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

ouldouali@limsi.fr

Résumé

Construire des modèles formels afin de les utiliser pour la planification de futurs actions est un problème crucial dans l'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 d'assurer la réparation des cassures durant l'exécution d'un 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 un HTN réactif appelé Disco et le planificateur STRIPS implémenté en Prolog, et 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 différrentes formalisations et représentations graphiques pour la définition des HTNs. Dans ce présent travail, nous utilisons une simple représentation en arbre pour la définition des HTNs qui sera expliquée plus en détails dans la section 4.1. Un exemple est représenté dans la FIGURE 1. Les HTNs sont généralement codés à la main et peuvent avoir de larges dimensions avec plusieurs niveaux d'hierarchies. Les HTNs partagent la même structure pour la décomposition des tâches en séquence (pouvant être partiellement ordonnée) de sous tâches avec plusieurs alternatives de décompositions (appelée parfois recipes) pour les différentes situations. En plus de la structure de décomposition, la plupart des HTNs sont définis avec des conditions ; des préconditions et des postconditions asosciées aux nœuds de l'arbre qui contôlent l'exécution de l'HTN.
À l'origine, les HTNs sont une éxtension hiérarchique des plans linéaires classiques (e.g.,STRIPS), et similairement aux plans classiques, les conditions associées aux tâches sont dites *symboliques*, i.e., elles sont écrites suivant une certaine forme de logique formelle qui permet à un moteur d'inférence logique de raisonner dessus. Par la suite, en réponse aux difficultés liées à la modèlisation des connaissances symboliques (voir section 3), une variante appelée HTNs *reactifs* a été developpée dans laquelle les conditions sont *procédurales*

i.e elles sont écrites dans un langage de programmation et évaluées par un interpréteur de ce langage de programmation. L'idée des HTNs réactifs est aussi utilisée dans le domaine des jeux vidéos sous le nom de *behaviour tree.* [1]

Le présent travail s'intéresse principalement aux HTNs réactifs et spécialement à la réparation des cassures qui peuvent survenir durant leurs 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. Le section 2 présente un exemple motivant notre approche.

La problématique de la réparation des cassures durant l'exécution a déjà été étudiée dans le domaine des HTNs symboliques [2, 9, 1, 10]. Ces travaux malgré leur efficacité ne sont pas directement pertinents, car ces techniques de réparation de plans reposent sur la définition symbolique de toutes les conditions dans le HTN ce qui les rendent inapplicables pour les HTNs réactifs. D'autre travaux ont proposé d'ajouter un module de planification symbolique pour les HTNs réactifs. Par exemple, Firby [5] proposait d'utiliser un planificateur pour ordonner les tâches dans la queue d'exécution du HTN ou de l'aider à choisir d'autres alternatives de décompositions ce qui permettrait au HTN de détecter les possibles situations problématiques avant qu'elles ne se produisent. Brom [3] proposait d'utiliser la planification afin d'assurer l'exécution des tâches avec des contraintes de temps. Cependant, aucun travail existant ne propose le développement d'un algorithme hybride combinant le procédural et le symbolique (voir section 4.2) comme nous le proposons dans ce travail.

Finalement, cette proposition est préliminaire bien qu'elle fut implémentée et testée sur des données générées synthétiquement. Cependant, la question de ses performances en pratique reste une question ouverte.

2 Motivations

Afin de mieux introduire et expliquer notre travail, nous présentons dans cette section un exemple simple mais intuitif d'une cassure dans l'exécution d'un HTN et sa réparation. l'idée de base de cet exemple présenté dans la FIGURE 2 est l'implémentation d'un robot en HTN pour transporter un objet a travers une porte fermée. Dans le HTN produit, la tâche bute *Transport*, est décomposée en trois étapes : ramasser l'objet (*pickup*), Manipuler la porte *(Navigate)* et poser l'object *(putdown)*. la tâche *Navigate* est a son tour décomposée en trois étapes : déverrouiller la porte *(unlock)*, l'ouvrir *(open)* et franchir la porte *(walkthru)*. Chacune de ses tâches est représentée par un ovale dans la

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

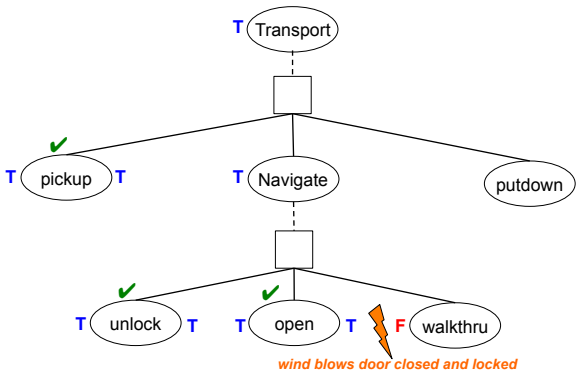


FIGURE 1 – cassure dans l'exécution du HTN après que le vent ait fait claquer la porte et la bloquer. Les coches indiquent les tâches dont l'exécution est terminée. "T" représente une condition dont l'évaluation a retourné *vrai* ; "F" représente une condition qui a retourner *faux*.

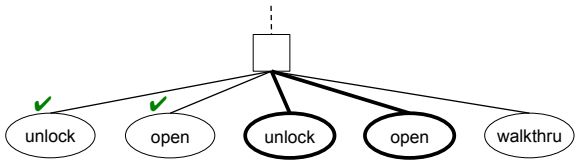
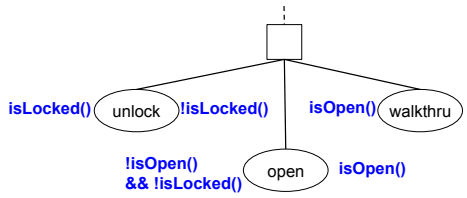


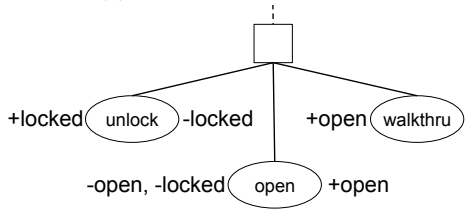
FIGURE 2 – Séquence de deux tâches primitives (en gras) ajoutée afin de planifier pour réparer la cassure de *Navigate* dans la FIGURE 1.

FIGURE 1. (Les boîtes représentent les différentes alternatives de décomposition qui peuvent être ignorées dans cet exemple). Le moment de l'exécution décrit dans FIGURE 1, dans lequel le robot a ramassé l'objet, déverrouillé et ouvert la porte avec succès. Cependant, avant que la précondition de la tâche *walkthru* ne soit évaluée, le vent souffla et la porte se ferma et se verrouilla. La précondition de la tâche *walkthru* évalue si la porte est ouverte et retourne faux. A ce moment, aucune tâche ne peut être exécutée dans le HTN et c'est ce que nous appelons *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). Par ailleurs, 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.

Dans les HTN réactifs, pré-et postconditions sont définies en langage procédural et évaluées par un interpréteur approprié à ce langage. FIGURE 3a montre les conditions procédurales définies dans le plan *Navigate* comme elles seraient typiquement écrites par exemple en JavaScript. En comparaison, FIGURE 3b montre la formalisation symbolique des mêmes tâches primi-



(a) Conditions procédurales



(b) Représentation symboliques des conditions

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

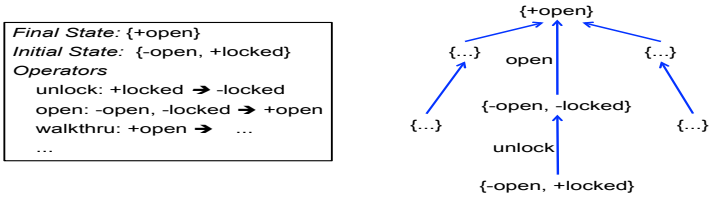


FIGURE 4 – réparation de la cassure FIGURE 1 comme un problème de planification STRIPS.

tives en STRIPS. Supposons que lorsque la cassure a était détectée dans FIGURE 1, le moteur d'exécution du HTN contenait la connaissance symbolique qui est dans 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 *walkthru* (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 HTH 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 Modélisation procédurale VS modélisation symbolique

La question que pourrait soulever la section précédente est pourquoi ne pas utiliser seulement la planification symbolique au lieu d'un HTN et appliquer les solutions existantes de réparation des cassures ?. La réponse à cette question mène directement à la problématique de modélisation. En effet, les planificateurs symboliques comme STRIPS requièrent une description symbolique *complète* et *correcte* de toutes les tâches primitives dans le domaine du problème. Les différents planificateurs utilisent différentes formalisations pour cela comme le (add/delete) liste dans la FIGURE 3b ou PDDL [6]. Cependant, tous ces planificateurs ont un point commun, si la description symbolique est incomplète ou incorrecte, alors les plans générés vont subir des cassures.

Malheureusement, la recherche en IA a montré que la modélisation d'une description logique du monde réel qui soit complète et correcte peut se révéler extrêmement difficile voir impossible pour des raisons de fins pratiques. La difficulté de la modélisation symbolique est la raison principale pour laquelle les HTNs réactifs ont été inventés. La connaissance est insérée dans les HTNs réactifs principalement a deux endroits : dans la structure de décomposition de l'arbre et dans le code pour définir les conditions procédurales (spécialement dans les conditions d'applicabilités pour les choix de décompositions). grâce à leur facilité de modélisation, les HTNs sont utilisés pour les problèmes de domaines complexes. En effet, il est bien connu que l'architecture hiérarchique facilite aux personnes l'organisation de leur pensées et leur permet de faire face à la complexité des tâches. De plus, les conditions procédurales comme une précondition n'est évaluée que dans l'état *courant* du monde, quant aux descriptions symboliques, elles doivent être vrais dans tous les mondes possibles. Ceci dit, les HTNs réactifs rencontrent des cassures, ce qui nous a mené à proposer notre solution ; à savoir une approche hybride dans laquelle un HTN réactif est augmenté avec de la connaissance symbolique afin d'aider à la réparation des cassures.

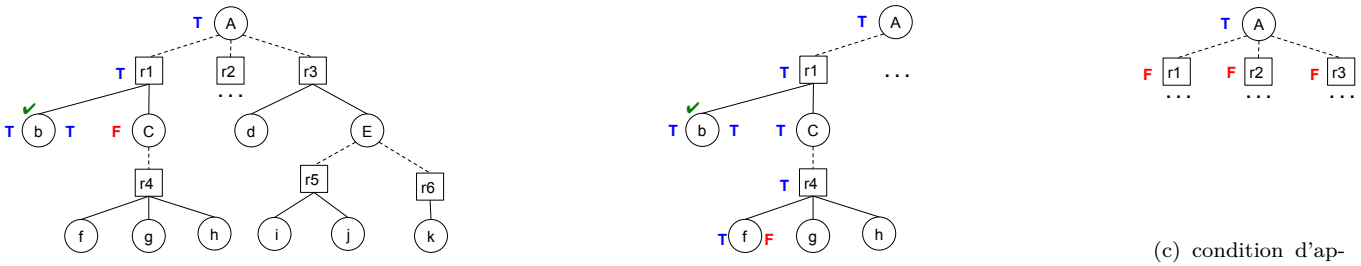
4 L'approche hybride

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

Un HTN réactif est un arbre biparti avec différents niveaux de nœuds *tâches* et nœuds de *décompositions*. Il est définit avec un nœud tâche au niveau de la racine et des feuilles. Ces dernières sont appelée *tâches primitives* et les autres sont appelées *tâches abstraites*. Les nœuds de l'arbre sont définis avec trois types de procédures booléennes, chacune étant évaluée uniquement dans l'état courant. Les nœuds tâches sont définis avec des *pré/postconditions* optionnelles et chaque nœud de décomposition a une *condition d'applicabilité*.

Les HTNs réactifs peuvent être représentés comme un arbre et/ou, où les tâches sont des "et", et les nœuds de



(a) précondition échouée (b) postcondition échouée (c) condition d'applicabilité échouée

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

décomposition sont des "ou". 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 dans la suite.
Si l'exécution du nœud courant est une tâche alors ses éventuelles préconditions 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 fausses alors l'exécution est arrêtée(cassure). Une fois l'exécution de la tâche terminée, ses éventuelles postconditions 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 (ex : le vent dans la section 2) ; ou a 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

Le 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. Cependant, pour ce même exemple, différents état buts peuvent être trouvés. Par exemple, supposons que *walkthru* ait une postcondition qui spécifie que le robot est dans la pièce à l'autre côté de la porte. Donc, après la cassure, le robot peut trouver une autre méthode de réparation ; trouver un plan pour satisfaire cette postcondition. (ex, sortir de la pièce actuelle par une autre porte, passer par autre chemin pour atteindre la pièce de destination). Cette procédure consiste à définir la postcondition comme le*but* d'un des candidats pour la réparation. De la même façon, supposons que *walkthru* n'a pas de représentation symbolique pour sa postcondition, mais la postcondition symbolique de *navigate* spécifie la location bute du robot. Dans ce cas la postcondition de *navigate* est considérée comme un état but de réparation. Il est a noté que les conditions d'applicabilités sont un cas a part pour le calcul des candidats. En effet, une condition d'applicabilité est considérée comme candidat dans l'unique cas où toutes ses sœurs dans l'arbre ont été évaluées comme fausses. L'idée est de construire l'ensemble des candidats le plus général possible qui inclut toutes les conditions du HTN qui n'ont pas déjà étaient validées durant l'exécution. Cette approche pourrait être excessivement générale, mais requière plus d'expérience pour construire une meilleure.

FIGURE 6 présente le pseudo code du système hybride conçu. La procédure principale, EXECUTE, execute un HTN jusqu'à ce qu'il retourne soit succès, soit une cassure est 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 changent pas durant l'exécution, Il n'est donc pas définit comme un argument explicite au planificateur symbolique.(voir section 4.3).

Plus en détail, on peut voir à la linge 6 que notre approche requière 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 le planificateur dans la section 2, chaque fonction symbolique comme "open" doit être associée à une procédure comme "isOpen()" afin de pouvoir calculer sa valeur dans l'état courant. Cette association est une des parties de la modélisation hybride de notre modèle (voir partie 4.3 pour plus de détails). La ligne 8 montre que les candidats sont triés par rapport à leurs distance du nœud courant dans l'arbre, en utilisant une

```
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 postcondition 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

simple métrique (Ex : le plus court chemin dans un arbre non dirigé). La raison principale est de favoriser des réparations locales qui préservent au maximum la structure originale du HTN. Cependant, nous n'avons pas encore testé les performances de cette heuristique. Finalement, à la ligne 12 de la procédure EXECUTE, quand un plan est trouvé, il doit être intégré dans le HTN. La difficulté de cette étape réside dans la distance entre le premier et le dernier nœud. Plus ces derniers sont distants plus les changement devant être effectués dans la structure du HTN sont compliqués

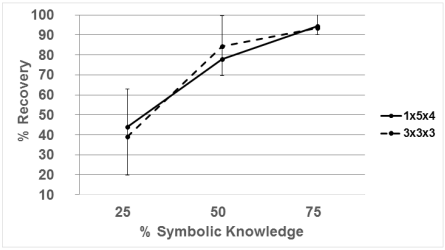
4.3 Méthodologie de modélisation

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 exposé précédemment, la difficulté liée à la réalisation d'une modélisation symbolique a mené 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 situer où il est préférable d'investir et l'effort à consacrer pour la 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 primaire consiste à réduire la modélisation symbolique aux tâches primitives, car nous espérons que l'utilisation d'une stratégie locale pour la réparation des cassures soit plus efficace. Le choix des opérateurs à introduire au planificateur linéaire ont une implication directe sur ses performances. En effet, seules les tâches primitives disposant d'une représentation symbolique de ses pré et postconditions 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 proposons de mettre en œuvre un outils qui aiderait à réduire la complexité de le modélisation hybride, qui pourrait reconnaitre les conventions entre les deux modélisations comme dans la FIGURE 3a et automatiquement générer sa représentation symbolique comme dans la FIGURE 3b. Il peut aussi garder l'historique des cassures et les utiliser pour conseiller l'auteur où il devrait ajouter des connaissances symboliques dans le HTN.

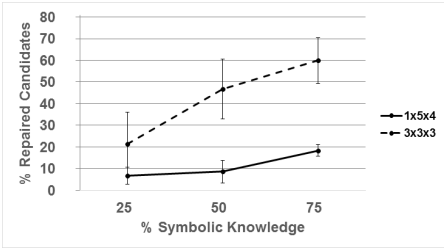
5 Implémentation et évaluation

Nous avons implémenté notre système hybride en Java en utilisant le standard ANSI/CEA-2012 [7] pour la modélisation du HTN et Disco [8] 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.
L'évaluation ultime de notre approche consiste à faire évoluer plusieurs agents dans des environnements dynamiques du monde réel et évaluer leur performances ainsi que leur robustesses face aux cassures. Entre temps, nous avons validé notre système avec une évaluation préliminaire sur des HTNs générés synthétiquement en variant le niveau de connaissance symbo-

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.

lique. Nous sommes partis d'une théorie simple qui préconise que plus il y'a des connaissances symboliques dans le HTN, mieux seront ses performances pour la réparation des cassures. FIGURE 7 montre les résultats obtenus qui confirment notre hypothèse. Nous avons testé deux configurations d'HTNs RxSxD, 3x3x3 et 1x5x4, 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. Pour chaque test, nous avons aléatoirement extrait toutes les 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. 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 notre 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 expliquer le compromis existant entre la modélisation symbolique et 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 mené pour valider le système et les résultats obtenus confirment nos théories de départ. Des futures perspectives sont envisagées. En effet, un travail de thèse est entamé qui consiste à intégrer notre modèle dans un système de dialogue social afin de gérer de manière opportuniste les éventuelles cassures.

Références

[1] AYAN, N. F., KUTER, U., YAMAN, F., AND GOLDMAN, R. P. 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] BOELLA, G., AND DAMIANO, R. A replanning algorithm for a reactive agent architecture. In *Artificial Intelligence : Methodology, Systems, and Applications*. Springer, 2002, pp. 183–192.

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

[4] EROL, K., HENDLER, J., AND NAU, D. S. Htn planning : Complexity and expressivity. In *AAAI* (1994), vol. 94, pp. 1123–1128.

[5] FIRBY, R. J. An investigation into reactive planning in complex domains. In *AAAI* (1987), vol. 87, pp. 202–206.

[6] GHALLAB, M., KNOBLOCK, C., WILKINS, D., BARRETT, A., CHRISTIANSON, D., FRIEDMAN, M., KWOK, C., GOLDEN, K., PENBERTHY, S., SMITH, D. E., ET AL. Pddl-the planning domain definition language.

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

[8] RICH, C., AND SIDNER, C. L. Using collaborative discourse theory to partially automate dialogue tree authoring. In *Intelligent Virtual Agents* (2012), Springer, pp. 327–340.

[9] VAN DER KROGT, R., AND DE WEERDT, M. Plan repair as an extension of planning. In *ICAPS* (2005), vol. 5, pp. 161–170.

[10] WARFIELD, I., HOGG, C., LEE-URBAN, S., AND MUNOZ-AVILA, H. Adaptation of hierarchical task network plans. In *FLAIRS Conference* (2007), pp. 429–434.