and PointOfInterest(medikit, area)
        Holding(medikit) forall medikit \
            if Medikit(medikit) and PointOfInterest(medikit, area)
    method medikit:
        precond:
            Medikit(medikit) and PointOfInterest(medikit, area)
        subtasks:
            AchieveCloseTo(medikit, area)
            PlaceInInventory(medikit, area)
    optimistic:
        PlaceInInventoryCost
    pessimistic:
        MoveToPointCost + PlaceInInventoryCost + 2 * Epsilon
    level:
        2

task PlaceAmmoInInventory(area):
    precond:
        AtAmmo(area) and At(area) and not Covered
    add:
        HoldingAmmo
    abstract:
        PointOfInterest(ammo, area) forall ammo \
            if Ammo(ammo) and PointOfInterest(ammo, area)
        Holding(ammo) forall ammo \
            if Ammo(ammo) and PointOfInterest(ammo, area)
    method ammo:
        precond:
            Ammo(ammo) and PointOfInterest(ammo, area)
        subtasks:
            AchieveCloseTo(ammo, area)
            PlaceInInventory(ammo, area)
    optimistic:
        PlaceInInventoryCost
    pessimistic:
        MoveToPointCost + PlaceInInventoryCost + 2 * Epsilon
    level:
        2

task PlaceKnifeInInventory(area):
    precond:
        AtKnife(area) and At(area) and not Covered
    add:
        HoldingKnife
    abstract:
        PointOfInterest(knife, area) forall knife \
            if Knife(knife) and PointOfInterest(knife, area)
        Holding(knife) forall knife \
            if Knife(knife) and PointOfInterest(knife, area)
```

```
    method knife:
        precond:
            Knife(knife) and PointOfInterest(knife, area)
        subtasks:
            AchieveCloseTo(knife, area)
            PlaceInInventory(knife, area)
    optimistic:
        PlaceInInventoryCost
    pessimistic:
        MoveToPointCost + PlaceInInventoryCost + 2 * Epsilon
    level:
        2

task PlaceGunInInventory(area):
    precond:
        AtGun(area) and At(area) and not Covered
    add:
        HoldingGun
    abstract:
        PointOfInterest(gun, area) forall gun \
            if Gun(gun) and PointOfInterest(gun, area)
        Holding(gun) forall gun \
            if Gun(gun) and PointOfInterest(gun, area)
    method gun:
        precond:
            Gun(gun) and PointOfInterest(gun, area)
        subtasks:
            AchieveCloseTo(gun, area)
            PlaceInInventory(gun, area)
    optimistic:
        PlaceInInventoryCost
    pessimistic:
        MoveToPointCost + PlaceInInventoryCost + 2 * Epsilon
    level:
        2

task PlaceToolInInventory(area):
    precond:
        AtTool(area) and At(area) and not Covered
    add:
        HoldingTool
    abstract:
        PointOfInterest(tool, area) forall tool \
            if Tool(tool) and PointOfInterest(tool, area)
        Holding(tool) forall tool \
            if Tool(tool) and PointOfInterest(tool, area)
    method tool:
        precond:
            Tool(tool) and PointOfInterest(tool, area)
        subtasks:
            AchieveCloseTo(tool, area)
```

```
            PlaceInInventory(tool, area)
    optimistic:
        PlaceInInventoryCost
    pessimistic:
        MoveToPointCost + PlaceInInventoryCost + 2 * Epsilon
    level:
        2

task PlaceGunWithNightvisionInInventory(area):
    precond:
        AtGunWithNightvision(area) and At(area) and not Covered
    add:
        HoldingGunWithNightvision
    abstract:
        PointOfInterest(gun, area) forall gun \
            if HasNightvision(gun) and PointOfInterest(gun, area)
        Holding(gun) forall gun \
            if HasNightvision(gun) and PointOfInterest(gun, area)
    method gun:
        precond:
            HasNightvision(gun) and PointOfInterest(gun, area)
        subtasks:
            AchieveCloseTo(gun, area)
            PlaceInInventory(gun, area)
    optimistic:
        PlaceInInventoryCost
    pessimistic:
        MoveToPointCost + PlaceInInventoryCost + 2 * Epsilon
    level:
        2

task PlaceLoadedGunInInventory(area):
    precond:
        AtLoadedGun(area) and At(area) and not Covered
    add:
        HoldingLoadingGun
    abstract:
        PointOfInterest(gun, area) forall gun \
            if Loaded(gun) and PointOfInterest(gun, area)
        Holding(gun) forall gun \
            if Loaded(gun) and PointOfInterest(gun, area)
    method gun:
        precond:
            Loaded(gun) and PointOfInterest(gun, area)
        subtasks:
            AchieveCloseTo(gun, area)
            PlaceInInventory(gun, area)
    optimistic:
        PlaceInInventoryCost
    pessimistic:
        MoveToPointCost + PlaceInInventoryCost + 2 * Epsilon
```

```
    level:
        2

task PlaceLoadedGunWithNightvisionInInventory(area):
    precond:
        AtLoadedGunWithNightvision(area) and At(area) and not Covered
    add:
        HoldingLoadedGunWithNightvision
    abstract:
        PointOfInterest(gun, area) forall gun \
            if HasNightvision(gun) and Loaded(gun) and \
                PointOfInterest(gun, area)
        Holding(gun) forall gun \
            if HasNightvision(gun) and Loaded(gun) and \
                PointOfInterest(gun, area)
    method gun:
        precond:
            HasNightvision(gun) and Loaded(gun) and PointOfInterest(gun, area)
        subtasks:
            AchieveCloseTo(gun, area)
            PlaceInInventory(gun, area)
    optimistic:
        PlaceInInventoryCost
    pessimistic:
        MoveToPointCost + PlaceInInventoryCost + 2 * Epsilon
    level:
        2

task AchieveUncovered:
    delete:
        Covered
    method:
        precond:
            not Covered
    method:
        precond:
            Covered
        subtasks:
            Uncover
    optimistic:
        0
    pessimistic:
        UncoverCost + Epsilon
    level:
        3

task AchieveCovered(area):
    precond:
        At(area) and AtCoverpoint(area)
    add:
        Covered
```

```
    delete:
        NotCloseToPoint
        CloseTo(point) forall point \
            if PointOfInterest(point, area) and not Coverpoint(point)
    abstract:
        CloseTo(coverpoint) forall coverpoint \
            if PointOfInterest(coverpoint, area) and Coverpoint(coverpoint)
    method:
        precond:
            Covered
    method:
        precond:
            not Covered
        subtasks:
            TakeCover(area)
    optimistic:
        0
    pessimistic:
        MoveToPointCost + TakeCoverCost + 3 * Epsilon
    level:
        3

task AchieveLighted(area):
    precond:
        At(area)
    add:
        Lighted(area)
    delete:
        Covered if not Lighted(area)
    abstract:
        NotCloseToPoint
        CloseTo(point) forall point if PointOfInterest(point, area)
    method:
        precond:
            Lighted(area)
    method controlbox:
        precond:
            not Lighted(area)
        subtasks:
            AchieveUncovered
            TurnLightOn(area)
    optimistic:
        0
    pessimistic:
        UncoverCost + MoveToPointCost + TurnLightOnCost + 4 * Epsilon
    level:
        4

task AchieveDark(area):
    precond:
        At(area)
```

```
    delete:
        Lighted(area)
        Covered if not Lighted(area)
    abstract:
        NotCloseToPoint
        CloseTo(point) forall point if PointOfInterest(point, area)
    method:
        precond:
            not Lighted(area)
    method controlbox:
        precond:
            Lighted(area)
        subtasks:
            AchieveUncovered
            TurnLightOff(area)
    optimistic:
        0
    pessimistic:
        UncoverCost + MoveToPointCost + TurnLightOffCost + 4 * Epsilon
    level:
        4

task Navigate(area1, area2, depth):
    precond:
        At(area1) and not Covered
    add:
        At(area2)
    delete:
        At(area1) if area1 != area2
        CloseTo(point) forall point \
            if CloseTo(point) and not PointOfInterest(point, area2)
    abstract:
        NotCloseToPoint
        Open(waypoint) forall waypoint \
            if Waypoint(waypoint) and not Open(waypoint)
        PointOfInterest(tool, area) forall tool, area \
            if Tool(tool) and PointOfInterest(tool, area)
        Holding(tool) forall tool if Tool(tool) and not Holding(tool)
        Lighted(area) forall area, waypoint \
            if Waypoint(waypoint) and PointOfInterest(waypoint, area) and \
               not Open(waypoint)
    method:
        precond:
            area1 == area2
    method area', waypoint:
        precond:
            area1 != area2 and depth < MaxDistance
            Connected(area', area2, waypoint) and Open(waypoint)
        subtasks:
            Navigate(area1, area', depth + 1)
            AchieveCloseTo(waypoint, area')
```

```
            Move(area', area2, waypoint)
    method area', waypoint:
        precond:
            area1 != area2 and depth < MaxDistance
            Connected(area', area2, waypoint) and not Open(waypoint)
            HoldingTool
        subtasks:
            Navigate(area1, area', depth + 1)
            AchieveCloseTo(waypoint, area')
            MakeAccessible(area', area2, waypoint)
            Move(area', area2, waypoint)
    method area', area'', waypoint:
        precond:
            area1 != area2 and depth < MaxDistance
            Connected(area', area2, waypoint) and not Open(waypoint)
            not HoldingTool and AtTool(area'')
        subtasks:
            Navigate(area1, area'', depth + 1)
            AchieveLighted(area'')
            PlaceToolInInventory(area'')
            Navigate(area'', area', depth + 1)
            AchieveCloseTo(waypoint, area')
            MakeAccessible(area', area2, waypoint)
            Move(area', area2, waypoint)
    optimistic:
        MoveCost
    pessimistic:
        (MaxDistance - depth) * (MoveToPointCost + MoveCost + 2 * Epsilon)
    level:
        5

task AchieveAt(area):
    add:
        At(area)
    abstract:
        NotCloseToPoint
        CloseTo(point) forall point if PointOfInterest(point, area)
        Open(waypoint) forall waypoint \
            if Waypoint(waypoint) and not Open(waypoint)
        PointOfInterest(tool, area') forall tool, area' \
            if Tool(tool) and PointOfInterest(tool, area')
        Holding(tool) forall tool if Tool(tool) and not Holding(tool)
        Lighted(area') forall area', waypoint \
            if Waypoint(waypoint) and PointOfInterest(waypoint, area') and \
               not Open(waypoint)
        Covered if Covered
    method:
        precond:
            At(area)
    method area':
        precond:
```

```
            At(area') and area' != area
        subtasks:
            AchieveUncovered
            Navigate(area', area, 0)
    optimistic:
        0
    pessimistic:
        MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
        UncoverCost + 2 * Epsilon
    level:
        6

task AchieveHoldingMedikit:
    add:
        HoldingMedikit
    abstract:
        PointOfInterest(medikit, area) forall medikit, area \
            if Medikit(medikit) and PointOfInterest(medikit, area)
        Holding(medikit) forall medikit if Medikit(medikit)
        At(area) forall area if AtMedikit(area)
        Ligthed(area) forall area if Area(area) and not Lighted(area)
        NotCloseToPoint
        CloseTo(point) forall point, area if PointOfInterest(point, area)
        Covered if Covered
    method:
        precond:
            HoldingMedikit
    method area:
        precond:
            not HoldingMedikit and AtMedikit(area)
        subtasks:
            AchieveAt(area)
            AchieveLighted(area)
            AchieveUncovered
            PlaceMedikitInInventory(area)
    optimistic:
        0
    pessimistic:
        MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
        UncoverCost + 2 * Epsilon + \
        MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
        UncoverCost + Epsilon + \
        MoveToPointCost + PlaceInInventoryCost + 3 * Epsilon
    level:
        7

task AchieveHoldingKnife:
    add:
        HoldingKnife
    abstract:
        PointOfInterest(knife, area) forall knife, area \
```

```
            if Knife(knife) and PointOfInterest(knife, area)
        Holding(knife) forall knife if Knife(knife)
        At(area) forall area if AtKnife(area)
        Ligthed(area) forall area if Area(area) and not Lighted(area)
        NotCloseToPoint
        CloseTo(point) forall point, area if PointOfInterest(point, area)
        Covered if Covered
    method:
        precond:
            HoldingKnife
    method area:
        precond:
            not HoldingKnife and AtKnife(area)
        subtasks:
            AchieveAt(area)
            AchieveLighted(area)
            AchieveUncovered
            PlaceKnifeInInventory(area)
    optimistic:
        0
    pessimistic:
        MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
        UncoverCost + 2 * Epsilon + \
        MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
        UncoverCost + Epsilon + \
        MoveToPointCost + PlaceInInventoryCost + 3 * Epsilon
    level:
        7

task AchieveHoldingAmmo:
    add:
        HoldingAmmo
    abstract:
        PointOfInterest(ammo, area) forall ammo, area \
            if Ammo(ammo) and PointOfInterest(ammo, area)
        Holding(ammo) forall ammo if Ammo(ammo)
        At(area) forall area if AtAmmo(area)
        Ligthed(area) forall area if Area(area) and not Lighted(area)
        NotCloseToPoint
        CloseTo(point) forall point, area if PointOfInterest(point, area)
        Covered if Covered
    method:
        precond:
            HoldingAmmo
    method area:
        precond:
            not HoldingAmmo and AtAmmo(area)
        subtasks:
            AchieveAt(area)
            AchieveLighted(area)
            AchieveUncovered
```

```
            PlaceAmmoInInventory(area)
    optimistic:
        0
    pessimistic:
        MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
        UncoverCost + 2 * Epsilon + \
        MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
        UncoverCost + Epsilon + \
        MoveToPointCost + PlaceInInventoryCost + 3 * Epsilon
    level:
        7

task AchieveHoldingGun:
    add:
        HoldingGun
    abstract:
        PointOfInterest(gun, area) forall gun, area \
            if Gun(gun) and PointOfInterest(gun, area)
        Holding(gun) forall gun if Gun(gun)
        At(area) forall area if AtGun(area)
        Ligthed(area) forall area if Area(area) and not Lighted(area)
        NotCloseToPoint
        CloseTo(point) forall point, area if PointOfInterest(point, area)
        Covered if Covered
    method:
        precond:
            HoldingGun
    method area:
        precond:
            not HoldingGun and AtGun(area)
        subtasks:
            AchieveAt(area)
            AchieveLighted(area)
            AchieveUncovered
            PlaceGunInInventory(area)
    optimistic:
        0
    pessimistic:
        MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
        UncoverCost + 2 * Epsilon + \
        MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
        UncoverCost + Epsilon + \
        MoveToPointCost + PlaceInInventoryCost + 3 * Epsilon
    level:
        7

task AchieveHoldingGunWithNightvision:
    add:
        HoldingGun
        HoldingGunWithNightvision
    abstract:
```

```
        PointOfInterest(gun, area) forall gun, area \
            if HasNightvision(gun) and PointOfInterest(gun, area)
        Holding(gun) forall gun if HasNightvision(gun)
        At(area) forall area if AtGunWithNightvision(area)
        Ligthed(area) forall area if Area(area) and not Lighted(area)
        NotCloseToPoint
        CloseTo(point) forall point, area if PointOfInterest(point, area)
        Covered if Covered
    method:
        precond:
            HoldingGunWithNightvision
    method area:
        precond:
            not HoldingGunWithNightvision and AtGunWithNightvision(area)
        subtasks:
            AchieveAt(area)
            AchieveLighted(area)
            AchieveUncovered
            PlaceGunWithNightvisionInInventory(area)
    optimistic:
        0
    pessimistic:
        MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
        UncoverCost + 2 * Epsilon + \
        MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
        UncoverCost + Epsilon + \
        MoveToPointCost + PlaceInInventoryCost + 3 * Epsilon
    level:
        7

task AchieveHoldingLoadedGun:
    add:
        HoldingLoadedGun
    abstract:
        PointOfInterest(item, area) forall item, area \
            if Ammo(item) or Loaded(item) and PointOfInterest(item, area)
        Holding(item) forall item if Ammo(item) or Gun(item)
        Loaded(gun) forall gun if Gun(gun) and not Loaded(gun)
        At(area) forall area if Area(area)
        Ligthed(area) forall area if Area(area) and not Lighted(area)
        NotCloseToPoint
        CloseTo(point) forall point, area if PointOfInterest(point, area)
        Covered if Covered
    method:
        precond:
            HoldingLoadedGun
    method area:
        precond:
            not HoldingLoadedGun and AtLoadedGun(area)
        subtasks:
            AchieveAt(area)
```

```
            AchieveLighted(area)
            AchieveUncovered
            PlaceLoadedGunInInventory(area)
    method:
        precond:
            not HoldingLoadedGun
        subtasks:
            AchieveHoldingGun
            AchieveHoldingAmmo
            ReloadGun
    optimistic:
        0
    pessimistic:
        2 * (
            MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
            UncoverCost + 2 * Epsilon + \
            MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
            UncoverCost + Epsilon + \
            MoveToPointCost + PlaceInInventoryCost + 3 * Epsilon) + \
        ReloadCost + 2 * Epsilon
    level:
        7

task AchieveHoldingLoadedGunWithNightvision:
    add:
        HoldingLoadedGun
        HoldingLoadedGunWithNightvision
    abstract:
        PointOfInterest(item, area) forall item, area \
            if Ammo(item) or (Loaded(item) and HasNightvision(item)) and \
               PointOfInterest(item, area)
        Holding(item) forall item if Ammo(item) or HasNightvision(item)
        Loaded(gun) forall gun if HasNightvision(gun) and not Loaded(gun)
        At(area) forall area if Area(area)
        Ligthed(area) forall area if Area(area) and not Lighted(area)
        NotCloseToPoint
        CloseTo(point) forall point, area if PointOfInterest(point, area)
        Covered if Covered
    method:
        precond:
            HoldingLoadedGunWithNightvision
    method area:
        precond:
            not HoldingLoadedGunWithNightvision
            AtLoadedGunWithNightvision(area)
        subtasks:
            AchieveAt(area)
            AchieveLighted(area)
            AchieveUncovered
            PlaceLoadedGunWithNightvisionInInventory(area)
    method:
```

```
        precond:
            not HoldingLoadedGunWithNightvision
        subtasks:
            AchieveHoldingGunWithNightvision
            AchieveHoldingAmmo
            ReloadGunWithNightvision
    optimistic:
        0
    pessimistic:
        2 * (
            MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
            UncoverCost + 2 * Epsilon + \
            MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
            UncoverCost + Epsilon + \
            MoveToPointCost + PlaceInInventoryCost + 3 * Epsilon) + \
        ReloadCost + 2 * Epsilon
    level:
        7

task AchieveFullHealth:
    delete:
        Injured
    abstract:
        PointOfInterest(medikit, area) forall medikit, area \
            if Medikit(medikit) and PointOfInterest(medikit, area)
        Holding(medikit) forall medikit \
            if Medikit(medikit) and Holding(medikit)
        At(area) forall area if Area(area)
        Ligthed(area) forall area if Area(area) and not Ligthed(area)
        NotCloseToPoint
        CloseTo(point) forall point, area if PointOfInterest(point, area)
        Covered if Covered
    method:
        precond:
            not Injured
    method:
        precond:
            Injured
        subtasks:
            AchieveHoldingMedikit
            UseMedikit
    optimistic:
        0
    pessimistic:
        MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
        UncoverCost + 2 * Epsilon + \
        MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
        UncoverCost + Epsilon + \
        MoveToPointCost + PlaceInInventoryCost + 3 * Epsilon + \
        UseMedikitCost + 2 * Epsilon
    level:
```

```
        8

task AchieveWounded(player):
    add:
        Wounded(player)
    abstract:
        PointOfInterest(item, area) forall item, area \
            if Item(item) and not Medikit(item) and PointOfInterest(item, area)
        Holding(item) forall item if Item(item) and not Medikit(item)
        At(area) forall area if Area(area)
        Ligthed(area) forall area if Area(area)
        NotCloseToPoint
        CloseTo(point) forall point, area if PointOfInterest(point, area)
        Covered
    method:
        precond:
            Wounded(player)
    method area:
        precond:
            not Wounded(player) and PointOfInterest(player, area)
            AtCoverpoint(area)
        subtasks:
            AchieveHoldingLoadedGunWithNightvision
            AchieveAt(area)
            AchieveDark(area)
            AchieveCovered(area)
            SneakKillFromCover(player, area)
    method area:
        precond:
            not Wounded(player) and PointOfInterest(player, area)
            not AtCoverpoint(area)
        subtasks:
            AchieveHoldingLoadedGunWithNightvision
            AchieveAt(area)
            AchieveDark(area)
            AchieveUncovered
            SneakKill(player, area)
    method area:
        precond:
            not Wounded(player) and PointOfInterest(player, area)
            AtCoverpoint(area)
        subtasks:
            AchieveHoldingLoadedGun
            AchieveAt(area)
            AchieveLighted(area)
            AchieveCovered(area)
            AttackRangedFromCover(player, area)
    method area:
        precond:
            not Wounded(player) and PointOfInterest(player, area)
            not AtCoverpoint(area)
```

```
    subtasks:
        AchieveHoldingLoadedGun
        AchieveAt(area)
        AchieveLighted(area)
        AchieveUncovered
        AttackRanged(player, area)
method area:
    precond:
        not Wounded(player) and PointOfInterest(player, area)
    subtasks:
        AchieveHoldingKnife
        AchieveAt(area)
        AchieveLighted(area)
        AchieveUncovered
        AttackMelee(player, area)
optimistic:
    0
pessimistic:
    """A level always contains a knife, so we can assess that a
    pessimistic solution consist in getting a knife and attacking
    the player in melee."""
    # Cost of getting a knife.
    MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
    UncoverCost + 2 * Epsilon + \
    MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
    UncoverCost + Epsilon + \
    MoveToPointCost + PlaceInInventoryCost + 3 * Epsilon + \
    # Cost of travelling to player area.
    MaxDistance * (MoveToPointCost + MoveCost + 2 * Epsilon) + \
    UncoverCost + 2 * Epsilon + \
    # Cost of lighting player area.
    MoveToPointCost + TurnLightOnCost + 3 * Epsilon + \
    # Cost of uncovering.
    UncoverCost + Epsilon + \
    # Cost of attacking player in melee
    MoveToPointCost + AttackMeleeCost + 2 * Epsilon + \
    # Add Epsilon to be sure that the cost will be stricly decreasing.
    Epsilon
level:
    8
```

Bibliographie

[1] ISO/IEC 14977 : 1996(E). Information technology – Syntactic metalanguage – Extended BNF . Technical report, International Organization for Standardization, Geneva, CH, 1996.

[2] Ron Alford, Pascal Bercher, and David W. Aha. Tight Bounds for HTN Planning. In *International Conference on Automated Planning and Scheduling*, pages 7–15. AAAI Press, 2015.

[3] Ronald Alford, Vikas Shivashankar, Ugur Kuter, and Dana S Nau. HTN Problem Spaces : Structure, Algorithms, Termination. In *SOCS*, 2012.

[4] John S. Anderson and Arthur M. Farley. Plan Abstraction Based on Operator Generalization. In Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith, editors, *AAAI*, pages 100–104. AAAI Press / The MIT Press, 1988.

[5] Fahiem Bacchus and Qiang Yang. The Downward Refinement Property. In John Mylopoulos and Raymond Reiter, editors, *IJCAI*, pages 286–293. Morgan Kaufmann, 1991.

[6] Fahiem Bacchus and Qiang Yang. Downward Refinement and the Efficiency of Hierarchical Problem Solving . *Artificial Intelligence*, 71(1) :43–100, 1994.

[7] Merrie Bergmann. *An Introduction to Many-valued and Fuzzy Logic : Semantics, Algebras, and Derivation Systems*. Cambridge University Press, 2008.

[8] M Bienkowski, ME des Jardins, and RV Desimone. SOCAP : System for Operations Crisis Action Planning. *Advanced Planning Technology*, pages 70–76, 1996.

[9] Susanne Biundo and Bernd Schattenberg. From Abstract Crisis to Concrete Relief A Preliminary Report on Combining State Abstraction and HTN Planning . In *Sixth European Conference on Planning*, 2001.

[10] Blai Bonet and Hector Geffner. Planning as Heuristic Search. *Artif. Intell.*, 129(1-2) :5–33, 2001.

[11] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near Optimal Hierarchical Path-finding. *Journal of game development*, 1(1) :7–28, 2004.

[12] Jaime Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Craig Knoblock, Steve Minton, and Manuela Veloso. Prodigy : An Integrated Architecture for Planning and Learning. *ACM SIGART Bulletin*, 2(4) :51–55, 1991.

[13] Luis A. Castillo, Juan Fernández-Olivares, Óscar García-Pérez, and Francisco Palao. Efficiently Handling Temporal Knowledge in an HTN Planner. In *ICAPS*, pages 63–72. AAAI, 2006.

[14] David Chapman. Planning for Conjunctive Goals. *Artificial intelligence*, 32(3) :333–377, 1987.

[15] Ken Currie and Austin Tate. O-Plan : the Open Planning Architecture. *Artif. Intell.*, 52(1) :49–86, 1991.

[16] Thomas L. Dean and Mark S. Boddy. An Analysis of Time-Dependent Planning. In *AAAI*, volume 88, pages 49–54, 1988.

[17] Roberto Desimone. SOCAP : Lessons Learned in Applying SIPE-2 to the Military Operations Crisis Action Planning Domain . In *Proceedings of the Spring Symposium on Practical Approaches to Scheduling and Planning*, pages 156–159, 1992.

[18] Filip Dvorak, Arthur Bit-Monnot, Félix Ingrand, and Malik Ghallab. A Flexible ANML Actor and Planner in Robotics. In *Planning and Robotics (PlanRob) Workshop (ICAPS)*, Portsmouth, United States, 2014.

[19] Stefan Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, pages 13–34, 2001.

[20] Kutluhan Erol. *Hierarchical Task Network Planning Formalization Analysis and Implementation* . PhD thesis, University of Maryland, 1996.

[21] Kutluhan Erol, James Hendler, and Dana S Nau. HTN Planning : Complexity and Expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.

[22] Kutluhan Erol, James A Hendler, and Dana S Nau. UMCP : A Sound and Complete Procedure for Hierarchical Task-Network Planning . In *AIPS*, volume 94, pages 249–254, 1994.

[23] Kutluhan Erol, James A Hendler, and Dana S Nau. Semantics for Hierarchical Task-network Planning. Technical report, DTIC Document, 1995.

[24] Kutluhan Erol, James A. Hendler, and Dana S. Nau. Complexity Results for HTN Planning. *Annals of Mathematics and Artificial Intelligence*, 18(1) :69–93, 1996.

[25] Kutluhan Erol, Dana S Nau, and Venkatramana S Subrahmanian. Complexity, Decidability and Undecidability Results for Domain-Independent Planning . *Artificial Intelligence*, 76(1) :75–88, 1995.

[26] Kutluhan Erol, Dana S Nau, and VS Subrahmanian. On the Complexity of Domain-Independent Planning. In *AAAI*, volume 1, pages 381–386, 1992.

[27] Juan A. Fernandez and Javier Gonzalez. Hierarchical Graph Search for Mobile Robot Path Planning. In *ICRA*, pages 656–661. IEEE Computer Society, 1998.

[28] Juan-Antonio Fernandez-Madrigal and Javier Gonzalez. Multihierarchical Graph Search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(1) :103–113, 2002.

[29] Richard E Fikes and Nils J Nilsson. STRIPS : A New Approach to the Application of Theorem Proving to Problem Solving . *Artificial intelligence*, 2(3) :189–208, 1972.

[30] Cipriano Galindo, Juan-Antonio Fernandez-Madrigal, and Javier González. Improving Efficiency in Mobile Robot Task Planning Through World Abstraction . *IEEE Trans. Robotics*, 20(4) :677–690, 2004.

[31] Peter Gorniak and Ian Davis. SquadSmart : Hierarchical Planning and Coordinated Plan Execution for Squads of Characters . In *AIIDE*, pages 14–19, 2007.

[32] Christophe Guettier. Solving Planning and Scheduling Problems in Network Based Operations. In *Principles and Practice of Constraint Programming CP*, volume 7, 2007.

[33] Eric A. Hansen and Rong Zhou. Anytime Heuristic Search. *J. Artif. Intell. Res.(JAIR)*, 28 :267–297, 2007.

[34] P.E. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2) :100–107, 1968.

[35] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning . In *AAAI*, pages 1007–1012, 2007.

[36] Nick Hawes. Real Time Goal Oriented Behaviour for Computer Game Agents. In *GAME-ON*, 2000.

[37] Malte Helmert, Patrik Haslum, and Jörg Hoffmann 0001. Explicit-State Abstraction : A New Method for Generating Heuristic Functions . In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 1547–1550. AAAI Press, 2008.

[38] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible Abstraction Heuristics for Optimal Sequential Planning . In *ICAPS*, pages 176–183, 2007.

[39] Hai Hoang. Planning to Coordinate : Using HTN to Coordinate Unreal Tournament Bots , 2005.

[40] Hai Hoang, Stephen Lee-Urban, and Héctor Muñoz-Avila. Hierarchical Plan Representations for Encoding Strategic Game AI . In *AIIDE*, 2005.

[41] Robert C Holte and Berthe Y Choueiry. Abstraction and Reformulation in Artificial Intelligence. *Philosophical Transactions of the Royal Society of London B : Biological Sciences*, 358(1435) :1197–1204, 2003.

[42] Robert C Holte, Chris Drummond, Maria B Perez, Robert M Zimmer, and Alan J MacDonald. Searching with Abstractions : A Unifying Framework and New High-Performance Algorithm . In *Proceedings of the 10th Canadian Conference on Artificial Intelligence*, pages 263–270, 1994.

[43] Robert C. Holte, Jeffery Grajkowski, and Brian Tanner. Hierarchical Heuristic Search Revisited. In *SARA*, volume 3607 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2005.

[44] Robert C. Holte, T. Mkadmi, Robert M. Zimmer, and Alan J. MacDonald. Speeding Up Problem Solving by Abstraction : A Graph Oriented Approach . *Artificial Intelligence*, 85(1-2) :321–361, 1996.

[45] Robert C Holte, Maria B Perez, Robert M Zimmer, and Alan J MacDonald. Hierarchical A* : Searching Abstraction Hierarchies Efficiently. In *AAAI/IAAI, Vol. 1*, pages 530–535, 1996.

[46] Troy Humphreys. Exploring HTN Planners Through Example. *Game AI Pro : Collected Wisdom of Game AI Professionals*, page 149, 2013.

[47] Okhtay Ilghami. Documentation for JSHOP2. Technical report, Technical Report CS-TR-4694, Department of Computer Science, University of Maryland, College Park, MD 20742 USA, 2006.

[48] Okhtay Ilghami and Dana S Nau. A General Approach to Synthesize Problem-Specific Planners. Technical report, DTIC Document, 2003.

[49] Éric Jacopin. Game AI Planning Analytics : The Case of Three First-Person Shooters. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[50] Subbarao Kambhampati and James A. Hendler. A Validation-Structure-Based Theory of Plan Modification and Reuse . *Artif. Intell.*, 55(2) :193–258, 1992.

[51] Craig A Knoblock. A Theory of Abstraction for Hierarchical Planning. In *Change of Representation and Inductive Bias*, pages 81–104. Springer, 1990.

[52] Craig A Knoblock. Abstracting the Tower of Hanoi. In *Proceedings of the Workshop on Automatic Generation of Approximations and Abstractions*, pages 13–23, 1990.

[53] Craig A. Knoblock. Learning Abstraction Hierarchies for Problem Solving. In *AAAI*, pages 923–928. AAAI Press / The MIT Press, 1990.

[54] Craig A Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, Carnegie Mellon University, 1991.

[55] Craig A. Knoblock. Search Reduction in Hierarchical Problem Solving. In Thomas L. Dean and Kathleen McKeown, editors, *AAAI*, pages 686–691. AAAI Press / The MIT Press, 1991.

[56] Richard E. Korf. Depth-first Iterative-deepening : An Optimal Admissible Tree Search . *Artif. Intell.*, 27(1) :97–109, September 1985.

[57] Raphaël Lallement, Lavindra de Silva, and Rachid Alami. HATP : An HTN Planner for Robotics. In *Planning and Robotics Workshop*, 2014.

[58] Charles Madeira. *Agents adaptatifs dans les jeux de stratégie modernes : une approche fondée sur l'apprentissage par renforcement*. PhD thesis, Université Paris 6, 2007.

[59] Bhaskara Marthi, Stuart Russell, and Jason Wolfe. Angelic Hierarchical Planning : Optimal and Online Algorithms (Revised) . Technical report, Technical Report UCB/EECS-2009-122, EECS Department, University of California, Berkeley, 2009.

[60] Bhaskara Marthi, Stuart J Russell, and Jason Wolfe. Angelic Semantics for High-Level Actions. In *ICAPS*, pages 232–239, 2007.

[61] David McAllester and David Rosenblatt. Systematic Nonlinear Planning. 1991.

[62] John McCarthy and Patrick J Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence . *Readings in artificial intelligence*, pages 431–450, 1969.

[63] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - The Planning Domain Definition Language. Technical Report TR-98-003, Yale Center for Computational Vision and Control,, 1998.

[64] Drew McDermott. Using Regression-Match Graphs to Control Search in Planning. *Artificial Intelligence*, 109(1) :111–159, 1999.

[65] Alexandre Menif, Christophe Guettier, and Tristan Cazenave. Planning and Execution Control Architecture for Infantry Serious Gaming . In *Planning in Games Workshop*, volume 31, 2013.

[66] Alexandre Menif, Éric Jacopin, and Tristan Cazenave. SHPE : HTN Planning for Video Games. In *Computer Games*, pages 119–132. Springer, 2014.

[67] Ministère de la défense. *Le manuel d'emploi de la section d'infanterie*, 1999.

[68] Ministère de la défense, École de l'infanterie. *Manuel d'emploi du groupe de combat d'infanterie en zone urbaine*, 2010.

[69] Héctor Muñoz-Avila, David W. Aha, Ulit Jaidee, Matthew Klenk, and Matthew Molineaux. Applying Goal Driven Autonomy to a Team Shooter Game. In *Proceedings of the Twenty-Third International Florida Artificial Intelligence Research Society Conference*, 2010.

[70] Héctor Munoz-Avila and Todd Fisher. Strategic Planning for Unreal Tournament Bots. In *AAAI Workshop on Challenges in Game AI*, 2004.

[71] D. S. Nau, D. W. Aha, and H. Muñoz-Avila. Ordered Task Decomposition. In *AAAI Workshop on Representational Issues for Real-World Planning Systems*. AAAI Press, 2000.

[72] Dana Nau. Game Applications of HTN Planning with State Variables. In *ICAPS Workshop on Planning in Games*, 2013.

[73] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP : Simple Hierarchical Ordered Planner. In *Proceedings of the 16th international joint conference on Artificial intelligence*, volume 2, pages 968–973. Morgan Kaufmann Publishers Inc., 1999.

[74] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP and M-SHOP : Planning With Ordered Task Decomposition. Technical Report CS TR 4157, University of Maryland, 2000.

[75] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. SHOP2 : An HTN Planning System. *J. Artif. Intell. Res.(JAIR)*, 20 :379–404, 2003.

[76] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, Héctor Muñoz-Avila, J. William Murdock, Dan Wu, and Fusun Yaman. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2) :34–41, 2005.

[77] Dana S. Nau, Héctor Muñoz-Avila, Yue Cao, Amnon Lotem, and Steven Mitchell. Total-Order Planning with Partially Ordered Subtasks. In *IJCAI*, pages 425–430. Morgan Kaufmann, 2001.

[78] Dana S Nau, Stephen JJ Smith, Kutluhan Erol, et al. Control Strategies in HTN Planning : Theory Versus Practice. In *AAAI/IAAI*, pages 1127–1133, 1998.

[79] Jeff Orkin. Symbolic Representation of Game World State : Toward Real-time Planning in Games . In *AAAI Challenges in Game AI Technical Report*, 2004.

[80] Jeff Orkin. Agent Architecture Considerations for Real-Time Planning in Games . In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment*, 2005.

[81] Jeff Orkin. Three States and a Plan : the AI of FEAR. In *Game Developers Conference*, volume 2006, page 4, 2006.

[82] Edwin PD Pednault. ADL and the State-Transition Model of Action. *Journal of logic and computation*, 4(5) :467–512, 1994.

[83] Edwin Peter Dawson Pednault. *Toward a Mathematical Theory of Plan Synthesis*. PhD thesis, Stanford University, 1987.

[84] J. Scott Penberthy and Daniel S. Weld. UCPOP : a Sound, Complete, Partial Order Planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, volume 92, pages 103–114, 1992.

[85] David Pittman. Command Hierarchies Using Goal-Oriented Action Planning. In *AI Game Programming Wisdom 4*. 2008.

[86] Ira Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1(3) :193–204, 1970.

[87] Ira Pohl. The Avoidance of (Relative) Catastrophe, Heuristic competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 12–17. Morgan Kaufmann Publishers Inc., 1973.

[88] Lutz Prechelt. Are Scripting Languages Any Good ? A Validation of Perl, Python, Rexx, and Tcl Against C, C++, and Java . *Advances in Computers*, 57 :205–270, 2003.

[89] Earl D. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial intelligence*, 5(2) :115–135, 1974.

[90] Earl D. Sacerdoti. *A Structure for Plans and Behavior*. PhD thesis, Stanford University, 1975.

[91] Earl D. Sacerdoti. The Nonlinear Nature of Plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 206–214, 1975.

[92] Lorenza Saitta and Jean-Daniel Zucker. *Abstraction in Artificial Intelligence and Complex Systems*. Springer, 2013.

[93] Geneviève Sella, Olivier Cherrier, Christophe Guettier, and Jacques Yelloz. Development and Experimentation of Collaborative Red Force Tracking in Service Oriented Architecture for Tactical Networking Systems . In *MILITARY COMMUNICATIONS CONFERENCE, 2011-MILCOM 2011*, pages 1529–1534. IEEE, 2011.

[94] Shirin Sohrabi, Jorge A Baier, and Sheila A McIlraith. HTN Planning with Preferences. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, pages 1790–1797, 2009.

[95] Shirin Sohrabi and Sheila A McIlraith. On Planning With Preferences in HTN. In *Proc. of the 12th Int'l Workshop on Non-Monotonic Reasoning (NMR)*, pages 241–248, 2008.

[96] Nathan R. Sturtevant and Michael Buro. Partial Pathfinding Using Map Abstraction and Refinement. In *AAAI*, pages 1392–1397. AAAI Press / The MIT Press, 2005.

[97] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific journal of Mathematics*, 5(2) :285–309, 1955.

[98] Austin Tate. Project Planning using a hierarchical non-linear planner. Research report 25, Artificial Intelligence Dpt., University of Edinburgh, 1976.

[99] Austin Tate. Generating Project Networks. In *Proceedings of the 5th international joint conference on Artificial intelligence*, volume 2, pages 888–893. Morgan Kaufmann Publishers Inc., 1977.

[100] Austin Tate, Brian Drabble, and Richard Kirby. O-Plan2 : an Open Architecture for Command, Planning and Control . In *Intelligent Scheduling*, 1994.

[101] Austin Tate, John Levine, Peter Jarvis, and Jeff Dalton. Using AI Planning Technology for Army Small Unit Operations. In *AIPS*, pages 379–386, 2000.

[102] Josh Tenenberg. Maintaining Logical Consistency in ABSTRIPS : a Formal Approach. Technical Report 205, University of Rochester Computer Science Department, 1987.

[103] Josh Tenenberg. *Abstarction in Planning*. PhD thesis, University of Rochester Computer Science Department, 1988.

[104] Josh D. Tenenberg. Inheritance in Automated Planning. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Representation and Reasoning*, pages 475–485, Toronto, 1989.

[105] Josh D Tenenberg. Abstracting First-order Theories. In *Change of Representation and Inductive Bias*, pages 67–79. Springer, 1990.

[106] Sylvie Thiébaux, Jörg Hoffmann, and Bernhard Nebel. In Defense of PDDL Axioms. *Artificial Intelligence*, 168(1) :38–69, 2005.

[107] Dirk van Dalen. *Logic and structure*. Universitext. Springer, fourth edition edition, 2008.

[108] William van der Sterren. Hierarchical Plan-Space Planning for Multi-unit Combat Maneuvers . *Game AI Pro : Collected Wisdom of Game AI Professionals*, page 169, 2013.

[109] Johan vand der Beek. Autonomous Squad Behavior in a Virtual Game Environment, 2007.

[110] Stavros Vassos and Michail Papakonstantinou. The SimpleFPS Planning Domain : A PDDL Benchmark for Proactive NPCs. In *Intelligent Narrative Technologies*, 2011.

[111] David E. Wilkins. Domain-Independent Planning Representation and Plan Generation. *Artificial Intelligence*, 22(3) :269 – 301, 1984.

[112] David E. Wilkins. *Practical Planning - Extending the Classical AI Planning Paradigm* . Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1988.

[113] David E Wilkins and Roberto V Desimone. Applying an AI Planner to Military Operations Planning. Technical Report 534, SRI International, 1993.

[114] Jason Andrew Wolfe. *Hierarchical Methods for Optimal Long-Term Planning*. PhD thesis, University of California, Berkeley, 2011.

[115] Qiang Yang. Formalizing Planning Knowledge for Hierarchical Planning. *Computational Intelligence*, 6 :12–24, 1990.

[116] Qiang Yang. *Intelligent Planning - A Decomposition and Abstraction Based Approach* . Artificial intelligence. Springer, 1997.

[117] Qiang Yang and Josh D. Tenenberg. ABTWEAK : Abstracting a Nonlinear, Least Commitment Planner. In *AAAI*, pages 204–209. AAAI Press / The MIT Press, 1990.

[118] Qiang Yang, Josh D Tenenberg, and Steve Woods. Abstraction in nonlinear planning. Technical Report CS-91-65, Computer Science Department, University of Waterloo, 1991.

[119] R. Michael Young, Martha E. Pollack, and Johanna D. Moore. Decomposition and Causality in Partial-Order Planning. In *Proceedings of Second International Conference on Artificial Intelligence and Planning Systems*, pages 189–193, 1994.

[120] Jean-Daniel Zucker. A Grounded Theory of Abstraction in Artificial Intelligence. *Philosophical Transactions of the Royal Society of London B : Biological Sciences*, 358(1435) :1293–1309, 2003.

Index

Algorithmes
- A*, 28–30, 33, 34, 45, 48, 55, 56, 151, 164, 168
- ABDHTN, 146–149, 172
- ABHTNPLAN, 87–91, 99, 100
- ABPHTN, 99, 100, 102, 103, 105, 106, 109, 112–115, 117–119, 121, 123, 149
- ABPLAN, 55–60, 87, 88
- AHA*, 44, 45
- Anytime-A*, 164
- Anytime-BFS-HTN, 164–170, 172
- BFS (largeur d'abord), 26, 27, 33
- BFS (meilleur d'abord), 154, 164
- DFS, 26, 27, 30, 33
- DHTN, 39, 45, 46, 146, 164, 165
- PHTN, 41, 42, 45, 46, 99, 100, 117, 118, 120–122, 173, 175–177, 180, 182, 185
- Séparation & évaluation, 29, 30, 33, 42, 67, 76, 151, 152, 164, 166, 167, 186, 190

Domaines de planification
- Affectation de cibles, 114, 117, 119, 213
- Logistics, 170, 176, 223
- Navigation dans un graphe hiérarchique, 114, 120, 121, 215
- Reconnaissance de bâtiment, 114–116, 207
- Robotbox, 170, 177, 219
- SimpleFPS, 75–77, 151, 170, 181, 185, 197, 231

Langages de planification
- ADL, 19, 21, 22, 24, 31, 52, 61, 80, 136, 138
- Calcul Situationnel, 21, 22
- HTDL, 13, 18, 63, 68–70, 74, 78, 80, 83–86, 90, 114, 117, 121, 136, 155, 171, 182, 193
- NCSTRIPS, 44
- PDDL, 34, 66, 75, 176
- STRIPS, 19, 21, 22, 31, 34, 53, 61

Langages de programmation
- C++, 34, 63, 67–69, 74, 76, 116, 173, 189, 191
- Java, 48, 76
- LISP, 68
- Python, 66, 68, 70, 115, 118, 172, 173

Planification
- Planification classique, 19–21, 31, 33, 34, 36, 39, 45, 51, 61, 82, 87, 88, 90, 124
- Planification hiérarchique angélique, 43–45, 47, 48, 79, 82, 151, 152, 154, 190
- Planification HTN, 13, 17–19, 35, 36, 38–40, 43, 45–49, 61, 62, 66, 67, 69, 76, 80–82, 87, 88, 124, 149, 151, 152, 168, 175, 186, 189, 190
- Planification par abstraction, 19, 49–51, 54, 61, 62, 99, 190
- Planification POP, 30–32, 39, 89
- Planification temporelle, 35, 90

Systèmes de planification
- ABSTRIPS, 49, 51, 53, 55, 60, 84, 125, 126, 148
- ABTWEAK, 56
- Carnival, 34, 49
- DPOCL, 39
- FAPE, 35
- GOAP, 33, 34, 49
- HATP, 41, 42
- HTNPLAN, 151
- HTNPLAN-P, 41, 42
- HTNPREF, 151
- JSHOP, 48
- JSHOP2, 63, 68, 69, 76, 77, 86
- NOAH, 31, 35, 39, 49, 79
- NONLIN, 31, 35, 39, 79, 192
- O-Plan, 39, 48

O-Plan2, 39
PlannedAssault, 17, 48, 151
PRIAR, 39
SHOP, 35, 41, 42, 47–49, 61, 63, 66, 74, 79, 80, 91, 117, 146, 151, 152, 189
SHOP2, 41, 42, 66, 67, 76, 88, 151, 152, 186
SHPE, 18, 63, 66–69, 74, 76–80, 88, 89, 91, 99, 117, 191
SIADEX, 35
SIPE, 35, 39
SIPE-2, 48
SNLP, 31
STRIPS, 53, 55
TWEAK, 31, 56
UCPOP, 31, 74
UMCP, 35, 39

Résumé

Cette thèse explore l'application de la planification HTN afin d'animer une section d'infanterie dans un simulateur informatique temps réel. Afin de produire des plans en ligne pour près de 40 soldats, on montre qu'il est possible d'optimiser le planificateur pour un domaine HTN en compilant les éléments de planifications en structures statiques et en procédures C++. On montre ensuite que la structure du problème se prête à une combinaison de la planification HTN avec la planification par abstraction, obtenue en modélisant des effets abstraits aux tâches composées. Sous certaines conditions, la recherche de solutions est alors accélérée en détectant les réseaux de tâches pour lesquels aucune solution n'est exécutable. Enfin, on montre que la structure du problème permet aussi de formuler des fonctions d'évaluation exploitables dans un algorithme de recherche heuristique non admissible, capable de retourner rapidement des solutions presque optimales.

Abstract

This thesis explores the application of HTN planning to the animation of an infantry platoon in a real-time simulation software. In order to achieve online planning for nearly 40 soldiers, we show that it is possible to optimize the planner for one HTN domain with a compilation of planning elements into C++ static structures and procedures. Then, we demonstrate that the problem structure lends itself to a combination of HTN planning with abstraction planning, achieved with the modelisation of abstract effects for compound tasks. In some conditions, we can detect those task networks that never lead to any executable solution, and therefore improve the search. Eventually, we show that the problem structure enables to formulate evaluation functions that can be input into a non admissible heuristic search algorithm, and that near optimal solutions can be obtained within a short run-time.

Mots Clés

Intelligence Artificielle, Planification Automatique, Planification HTN, Abstraction, Hiérarchie, Architecture, Logique, Simulation , Jeu sérieux, Infanterie

Keywords

Artificial Intelligence, Automated Planning, HTN Planning, Abstraction, Hierarchies, Architecture, Logic, Simulation, Serious Gaming, Infantry