Réparation de plans dans les HTNs réactifs en utilisant la planification symbolique

Lydia Ould Ouali[1] Charles Rich[2] Nicolas Sabouret[1]

[1] LIMSI-CNRS, UPR 3251, Orsay, France & Univ. Paris-Sud, Orsay, France
[2] Worcester Polytechnic Institute, Worcester, MA, USA

Résumé

Construire des modèles formels pour planifier les prochaines actions d'un agent est un problème crucial en Intelligence Artificielle. Dans ce travail, nous proposons de combiner deux approches connues, à savoir les réseaux de tâches hiérarchiques (HTNs) et la planification symbolique linéaire. La motivation principale de cette approche hybride est de d'assurer la réparation des cassures durant l'exécution du HTN en invoquant dynamiquement un planificateur symbolique. Ce travail nous a aussi permis de mettre en exergue les problèmes liés à la combinaison de la modélisation procédurale et symbolique. Nous avons implémenté notre approche qui combine une HTN réactif appelé Disco et le planificateur STRIPS implémenté en Prolog, et nous avons conduit des évaluations préliminaires.

Mots Clef

HTN réactif, planification symbolique, réparation de cassures.

Abstract

Building formal models of the world and using them to plan future action is a central problem in artificial intelligence. In this work, we combine two well-known approaches to this problem, namely, reactive hierarchical task networks (HTNs) and symbolic linear planning. The practical motivation for this hybrid approach was to recover from breakdowns in HTN execution by dynamically invoking symbolic planning. This work also reflects, however, on the deeper issue of tradeoffs between procedural and symbolic modeling. We have implemented our approach in a system that combines a reactive HTN engine, called Disco, with a STRIPS planner implemented in Prolog, and conducted a preliminary evaluation.

Keywords

Reactive HTN, symbolic linear planning, breakdowns, recovery.

1 Introduction

Les réseaux de tâches hiérarchiques ou HTN [4] (Hierarchical task networks) sont largement utilisés pour contrôler des agents et des robots évoluant dans des environnements dynamiques. Il existe dans la littérature différentes formalisations et représentations graphiques. Nous utiliserons ici une simple représentation en arbre, avec différents niveaux de nœuds *tâches* et des nœuds de *décompositions*. Un HTN est donc défini avec un nœud tâche au niveau de la racine. Les feuilles constituent des *tâches primitives*, correspondant aux actions de l'agent dans le monde. Les autres nœuds de l'arbre définissent soit des *tâches abstraites* à décomposer (c'est par exemple le cas de la racine de l'arbre), soit des *recettes* décrivant une décomposition possible en sous-tâches (elles-mêmes décomposables à leur tour, jusqu'à atteindre des tâches primitives).
Ces HTNs sont généralement codés à la main par un modélisateur qui définit la décomposition des tâches en recettes et en sous-tâches (éventuellement partiellement ordonnées). En plus de cette décomposition, la plupart des HTNs sont définis avec des conditions pour le contrôle de l'exécution. Ainsi, le modélisateur peut définir pour chaque recette des *conditions d'applicabilité* qui déterminent l'applicabilité de la recette dans l'état courant, et pour chaque tâche des préconditions et des post-conditions. Une tâche ne peut être effectuée que si ses préconditions sont valides et une tâche est considérée comme réussie si ses post-conditions sont satisfaites.
À l'origine, les HTN sont une extension hiérarchique des plans linéaires classiques (e.g. STRIPS [5]), et comme pour les modèles classiques, les conditions associées aux tâches sont dites *symboliques*, *i.e.* elles sont écrites en logique formelle qui permet à un moteur d'inférence de raisonner dessus. La principale difficulté, comme l'a souligné [8], réside dans la modélisation complète et fidèle d'une monde complexe à l'aide de prédicats. En réponse à ces difficultés liées à la modèlisation des connaissances symboliques, une variante appelée HTNs *réactifs* a été developpée dans laquelle les conditions sont *procédurales*, *i.e.* elles sont écrites dans un langage de programmation et évaluée

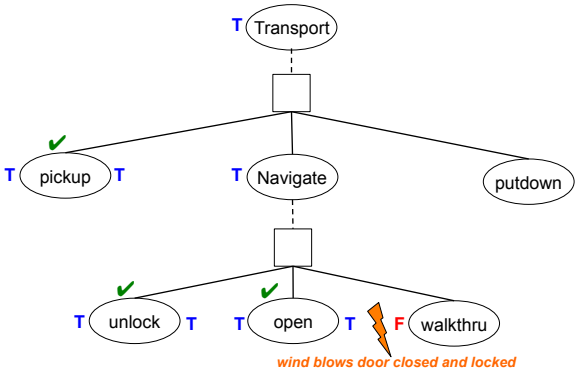


FIGURE 1 – Cassure dans l'exécution du HTN après que le vent a fait claquer la porte et l'a bloquée. Les coches indiquent les tâches dont l'exécution est terminée. "T" représente une condition dont l'évaluation a retourné *vrai* ; "F" représente une condition qui a retourné *faux*.

par un interpréteur de ce langage de programmation. Les HTNs réactifs sont utilisés par exemple dans le domaine des jeux vidéos sous le nom de *behaviour tree* [1]. Notre travail porte sur les HTNs réactifs et, en particulier, sur la réparation des cassures (ou *breakdowns*) qui peuvent survenir durant l'exécution. L'idée principale est d'ajouter une petite proportion de conditions symboliques au HTN réactif afin de permettre à un planificateur linéaire de produire des plans de réparation locaux.

2 Motivations

Afin de mieux introduire et expliquer notre travail, considérons un exemple simple mais intuitif d'une cassure dans l'exécution d'un HTN et sa réparation, illustré dans la FIGURE 1. Il s'agit d'implémenter en HTN le comportement d'un robot pour transporter un objet a travers une porte fermée. Dans le HTN, la tâche but *transport*, est décomposée en trois étapes : ramasser l'objet (*pickup*), manipuler la porte *(navigate)* et poser l'object *(putdown)*. La tâche *navigate* est à son tour décomposée en quatre étapes : déverrouiller la porte *(unlock)*, l'ouvrir *(open)* et franchir la porte *(walkthrough)*. Chacune de ces tâches est représentée par un ovale dans la FIGURE 1. Les boîtes représentent les différentes alternatives de décomposition qui peuvent être ignorées dans cet exemple. Au moment de l'exécution décrit dans la FIGURE 1, le robot a ramassé l'objet, déverrouillé et ouvert la porte avec succès. Cependant, avant que la précondition de la tâche *walkthrough* ne soit évaluée, le vent souffle et la porte de ferme et se verrouille (l'exemple ne vise pas à être réaliste mais à illustrer notre propos). La précondition de la tâche *walkthrough* évalue si la porte est ouverte et retourne faux. À ce moment, aucune tâche ne peut être exécutée

1. See http ://aigamedev.com/open/article/popular-behavior-tree-design

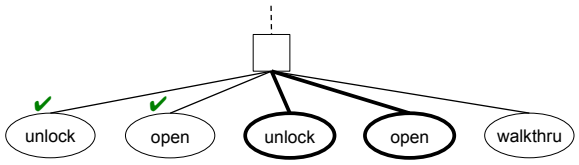
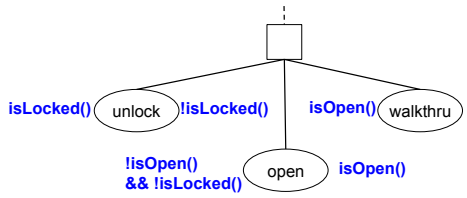


FIGURE 2 – Séquence de deux tâches primitives (en gras) ajoutées au plan hiérarchique de la tâche "navigate" pour réparer la cassure de la FIGURE 1.

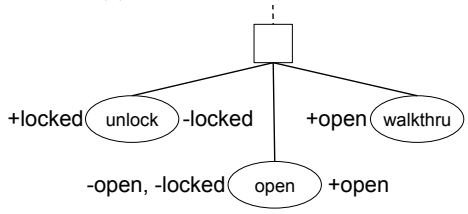
dans le HTN et c'est ce que nous appelons une *cassure*. Il n'est pas rare qu'un HTN réactif rencontre des cassures, spécialement dans le cas d'exécution dans des environnements dynamiques et complexes. Cependant, en analysant la cassure produite dans cet exemple, la solution montrée dans la FIGURE 2 parait évidente : déverrouiller la porte et l'ouvrir. Trouver une solution pour cette cassure est un problème trivial pour un planificateur linéaire symbolique comme STRIPS si les (pré/post)-conditions des tâches primitives correspondantes étaient spécifiées symboliquement.
Or justement, ce qui caractérise un HTN réactif, c'est que *les pré et post-conditions sont définies à l'aide de scripts* (elles sont définies en langage procédural et évaluées par un interpréteur approprié à ce langage) et *ne peuvent donc pas être manipulés comme des connaissances symboliques*. La FIGURE 3a montre les conditions procédurales définies dans le plan *Navigate* comme elles seraient typiquement écrites par exemple en JavaScript. Par exemple, la procédure "isOpen()" appelle un code spécifique dans le système de capteurs du robot pour évaluer si la porte est actuellement ouverte. À titre de comparaison, la FIGURE 3b montre la formalisation des mêmes tâches en STRIPS.
Supposons que lorsque la cassure a été détectée dans l'exemple de la FIGURE 1, le moteur d'exécution du HTN contenait la connaissance symbolique qui est fournie sur la FIGURE 3b. La réparation de la cassure pourrait donc être traitée comme une problème de planification STRIPS (voir FIGURE 4) dans lequel l'état initial est représenté par l'état courant du monde et l'état final est la précondition échouée de la tâche *walkthrough* (la porte est ouverte). Un simple chainage arrière peut rapidement trouver une séquence d'actions (à savoir déverrouiller et ouvrir la porte). Le plan produit est ensuite intégré au HTN comme dans la FIGURE 2 afin de permettre au HTN de reprendre son exécution. Le but de cet article est de généraliser la solution de cet exemple en algorithme de réparation de cassures qui soit indépendant des applications et auquel on associe une méthodologie de modélisation des HTNs réactifs.

3 Travaux connexes

La question que pourrait soulever la section précédente est pourquoi ne pas utiliser seulement la planification symbolique au lieu d'un HTN réactif et ap-



(a) Conditions procédurales



(b) Représentation symbolique des conditions

FIGURE 3 – Conditions procédurales versus symboliques pour le plan Navigate.

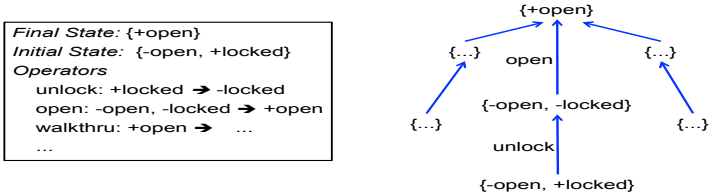


FIGURE 4 – Réparation de la cassure de la FIGURE 1 comme un problème de planification STRIPS.

pliquer les solutions existantes de réparation des cassures ? a problématique de la réparation des cassures durant l'exécution a déjà été étudiée dans le domaine des HTNs symboliques et de nombreux modèles ont été proposés[2, 11, 1, 12]. Ces techniques de réparation de plans sont généralement très efficaces.

La réponse à cette question réside dans le rapport entre la modélisation du problème et l'exécution du plan produit. En effet, les planificateurs symboliques basées sur des connaissances symboliques linéaires (en langage STRIPS ou PDDL [7]) ou hiérarchiques (à l'aide de HTNs) requièrent une description *complète* et *correcte* de toutes les tâches dans le domaine du problème. Si la description symbolique est incomplète ou incorrecte, les plans générés vont subir des cassures à l'exécution.

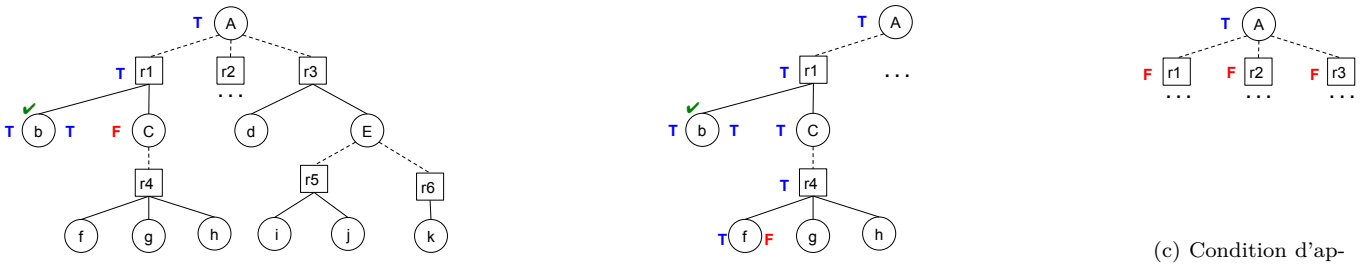
Malheureusement, la recherche en IA [8] a montré que la modélisation d'une description logique du monde réel qui soit complète et correcte peut se révéler extrêmement difficile voir impossible en pratique. C'est pour cela que les HTNs réactifs ont été inventés : la connaissance est insérée dans les HTNs réactifs principalement à deux endroits : dans la structure de décomposition de l'arbre et dans le code pour définir les conditions procédurales (en particulier dans les conditions d'applicabilité pour les choix de décompositions). L'intérêt des HTNs réactifs provient de la facilité de modélisation sur des problèmes dont le domaine de connaissance est complexe pour être modélisé de manière symbolique. De plus, les conditions procédurales ne sont évaluées que dans l'état *courant* du monde (alors que les descriptions symboliques doivent être vrais dans tous les mondes possibles). Ceci n'empêche pas les HTNs réactifs de rencontrer des cassures, comme dans notre exemple, et c'est ce qui nous a conduit à proposer une approche hybride dans laquelle un HTN réactif est augmenté avec de la connaissance symbolique afin d'aider à la réparation des cassures.
D'autre travaux avant nous ont proposé d'ajouter un module de raisonnement symbolique pour étendre les HTNs réactifs. Par exemple, Firby [6] propose d'ordonner les tâches dans la file d'exécution du HTN et de choisir d'autres alternatives de décompositions ce qui permet au HTN de détecter les situations problématiques avant qu'elles ne se produisent. Brom [3] propose d'utiliser la planification afin d'assurer l'exécution des tâches avec des contraintes de temps. Cependant, aucune de ces deux approches ne propose d'étudier la possibilité de réparer des HTN réactifs à l'aide de planification symbolique. Notre originalité est de proposer un algorithme hybride combinant le procédural et le symbolique.

4 L'approche hybride

Dans cette section, nous proposons une généralisation sur deux niveaux de l'exemple de la section 2 en considérant : (1) les différents types de cassures, (2) l'ensemble des états finaux possibles. Nous présenterons d'abord l'algorithme général de réparation de cassures et discuterons ensuite la méthodologie de modélisation associée.

4.1 Les HTNs réactifs

Les HTNs réactifs peuvent être représentés comme un arbre et/ou, où les tâches sont des "et", et les nœuds de décomposition sont des "ou". Contrairement aux HTNs symboliques, les HTNs réactifs ne visent pas à anticiper le futur et construire un plan complet afin de l'exécuter par la suite. Ils se content seulement de calculer la prochaine tâche a exécuter à partir de l'état courant du monde. L'exécution est en profondeur d'abord, avec un parcours de gauche à droite à partir du nœuds racine, durant lequel les conditions sont évaluées, comme décrit ci-après.
Si l'exécution du nœud courant est une tâche alors ses éventuelles préconditions représentés en procédures booléennes sont évaluées. Si elles retournent faux, alors l'exécution est interrompue *(cassure)* ; sinon l'exécution continue. Si la tâche est primitive, elle est directement exécutée pour changer l'état du monde. Sinon, si elle est abstraite alors les conditions d'applicabilités des nœuds fils (décompositions) sont évalués dans l'ordre jusqu'à l'obtention d'un nœud qui retourne vrai. L'exécution continue alors avec cette décomposition. Si toutes les conditions d'applicabilités sont



(a) Précondition échouée (b) post-condition échouée (c) Condition d'applicabilité échouée

FIGURE 5 – Exemples des trois types de cassures dans l'exécution d'un HTN.

fausses alors l'exécution est arrêtée *(cassure)*. Une fois l'exécution de la tâche terminée, ses éventuelles post-conditions sont évaluées, si elles retournent faux, alors l'exécution est interrompue *(cassure)*. Sinon l'exécution continue. Si le nœud courant est un nœud de décomposition, alors les nœuds fils (tâches) sont exécutées dans l'ordre.
La FIGURE 5 résume les trois types de cassures qu'un HTN peut rencontrer durant son exécution. La cassure dans l'exemple de la section 2 est causé par une précondition échouée. Cependant, cette taxonomie n'est pas apte à distinguer les raisons sous-jacentes qui peuvent provoquer des cassures. En effet, une cassure peut être causée par un changement inattendu dans le monde (*e.g.* le vent dans la section 2) ou à cause d'une erreur de programmation (condition mal codée, structure d'arbre erronée). L'impossibilité de détecter les causes des cassures est une limitation intrinsèque des HTN réactifs.

4.2 Algorithme de réparation de plans

La généralisation la plus significative de l'algorithme sur l'exemple de la section 2, concerne le choix de l'état final pour le planificateur linéaire. En effet, pour que l'algorithme de réparation fonctionne il a besoin d'avoir en entrée un état but à atteindre. Cependant, pour ce même exemple, différents état buts peuvent être trouvés. Par exemple, supposons que *walkthrough* ait une post-condition qui spécifie que le robot est dans la pièce de l'autre côté de la porte. Après la cassure, le robot peut trouver une autre méthode de réparation : trouver un plan pour satisfaire cette post-condition (par exemple, sortir de la pièce actuelle par une autre porte...). Cette procédure consiste à définir la post-condition comme un *état but* candidat pour la réparation. De la même façon, supposons que *walkthrough* n'a pas de représentation symbolique pour sa post-condition, mais que la post-condition symbolique de *navigate* spécifie l'emplacement but du robot. Dans ce cas la post-condition de *navigate* est considérée comme un état but de réparation. De manière similaire, si on suppose que la cassure dans notre exemple a été provoquée par l'échec de la post-condition de *unlock*, la représentation symbolique de la précondition de *walkthrough* ainsi que celles des post-conditions de *walkthrough* et de *navigate* sont de bons candidats pour la réparation.

Sur la base de ce raisonnement, l'algorithme que nous présentons ci-dessous repose sur l'idée de construire les états but candidats pour la procédure de réparation à partir de l'ensemble des conditions du HTN affectées par la cassures (dont l'évaluation retourne faux) et qui n'ont pas encore été validées durant l'exécution. Nous pensons que cette première approximation est trop grossière (trop de candidats sont considérés) mais nous avons besoin d'étudier notre approche sur des exemples plus concrêts pour concevoir une meilleure heuristique de recherche. Il est a noter qu'une condition d'applicabilité est considérée comme candidat dans l'unique cas où toutes ses sœurs dans l'arbre ont été évaluées comme fausses.

La FIGURE 6 présente le pseudo-code du système hybride ainsi conçu. La procédure principale, EXECUTE, execute un HTN jusqu'à ce qu'il se termine (succès) ou jusqu'à ce qu'une cassure soit détectée. Le code de réparation des cassures commence à la ligne 5. La sous procédure, FINDCANDIDATES va récursivement parcourir le HTN afin de calculer les candidats cibles pour la procédure de réparation. La procédure SYMBOLICPLANNER n'est pas décrite car n'importe quel planificateur linéaire peut être utilisé. De plus, comme l'ensemble des opérateurs symboliques ne change pas durant l'exécution, il n'est pas défini comme un argument explicite du planificateur symbolique. On peut voir à ligne 6 que notre approche requiert une méthode pour calculer, à partir de l'état courant, l'état initial du planificateur symbolique dans un formalisme symbolique que le planificateur puisse exploiter. Par exemple, pour l'exemple de la section 2, chaque fonction symbolique comme "open" doit être associée à une procédure comme "isOpen()" donnant sa valeur dans l'état courant. C'est là que réside la difficulté de la modélisation hybride (discutée dans la section 4.3). La ligne 8 montre que les candidats sont triés par rapport à leurs distance du nœud courant dans l'arbre, en utilisant une simple métrique (par exemple le plus court chemin dans un arbre non-orienté). Notre but est ainsi de favoriser des réparations locales qui préservent au maximum la structure originale du HTN. Cependant, nous n'avons pas encore testé les perfor-

```
1: procedure EXECUTE(htn)
2:     while htn is not completed do
3:         current ← next executable node in htn
4:         if current ≠ null then execute current
5:         else [breakdown occurred]
6:             initial ← symbolic description of current world state
7:             candidates ← FindCandidates(htn)
8:             sort candidates by distance from current
9:             for final ∈ candidates do
10:                plan← SymbolicPlanner(initial,final)
11:                if plan ≠ null then
12:                    splice plan into htn between current and final
13:                    continue while loop above
14:            Recovery failed!

15: procedure FINDCANDIDATES(task)
16:    conditions ← ∅
17:    pre ← symbolic precondition of task
18:    if pre ≠ null ∧ procedural prec of task has not evaluated to true then
19:        add pre to conditions
20:    post ← symbolic post-condition of task
21:    if post ≠ null ∧ procedural postc of task has not evaluated to true then
22:        add post to conditions
23:    applicables ← ∅
24:    allFalse ← true
25:    for decomp ∈ children of task do
26:        for task ∈ children of decomp do FindCandidates(task)
27:        if allFalse then
28:            if procedural appl condition of decomp has evaluated to false then
29:                app ← symbolic applicability condition of decomp
30:                if app ≠ null then add app to applicables
31:            else allFalse ← false
32:    if allFalse then add applicables to conditions
33:    return conditions
```

FIGURE 6 – Pseudocode de l'exécution et réparation du système HTN réactif hybride

mances de cette heuristique. Enfin, à la ligne 12 de la procédure EXECUTE, quand un plan est trouvé, il doit être intégré dans le HTN pour être exécuté. Une fois le plan de réparation exécuté, le HTN récupéré la tâche de la condition réparée pour poursuivre son l'exécution. Dans l'exemple de la figure FIGURE 4 la condition réparée est "isOpen()". Par conséquent, la prochaine tâche à exécuter après le plan de réparation est la tâche où la condition est utilisée, à savoir *walkthrough*.

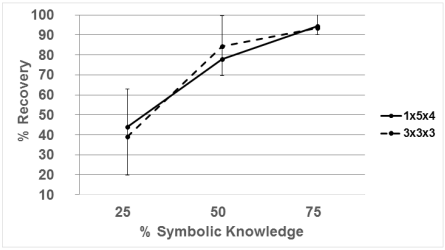
4.3 Méthodologie de modélisation

Le système hybride prend avantage de la connaissance symbolique construite par l'auteur du HTN, sachant que toute condition dans le HTN peut, ou non, avoir une représentation symbolique. Comme nous l'avons annoncé précédemment, la difficulté liée à la réalisation d'une modélisation symbolique a conduit les auteurs des HTNs à utiliser la modélisation procédurale. Il existe donc deux problèmes méthodologiques liées à la conception d'un HTN hybride. D'une part, il faut déterminer où il est préférable d'investir l'effort nécessaire de modélisation symbolique. D'une autre part, il faut définir une stratégie qui facilite à l'auteur le processus de mixage entre le procédural et le symbolique. Notre intuition première est qu'il faut réduire la modélisation symbolique aux tâches primitives, car nous espérons que l'utilisation d'une stratégie locale pour la réparation des cassures sera plus efficace. Le choix des opérateurs à introduire au planificateur linéaire a une implication directe sur ses performances. En effet, seules les tâches primitives disposant d'une représentation symbolique de ses pré et post-conditions peuvent être incluses dans l'ensemble des opérateurs. Cependant, si une tâche abstraite dispose de ces caractéristiques, elle peut alors être utilisée pour la réparation des cassures, en utilisant durant son exécution une de ses décompositions valables dans l'état courant.
Nous envisageons de développer un outil qui aiderait à simplifier la modélisation hybride en détectant les liens entre les deux modélisations comme dans la FIGURE 3a et générer automatiquement la représentation symbolique comme dans la FIGURE 3b. L'outil pourrait aussi garder l'historique des cassures et les utiliser pour guider l'auteur dans l'ajout de connaissances symboliques dans le HTN.

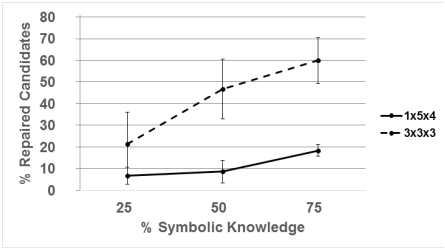
5 Implémentation et évaluation

Nous avons implémenté notre système hybride en Java en utilisant le standard ANSI/CEA-2012 [9] pour la modélisation du HTN et Disco [10] comme le moteur d'exécution de ce HTN. Concernant le planificateur symbolique, nous avons utilisé une simple implémentation de STRIPS exécutée dans un environnement Java de Prolog [2]. L'utilisation de Prolog facilite l'ajout de règles de raisonnement symboliques durant le processus de planification.
Une véritable évaluation de notre approche consisterait à faire évoluer plusieurs agents dans des environnements dynamiques du monde réel et évaluer leurs performances ainsi que leur robustesse face aux cassures. En attendant, nous avons validé notre système avec une expérimentation préliminaire sur des HTNs générés synthétiquement en variant le niveau de connaissance symbolique. Nous sommes partis d'une théorie simple qui préconise que plus il y'a des connaissances symboliques dans le HTN, meilleures seront ses performances pour la réparation des cassures.

2. See http ://tuprolog.apice.unibo.it



(a) Ration des réparation/cassure



(b) Proportion des candidats réparés

FIGURE 7 – Résultats des expérimentations sur des HTNs générés synthétiquement avec différents niveaux de connaissances symbolique.

La FIGURE 7 montre les résultats obtenus qui confirment notre hypothèse. Nos HTN synthétiques sont caractérisés par trois dimensions $R \times S \times D$, où R est le facteur de ramification d'une décomposition (recipe), S le facteur de ramification d'une tâche, D étant la profondeur du HTN. Nous avons testé deux configurations de HTNs de dimensions respectivement $3 \times 3 \times 3$ et $1 \times 5 \times 4$. Pour chaque test, nous avons aléatoirement extrait des combinaisons possibles de connaissance symbolique sur trois différent niveaux ; 25% , 50%, 75% (le pourcentage des conditions qui ont une représentation symbolique). Nous nous sommes restreints à ces modélisations pour des questions de temps d'exécution. La FIGURE 7a montre l'évolution des performances de réparation des cassures en fonction de l'augmentation du niveaux de connaissance symbolique. Dans la FIGURE 7b, nous nous sommes intéressés à la proportion des problèmes de planification soumis au planificateur (buts candidats) qui ont été correctement résolus. Nous pouvons remarquer que cette dernière augmente aussi en fonction du niveaux de connaissance symbolique (pour ces tests, nous avons modifié notre système pour qu'il génère un plan pour tous les candidats).

6 Conclusion

Nous avons présenté dans cet article une première contribution pour la réparation des cassure qui consiste à augmenter un HTN réactif avec un planificateur linéaire symbolique qui propose des plans de réparation. Nous avons ensuite soulevé la question sous-jacente à la modélisation hybride de notre système et nous avons expliqué le compromis existant entre la modélisation symbolique et la modélisation procédurale. Une implémentation en Java a été proposé qui combine un HTN réactif appelé Disco et le planificateur STRIPS implémenté en Prolog. Des tests sur des HTNs synthétiques ont été menés pour valider le système et les résultats obtenus confirment nos théories de départ. Notre proposition reste préliminaire, bien qu'elle est implémentée et testée sur des données générées synthétiquement, car la question de ses performances en pratique reste une question ouverte. Nous poursuivons actuellement ce travail dans une thèse dont l'objectif est d'intégrer notre modèle dans un système de dialogue social afin de gérer de manière opportuniste les éventuelles cassures.

Références

[1] Ayan et al. Hotride : Hierarchical ordered task replanning in dynamic environments. In *Planning and Plan Execution for Real-World Systems–Principles and Practices for Planning in Execution : Papers from the ICAPS Workshop. Providence, RI*, 2007.

[2] G. Boella and R. Damiano. A replanning algorithm for a reactive agent architecture. In *Artificial Intelligence : Methodology, Systems, and Applications*, pages 183–192. Springer, 2002.

[3] C. Brom. Hierarchical reactive planning : Where is its limit. *Proceedings of MNAS : Modelling Natural Action Selection. Edinburgh, Scotland*, 2005.

[4] K. Erol, J. Hendler, and D. S. Nau. Htn planning : Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.

[5] R. E. Fikes and N. J. Nilsson. Strips : A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3) :189–208, 1972.

[6] R. J. Firby. An investigation into reactive planning in complex domains. In *AAAI*, volume 87, pages 202–206, 1987.

[7] D. E. Ghallab, Malik et al. Pddl-the planning domain definition language. 1998.

[8] Y. Gil. Acquiring domain knowledge for planning by experimentation. Technical report, DTIC Document, 1992.

[9] C. Rich. Building task-based user interfaces with ansi/cea-2018. *Computer*, 42(8) :20–27, 2009.

[10] C. Rich and C. L. Sidner. Using collaborative discourse theory to partially automate dialogue tree authoring. In *Intelligent Virtual Agents*, pages 327–340. Springer, 2012.

[11] R. Van Der Krogt and M. De Weerdt. Plan repair as an extension of planning. In *ICAPS*, volume 5, pages 161–170, 2005.

[12] I. Warfield, C. Hogg, S. Lee-Urban, and H. Munoz-Avila. Adaptation of hierarchical task network plans. In *FLAIRS Conference*, pages 429–434, 2007.