Projet de Recherche

Architecture de contrôle hybride pour une planification robuste

Réalisé par : Koudjo AGBEDANOU

Encadrant :

M. Halim Djerroud

LISV
Laboratoire d'ingénierie des systèmes de Versailles

UVSQ
université PARIS-SACLAY

Hiver 2026

Table des matières

Introduction

La manipulation robotique autonome, et plus particulièrement la tâche de rangement d'objets dans un environnement même structuré, constitue un défi majeur de l'intelligence artificielle contemporaine. Dans ce contexte, l'utilisation de bras manipulateurs légers, tels que le *xArm6 Lite*, couplés à des systèmes de vision par marqueurs *ArUco* (caméra ZED2i), impose le développement d'architectures décisionnelles capables de concilier stratégie à long terme et réactivité immédiate.

Traditionnellement, cette problématique est abordée via le paradigme de la planification automatique déterministe utilisant des planificateurs basés sur un langage de description standardisé PDDL (*Planning Domain Definition Language*) [1]. Ces approches, incarnées par des planificateurs performants tels que *Fast Downward* [2], permettent de générer des séquences d'actions optimales en supposant un monde clos [1] et statique. Cependant, comme il sera observé dans l'architecture en boucle ouverte, un agent s'avère extrêmement vulnérable aux perturbations environnementales imprévues.

Le travail présenté dans ce rapport retrace la transition d'un système rigide (boucle ouverte) vers une architecture en boucle fermée capable de planification réactive. Si cette réactivité événementielle peut être structurée localement par des Arbres de Comportement (*Behavior Trees ou BT*) [3, 4, 5], une approche hybride a été privilégié où la détection sensorielle pilote directement une replanification globale, garantissant ainsi le maintien d'une cohérence logique optimale [6, 7].

Malgré l'implémentation des ces algorithmes pour introduire la réativité dans des systèmes utilisant du PDDL, une limite est atteinte dès lors que l'incertitude ne porte plus seulement sur la position des objets, mais sur la persistance de leur connaissance. Un cas critique peut se manifester lorsqu'un cube est déplacé vers une zone d'occlusion : le robot détecte une absence mais ne peut raisonner sur la continuité de l'existence de l'objet. Pour franchir cette étape, il devient nécessaire d'évoluer vers la planification épistémique [8, 9]. La "planification épistémique" permet au robot de diagnostiquer un doute et de planifier des actions d'acquisition de connaissances, ouvrant la voie à une manipulation véritablement consciente de son environnement [10, 11, 12].

Enfin, il convient de définir un *planificateur* comme étant le moteur computationnel au cœur des architectures robotiques dites «intelligentes». Son rôle est d'explorer l'espace des états possibles pour transformer un état initial brut en une séquence ordonnée d'actions permettant d'atteindre un but défini.

L'objectif de ce rapport est de détailler l'implémentation de cette architecture réactive, d'en analyser les performances face aux perturbations, et d'identifier les verrous technologiques justifiant le passage futur à un raisonnement épistémique.

1. monde clos : Tout fait qui n'est pas explicitement déclaré comme vrai dans l'état courant est considéré comme faux.

1 La Planification Classique : Le PDDL standard

La planification automatique repose sur la capacité d'un agent à déterminer de manière autonome une séquence d'actions menant d'un état initial à un état final désiré. Le langage PDDL constitue la pierre angulaire de ce domaine en offrant un formalisme standardisé basé sur la logique des prédicats du premier ordre [1].

Le PDDL repose sur une séparation stricte entre la modélisation des lois physiques du monde et la description d'une mission spécifique. Cette modularité s'articule autour de deux fichiers fondamentaux :

1.1 Le Fichier de Domaine

Le domaine définit le cadre d'existence du robot. Il regroupe les éléments immuables de l'environnement :

- **Types :** Hiérarchie des objets manipulables (ex : `cube`, `zone`, `gripper`).
- **Prédicats :** Propriétés logiques pouvant être vraies ou fausses à un instant t. Dans notre cas : `(at ?cube ?zone)`, `(clear ?cube)`, ou `(hand-empty)`.
- **Actions :** Opérateurs de transition d'état. Chaque action est définie par : *les Paramètres, les Pré-conditions et les Effets*

1.2 Le Fichier de Problème

Le fichier de problème instancie le domaine pour une tâche précise. Il définit :

- **Objects :** La liste réelle des entités présentes (ex : `cube_A`, `cube_B`, `table`).
- **Initial State ($\mathcal{I}$) :** La « photographie » complète du monde au début de l'opération, telle que perçue par la caméra.
- **Goal State ($\mathcal{G}$) :** L'objectif partiel ou total à atteindre (ex : tous les cubes triés par ordre (chiffre) suivant l'axe X).

Lorsqu'une action $a \in A$ est exécutée dans un état s, elle produit un nouvel état s' par l'application de ses effets. La planification consiste alors à trouver un chemin (une trajectoire) dans ce graphe d'états entre $\mathcal{I}$ et un état $s \in \mathcal{G}$.

2 Le Planificateur : Moteur de l'Intelligence Artificielle Symbolique

Le planificateur est l'entité logicielle chargée de résoudre le fossé entre l'état actuel du monde et l'objectif fixé. Contrairement à un algorithme classique où l'humain code la procédure de résolution, le planificateur reçoit uniquement les *règles du jeu* (le domaine), l'état initiale du monde et la *l'état cible* (le problème). Il doit alors « inventer » la solution.
Parmis les familles de planificateurs, on retrouve :

2.1 Planificateurs Classiques (Déterministes)

- **Mécanisme :** Ils transforment le domaine en un graphe d'états et utilisent des heuristiques de relaxation (telles que l'ignorance des effets négatifs) pour guider la recherche dans des espaces à forte combinatoire. [2]

— **Avantages :** Ils sont extrêmement performants pour des problèmes à large échelle, comme le tri simultané de six cubes.
— **Défauts :** Ils sont intrinsèquement incapables de gérer le temps réel ou l'incertitude sans l'adjonction d'une couche logicielle de replanification externe.

Comme outils de référence, on peut retrouver dans cette catégorie Fast Downward, LAMA.

2.2 Planificateurs Temporels (PDDL 2.1+)

— **Mécanisme :** Ils gèrent des actions ayant une durée déterminée (*durative actions*) et des contraintes temporelles strictes (*deadlines*). [1]
— **Avantages :** Ils permettent d'optimiser le temps de cycle, par exemple en déplaçant le *xArm6* vers une zone de saisie pendant que la caméra *ZED2i* termine le post-traitement de l'image.
— **Défauts :** La recherche est mathématiquement beaucoup plus complexe, ce qui peut générer des temps de calcul incompatibles avec une réactivité fluide.

On peut retrouver dans cette catégorie POPF, OPTIC comme des outils de référence.

2.3 Planificateurs Probabilistes (PPDDL et MDP)

L'incertitude physique peut être modélisée via l'extension *PPDDL* (*Probabilistic PDDL*). Ces planificateurs, tels que *PRP* ou *GURT*, ne cherchent plus une simple séquence d'actions, mais une **politique de décision** [12].

— **Mécanisme :** Ils s'appuient sur le cadre mathématique des *Processus de Décision Markoviens* (MDP) pour déterminer l'action optimale à chaque état, en tenant compte des probabilités d'échec (ex : un cube qui glisse de la pince).
— **Avantages :** Ils offrent une robustesse native face aux aléas physiques sans nécessiter de replanification totale.
— **Défauts :** Ils subissent une explosion combinatoire rapide, limitant leur usage en temps réel sur des configurations d'objets nombreuses.

3 Architectures Réactives et Robustesse d'Exécution

3.1 La Planification en Boucle Fermée (Replanning)

La méthode la plus directe pour gérer l'incertitude consiste à intégrer le planificateur dans une boucle itérative de surveillance et d'ajustement [7].

— **Principe :** L'agent ne se contente pas d'exécuter un plan statique ; il compare continuellement l'état perçu par ses capteurs avec l'état théorique attendu [7].
— **Fonctionnement :** Si une divergence significative est détectée entre l'observation et le modèle, le système interrompt l'exécution pour générer un nouveau plan depuis la situation réelle [2, 7].
— **Limites :** Cette approche engendre une latence décisionnelle, car chaque imprévu déclenche un calcul global, ce qui peut nuire à la fluidité du comportement [2].

3.2 La Planification Conditionnelle (Contingency Planning)

— **Principe :** Au lieu d'une trajectoire linéaire, le planificateur produit un arbre de décision [13].

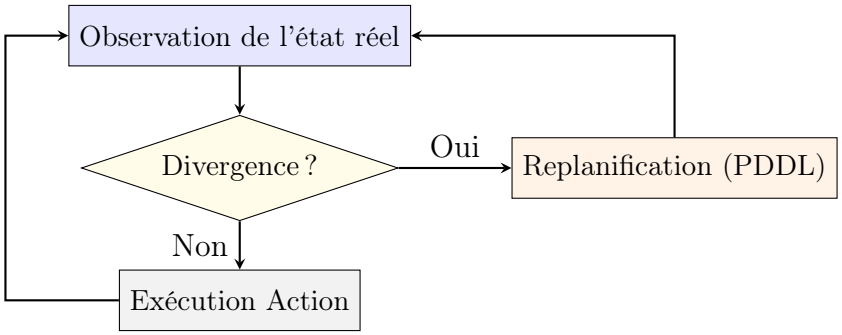

FIGURE 1 – Architecture de replanification en boucle fermée [7].

— **Fonctionnement :** Le plan inclut des points de décision basés sur le résultat d'observations sensorielles prédéfinies (ex : `IF action_success THEN ... ELSE ...`) [13].
— **Avantages :** Permet une réaction immédiate car les comportements de secours sont pré-calculés.
— **Limites :** Le système souffre d'une explosion combinatoire, rendant impossible la modélisation de toutes les éventualités dans un monde ouvert [13].

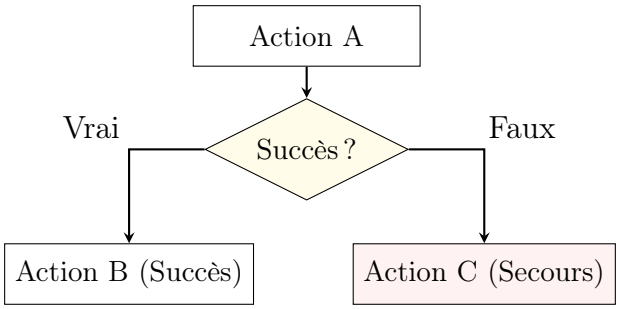

FIGURE 2 – Structure décisionnelle d'un plan conditionnel [13].

3.3 Les Behavior Trees (Arbres de Comportement)

— **Principe :** Ils organisent la logique de décision en nœuds de contrôle (Séquences, Sélecteurs) et nœuds d'exécution [3].
— **Fonctionnement :** Un signal périodique (*Tick*) traverse l'arbre. Chaque nœud renvoie un statut : *Success, Failure* ou *Running* [3, 4].
— **Réactivité :** Le nœud *Sélecteur* essaie ses enfants jusqu'à en trouver un qui réussisse, permettant des récupérations d'erreurs locales sans interrompre la logique globale [5, 6].

4 Analyse du modèle de référence

4.1 Spécifications matérielles

Le projet utilise un cobot Ufactory xArm6 Lite couplé à une caméra RGB-D (ZED2i) afin de percevoir la scène en temps réel et d'interagir avec des objets (cubes). La caméra fournit à la fois l'image couleur et la profondeur, ce qui permet d'estimer la pose des objets (repères ArUco) et de reconstruire leur position 3D pour guider les actions de saisie/dépôt.

FIGURE 3 – Robot manipulateur : xArm6 Lite

FIGURE 4 – Caméra ZED2i

4.2 Fonctionnement linéaire et séquentiel : Boucle Ouverte

Dans cette version, le processus suit une cascade d'étapes rigides et indépendantes :

— **Perception ponctuelle** : La lecture de l'état du monde via les marqueurs ArUco ne se fait qu'à la demande explicite de l'utilisateur (`Read World State`). Une fois cette lecture effectuée, le système considère que l'environnement est figé.

— **Planification statique** : Le plan d'action (PDDL) est généré sur la base d'un instantané (*snapshot*) des positions des cubes.

— **Exécution aveugle** : Une fois la commande `Execute` lancée, le robot parcourt la liste des instructions (`pick` et `place`) sans aucune vérification intermédiaire.

4.3 Limites et rigidité du système

L'analyse de cette architecture met en évidence plusieurs vulnérabilités majeures face aux conditions réelles d'utilisation :

— **Absence de rétroaction** : Le programme ne compare jamais les coordonnées théoriques du plan avec les coordonnées réelles lues par la caméra pendant l'exécution.

— **Sensibilité aux perturbations** : Si un opérateur déplace manuellement un cube alors que le robot est en mouvement, ce dernier continuera sa trajectoire vers l'ancienne position, entraînant inévitablement un échec de la tâche (saisie dans le vide).

— **Blocage de l'interface** : L'exécution étant gérée de manière synchrone, l'interface utilisateur reste en attente de la fin du cycle complet du robot avant de pouvoir reprendre la main sur d'éventuelles commandes de sécurité.

Dans cette version le système remplit sa mission de preuve de concept mais s'avère insuffisante pour une collaboration homme-robot sécurisée. Pour pallier ces défauts, il est nécessaire d'évoluer vers un système **réactif** capable de surveiller l'environnement en temps réel et de recalculer sa stratégie de manière autonome dès qu'une incohérence est détectée.

5 Architecture proposée : Passage à la boucle fermée

Pour répondre aux limitations du prototype initial, il est introduit un changement de paradigme : le passage d'une exécution séquentielle à un contrôle en **boucle fermée** (*closed-loop*). Dans ce modèle, le système ne se contente plus d'émettre des ordres, mais surveille en permanence l'adéquation entre ses prévisions et la réalité physique.

Le passage d'un automate rigide à un système réactif a nécessité la sélection d'outils et de méthodes spécifiques. Cette section justifie ces choix pour garantir la fluidité et la sécurité de l'interface.

5.1 L'architecture Multi-threadée (Séparation des flux)

Le choix d'une architecture multi-threadée (via la bibliothèque `threading`) est une réponse directe au blocage constaté dans la version initiale.

- **Le problème du blocage** : Dans un système à fil unique (*single-thread*), l'exécution d'un plan robotique sature le processeur, rendant l'interface utilisateur (`Pygame`) "gelée" et incapable de détecter un mouvement de souris ou une commande d'arrêt d'urgence.
- **La solution** : L'utilisation de `threading.Thread` permet au cycle de rafraîchissement de Pygame (60 FPS) de rester indépendant des délais de latence du robot.

Ce choix est indispensable pour maintenir une sécurité logicielle ; si le robot entre en collision, l'interface doit rester capable d'envoyer un signal d'arrêt d'urgence (`set_state(4)`) instantanément.

5.2 La Planification Réactive par PDDL : Le moteur de décision adaptatif

L'utilisation du langage PDDL constitue le cœur décisionnel du système depuis sa version initiale. Cependant, le passage à celui implémenté ici a transformé cet outil d'un simple générateur d'instructions statiques en un véritable moteur de décision adaptatif capable de gérer l'imprévu.

5.2.1 De la planification statique à la planification dynamique

Il est important de noter que le modèle de référence intégrait déjà les capacités de planification PDDL. Néanmoins, son utilisation y était limitée :

- **Approche initiale (Boucle ouverte)** : Le plan était généré une seule fois avant le début de l'exécution. Si l'environnement changeait après ce calcul, le plan devenait immédiatement caduc, car le robot ne disposait d'aucun mécanisme pour solliciter à nouveau le planificateur.
- **Approche proposée (Réactive)** : Le PDDL est désormais utilisé comme un service *à la demande*. Chaque détection de perturbation déclenche un nouveau cycle de planification complet.

5.2.2 Décomposition du processus de replanification

Lorsqu'une perturbation est détectée, le système réinitialise son cycle de réflexion selon quatre étapes clés :

1. **Extraction de l'état initial courant** : Le système interroge l'objet `Environment` pour obtenir les positions exactes (x, y, z) de tous les cubes au moment précis de l'incident. Ces données réelles remplacent les données obsolètes du plan précédent.
2. **Génération dynamique du fichier** `problem.pddl` : Le `PDDLProblemGenerator` crée un nouveau fichier décrivant la situation actuelle tout en conservant le but final (*goal state*) inchangé (ex : tri par ordre croissant).
3. **Résolution symbolique** : Le `PDDLPlanner` invoque le solveur pour trouver une nouvelle séquence d'actions optimales. Ce processus garantit que si un cube a été déplacé dans une zone qui bloque le chemin d'un autre, le planificateur le déplacera en priorité.
4. **Injection dans le flux d'exécution** : Le nouveau plan est transmis au thread d'exécution via un *replan_ callback*. L'interface graphique est mise à jour dynamiquement pour afficher les nouvelles étapes, préfixées par R (pour Replan).

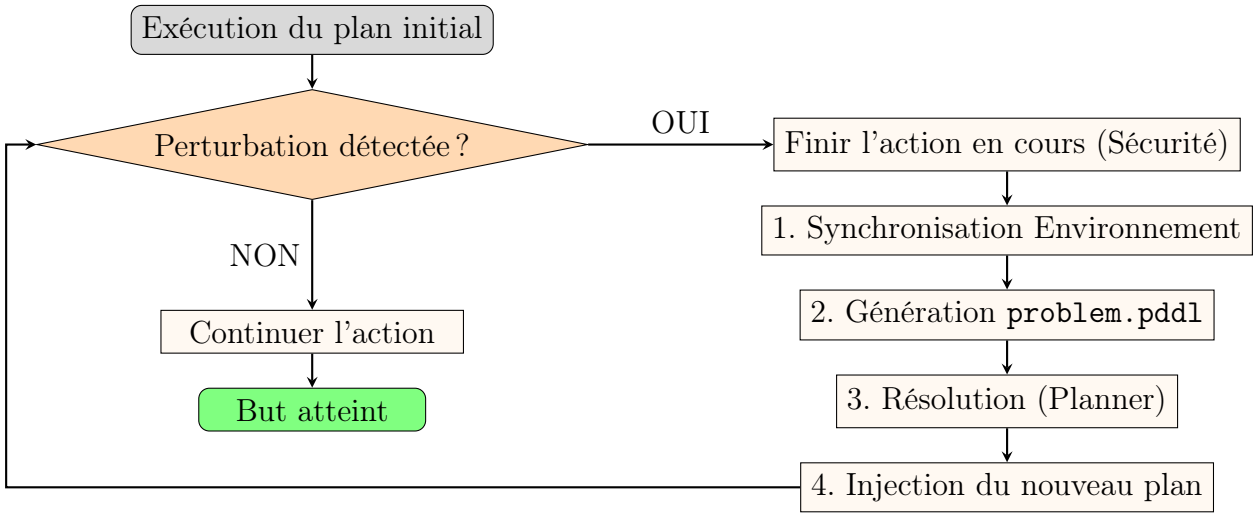

FIGURE 5 – Architecture du système de planification réactive.

5.2.3 Comparaison avec le PPDDL et gestion de l'incertitude

Une alternative à l'approche retenue aurait pu être l'utilisation du **PPDDL** (*Probabilistic PDDL*). Cependant, ce choix a été écarté pour des raisons liées à la nature des perturbations observées :

- **Incertitude exogène vs endogène** : Le PPDDL est conçu pour gérer des actions dont le résultat est incertain (ex : une pince qui glisse avec une probabilité de 10 %). Dans notre système, les actions du robot sont fiables (endogènes), mais l'environnement est soumis à des changements imprévus causés par l'utilisateur (exogènes).
- **Complexité de calcul** : La planification probabiliste est extrêmement gourmande en ressources. Pour un système devant réagir en temps réel à un déplacement de cube, la régénération rapide d'un problème PDDL déterministe s'avère beaucoup plus performante qu'une recherche d'optimum dans un arbre d'états probabilistes.

Il a donc été privilégié une **réactivité par replanification déterministe**. Le système traite chaque changement comme un nouvel état certain, ce qui simplifie la logique tout en garantissant une réponse quasi instantanée du planificateur.

5.2.4 Justification de l'absence de Behavior Trees (BT) comme couche réactive

Une architecture alternative aurait consisté à implémenter un *Behavior Tree* juste après le planificateur. Dans ce scénario, le BT servirait de superviseur local capable de détecter un changement environnemental et de forcer une demande de replanification. Bien que pertinente, cette approche n'a pas été retenue pour les raisons suivantes :

- **Redondance fonctionnelle** : Ce rôle de surveillance est déjà assuré de manière plus légère par le thread de *callbacks*. Ajouter un BT introduirait une couche logicielle supplémentaire complexe sans gain réel de réactivité par rapport au système d'événements (`replan_event`) développé.
- **Complexité de la communication descendante** : Un BT nécessite de modéliser explicitement les conditions de retour au planificateur pour chaque branche d'exécution. Dans notre système, la réactivité est centralisée : dès qu'une perturbation est détectée, le système de *callback* informe le thread d'exécution qui sollicite directement le moteur

PDDL. Cette approche directe est plus efficace pour des tâches de tri où le contexte global prévaut sur le réflexe local.

— **Gestion centralisée de la logique** : L'utilisation d'un BT pour relayer l'information au planificateur créerait un double raisonnement. Il a été préféré que la détection soit purement sensorielle (UI/Vision) et que le raisonnement soit purement logique (PDDL), sans intermédiaire décisionnel. Cela garantit que le robot ne prend pas de décisions micro-réactives qui pourraient entrer en conflit avec l'optimalité globale du plan de tri.

5.3 Mécanisme de Communication par Événements et Callbacks

La liaison entre le thread de calcul et l'interface graphique repose sur un système de *callbacks* (fonctions de rappel) et de *Flags*.

— **Le choix du `replan_event`** : Utiliser un `threading.Event()` est plus efficace qu'une variable booléenne simple car il est thread-safe et permet une mise en attente sans consommation CPU inutile.

— **L'usage des Callbacks d'exécution** : La fonction `progress_callback` permet au thread d'exécution de notifier l'UI de chaque étape franchie (`pick` réussi, `place` effectué).

Ce couplage lâche (*loose coupling*) permet de mettre à jour le plan visuel et les animations de manière asynchrone, évitant ainsi tout scintillement ou décalage entre la position réelle du robot et sa représentation à l'écran.

5.4 Gestion Géométrique de la Portée (Reachability)

Un aspect crucial de la planification réactive est le filtrage des actions impossibles. Si une perturbation déplace un cube hors du rayon d'action du robot (> 42 cm), le système identifie que les pré-conditions de l'action `pick` ne peuvent plus être remplies. Plutôt que de bloquer le programme, le système marque l'action comme SKIP dans l'interface et poursuit le tri des autres objets, garantissant une continuité de service.

— **Pourquoi le calculer dans l'UI ?** : Si le calcul n'était fait que par le robot, l'interface pourrait valider des actions impossibles, créant une confusion pour l'utilisateur.

— **Implémentation** : L'usage des méthodes `_clamp_to_table` et `_clamp_storage_to_reach` force les coordonnées dans des limites sécurisées avant même que le planificateur ne soit sollicité.

Cette façon de faire évite les erreurs de cinématique inverse du xArm6 qui pourraient survenir si le robot tentait d'atteindre un cube trop éloigné suite à une perturbation mal gérée.
Le passage d'une boucle ouverte à une planification réactive ne peut être pleinement exploité sans un retour d'information visuel précis. La section suivante détaille comment l'ergonomie et les choix d'UX/UI permettent de traduire les calculs complexes du planificateur en indicateurs clairs pour l'utilisateur final.

6 Ergonomie et retour d'information (UX/UI)

6.1 Transparence du système et retour d'état en temps réel

Pour que l'opérateur puisse superviser efficacement le robot, l'interface doit refléter instantanément les changements internes du système :

— **Système de Toasts et Badges** : L'utilisation de notifications éphémères (*Toasts*) informe l'utilisateur de chaque étape critique (ex : Detection validated ou Replanning

requested). Des badges colorés indiquent l'état actuel de la machine (`IDLE`, `EXECUTING`, `WORLD_LOADED`), évitant toute confusion sur la phase opérationnelle.

— **Suivi visuel du plan** : Le panneau de droite affiche la liste des actions générées par le PDDL. Contrairement à la version initiale, cette liste est dynamique : elle met en évidence l'action en cours et insère visuellement les segments de replanification, rendant le processus de recalcul totalement transparent.

6.2 Visualisationl et animations

L'interface graphique ne se contente plus d'être un tableau de bord statique, elle devient une fenêtre de prédiction :

— **Interpolation de mouvement** : Pendant l'exécution, le déplacement des cubes à l'écran est lissé par une interpolation linéaire. Cela permet de faire correspondre la simulation visuelle avec la vitesse réelle du bras xArm6, facilitant la surveillance par l'opérateur.

— **Représentation de la portée (Reachability)** : Un cercle cyan de 42 cm entoure le robot, matérialisant sa zone de travail sécurisée. Une zone d'ombre rouge indique la zone morte centrale (*dead zone*), prévenant l'utilisateur contre les placements de cubes impossibles à saisir.

6.3 Interaction directe et simulation de perturbations

L'ergonomie a été repensée pour permettre une interaction bidirectionnelle :

— **Manipulation par Drag & Drop** : L'utilisateur peut déplacer les cubes ou la zone de stockage à la souris. Cette fonctionnalité n'est pas seulement esthétique : elle permet de tester la robustesse du système réactif en simulant des perturbations environnementales de manière contrôlée.

— **Validation contextuelle des cibles** : Le clic droit permet de définir une cible pour l'action `Move Cube`. L'interface vérifie immédiatement si la cible est valide et à portée avant de permettre la génération du problème PDDL, agissant comme un filtre d'erreurs en amont de l'exécution.

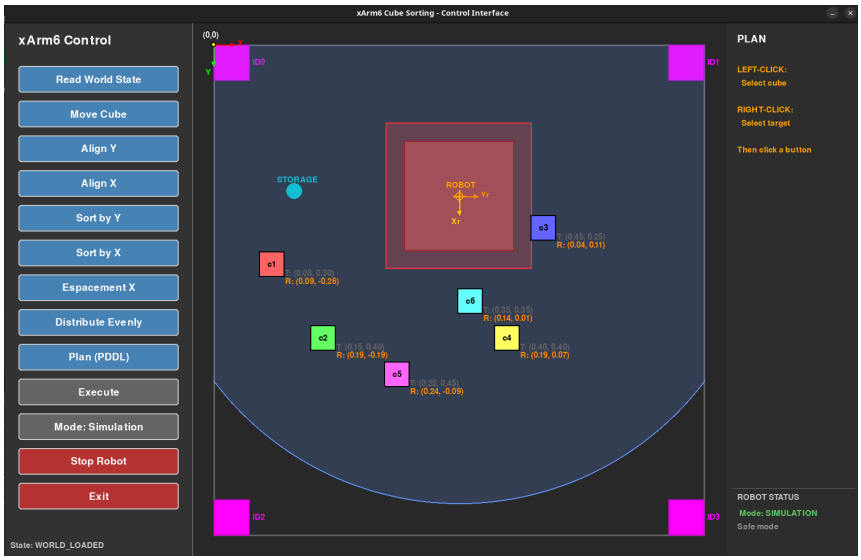

FIGURE 6 – UI avec World State chargé de l'interface de contrôle initiale.

FIGURE 7 – UI avec World State chargé de l'interface de contrôle implémentée

7 Bilan technique : Réactivité vs Rigidité

7.1 Comparaison des performances globales

— **Robustesse opérationnelle** : Replanification rapide → missions maintenues malgré les changements.
— **Fluidité de l'interaction** : Multi-threading → contrôle opérateur et arrêt d'urgence toujours disponibles.

Caractéristique	Version Initiale (Statique)	Version Finale (Réactive)
Boucle de contrôle	Ouverte (aveugle)	Fermée (supervisée)
Planification PDDL	Unique au démarrage	Dynamique et à la volée
Réaction aux erreurs	Échec critique / Collision	Re-calcul du plan (GPS)
Interface	Bloquée durant l'action	Fluide et interactive
Gestion de portée	Aucune vérification	Filtrage géométrique préventif

TABLE 1 – Synthèse comparative des deux architectures de contrôle.

7.2 Limites de la méthode : Vers une gestion de la connaissance

Bien que robuste, l'approche par replanification réactive se heurte à des limites structurelles liées au traitement de l'information :

— **Hypothèse du monde clos** : Le système considère que tout ce qui n'est pas vu par la caméra n'existe pas. Il est incapable de gérer l'absence d'information ou de planifier des actions pour *vérifier* une donnée incertaine.
— **Absence de persistance cognitive** : Le robot n'a pas de mémoire des états passés. Si un cube est temporairement occulté, il disparaît de sa représentation du monde, empêchant toute anticipation basée sur une déduction logique.
— **Invisibilité des intentions** : Le système réagit aux déplacements physiques sans comprendre la causalité humaine. Il traite une perturbation comme une erreur de coordonnées plutôt que comme une interaction intentionnelle.

Cette incapacité à traiter l'information en tant qu'état dynamique et incertain marque la limite supérieure du système actuel. Pour franchir cette étape, il devient nécessaire d'évoluer vers une architecture capable de modéliser non seulement la position des objets, mais aussi le degré de connaissance que le robot possède sur son environnement.

8 Approche par Planification Épistémique

8.1 Principes Fondamentaux de la Planification Épistémique

La planification épistémique étend la planification classique en introduisant la notion d'état de croyance (*Belief State*). L'objectif du planificateur n'est plus seulement d'atteindre une configuration physique du monde, mais d'atteindre un état de connaissance spécifique (par exemple : *"Je veux savoir où est le cube 2"*). [12]

Le fonctionnement interne du planificateur, intégrant les boucles de perception et de contingence, peut être résumé par le processus suivant :

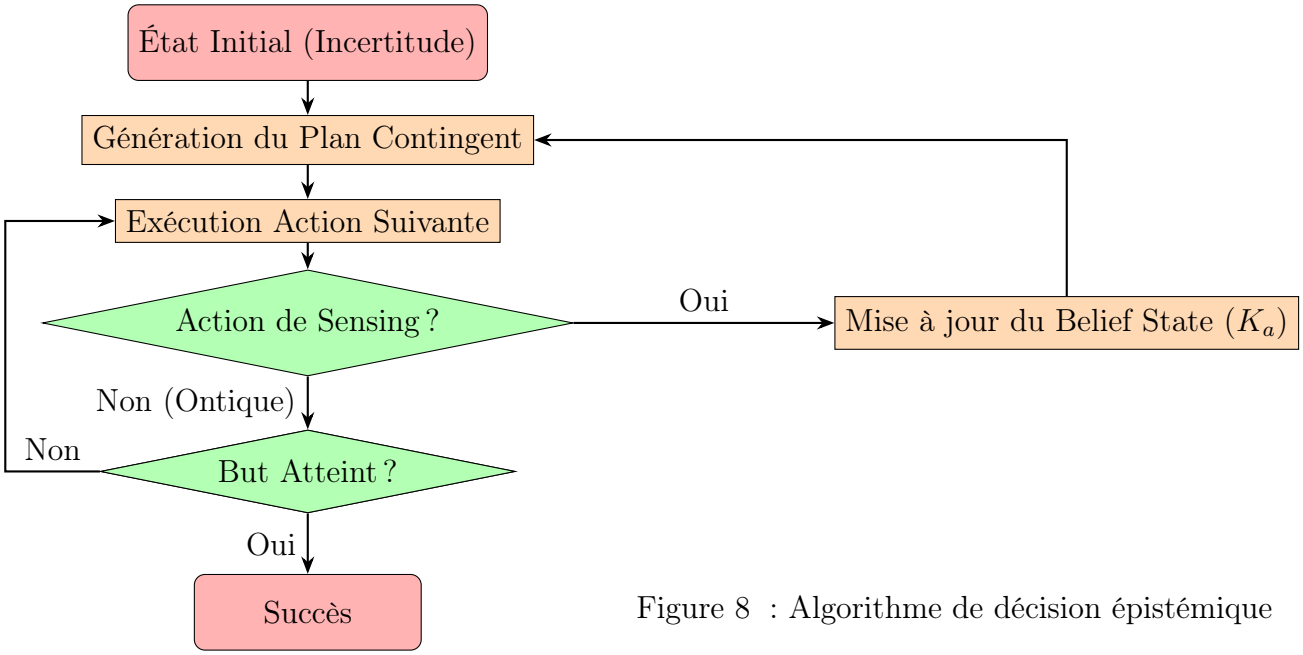

Figure 8 : Algorithme de décision épistémique

8.2 Vers un besion d'une implémentation dédiée

Bien que la littérature académique propose plusieurs planificateurs épistémiques théoriques (tels que RP-MEP ou PKS), l'implémentation d'une solution sur mesure s'avère nécessaire pour notre cas d'application. Cette décision repose sur deux contraintes majeures.

— **Optimisation pour le temps réel** : En se focalisant sur une logique épistémique binaire (connu/inconnu) plutôt que sur des croyances imbriquées complexes, notre système réduit le temps de calcul à quelques millisecondes pour garantir une réactivité fluide.
— **Perception active intégrée** : L'intégration directe des contraintes physiques et géométriques de la caméra ZED2i dans les préconditions d'actions permet au robot de planifier des mouvements spécifiques pour lever des incertitudes ou dégager son champ de vision.

Conclusion

Le travail présenté dans ce rapport a permis de transformer une interface de commande robotique rudimentaire en un système de contrôle adaptatif et résilient pour le bras *xArm6*. Le passage d'une architecture en boucle ouverte, caractérisée par une exécution aveugle et rigide, à un système en boucle fermée représente une avancée majeure dans la gestion des interactions au sein d'un environnement dynamique.

Grâce à l'implémentation d'une surveillance asynchrone et d'un mécanisme de planification réactive basé sur le langage `PDDL`, le robot est désormais capable de détecter des perturbations physiques et de recalculer son plan d'action de manière optimale sans interruption humaine. Cette approche hybride combine la rigueur logique des planificateurs déterministes, tels que *Fast Downward*, avec la flexibilité nécessaire à la manipulation collaborative. L'interface utilisateur développée facilite cette supervision en offrant un retour d'état transparent via des outils de visualisation augmentée et de gestion dynamique de la portée.

Malgré ces succès, l'étude a mis en évidence des limites intrinsèques liées à la dépendance du système envers une perception parfaite du monde clos. Pour franchir le stade de la simple réaction physique, les travaux futurs devront s'orienter vers l'intégration de planificateurs épistémiques. Ce paradigme permettrait au système de raisonner explicitement sur ses propres croyances et incertitudes, ouvrant la voie à une perception active où le robot peut planifier des actions pour acquérir de l'information.

En conclusion, l'architecture développée constitue une base solide pour la robotique de service, prouvant que l'association de la planification symbolique et d'une supervision en temps réel est une solution viable pour relever les défis de l'autonomie adaptative.

Références

[1] M. Fox and D. Long, "Pddl2.1 : An extension to pddl for expressing temporal planning domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003.

[2] M. Helmert, "The fast downward planning system," in *Journal of Artificial Intelligence Research*, vol. 26, 2006, pp. 191–246.

[3] M. Colledani and P. Ögren, *Behavior trees in robotics and AI : An introduction.* CRC Press, 2018.

[4] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of behavior trees in robotics and ai," *IEEE Transactions on Robotics*, vol. 38, no. 3, pp. 2025–2044, 2022, l'état de l'art le plus récent sur les BT en robotique.

[5] C. Paxton, J. Felix, D. Vibhavari, B. Jonathan, and D. H. Gregory, "Costar : Instructing collaborative robots with behavior trees and vision," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 5640–5647.

[6] J. Tanevski *et al.*, "Pddl and behavior trees : A hybrid approach to task planning and execution," *Frontiers in Robotics and AI*, 2020, idéal pour justifier le flowchart que nous avons créé.

[7] S. Macenski, F. M. Cano, J. R.-L. Francisco, and V. Matellan, "Plansys2 : A planning system for ros2," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.

[8] T. Le, F. Fabiano, T. C. Son, and E. Pontelli, "Epistemic planning : Theory and application," *Theory and Practice of Logic Programming*, vol. 22, no. 5, pp. 721–736, 2022, excellent résumé récent des techniques DEL.

[9] C. Muise, V. Belle, P. Felli, S. McIlraith, T. Miller, A. Pearce, and T. C. Son, "Planning for embodied agents with episodic memory and epistemic reasoning," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2015.

[10] F. Fabiano, A. Burigana, A. Dovier, and E. Pontelli, "Eplano : An epistemic planner based on functional dependencies," in *Proceedings of the 35th Italian Conference on Computational Logic*, 2020.

[11] A. Smith and F. Fabiano, "Scaling epistemic planning via symbolic model compression," *Artificial Intelligence Journal*, 2025, travail pionnier sur l'utilisation des ADD pour la planification épistémique.

[12] L. Dupont and T. C. Son, "Probabilistic epistemic planning for robotic manipulation," in *Proceedings of the 37th International Conference on Automated Planning and Scheduling (ICAPS)*, 2026.

[13] C. Muise, S. McIlraith, and J. C. Beck, "Pddl3.1 : Explicit contingency planning," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2014, référence clé pour l'extension du PDDL vers les choix conditionnels.

Annexes : Extraits de code de l'implémentation réactive

A. Structure du superviseur asynchrone (Threading)

Le code suivant illustre l'isolation du thread d'exécution par rapport au thread d'interface utilisateur (UI). Cette séparation permet à l'UI de rester à l'écoute de l'environnement pendant que le robot opère.

```
# Initialisation du thread d'exécution dans la classe RobotControlApp
def start_execution(self):
    self.stop_exec_event.clear()
    self.replan_event.clear()

    # Lancement du thread pour ne pas bloquer l'UI Pygame
    self.exec_thread = threading.Thread(
        target=self._exec_fn,
        daemon=True
    )
    self.exec_thread.start()
```

B. Détection et Signalisation d'une Perturbation

Cet extrait montre comment le Drag & Drop utilisateur déclenche un événement de replanification. L'interaction utilisateur est traitée comme une perturbation sensorielle.

```
# Dans la gestion des événements Pygame (MOUSEBUTTONUP)
if self.dragging:
    self.perturbation_detected = True
    self.replan_reason = "cube_moved"

    # Signalement au thread d'exécution qu'un nouveau plan est requis
    if self.state == UIState.EXECUTING:
        self.replan_event.set()
        self.toast.push("Cube moved -> replanning...", "warn")
```

C. La Boucle de Replanification Réactive (GPS du robot)

C'est ici que réside le cœur de la réactivité. Avant chaque action, le système vérifie si un nouveau plan a été sollicité. Si c'est le cas, il appelle le moteur PDDL pour recalculer la stratégie.

```
# Au sein de la boucle execute_plan
while i < len(plan):
    # Vérification du drapeau de replanification (Feedback Loop)
    if replan_event is not None and replan_event.is_set():
        replan_event.clear()

        # Appel au callback pour générer un nouveau problème PDDL
        new_plan = replan_callback()
```

```
        if new_plan:
            plan = new_plan # Mise à jour dynamique du plan
            i = 0           # Reset du curseur d'exécution
            continue        # Reprise immédiate avec le nouveau plan
```

D. Gestion de la Portée Géométrique (Safety Layer)

Avant d'envoyer les coordonnées au robot, l'interface effectue un filtrage pour garantir que les cubes sont dans la zone atteignable (42 cm).

```
def _clamp_storage_to_reach(self, x, y, reach_radius_m=0.42):
    rx, ry = self.visualizer.robot_position
    dx = x - rx
    dy = y - ry
    distance = (dx**2 + dy**2)**0.5

    if distance <= reach_radius_m:
        return (x, y)

    # Projection sur la limite du cercle atteignable
    scale = reach_radius_m / distance
    return (rx + dx * scale, ry + dy * scale)
```

E. Synchronisation de l'état du monde vers le PDDL

Cette fonction permet de rafraîchir la base de données interne avant chaque replanification. Elle garantit que le fichier `problem.pddl` généré reflète la réalité exacte du terrain au moment de la perturbation.

```
def _sync_environment_from_world_state(self):
    """
    Met à jour l'objet Environment avec les positions actuelles des cubes
    et de la zone de stockage avant de solliciter le planificateur.
    """
    # Récupération des données depuis le monde simulé ou la vision
    world_state = self.visualizer.get_world_state()

    for cube_id, pos in world_state['cubes'].items():
        # Mise à jour des coordonnées cartésiennes dans l'objet métier
        self.environment.update_cube_position(cube_id, pos['x'], pos['y'])

    # Synchronisation de la cible (Storage)
    storage_pos = world_state['storage_area']
    self.environment.set_storage_target(storage_pos['x'], storage_pos['y'])

    self.toast.push("Environment synced for PDDL", "info")
```