Flexibilité dans les plans pour une planification centrée humain

Cyrille Martin[1], Humbert Fiorino[2], and Gaëlle Calvary[3]

Université Joseph Fourier, Grenoble INP, CNRS UMR 5217
Laboratoire d'Informatique de Grenoble
1. cyrille.martin@imag.fr
2. humbert.fiorino@imag.fr
3. gaelle.calvary@imag.fr

Résumé : La flexibilité de l'interaction est une propriété fondamentale en interaction Homme-Machine. Elle est exacerbée en informatique ubiquitaire où l'utilisateur final devient le programmeur de son environnement : il convient dès lors de produire des plans partiellement résolus. Les plans dits *flexibles* permettent de différer certains choix de conception à l'exécution. Dans cet article, nous étudions des plans flexibles car permettant à l'utilisateur de décider de l'ordre d'application d'un traitement spécifique (corps de boucle) à un ensemble d'objets (variants de boucle).
Nous proposons l'algorithme λ-graphplan fondé sur l'approche de planification de graphe. λ-graphplan n'a besoin d'aucune connaissance spécifique pour fournir des plans flexibles. En relâchant des contraintes mutex dans le graphe de planification, λ-graphplan découvre les variants de boucle et construit des macro-actions constituant le corps des boucles. Nous montrons que λ-graphplan est performant aussi bien avec des domaines "itératifs" que "linéaires".

1 Introduction

Aujourd'hui, les technologies liées à l'informatique ubiquitaire offrent de nombreuses ressources (services et données) aux personnes connectées. Elles conduisent à des espaces numériques variés, variables et imprévisibles : les ressources de calcul et de communication apparaissent et disparaissent opportunément. Ainsi, les utilisateurs interagissent avec des espaces numériques dynamiques pouvant, en conséquence, créer des besoins opportunistes. Dès lors, le défi est de développer des systèmes interactifs capables de s'adapter aux besoins dynamiques des utilisateurs Aarts & de Ruyter (2009) et aux ressources disponibles Weiser (1999). En particulier, il n'est plus possible de supposer qu'un service spécifique sera disponible à un instant donné de l'interaction. Plus vraisemblablement, l'utilisateur interagira avec un ensemble de services pour satisfaire son objectif.

La plupart des utilisateurs ne peuvent pas gérer la composition de services eux-mêmes Guzdial *et al.* (1992) en raison de leur manque de compétences, connaissances, habilité etc. Par conséquent, la composition doit être automatique. Cet article explore la planification automatisée pour composer des services tout en tenant compte des différents contextes d'utilisation Rao & Su (2004). Par exemple, dans l'étude de cas de "l'agence de voyage", le but de l'utilisateur est de préparer un voyage à travers différentes villes telles que `Paris`, `Rome` et `Berlin`. À cette fin, l'utilisateur a accès à des services lui permettant de réserver des logements, des restaurants, d'obtenir des informations sur une ville, etc. Supposons que, pour chaque ville, l'utilisateur veuille trouver un hôtel *deux étoiles* et un restaurant *végétarien*. Cela nécessite que l'utilisateur spécifie ses destinations et ses préférences pour chacun des services, ce qui rend la tâche répétitive. Une composition automatique de services fournirait un plan solution représentant une séquence ordonnée d'actions correspondant aux réservations d'hôtels et de restaurants en accord avec les préférences de l'utilisateur. La séquence d'actions (le choix d'un hôtel puis d'un restaurant) est une macro-action appliquée à chaque ville ciblée. Toutefois, l'ordre des villes à visiter est une décision qui devrait être laissée à l'utilisateur au moment de l'exécution. En effet, seul l'utilisateur connaît certaines contraintes

cachées dépendantes du contexte, comme la nécessité d'apporter des cadeaux de `Paris` à ses parents habitant `Berlin` avant de retrouver des amis à `Rome`.

Un *plan flexible* laisse l'utilisateur décider de l'ordre de traitement des objets soumis à un traitement spécifique (appelé macro-action car composé d'un ensemble d'actions considéré comme une unique action), ce qui est essentiel dans les approches centrées humain. À notre connaissance, les planificateurs capables de trouver des plans flexibles requièrent l'identification des objets soumis à la flexibilité, ce qui n'est pas compatible avec les exigences de l'informatique ubiquitaire. Dans cet article, nous proposons un algorithme de planification qui calcule des plans flexibles à partir de domaine et des problèmes de planification de type STRIPS, c'est-à-dire sans information quant à la flexibilité attendue. Gripper Bonet *et al.* (2009) est utilisé pour illustrer nos concepts tout au long de cet article : *un robot peut attraper une balle (l'opérateur de planification `PICK`), la relâcher (`DROP`) et changer de pièce (`MOVE`). Dans l'état initial, la pièce de gauche (`Left`) contient n balles, celle de droite (`Right`) est vide et le robot se tient dans la pièce vide.* L'utilisateur a pour objectif de *déplacer les n balles de la pièce `Left` vers `Right`* au moyen du robot. Le plan suivant est dit flexible puisque il répond au besoin de l'utilisateur tout en le laissant libre de l'ordre de traitement des balles. Le corps de la boucle *while* est une *macro-action*. Les balles sont les *variants de boucle*. Le

```
1 B ← {Ball_1, ..., Ball_n };
2 while B is not empty do
3 |   User selects ball ∈ B ;
4 |   B ← B − ball ;
5 |   MOVE(Right, Left);
6 |   PICK(ball, Left);
7 |   MOVE(Left, Right);
8 |   DROP(ball, Right);
```

domaine de planification pour la variante de Gripper utilisée dans cet article ainsi qu'un exemple de problème (avec le nombre de balles $n = 2$) sont disponibles à l'annexe A.

Nous proposons une revue de l'état de l'art dans le paragraphe 2. L'approche "graph-based planning" est présentée au paragraphe 3. Notre proposition est décrite au paragraphe 4 et évaluée au paragraphe 5.

2 Travaux connexes

À notre connaissance, il n'existe pas d'approche en planification conçue pour la flexibilité, c'est-à-dire pour laisser libre l'ordonnancement des objets à traiter dans une macro-action. Toutefois, deux approches sont connexes.

En Model-Based Planning (MBP) Giunchiglia & Traverso (2000), les domaines de planification sont des graphes qui représentent la dynamique de l'espace d'états : les nœuds sont des états du monde ; les arcs correspondent aux actions exécutables sur ces états. La solution d'un problème de planification est une politique, c'est-à-dire un ensemble de tuples ⟨état atteint, action à exécuter⟩. Comme un domaine non déterministe implique des changements incertains d'états en MBP Cimatti *et al.* (2003), certaines politiques sont équivalentes à des plans avec boucles. En effet, pour résoudre le problème Gripper en MBP, le domaine peut être décrit en tant qu'espace d'états où les deux pièces peuvent être vides ou non et où l'action non déterministe `PICK` peut vider ou non la pièce considérée. L'objectif de l'utilisateur est alors de vider la pièce `Left` et non de déplacer les balles dans la bonne pièce. Ainsi, les balles en tant qu'objets de planification ne sont pas explicitées. Le plan solution est un chemin dans le graphe : il est implicitement encodé dans le domaine via la notion de balle comme variant de boucle. Seul le concepteur du domaine sait que les pièces contiennent des balles et que l'action `PICK` s'applique aux balles. En conséquence, avec ce domaine de planification, il n'est pas possible de spécifier un objectif concernant un sous ensemble de balles. Ces approches supposent que tous les objectifs des utilisateurs soient anticipés au moment de la

conception par la description du domaine de planification.

Dans les travaux de "*generalized planning*", les plans résolvent un ensemble de problèmes instances d'une même classe. Une classe de problèmes contient une description de *variables* dont les valeurs correspondent aux différentes instances de problèmes.
Dans Hu & Levesque (2009), la variable est un entier qui indique le nombre d'objets à traiter. Cette variable est utilisée comme variant de boucle dans le plan engendré. Par exemple, dans Gripper, le nombre de balles est une variable. Le but de l'utilisateur n'est pas explicitement décrit comme une collection de balles à déplacer. Par conséquent, le plan solution peut fonctionner avec n'importe quel nombre de balles. Toutefois, il n'est notamment pas possible de résoudre des objectifs sur d'autres variants.
Srivastava *et al.* (2008) utilise la notion de *prédicats d'abstraction* pour définir des *rôles*. Ces rôles rendent possible la transformation d'un exemple de plan en un espace d'états abstrait. De cette transformation, l'algorithme trouve des boucles sur les objets jouant le rôle défini. Le plan peut alors être appliqué à n'importe quel problème instance de la classe correspondante. Pour résoudre Gripper, un rôle "balle dans la pièce `Left`" doit être défini de façon à abstraire toutes les balles de la pièce de gauche dans un unique objet abstrait. Une limite de cette approche est que les rôles doivent être définis à la conception à travers la liste explicite des *prédicats d'abstraction*.

3 Fondements de λ-graphplan

λ-graphplan est une extension de l'approche "graph-based planning" Blum & Furst (1997). Un graphe de planification $\mathcal{G}$ pour un domaine et un problème donnés est une représentation condensée de l'espace d'états. Il est composé de couches d'actions $\mathcal{A}_i$ et de couches de propositions $\mathcal{P}_i$. $\mathcal{A}_i$ (resp. $\mathcal{P}_i$) contient les actions (resp. les propositions) qui peuvent être déclenchées (resp. atteintes) au pas de temps i.

Les préconditions d'une action $a \in \mathcal{A}_i$ sont des propositions de $\mathcal{P}_{i-1}$, alors liées à a. De même, les propositions de $\mathcal{P}_i$ sont des effets positifs ou négatifs des actions de $\mathcal{A}_i$. De plus, chaque proposition $p \in \mathcal{P}_{i-1}$ est la précondition d'une action de propagation *noop-p* de $\mathcal{A}_i$ qui ne possède qu'un seul effet positif $p \in \mathcal{P}_i$ et aucun effet négatif.

Deux actions a_1 et a_2 de $\mathcal{A}_i$ sont liées par une relation *mutex* si soit un effet négatif de a_1 est une précondition ou un effet positif de a_2, soit une précondition de a_1 est liée par une relation mutex avec une précondition de a_2 dans $\mathcal{P}_{i-1}$. Deux propositions p_1 et p_2 de $\mathcal{P}_i$ sont *mutex* si toute action de $\mathcal{A}_i$ qui a pour effet positif p_1 est mutex avec toutes les actions de $\mathcal{A}_i$ qui produisent p_2, et si aucune action de $\mathcal{A}_i$ ne produit simultanément p_1 et p_2. $\mu\mathcal{A}_i$ et $\mu\mathcal{P}_i$ désignent respectivement les ensembles de relations mutex entre les actions de $\mathcal{A}_i$ et entre les propositions de $\mathcal{P}_i$. Les relations mutex sont binaires et symétriques. Le sens à leur donner est que deux actions (resp. propositions) en relation mutex au pas de temps i, autrement dit dont le couple appartient à $\mu\mathcal{A}_i$ (resp. à $\mu\mathcal{P}_i$), ne peuvent pas être déclenchées (resp. atteintes) simultanément.

$\mathcal{P}_0$ représente l'état initial du problème. Le graphe de planification est *étendu* pas à pas selon les règles suivantes : une action est ajoutée à $\mathcal{A}_i$ si ses préconditions ne sont pas mutex dans $\mathcal{P}_{i-1}$. Une proposition p est ajoutée à $\mathcal{P}_i$ si une action de $\mathcal{A}_i$ a p pour effet positif ou négatif. Le graphe de planification est étendu jusqu'à ce que la couche courante $\mathcal{P}_g$ contienne un état consistant de l'objectif de planification, c'est-à-dire que toutes les propositions de l'objectif ne sont pas mutex entre elles. Lorsque $\mathcal{P}_g$ est atteint, Graphplan Blum & Furst (1997) tente d'*extraire* un plan de $\mathcal{G}$: chaque proposition sélectionnée de la couche $\mathcal{P}_i$ doit être produite par au moins une action de la couche $\mathcal{A}_i$. Les actions qui produisent les propositions sélectionnées doivent former un ensemble sans relation *mutex* entre elles. Toutes les préconditions de cet ensemble d'actions de $\mathcal{A}_i$ sont sélectionnées dans $\mathcal{P}_{i-1}$ et ainsi de suite. Ce processus de sélection commence avec les propositions formant l'objectif de planification dans la dernière couche propositionnelle de $\mathcal{G}$ et s'arrête lorsque $\mathcal{P}_0$ est atteint. Si aucun plan n'est trouvé, le graphe de planification est étendu avec une couche supplémentaire, jusqu'à ce qu'un plan soit extrait ou que la condition de terminaison soit atteinte.

Tout comme Graphplan, λ-graphplan progresse par étapes successives d'extension et d'extraction (Figure 1). Il utilise la même condition de terminaison et la même définition des relations mutex.

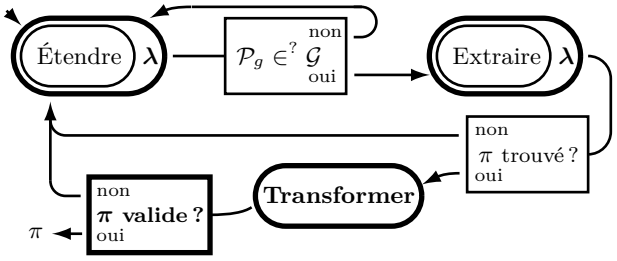

FIGURE 1 – Illustration de l'algorithme λ-graphplan : les extensions à Graphplan sont en **gras**.

En revanche, en plus de ces principes, il introduit le concept de λ-mutex pour l'étiquetage des actions candidates pour une structure de boucle. Un plan λ-solution est un plan en couches de telle façon que les actions de chaque couche sont ou bien sans relation mutex ou forment des cliques sous la relation λ-mutex : la relation λ-mutex est un relâchement des contraintes dues aux relations mutex. Lorsque π, un plan λ-solution, est découvert, λ-graphplan applique successivement des transformations sur π pour découvrir des variants de boucle et construire des structures de boucle. Si la transformation réussit, le plan π modifié est retourné. Sinon, le processus d'extension est repris.

4 Algorithme λ-graphplan

Dans les paragraphes suivants, un *opérateur* o est un 3-tuple $\langle pre(o), \mathit{eff}^+(o), \mathit{eff}^-(o)\rangle$ où $pre(o)$, $\mathit{eff}^+(o)$ et $\mathit{eff}^-(o)$ sont des ensembles de prédicats partageant un ensemble de variables $param(o)$. Ils représentent respectivement les préconditions, les effets positifs et négatifs de l'opérateur. Une *action* a est le résultat de l'application d'une substitution σ sur un opérateur o. Chaque prédicat de $pre(o)$, $\mathit{eff}^+(o)$, $\mathit{eff}^-(o)$ devient une proposition. Cela signifie que pour chaque variable v de $param(o)$, une substitution $(v\backslash t)$ existe dans σ et que $a = \sigma(o)$. Par exemple, `MOVE (Left, Right)` est une action dérivée de l'opérateur `MOVE (`$room_1$`,` $room_2$`)` par l'application de la substitution $\{(room_1\backslash$ `Left`$)\,;\,(room_2\backslash$ `Right`$)\}$. Une *action partiellement instanciée* résulte de $\sigma(o)$ s'il existe v dans $param(o)$ tel que $(v\backslash t)$ n'existe pas dans σ. Par exemple, `PICK (`*ball*`, Left)` est une action partiellement instanciée.

4.1 Étendre

λ-graphplan ajoute des relations appelées λ-mutex de façon à relâcher certaines contraintes dans le processus d'extraction de Graphplan : les actions d'une étape du plan ne doivent toujours pas être mutex entre elles, sauf dans le cas des actions susceptibles d'appartenir à une macro-action. Ces dernières sont différenciées dans le graphe de planification par les relations λ-mutex. Ces relations concernent les actions mutex dérivées d'un même opérateur. Plus formellement :

Definition 1 (λ-mutex– Actions)
Deux actions a_1 et a_2 de la couche $\mathcal{A}_i$ sont λ-mutex *si :*
- $(a_1, a_2) \in \mu\mathcal{A}_i$ *; et*
- *il existe un opérateur o et les substitutions σ_1 et σ_2 tels que $a_1 = \sigma_1(o)$ et $a_2 = \sigma_2(o)$*

Les relations λ-mutex entre les actions sont ajoutées dans le graphe de planification lors de son expansion. $\lambda\mathcal{A}_i$ est l'ensemble des actions λ-mutex : $\forall i, \lambda\mathcal{A}_i \subseteq \mu\mathcal{A}_i$. Si $\sigma_a = \sigma_1 \cap \sigma_2$, alors $a_a = \sigma_a(o)$ est une action partiellement instanciée qui représente les deux actions a_1 et a_2. Ainsi, a_a sera essentielle pour la construction de macro-actions et donc des structures de boucle.

Les propositions λ-mutex sont celles qui sont accessibles au pas de temps i et qui dérivent d'actions pouvant être intégrées à une structure de boucle. Plus formellement :

Definition 2 (λ-mutex– Propositions)
Deux propositions p_1 et p_2 de la couche $\mathcal{P}_i$ sont λ-mutex *si :*
- $(p_1, p_2) \in \mu\mathcal{P}_i$ *; et*

– *chaque action de la couche $\mathcal{A}_i$ qui a p_1 en tant qu'effet positif est* λ-mutex *avec chaque action de la couche $\mathcal{A}_i$ qui produit p_2.*

$\lambda\mathcal{P}_i$ est l'ensemble des propositions λ-mutex de la couche $\mathcal{P}_i$ et $\forall i, \lambda\mathcal{P}_i \subseteq \mu\mathcal{P}_i$.

À chaque couche i du graphe de planification $\mathcal{G}$, les informations disponibles sont $\mathcal{A}_i$, $\mu\mathcal{A}_i$, $\lambda\mathcal{A}_i$, $\mathcal{P}_i$, $\mu\mathcal{P}_i$ et $\lambda\mathcal{P}_i$.

4.2 Extraire

Pour chaque couche i, $\mu\mathcal{A}_i \setminus \lambda\mathcal{A}_i$ est l'ensemble des couples d'actions qui ne peuvent pas être sélectionnés ensemble. De même, $\mu\mathcal{P}_i \setminus \lambda\mathcal{P}_i$ contient les couples de propositions qui ne peuvent pas être atteints ensemble à la couche i. En particulier, l'objectif est considéré comme accessible à la couche $\mathcal{P}_g$ même si celle-ci contient des propositions λ-mutex entre elles. Dans le domaine Gripper, l'objectif est formé des propositions `at Left Ball`$_i$ pour chaque balle i. Graphplan développe le graphe de planification jusqu'à ce que les propositions formant l'objectif ne contiennent plus de relations mutex, c'est-à-dire jusqu'à ce que la couche $\mathcal{P}_7$ soit atteinte (quel que soit le nombre de balles). λ-graphplan arrêtera le développement du graphe de planification à la couche $\mathcal{P}_4$ car les propositions formant l'objectif sont toutes λ-mutex entre elles. Ainsi, λ-graphplan essaie généralement d'extraire un plan avant Graphplan (en terme de nombre de couches à développer). Dans Gripper, le plan extrait par λ-graphplan est le suivant :

---

1 `MOVE(Right, Left)`;
2 `PICK(Ball`$_1$`, Left)`, ..., `PICK(Ball`$_n$`, Left)`;
3 `MOVE(Left, Right)`;
4 `DROP(Ball`$_1$`, Right)`, ..., `DROP(Ball`$_n$`, Right)`;

Ce plan contient des incohérences puisque le robot n'est pas en capacité d'attraper plus d'une balle en même temps. Ce plan doit subir des transformations de façon à insérer des structures de boucles pour permettre le traitement de chaque balle l'une après l'autre.

4.3 Transformer

L'ensemble des transformations se fait en trois étapes. La première consiste à trouver les variants de boucle dans le plan extrait, tout en "réduisant" certaines actions dans des actions partiellement instanciées. Les variables de ces actions partiellement instanciées prennent pour valeur les variants de boucle découverts auparavant. Ensuite, la "fermeture interne" construit le corps des boucles par agrégation d'actions dans des macro-actions. Finalement, la "fermeture externe" complète les macro-actions de façon à ce que leurs effets et préconditions soient cohérents pendant les itérations.

4.3.1 Actions généralisées et variants de boucle

Les actions λ-mutex sont candidates à la formation de boucles. Mais la première étape consiste à rechercher une représentation commune de ces actions, de façon à révéler les objets de planification utilisés dans la boucle à former, c'est-à-dire les *variants de boucle*. Par exemple, *ball* $\in$ $\{$`Ball`$_1, \ldots,$ `Ball`$_n\}$ est exploité par les actions `Pick(Ball`$_1$`, Left)`, `Pick(Ball`$_2$`, Left)` etc. Ces actions sont généralisées dans une action `Pick(`*ball*`, Left)`. Plus formellement, une *action généralisée* α (pour l'étape i) est une action partiellement instanciée définie comme suit :

– Soit $\mathcal{S}_i = \{a_1, \ldots, a_n\}$ un ensemble d'actions tel que $\forall a_a, a_b \in \mathcal{S}_i, (a_a, a_b) \in \lambda\mathcal{A}_i$, autrement dit $\mathcal{S}_i$ sous la relation λ-mutex est une clique : cela garantit que les actions sont dérivées du même opérateur ;
– $\forall a \in \mathcal{S}_i, \exists\sigma \mid a = \sigma(\alpha)$, autrement dit α représente toutes les actions de $\mathcal{S}_i$. $var(\alpha)$, les variables de α sont les variants de boucle.

Les actions généralisées sont composées de :

– $spre(\alpha)$, l'ensemble des propositions partagées ("*shared preconditions*") dans les préconditions des actions de $\mathcal{S}_i$: $spre(\alpha) = \bigcap\{pre(a) \mid a \in \mathcal{S}_i\}$. De même, $seff^+(\alpha)$ et $seff^-(\alpha)$ sont les

ensembles de propositions partagées dans les effets respectivement positifs et négatifs des actions de $\mathcal{S}_i$. Les propositions partagées représentent les préconditions et effets qui n'affectent pas les variants ;
- *upre*(α), l'ensemble des propositions non partagées ("*unshared preconditions*") dans les préconditions des actions de $\mathcal{S}_i$: *upre*$(\alpha) = \bigcup\{pre(a) \mid a \in \mathcal{S}_i\} \setminus spre(\alpha)$. De même, $ueff^+(\alpha)$ and $ueff^-(\alpha)$ sont les ensembles de propositions non partagées dans les effets respectivement positifs et négatifs des actions de $\mathcal{S}_i$. Les propositions non partagées représentent les préconditions et effets liés aux variants de boucle.

De plus, comme les actions non généralisées n'affectent pas les variants de boucles, leurs préconditions et effets sont "partagés". Dans l'exemple Gripper, le plan ainsi transformé devient le suivant :

```
1 MOVE(Right, Left);
2 PICK(ball ∈ {Ball_1, ..., Ball_n }, Left);
3 MOVE(Left, Right);
4 DROP(ball ∈ {Ball_1, ..., Ball_n }, Right);
```

Après avoir trouvé les actions généralisées appropriées pour toutes les actions sous relations λ-mutex, la transformation suivante a pour objectif de trouver les actions généralisées qui doivent appartenir au même corps de boucle.

4.3.2 Fermeture interne

Pour cela, λ-graphplan construit des *macro-actions paramétrées*. Nous utilisons une notion de macro-action très proche de celle de macro-opérateur dans Botea *et al.* (2005); Coles *et al.* (2007); Newton *et al.* (2007). Une macro-action $\mathcal{M}$ est considérée comme une *action unique* sous la forme d'un sous-plan : $\mathcal{M} = \langle \pi_1, \ldots, \pi_n \rangle$ où π_i sont les ensembles d'actions et d'actions généralisées. La construction d'une macro-action débute par une macro-action vide $\mathcal{M} = \langle\rangle$ et chaque ensemble de propositions partagées ou non est vide. Lorsqu'un ensemble d'actions (généralisées ou non) $\pi_i = \{a_1, \ldots, a_n\}$ est concaténé à $\mathcal{M}$, les ensembles *spre*$(\mathcal{M})$, *upre*$(\mathcal{M})$ etc. sont mis à jour. Les règles de mise à jour sont les suivantes :
- les préconditions partagées de $\mathcal{M}$ sont mises à jour avec les préconditions partagées de chaque action dans π_i qui ne sont pas des effets positifs partagés de $\mathcal{M}$. De même, les préconditions non partagées de $\mathcal{M}$ sont mises à jour avec les préconditions non partagées de chaque action dans π_i qui ne sont pas des effets positifs non partagés de $\mathcal{M}$;
- aux effets positifs partagés de $\mathcal{M}$ sont ajoutés les effets positifs partagés de chaque action a de π_i et enlevés les effets négatifs partagés de chaque action a ; les effets positifs non partagés sont mis à jour de façon similaire ;
- aux effets négatifs partagés de $\mathcal{M}$ sont ajoutés les effets négatifs partagés de chaque action a de π_i et enlevés les effets positifs partagés de chaque action a ; les effets négatifs non partagés sont mis à jour de façon similaire ;

Les actions à ajouter dans les macro-actions sont sélectionnées de la façon suivante :
- toutes les actions généralisées doivent être incluses dans des macro-actions qui manipulent les mêmes variants et qui sont liés par des propositions produites et consommées : $\mathcal{C}_k$ est une action généralisée et $\mathcal{M}_i$ une macro-action de telle façon que *upre*$(\mathcal{C}_k) \cap ueff^+(\mathcal{M}_i) \neq \emptyset$ et $\mathcal{C}_k$ apparaît après $\mathcal{M}_i$. Dans ce cas, $\mathcal{C}_k$ est incluse dans $\mathcal{M}_i$. Par exemple, à l'étape 2 l'action généralisée $\mathcal{C}_{PICK}$ = `PICK (`*ball* $\in$ `{Ball`$_1$`, ..., Ball`$_n$`}, Left)` a pour effet d'attraper les balles, c'est-à-dire de produire les propositions `carry Ball`$_i$ pour chaque balle i. Ces propositions appartiennent donc à l'ensemble $ueff^+(\mathcal{C}_{PICK})$, mais aussi à l'ensemble *upre* de l'action généralisée $\mathcal{C}_{DROP}$ dérivée des actions `DROP` à l'étape 4. Cela implique que ces deux actions généralisées sont incluses dans la même macro-action ;
- des actions sont ajoutées dans les macro-actions de façon à provoquer la *fermeture interne*, c'est-à-dire pour soutenir les préconditions partagées des actions généralisées. Par exemple, pour déclencher $\mathcal{C}_{PICK}$, le robot doit être dans la pièce de gauche, autrement dit la proposition `at-g Left` appartient à *spre*$(\mathcal{C}_{PICK})$. Cette précondition est nécessairement produite à cet instant du plan, puisque $\mathcal{C}_{PICK}$ est la première action généralisée de la macro-action : des

actions des pas de temps précédents ont produit cette proposition. Soit s l'état du monde juste avant le commencement de la macro-action. L'une des préconditions de $\mathcal{C}_{DROP}$ est la proposition `at-g Right`. Parce que cette proposition n'est pas produite par l'état résultant de l'application de $\mathcal{C}_{PICK}$ sur s, une action entre $\mathcal{C}_{PICK}$ et $\mathcal{C}_{DROP}$ doit pouvoir produire `at-g Right`. Cette action existe obligatoirement puisque $\mathcal{C}_{DROP}$ a été extraite (toutes les actions généralisées dans $\mathcal{C}_{DROP}$ sont atteignables) : cette action est `MOVE(Left, Right)`. Finalement, la macro-action est formée par $\mathcal{C}_{PICK}$, `MOVE(Left, Right)` et enfin $\mathcal{C}_{DROP}$.

Une fois le robot dans la pièce de gauche, cette macro-action permet d'attraper une balle et de la relâcher dans l'autre pièce.

---

parameter: *ball* $\in$ {$\texttt{Ball}_1$,..., $\texttt{Ball}_n$ }
1 `PICK(`*ball,* `Left);`
2 `MOVE(Left,` `Right);`
3 `DROP(`*ball,* `Right);`

4.3.3 Fermeture externe

Les macro-actions ne suffisent pas pour créer des boucles. Il faut en plus s'assurer du bon déroulement des itérations successives, c'est-à-dire que les états du monde entre le début et la fin de la macro-action doivent partager des propositions qui représentent les invariants de boucle : ces éléments sont les propositions formant les préconditions partagées de la macro-action considérée. Cela permet d'assurer la bonne exécution de chaque action dans la macro-action. Ainsi, si l'état du monde à la fin de la macro-action supporte les préconditions partagées de la macro-action, la macro-action pourra être exécutée sans problème.

Dans notre exemple, la macro-action $\mathcal{M}$ permet de déplacer une balle de la pièce de gauche dans celle de droite. Les préconditions partagées *spre*($\mathcal{M}$) de la macro-action contiennent la proposition `at-g Left`, ce qui signifie que le robot doit être dans la pièce de gauche pour attraper une balle. Un des effets de $\mathcal{M}$ est de déplacer le robot dans la pièce de droite. Autrement dit, `at-g Left` appartient aux effets négatifs partagés de $\mathcal{M}$, et `at-g Right` à ses effets positifs partagés. Ainsi, l'état du monde s au début de $\mathcal{M}$ est différent de l'état t atteint par l'application des effets partagés ($seff^-$ et $seff^+$) depuis l'état s : c'est un nouveau problème de planification où *spre*($\mathcal{M}$) est l'objectif de planification et t l'état initial. S'il est résolu par λ-graphplan, le plan solution est ajouté à la macro-action. Dans Gripper, ce plan est composé d'une action, `MOVE(Right, Left)`, qui est ajoutée à $\mathcal{M}$.

Si λ-graphplan réussit la fermeture de toutes les macro-actions, il les transforme en structures de boucle et permet à l'utilisateur final de sélectionner l'ordre d'exécution pour les objets liés au variant de boucle. Pour le dernier objet traité dans la structure de boucle, les actions ajoutées pour permettre la fermeture externe de la macro-action ne doivent pas être exécutées. En effet, le plan initial a été extrait en considérant un unique objet. Ainsi, l'état attendu en sortie de boucle est le résultat de l'application de la macro-action sans les actions pour la fermeture externe de la boucle. Dans Gripper, le plan retourné est le suivant :

---

1 `MOVE(Right,` `Left);`
2 $\mathcal{B} \leftarrow$ {$\texttt{Ball}_1$,..., $\texttt{Ball}_n$ };
3 **while** *true* **do**
4 | User selects *ball* in $\mathcal{B}$;
5 | $\mathcal{B} \leftarrow \mathcal{B} - ball$;
6 | `PICK(`*ball,* `Left);`
7 | `MOVE(Left,` `Right);`
8 | `DROP(`*ball,* `Right);`
9 | **if** $\mathcal{B}$ *is empty* **then** break;
10 | `MOVE(Right,` `Left);`

Pour rappel, les étapes 6 à 10 forment la macro-action, l'étape 7 a permis la fermeture interne de

Monde des blocs			
n	Graphplan perf. moy (ms)	λ-graphplan perf. moy (ms)	coût supp. (%)
2	24	25	4
4	34	39	10
6	94	134	30
8	160	263	39
10	189	295	36
12	292	396	26
14	357	514	31
16	468	644	27
18	574	828	31
20	823	1245	34

TABLE 1 – Les performances moyennes en *ms* de Graphplan et λ-graphplan pour 100 exécutions de chaque problème du "monde des blocs" avec différents nombres n de blocs. Le coût supplémentaire de λ-graphplan par rapport à Graphplan est donné en pourcentage.

cette macro-action et l'étape 10 sa fermeture externe (à ne pas exécuter à la dernière itération, pour être dans l'état attendu par les actions suivantes).

5 Évaluation de λ-graphplan

Le calcul des relations λ-mutex n'est pas plus coûteux que celui des relations mutex. Ainsi, comme pour Graphplan, la construction du graphe de planification a un coût polynomial Blum & Furst (1997).

Nous évaluons les performances [1] de λ-graphplan en tant qu'extension de Graphplan, c'est-à-dire en mesurant le coût du calcul de la flexibilité dans les plans. Tout d'abord, nous étudions ces performances dans la situation la plus défavorable. Cela se produit avec les domaines "linéaires", c'est-à-dire ceux qui n'ont qu'une solution séquentielle. C'est par exemple le cas avec un "monde des blocs" dans lequel les n blocs d'une pile doivent être posés sur la table. Nécessairement, le plan solution dépile le bloc du sommet, le dépose sur la table et ainsi de suite (il n'existe donc aucune option ouverte pour l'utilisateur final). Le tableau 1 compare les résultats obtenus entre Graphplan et λ-graphplan : λ-graphplan est entre 30% et 40% plus lent que Graphplan, ce qui représente la charge liée à la gestion des relations λ-mutex.

Le domaine Gripper est favorable à λ-graphplan (cf. tableau 2) : ses performances sont largement meilleures que celle de Graphplan. Pour 3 balles, λ-graphplan trouve une solution 51% plus rapidement que Graphplan. Cela est dû à la taille réduite des plans trouvés par λ-graphplan, ce qui limite l'extension du graphe de planification et donc le coût lié à l'extraction de la solution. λ-graphplan calcule ici des cliques maximales sur les ensembles d'actions λ-mutex de façon à créer les actions généralisées. Le problème du calcul des cliques maximales est NP-complet. Bien que ce calcul n'est fait qu'une fois par couche du graphe de planification, c'est une part importante du processus d'extraction qui dépend du nombre d'actions λ-mutex. Lors de nos tests avec Gripper, le coût relatif pour le calcul des cliques devient supérieur à 50% au-delà de 8 balles.

6 Conclusion

Dans cet article, nous avons présenté λ-graphplan, un algorithme de planification fondé sur Graphplan permettant le calcul de plans flexibles. La flexibilité porte sur l'ordonnancement de tâches. Contrairement aux autres planificateurs, aucune connaissance relative à la flexibilité n'est

1. Tous les tests sont exécutés sur un processeur Intel Core 2 Duo de 2.6 GHz, avec 4 Go de mémoire vive DDR2 à 667 MHz.

Gripper			
n	Graphplan perf. moy (ms)	λ-graphplan perf. moy (ms)	coût (%) max-clique
1	27	30	0.0
2	37	40	2.7
3	68	45	3.1
4	328	57	3.9
5	471	58	6.6
6	947	72	9.0
7	3507	100	27.7
8	16241	203	59.5
9	–	406	79.4
10	–	687	86.6

TABLE 2 – Les performances moyennes en *ms* de Graphplan et λ-graphplan pour 100 exécutions de chaque problème du domaine Gripper avec différents nombres n de balles. Le coût max-clique est le temps relatif utilisé au calcul des cliques maximales par rapport au temps total du calcul de la solution.

déclarée à la conception : les variants et les corps de boucle sont découverts. Nous montrons que cette flexibilité ne s'obtient pas au détriment de la performance : l'algorithme est même plus efficace que Graphplan sur certains domaines.

Ce travail est une contribution à la planification sensible au contexte. À moyen terme, l'effort sera porté sur les boucles imbriquées.

Remerciements

Merci à Damien PELLIER pour pddl4j.

A Gripper en PDDL (domaine et problème)

Le code suivant est le domaine de planification correspondant à Gripper, écrit en PDDL.

```
(define (domain GRIPPER)
    (:requirements :strips)
    (:predicates
        (room ?room) (ball ?ball)
        (g-at ?room) (at ?ball ?room)
        (carry ?ball)  (free))
    (:action MOVE
        :parameters (?from ?to)
        :precondition (and (room ?from)
                           (room ?to)
                           (g-at ?from))
        :effect       (and (g-at ?to)
                      (not (g-at ?from))))
    (:action PICK
        :parameters (?ball ?room)
        :precondition (and (ball ?ball)
                           (room ?room)
                           (g-at ?room)
                           (at ?ball ?room)
                           (free))
        :effect       (and (carry ?ball)
                      (not (at ?ball ?room))
```

```
                    (not (free))))
    (:action DROP
        :parameters (?ball ?room)
        :precondition (and (ball ?ball)
                           (room ?room)
                           (g-at ?room)
                           (carry ?ball))
        :effect       (and (at ?ball ?room)
                           (free)
                      (not (carry ?ball)))))
```

Le problème de planification utilisé comme illustration tout au long de cet article utilise n balles. Nous proposons ci-dessous la description en PDDL de ce problème pour $n = 2$.

```
(define (problem TwoBalls)
    (:domain Gripper)
    (:requirements :strips)
    (:objects Right Left Ball1 Ball2)
    (:init
        (room Right)
        (room Left)
        (g-at Right)
        (free)
        (ball Ball1)
        (ball Ball2)
        (at Ball1 Left)
        (at Ball2 Left)
    )
    (:goal
        (and
            (at Ball1 Right)
            (at Ball2 Right))))
```

Références

Aarts E. & de Ruyter B. (2009). New research perspectives on ambient intelligence. *JAISE*, **1**, 5–14.

Blum A. & Furst M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, **90**(1), 281–300.

Bonet B., Palacios H. & Geffner H. (2009). Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*.

Botea B., Enzenberger M., Müller M. & Schaeffer J. (2005). Macro-FF : Improving AI planning with automatically learned macro-operators. *JAIR*, **24**, 581–621.

Cimatti A., Pistore M., Roveri M. & Traverso P. (2003). Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, **147**, 35–84.

Coles A., Fox M. & Smith A. (2007). Online identification of useful macro-actions for planning. In *ICAPS*.

Giunchiglia F. & Traverso P. (2000). Planning as model checking. In *ECP*.

Guzdial M., Reppy J. & Smith R. (1992). *Report of the 'User/Programmer Distinction' working group*, p. 367–383. A. K. Peters, Ltd.

Hu Y. & Levesque H. (2009). Planning with loops : Some new results. In *ICAPS Workshop on Generalized Planning*.

Newton H., Levine J., Fox M. & Long D. (2007). Learning macro-actions for arbitrary planner and domains. In *ICAPS*.

Rao J. & Su X. (2004). A survey of automated web service composition methods. In *SWSWPC*.

Srivastava S., Immerman N. & Zilberstein S. (2008). Learning generalized plans using abstract counting. In *AAAI Conference on Artificial intelligence*.

Weiser M. (1999). The computer for the 21st century. *SIGMOBILE MC2R*, **3**, 3–11.