INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Une bibliothèque pour des moteurs de pilotage de programmes

Régis Vincent, Sabine Moisan, Monique Thonnat

N° 3011

Octobre 1996

THÈME 3

Rapport de recherche

ISSN 0249-6399

Une bibliothèque pour des moteurs de pilotage de programmes

Régis Vincent, Sabine Moisan, Monique Thonnat

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Orion

Rapport de recherche n° 3011 — Octobre 1996 — 27 pages

Résumé : Notre objectif est de faciliter la construction de systèmes à base de connaissances (SBC). Pour cela, nous construisons des moteurs *dédiés* à une classe de problèmes. Les travaux exposés dans ce rapport proposent de baser la construction de ces moteurs sur des composants *réutilisables*. Le but est d'améliorer la rapidité de construction et de modification des moteurs ainsi que la sécurité de programmation. Dans un premier temps, nous avons décidé de nous restreindre à une classe de problèmes particulière : les problèmes de *pilotage de programmes*, dont une sous-partie importante est la tâche de planification. A partir de différents modèles de moteurs de pilotage de programmes, nous avons défini une première *bibliothèque de structures de données et d'instructions de base* pour le pilotage de programmes et nous l'avons utilisée pour l'écriture de nouveaux moteurs.

Mots-clé : bibliothèque de composants, pilotage de programmes, planification, système à base de connaissances.

(Abstract: pto)

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

A library for program supervision engines

Abstract: Our aim is to provide easy means to construct knowledge-based systems (*KBS*). For this purpose, we design *engines* dedicated to a class of problems. The work presented in this report proposes to base the construction of such engines on *reusable components*. Thereby we aim at speeding both the engine construction and modification as well as securing the coding. To begin with, we restricted ourselves to a particular class of problems: those of *program supervision*, an important sub-part of which is planning. Based on different program supervision models, we have defined a first *library of components and elementary instructions* for program supervision and we have used it to code new engines.

Key-words: library of components, program supervision, planning, knowledge-based systems

1 Introduction

Un système à base de connaissances (SBC) est composé d'un *moteur* générique de raisonnement et d'une base de connaissances. Dans le but de faciliter la construction de SBC, nous construisons des moteurs qui ne sont pas liés à une expertise particulière, mais qui sont cependant *dédiés* à une classe de problèmes. Les classes de problèmes envisagées sont le pilotage automatique de programmes et l'interprétation d'images. L'approche que nous proposons dans ce rapport est motivée par nos expériences sur les différents moteurs que nous avons développés, dans le cadre de projets de recherche ou industriels. Nous avons déjà construit plusieurs outils dans des domaines différents, par exemple OCAPI [22, 16, 15] dédié au pilotage de programmes en traitement d'images, PLANETE [19] dédié au pilotage de programmes temps-réel et CLASSIC [23, 4] dédié à la la classification.

Malgré leurs différences, les moteurs de ces outils reposent sur les mêmes principes :

– ils encapsulent un mécanisme lié à une stratégie de raisonnement particulière. Par exemple pour OCAPI et PLANETE, la stratégie consiste principalement à imbriquer une phase de planification et une phase d'exécution;

– ils s'appuient sur un certain nombre de structures de données permettant de représenter la connaissance spécifique à la classe de problèmes visée.

Pour construire de tels moteurs il est nécessaire de se baser sur une modélisation des connaissances et des méthodes de résolution de problème qui interviennent dans les classes de problèmes étudiées. Le développement d'outils dédiés fournit des systèmes bien adaptés à l'expression des connaissances et largement utilisables dans différents domaines d'applications ([2], [20]). Cependant, à chaque moteur construit correspond une stratégie souvent figée et difficile à modifier si on souhaite élargir la gamme d'applications visées. Toute évolution des spécifications, des besoins ou des contraintes nécessite de re-configurer, voire de re-écrire, les outils construits.

Nous proposons de pallier cet inconvénient en construisant ces moteurs eux-mêmes à partir de composants de base, regroupés au sein d'une bibliothèque. L'utilisation d'une telle approche permet de décrire une stratégie de raisonnement à un niveau abstrait, plus simplement et plus rapidement qu'avec un langage de programmation, et de la faire évoluer plus facilement. En effet, le travail de réécriture de moteurs en différents langages ou l'écriture de variantes d'un même moteur, nécessaires pour des tests ou des domaines d'applications différents, sont très fastidieux et coûteux en temps.

Les avantages d'une telle approche sont :

- de pouvoir tester plusieurs versions du même moteur, sur une même base de connaissance, sans réécrire la base de connaissances ;
- de permettre un prototypage rapide et efficace ;
- d'augmenter la sûreté de programmation, en évitant une programmation trop bas niveau ;
- d'avoir un nombre minimal de composants permettant d'implanter une gamme de moteurs.

Nous nous sommes dans un premier temps focalisés sur le pilotage de programmes, car c'est pour cette classe de problèmes que nous avons le plus d'expérience. En effet nous avons développé des moteurs de pilotage en différents langages (C, C++, Lisp), sous différents systèmes d'exploitation (UNIX, VxWorks) et avec différentes variantes, notamment en ce qui concerne les stratégies de planification (voir [21, 8, 28, 18]).

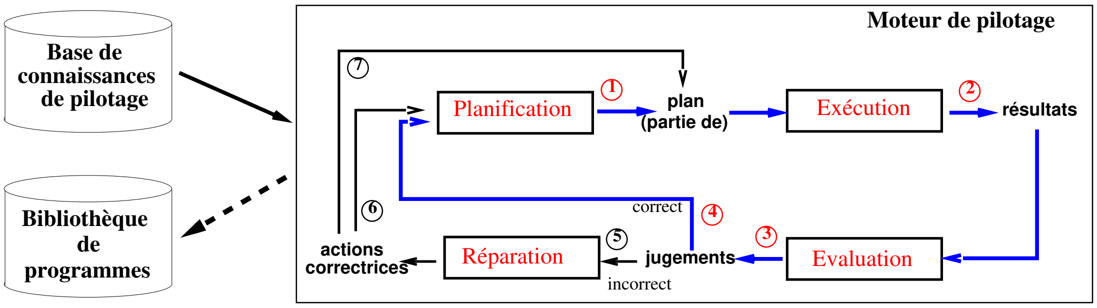

FIG. 1 – *La première phase de raisonnement est la planification, qui construit un plan ou une partie de plan (1) , pour résoudre un but donné, sur des données particulières. La deuxième phase exécute ce plan (ou partie de plan) et produit des résultats (2). Ces résultats sont ensuite jugés dans la phase d'évaluation (3). Si les résultats sont jugés corrects (4), le pilotage continue. Si des erreurs sont détectées (5), la phase de réparation décide des actions de correction à entreprendre : soit une re-exécution du même plan (7), soit une replanification (6).*

Un système à base de connaissances en pilotage de programmes se compose d'un ensemble de programmes existants, d'une base de connaissances sur l'utilisation de ces programmes et d'un moteur de pilotage. Le moteur de pilotage de programmes est composé de quatre modules : un planificateur, un exécuteur, un évaluateur et un réparateur. La figure 1 montre le rôle et les interactions entre les différents modules.

Pour construire des moteurs, il faut disposer d'une part de structures représentant les concepts intervenant dans la classe de problèmes visée et d'autre part d'instructions correspondant aux stratégies de raisonnement dans cette classe de problèmes. La bibliothèque de composants pour les moteurs de pilotage contient donc des structures de données (sous forme de classes) et des instructions de base manipulant ces structures, pour écrire des moteurs de pilotage de programmes. Nous avons porté un effort particulier sur la phase de planification [27, 12], car c'est une sous-tâche importante en pilotage de programmes (et en intelligence artificielle en général).

Cet article se décompose comme suit. Après une présentation l'approche que nous avons choisie (section 2) nous détaillons le contenu de la bibliothèque (section 3). Puis, la section 4 développe des exemples de moteurs de pilotage constructibles avec cette bibliothèque et la section 5 présente quelques approches similaires par leurs objectifs ou par la mise en œuvre de ces objectifs. Enfin la dernière section montre des résultats.

2 Approche proposée

Les moteurs que nous construisons incluent une modélisation fine de la classe de problèmes auxquels ils sont dédiés, c'est à dire une organisation des connaissances et un contrôle du raisonnement adaptés. Pour une même classe de problèmes, on peut vouloir envisager des outils plus ou moins sophistiqués (modèles de complexités différentes) ou des outils identiques mais dans des langages ou sur des machines différentes (même modèle mais implantations différentes).

La démarche que nous avons choisie consiste à abstraire à partir des différents modèles, des composants communs partageables. Ces composants correspondent aux structures de données et aux instructions de la bibliothèque. Ils doivent permettre d'une part, de modéliser une stratégie pour un nouveau moteur [25, 26] à un niveau abstrait, facile à comprendre et à modifier, et d'autre part d'implanter de nouveaux moteurs sans réécriture de code massive. Ceci permet de mieux séparer modélisations et implantations, tout en facilitant le passage des unes aux autres. La principale difficulté réside dans la détermination de la granularité de ces composants. Un trop grand niveau d'abstraction cache mieux les détails d'implantation, mais restreint la richesse et la souplesse d'expression de stratégies variées. A l'opposé, des instructions trop proches d'un langage de programmation ne permettent pas au concepteur de décrire une stratégie à un niveau d'abstraction suffisant et laissent à sa charge trop de détails d'implantation.

Nous avons actuellement développé :

- une bibliothèque de classes représentant les structures de données élémentaires, utilisées par des instructions (instructions de raisonnement), qui peuvent facilement être combinées pour modéliser différents mécanismes de pilotage ;
- des implantations efficaces et portables pour cette bibliothèque. Pour cela, nous avons choisi d'implanter la bibliothèque en C++ pour sa large utilisation et sa portabilité, ainsi qu'en Lisp, pour ses facilités de maquettage.

Cette approche permet une vision unifiée de différents moteurs et fournit une base commode de comparaison entre stratégies de pilotages. De plus, la construction d'un nouveau moteur, à partir de cette bibliothèque est plus rapide, plus facile et plus fiable que son écriture à partir de rien.

Dans le contexte dans lequel nous nous plaçons (développement de moteurs génériques applicables à des domaines d'applications variés), il existe trois rôles séparés, le plus souvent remplis par des personnes ou des équipes différentes. Le concepteur construit un moteur générique (par exemple notre équipe a conçu le moteur de pilotage de programmes OCAPI), l'expert décrit une base de connaissances (par exemple un spécialiste en vision décrit des programmes appliqués au traitement de séquences d'images) et enfin l'utilisateur final cherche à traiter des problèmes particuliers (par exemple un médecin en imagerie médicale ou un géologue pour la classification en imagerie satellitaire).

La bibliothèque proposée s'adresse au concepteur afin de faciliter son travail de construction ou de mise au point de moteurs. Lorsqu'un concepteur conçoit un nouveau moteur, il définit la stratégie de ce moteur, qu'il sait exprimer sous forme d'une spécification telle que celle présentée figure 2. Cette spécification est celle d'un planificateur totalement ordonné tel que Minton la décrit dans [14]. Dans ce cas la stratégie se décompose en 6 étapes : une vérification de fin, la sélection du meilleur but pour résoudre une précondition d'un opérateur O_{need} , la sélection du meilleur opérateur pour résoudre le but, l'ordonnancement des opérateurs entre eux, puis la mise à jour du but et enfin l'appel récursif.

Le rôle de la bibliothèque de composants est de faciliter le passage d'une spécification abstraite vers une implantation concrète.

TO(P,G)

1. **Termination check:** If G is empty, report success and return solution plan P.
2. **Goal selection:** Let c = select-goal(G), and let O_{need} be the plan step for wich c is a precondition.
3. **Operator selection:** Let O_{add} be an operator in the library that adds c. If there is no such O_{add}, then terminate and report failure. *Choice point: all such operators must be considered for completeness.*
4. **Ordering selection:** Let O_{del} be the last deleter of c. Insert O_{add} somewhere between O_{del} and O_{need}, call the resulting plan P'. *Choice point: all such positions must be considered for completeness.*
5. **Goal updating:** Let G' be the set of preconditions in P' that are not true.
6. **Recursive invocation:** TO(P',G').

FIG. 2 – *TO(P,G) est la spécification d'une stratégie de planification totalement ordonnée : "Total Order Planning", avec P, le plan solution construit et G l'ensemble des préconditions non vérifiées.*

3 Une bibliothèque pour le pilotage de programmes

Nous avons développé une première bibliothèque pour fournir des instructions et des structures de base à un concepteur de moteurs de pilotage. Nous avons particulièrement insisté sur la phase de planification, qui peut présenter beaucoup de variantes ([21]).

Les instructions de la bibliothèque travaillent sur des structures de données communes qui représentent les entités qui interviennent en pilotage de programmes, telles que : actions, plans, buts à atteindre. Les structures de données sont représentées par des classes de la bibliothèque et les instructions sont des fonctions ou des méthodes de ces classes. Le concepteur dispose des instructions de base qui utilisent ou manipulent les structures de données. Grâce à un langage simple[1] manipulant ces instructions il peut ainsi écrire différents moteurs.

1. Actuellement, il s'agit d'une sous-partie du langage de développement des fonctions de la bibliothèque, nous envisageons un langage spécifique pour l'avenir (voir [9]).

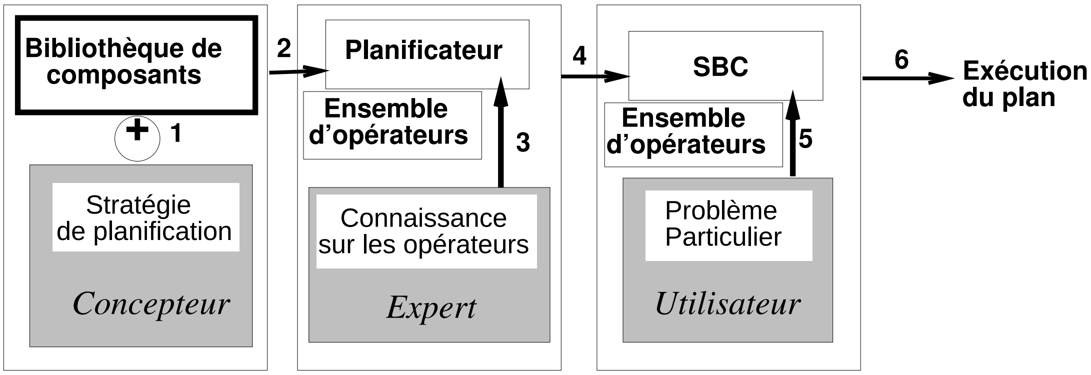

FIG. 3 – *Génération de moteurs de pilotage de programmes*

La figure 3 présente l'ensemble des étapes nécessaires à la création d'un système à base de connaissances, en utilisant notre bibliothèque. Un concepteur qui veut implanter une stratégie de pilotage particulière (1) peut choisir les structures et les instructions dont il a besoin dans la bibliothèque. Il décrit, grâce à ces instructions, la stratégie de pilotage souhaitée (par exemple basée sur une planification à base de squelettes ou sur une planification non-ordonnée à la STRIPS [3]). Le résultat (2) est un moteur de pilotage, c'est à dire un système, vide de connaissances, permettant de construire des SBC de pilotage.

Lorsque ce moteur général est construit, un expert d'un domaine particulier (par exemple en traitement d'images, en calcul scientifique ou en robotique) peut décrire dans une base de connaissances un ensemble d'opérateurs (par exemple des programmes de traitement d'images, des programmes de calcul scientifique ou des opérateurs de robotique), ainsi que sa connaissance sur l'utilisation de ces opérateurs (3). La connaissance consiste en la description des caractéristiques des opérateurs, de leurs arguments d'entrée et de sortie, des alternatives possibles, etc (cf section 3.1). Ceci constitue une base de connaissances pour le moteur général et le résultat est un système à base de connaissances (4).

Une fois ce SBC terminé, un utilisateur peut s'en servir pour résoudre un problème qui nécessite de planifier et d'exécuter des opérateurs de l'ensemble donné par l'expert (5). Par exemple, un biologiste peut demander un comptage automatique de certaines cellules dans des images, un astrophysicien peut souhaiter calculer la masse de galaxies à partir de sources signal et images, ou un système autonome embarqué sur une voiture peut demander la détection des obstacles avoisinants. Le résultat final (6) est l'exécution d'un plan d'opérateurs (de calcul, de traitement d'image, de robotique), qui résout le problème de l'utilisateur final.

Pour nos besoins d'expression de différents moteurs de pilotage, suivant différentes stratégies courantes, nous avons identifié certains composants qui interviennent dans l'activité de pilotage. Ces composants ont été spécifiés à la fois du point de vue du génie logiciel (réutilisation de logiciels) et du point de vue de l'intelligence artificielle et du traitement des données, dans le but d'aboutir à des représentations largement (re)utilisables.

3.1 Structures de données de la bibliothèque

Nous avons identifié les principaux concepts importants en pilotage de programmes :

- les **données** et les objets du domaine, qui contiennent l'information nécessaire à la description du problème de l'utilisateur final; ils sont stockés dans une base de faits ;
- les **opérateurs** terminaux qui correspondent aux programmes exécutables et manipulent les données ;
- les **schémas de plans** qui sont des squelettes de plans, fournis par l'expert et qui décrivent des réseaux de connexions (choix, séquence, spécialisation ...) connus entre certains opérateurs pour atteindre un but. Chaque élément d'un schéma de plan peut aussi être décomposé en sous-éléments.
- les **buts** qui expriment les contraintes sur l'état final souhaité ;
- les **requêtes** qui sont des instanciations des buts sur des données particulières, sous des contraintes particulières ;
- les **plans** qui sont les résultats de la planification, c.à.d. des listes partiellement ordonnées d'opérateurs ou de squelettes de plans.

Voici à titre d'exemple, une description détaillée des structures de données représentant deux de ces concepts : les opérateurs et les schémas de plans.

La représentation des **opérateurs** terminaux comprend :

- une fonctionnalité (information sur la fonction que l'opérateur permet d'atteindre) ;
- une action (par exemple la syntaxe d'appel d'un programme);

- des effets : actions "*secondaires*" de cet opérateur sur les données et les objets du domaine :
- des informations sur les arguments (par exemple leurs noms, types, signature) ;
- des informations sémantiques : caractéristiques (données par l'expert), contraintes, pré et post conditions (sur les données) ;
- des critères d'évaluation des résultats, d'initialisation et d'ajustement des paramètres.

Les **schémas de plans** peuvent se représenter sous forme d'arbres, dont les feuilles sont connectées aux opérateurs terminaux. Les schémas de plans contiennent des informations sur les flots de données entre leurs éléments, ainsi qu'une connaissance "tactique" (par exemple des heuristiques pour décider du parcours de l'arbre).

3.2 Instructions de raisonnement de la bibliothèque

Les instructions de base implantent les mécanismes de raisonnement utiles pour réaliser la planification et le contrôle d'exécution, nécessaires au pilotage. Ces instructions doivent permettre d'implanter différentes stratégies, pour obtenir différents comportements. Elles utilisent les structures de données définies plus haut.

On peut séparer les instructions en trois types. Les instructions d'accès aux structures et aux données (par exemple *get_preconditions*, *pop*, *push*), les instructions élémentaires et les instructions *complexes*. Les instructions élémentaires sont très génériques, elles effectuent des opérations simples (par exemple *choose_one*, *select*). Les instructions *complexes* sont de granularité moindre, mais elles permettent de "cacher" des opérations complexes. Par exemple la planification non hiérarchique nécessite de manipuler des requêtes (instances de buts), nous avons défini des fonctions qui créent de nouvelles requêtes (*create_new_request*), qui mettent à jour ces requêtes après une exécution (*update_request*), etc.

Quelques unes de ces instructions sont décrites ci-dessous :

3.2.1 Instructions d'accès aux données

- *get_preconditions* : récupération de la liste des pré-conditions d'un opérateur.
- *push* : empilement d'un objet dans une pile.

3.2.2 Instructions élémentaires

- *select* : sélection d'objets (par exemple d'opérateurs), c'est une opération globale qui utilise un critère de tri grossier pour extraire des objets intéressants dans un ensemble. Cette opération utilise des critères qui dépendent des objets concernés. Par exemple pour les opérateurs terminaux le critère peut être : ceux qui partagent la même fonctionnalité, ceux qui modifient un objet d'intérêt (d'après leurs effets), ou ceux qui peuvent s'exécuter sur le matériel disponible.

$$\begin{array}{ccc} & \text{critère de sélection} & \\ & \downarrow & \\ \text{ensemble d'objets} \rightarrow & \boxed{\text{select}} & \rightarrow \text{objets retenus} \end{array}$$

- *choose_ one* : choix d'un objet le mieux approprié à la situation courante, parmi un ensemble (souvent ceux déjà retenus par *select*). Cette opération utilise un critère plus fin qui permet de déterminer la “valeur” d'un objet en fonction de ses caractéristiques propres (par exemple l'opérateur le plus rapide à exécuter) ou de l'état courant.

$$\begin{array}{ccc} & \text{critère de choix} & \\ & \downarrow & \\ \text{ensemble d'objets} \rightarrow & \boxed{\text{choose\_one}} & \rightarrow \text{meilleur objet} \end{array}$$

- *prepare* : préparation d'un opérateur avant son exécution. Ceci correspond par exemple à l'instanciation des paramètres formels par les paramètres réels et la propagation des contraintes issues des exécutions précédentes.

$$\text{opérateur} \rightarrow \boxed{\text{prepare}} \rightarrow \text{opérateur prêt à l'exécution}$$

D'autres instructions sont nécessaires, par exemple en planification non linéaire, différentes instructions d'ordonnancement permettent de construire un plan, telles que l'insertion d'un nouvel élément dans un plan, le réordonnancement d'un plan selon certaines contraintes. En planification hiérarchique des instructions pour le raffinement de plan ou les interactions entre niveaux hiérarchiques (flots de données et répartition des données, par exemple *unify* qui unifie le flot de données dans un schéma de plan) sont aussi implantées.

Le contrôle d'exécution requiert aussi des instructions, comme le diagnostic d'une erreur d'exécution, l'évaluation de la correction d'un résultat ou simplement l'exécution réelle d'un opérateur. Ces instructions sont aussi implantées dans la bibliothèque : *assess-operator*, *execute*, etc (cf Annexe A.1).

4 Exemples d'utilisation de la bibliothèque

Afin d'expliciter sur des exemples l'utilisation des structures de données et des instructions de base, détaillons la construction de la partie planification de deux moteurs particuliers. Pour des raisons de simplicité nous avons choisi comme premier exemple un moteur avec une planification hiérarchique à base de squelettes, où exécution et planification sont imbriquées. Le deuxième exemple est un moteur avec une planification partiellement ordonnée dérivé du premier.

4.1 Un moteur utilisant une planification hiérarchique

Un tel moteur se sert de décompositions de problèmes en sous-problèmes sous forme de squelettes de plans fournis par l'expert. C'est pourquoi, une base de connaissances pour un moteur de ce type contient principalement un ensemble de squelettes de plans, dont les feuilles sont des opérateurs terminaux. Les structures de données intéressantes dans ce cas sont donc les schémas de plan (pour représenter les squelettes) et les opérateurs (cf. §3.1).

Le comportement souhaité pour ce type de moteur est le suivant :

1. sélectionner le meilleur squelette en fonction du but donné par l'utilisateur final ;
2. si l'opérateur courant est une feuille du squelette (opérateur terminal), il faut préparer l'opérateur pour son exécution, vérifier les pré-conditions, l'exécuter, puis tester ses post-conditions.
3. si l'opérateur courant n'est pas une feuille du squelette, il faut le raffiner en expansant ses fils. Il existe quatre façons différentes de décomposer un père, selon le type de lien entre ses fils (décomposition en séquence, en parallèle, décomposition avec étape optionnelle ou choix). Le raffinement d'une décomposition avec étape optionnelle ou de choix se fait grâce à des critères fournis par l'expert;
4. raffiner récursivement chacun des fils jusqu'aux opérateurs terminaux.

A partir de la spécification de cette stratégie de moteur, le code à écrire par le concepteur du moteur, suit l'algorithme précédent. Ce code s'écrit grâce à un langage simple, avec quelques structures de contrôles usuelles (boucles, conditionnelles, etc). Les instructions de base de la bibliothèque qui sont utiles au concepteur dans ce cas sont donc : *choose_one*, *prepare*, *execute*, etc. Le code du corps du moteur, c'est à dire les étapes de raffinement et d'exécution (2, 3 et 4) à écrire par le concepteur est donné figure 4.

```
hierarchical-supervisor(execution_stack)                                    2
 do
  if empty?(execution_stack)
   then stop()
   "nothing left to execute"
  else current:= pop(execution_stack)
   if terminal(current)
   "It is an executable operator"
   then prepare(current)
   "prepare it to execution i.e instantiate parameters, etc."
     if preconditions_ok(current)
      then execute(current)
      "actual execution"
      if postconditions_ok(current)
-----------------------------------------------------------------------------
       then hierarchical-supervisor(execution_stack)                        4
      else error
   else error
-----------------------------------------------------------------------------
  else sons:= get_sons(current)                                             3
  case connection(current,sons)
  "selection on the type of decomposition in the plan skeleton"
   Sequence: push(execution_stack, sons)
   Or: push(execution_stack,choose_one(sons,criterium(current)))
   Optional: if criterium(current)
     then push(execution_stack, sons)
   other: error
  esac
-----------------------------------------------------------------------------
  hierarchical-supervisor(execution_stack)                                  4
end hierarchical-supervisor
```

FIG. 4 – *Exemple du code d'un moteur de pilotage hiérarchique écrit avec la bibliothèque de composants. Les numéros correspondent aux étapes de l'algorithme.*

Une fois ce code écrit, le concepteur utilise un compilateur fourni avec la bibliothèque pour obtenir le moteur de pilotage à base de squelettes, correspondant à sa spécification. Il est ensuite possible de lier une base de connaissances avec ce moteur pour obtenir un système à base de connaissances (cf. figure 3 page 8).

Pour obtenir un autre comportement, il suffit au concepteur de modifier ce code. Par exemple dans le cas où l'on ne veut pas exécuter réellement les opérateurs, le concepteur peut écrire un planificateur "pur", fournissant un plan en se basant simplement sur les pré et post conditions des opérateurs. Les modifications à apporter au code sont minimes, par exemple il faudrait remplacer la ligne "then *execute*(*current*)" par "then *push*(*stack, current*)" pour construire au fur et à mesure un plan et rajouter "*return*(*stack*)" en fin du code du planificateur. Une nouvelle compilation de ce code fournirait alors un autre planificateur.

4.2 Un moteur utilisant une planification partiellement ordonnée

Le second exemple de moteur de pilotage présenté utilise une stratégie de planification partiellement ordonnée, en se basant sur un dérivé du modèle de TO(P,G) donné en exemple dans la première partie.

PO(P,G)

1. **Termination check:** If G is empty, report success and return solution plan P.
2. **Goal selection:** Let c = select-goal(G), and let O_{need} be the plan step for wich c is a precondition.
3. **Operator selection:** Let O_{add} be an operator in the library that adds c. If there is no such O_{add}, then terminate and report failure. *Choice point: all such operators must be considered for completeness.*
4. **Ordering selection:** Insert O_{add} before O_{need}, call the resulting plan P'.
5. **Goal updating:** Let G' be the set of preconditions in P' that are not true.
6. **Recursive invocation:** PO(P',G').

FIG. 5 – *Spécification d'une stratégie de planification partiellement ordonnée.*

PO(P,G) (pour "Partial Order", voir figure 5) est la spécification de cette stratégie de planification. La différence entre l'algorithme TO(P,G) et PO(P,G) se situe dans la partie ordonnancement. Dans ce planificateur, il n'est pas possible d'effacer un fait. L'ordonnancement est donc plus simple puisque O_{del} n'existe pas, il suffit de trouver O_{need}.

L'algorithme PO consiste à sélectionner l'opérateur résolvant les contraintes du but courant ou d'une partie du but (3), puis à regarder si cet opérateur est exécutable. Si oui on l'exécute (4) (dans le cas d'une exécution imbriquée avec la phase de planification). Sinon il faut modifier le but pour ôter les conditions potentiellement résolues par l'opérateur et rajouter comme nouvelles contraintes du but, les préconditions de cet opérateur (5). Enfin on ajoute cet opérateur dans le plan, et on appelle récursivement le planificateur sur le nouveau but (6).

Le code du moteur utilisant un tel planificateur est présenté figure 6.

5 Approches similaires

Le besoin de diminuer le temps et le travail de re-écriture de codes et de fournir des environnements de développement uniformes s'est fait sentir dans différents domaines de l'intelligence artificielle. La nécessité de gérer la prolifération de codes, de tester de nouveaux algorithmes ou de comparer des algorithmes entre eux a motivé beaucoup de travaux. C'est ainsi que des équipes aux objectifs différents (construction de générateurs de SBC, génération d'outils d'acquisition de connaissances, etc) se sont retrouvées dans une approche orientée génie logiciel de développement de méta-outils. Parmi eux, nous en citons ici certains dont la démarche se rapproche de la notre, par leurs objectifs ou par la mise en œuvre de ces objectifs.

PROTÉGÉ-II [24] propose une architecture configurable et extensible pour générer des outils d'acquisition des connaissances. Ce système fournit des mécanismes réutilisables pour différentes méthodes de résolution de problèmes, grâce à des bibliothèques pré-définies de classes de tâches et de méthodes. Un langage de tâches et de sous-tâches permet de construire de nouvelles méthodes de résolution de problèmes par assemblage des méthodes de base. L'objectif principal est de générer automatiquement des éditeurs d'acquisition de connaissances personnalisés pour l'expert d'un domaine.

Araki dans [1], décrit un système nommé **ARES/SCH** dédié à une tâche spécifique : l'ordonnancement. Son but est de permettre l'écriture facile de moteurs d'ordonnancement à partir d'une bibliothèque de “tâches primitives” (environ une vingtaine, indépendantes du domaine d'application). Ces tâches primitives ont une granularité assez fine, ce qui signifie qu'elles n'implantent pas de stratégies en elles-mêmes (contrairement à d'autres systèmes comme CAKE [5]). La stratégie désirée

```
partial-order-supervisor(execution_stack)                                   1
 do
   if constraints(current_request())
     then stop() "End"
```

```
   collect_all (Set_of_Selection)                                           3
   select(Set_of_Selection, applicability-criterium)
   if empty?(Set_of_Selection)
     error ("supervisor no-schema-to-apply ", current_request())
   sort(Set_of_Selection, comparaison-criterium)
   push(execution_stack, choose_one(Set_of_Selection,choice-criterium))
```

```
   while and( not(empty?(execution_stack)),                                 4
     if precondition_ok(top(execution_stack)))
     then do
     node:= top(execution_stack))
       prepare(node)
       execute (node)
       if postcondition_ok(pop(execution_stack), postconditions)
       then push(Request_stack,update_request(pop(Request_stack)))
       else error
     done
     else error
```

```
   if (constraints(current_request))                                        5
     then stop() "End"
     else if not(empty?(execution_stack))
       then push(Request_stack,
         create_new_request(pop(Request_stack), top(execution_stack)))
       fi
     fi
```

```
   partial-order-supervisor(execution_stack)                                6
   do
 end partial-order-supervisor(execution_stack)
```

FIG. 6 – *Exemple de code d'un moteur de pilotage avec planification partiellement ordonnée écrit avec la bibliothèque. Les numéros correspondent aux étapes de l'algorithme.*

est décrite au moment de la conception d'un nouveau moteur par un graphe de flot de tâches primitives. La compilation de ce graphe fournit le nouveau moteur. Cette technique a été employée dans différentes applications en ordonnancement, par exemple dans une usine de test de composants LSI, de montage de mémoires.

La bibliothèque de classes C++, $\mathcal{MLC}$++ [7] propose à la communauté de recherche en apprentissage automatique une plate-forme commune, principalement pour des comparaisons ou des hybridations entre algorithmes d'apprentissage. Cette bibliothèque fournit des structures de données communes riches ainsi que des implantations d'algorithmes d'apprentissage connus. Les chercheurs peuvent soit utiliser ces algorithmes tels quels sur différents jeux de données, soit en dériver des variantes assez facilement. Le principal effort de ce projet a porté sur la définition des classes C++ utiles aux différents algorithmes, et en particulier celles qui permettent les conversions de données ou la fourniture de statistiques.

Comparaison

Les éléments des bibliothèques fournis par PROTÉGÉ-II sont d'un grain plus gros que ceux que nous proposons. De plus, contrairement à PROTÉGÉ-II [24], nous ne nous intéressons pas à l'aspect acquisition des connaissances, mais à l'aspect développement de moteurs.

Les travaux réalisés dans le cadre d'ARES/SCH en ce qui concerne la construction de moteurs à partir de composants primitifs, sont proches des nôtres, mises à part les différences dues à la tâche visée (ordonnancement vs planification). Même si l'objectif de $\mathcal{MLC}$++ est un peu différent, les problèmes de représentation des données et des structures communes sont similaires aux nôtres.

Ce type d'approche modulaire s'avère bien adapté pour :

- créer de nouveaux moteurs, car la définition de composants primitifs fournit une représentation à un niveau d'abstraction proche de l'expression du concepteur. C'est par exemple le cas des graphes de tâches primitives proposés par ARES/SCH qui correspondent à la représentation classique des méthodes d'ordonnancement;
- re-configurer des moteurs existants, soit pour tester différentes versions sur une même application, soit pour étendre un moteur existant à de nouveaux besoins. Araki cite l'exemple de dix versions d'ordonnanceurs produites en moins de trois mois pour la même application.

6 Résultats

Cette bibliothèque nous a permis de développer différents types de moteurs de pilotage en plus de ceux présentés section 4. Ainsi, nous avons écrit six moteurs de planification ayant des fonctionnalités spécifiques, dont nous avons développé plusieurs versions. Le tableau suivant résume les différentes fonctionnalités de chaque moteur.

	NL-II	Pulsar DD	PEGASE	PULSAR IU
Type de planification	Non ordonnée	Non ordonnée	Ordonnée	Non ordonnée
Raisonnement dirigé par	les données	les données	les buts	les données
Exécution	Non	Oui	Oui	Oui
Replanification	Non	Non	Oui	Oui
Résolution de contraintes	Non	Non	Non	Non
Nb de version	2	1	3	2

	PULSAR IP	PHENIX
Type de planification	Partiellement ordonnée	Partiellement ordonnée
Raisonnement dirigé par	les buts	les buts
Exécution	Oui	Oui
Replanification	Oui	Oui
Résolution de contraintes	Oui	Oui
Nb de version	1	2

Il est intéressant de noter qu'en tout, nous avons développé plus de 11 moteurs en un an. Afin de comparer les différents moteurs, nous avons utilisé la même base de connaissances comme test (voir [11]). Nous avons choisi comme critère de comparaison la qualité du plan obtenu, sans tenir compte de la complexité de la recherche, ou des performances en temps.

Tous ces moteurs s'appuyant sur des structures communes, la base de connaissances de test n'a pas eu à être modifiée. Cette technique de conception nous a permis de tester facilement différentes variantes du même type de moteur simplement en modifiant la spécification de sa stratégie et le code correspondant.

7 Conclusion

La bibliothèque comporte environ 22 fonctions élémentaires (voir l'annexe A.1), 24 fonctions d'accès (annexe A.3) et 4 fonctions complexes (annexe A.2) qui permettent d'écrire la plupart des mécanismes de pilotage courants.

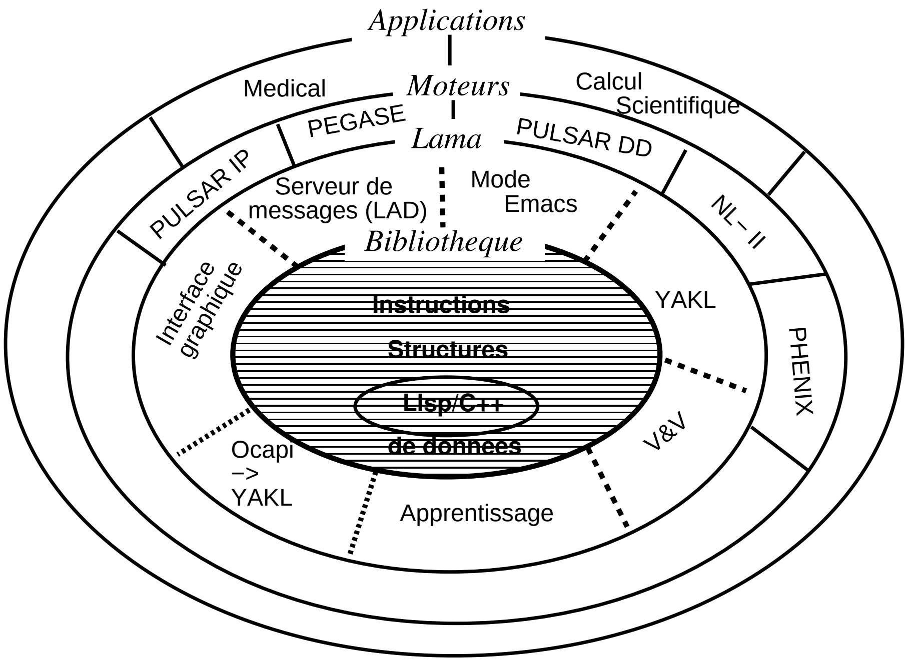

FIG. 7 – *L'architecture de* LAMA *est en couche. Chaque couche correspond à un niveau d'expertise. La bibliothèque que nous avons développée est située au centre (en rayé), le niveau* LAMA *est celui de la plate-forme avec les outils tels que le langage de description de bases de connaissances* YAKL, *des outils de vérification (V&V), des modes de développement Emacs et des outils optionnels comme les interfaces graphiques ou l'apprentissage. Les moteurs de pilotages sont basés sur* LAMA *et le niveau supérieur est celui des concepteurs de base de connaissances (Applications).*

Associée à cette bibliothèque, nous avons crée une plate-forme de développement (LAMA : **L**ibrary **A**nd platfor**M** for pl**A**nning)). Elle comprend, outre la bibliothèque de composants, un langage de description de base de connaissances (YAKL), un outil de vérification de base de connaissances(voir [10]), des mécanismes de récupérations d'erreurs (voir [17], [6] et [13]), des interfaces graphiques, et un serveur de messages

(LAD). Le schéma 7 présente l'intégration des différents modules. L'avantage considérable de cette plate-forme est de pouvoir disposer d'un environnement commun à tous les moteurs écrits avec cette bibliothèque tels que des interfaces, un langage d'écriture de base de connaissances ou un vérificateur syntaxique communs. Il est possible de tester plusieurs moteurs sur une même base de connaissances ou d'essayer plusieurs classes d'applications avec le même moteur tout en gardant le même environnement de développement.

Actuellement, nous avons finalisé une première beta-version de LAMA afin de la diffuser. Nous travaillons sur des extensions des mécanismes de contrôle d'exécution qui sont aussi nécessaires afin de couvrir les besoins de la classe de problèmes du pilotage de programmes. Les extensions futures porteront sur l'adaptation de la bibliothèque à l'écriture de moteurs pour d'autres classes de problèmes, comme l'interprétation automatique d'images.

Références

[1] D. Araki, K. Narimatsu, and S. Kojima. Task-oriented Modeling of Job-shop Scheduling Knowledge. In *Proceedings of the Third Japanese Knowledge Acquisition for Knowledge-based Systems Workshop*, pages 17–31. JKAW-94, November 1994.

[2] W. J. Clancey. Model construction operators. *Artificial Intelligence*, 53:1–115, 1992.

[3] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[4] C. Granger. *Reconnaissance d'objets par mise en correspondance en vision par ordinateur*. PhD thesis, Université de NICE, June 1985.

[5] M. Hori, Y. Nakamura, and T. Hama. Methodology for configuring scheduling engines with task-specific components. In *Proceedings of the Second Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 215–229. JKAW'92, 1992.

[6] A. E. Howe. Analyzing failure recovery to improve planner design. In *Proc. of AAAI-92*, pages 387–392, San-Jose, CA, 1992.

[7] Ron Kohavi, George John, Richard Long, David Manley, and Karl Pfleger. $\mathcal{MLC}++$: A Machine Learning Library in C++. In *Tools with Artificial Intelligence*, November 1994.

[8] H. Laruelle, P. Régnier, and S. Thiébaux. Dossier plannification et action. Rapport IRIT 94/1/R, Institut de Recherche en Informatique de Toulouse (IRIT), 1994.

[9] R. Levinson. A general programming language for unified planning and control. *Artificial Intelligence*, 76:319 – 375, 1995.

[10] M. Marcos, S. Moisan, and A. P. del Pobil. Verification and Validation of Knowledge-Based Program Supervision Systems. In IEEE, editor, *IEEE International Conference on Systems, Man and Cybernetics*, Vancouver, Canada, October 1995.

[11] S. Mathieu-Marni, S. Moisan, and R. Vincent. A knowledge based system for the determination of land use mixing and the classification of a multi-spectral satellite image. *International Journal of Remote Sensing*, 17:1483 – 1492, 1996.

[12] D. McDermott and J. Hendler. Planning : What it is, what it could be, an introduction to the special issue on planning and scheduling. *Artificial Intelligence*, 76:1 – 16, 1995.

[13] P. Mi and W. Scacchi. Articulation : An integrated approach to the diagnosis, replanning, and rescheduling of software process failures. In *Proceedings of KBSE'93*, 1993.

[14] S. Minton, J. Bresina, and M. Drummond. Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research*, 2:227 – 262, December 1994.

[15] S. Moisan, F. Ployette, M. Thonnat, V. Clément, and R. Vincent. Documentation de l'interface du Logiciel OCAPI, Version 2.0. Technical Report RT-0186, I.N.R.I.A. Sophia Antipolis, 1995.

[16] S. Moisan, R. Vincent, M. Thonnat, V. Clément, and J. van den Elst. Manuel de Référence du logiciel OCAPI version *2.0*. Technical Report RT-0183, I.N.R.I.A. Sophia Antipolis, 1995.

[17] S. Moisan, R. Vincent, J. van den Elst, and F. van Harmelen. Towards an intelligent failure handling mechanims in program supervision. In *Proceedings of KBUP'95*, 1995.

[18] E.D. Sacerdoti. The non-linear nature of plans. In *Advanced papers of IJCAI-75*, pages 206–214, 1975.

[19] C. Shekhar, S. Moisan, and M. Thonnat. Real-time perception program supervision for vehicule driving assistance. In *International Conference on Recent Advances in Mechatronics ICRAM'95*, Istambul, Turkey, August 1995.

[20] L. Steels. Components of expertise. *AI Magazine*, 11(2):28–49, 1990.

[21] A. Tate, J. Hendler, and M. Drummond. A review of ai planning techniques. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in planning*, pages 26–49. Morgan-Kaufmann Publishers, 1990.

[22] M. Thonnat, V. Clement, and J. van den Elst. Supervision of perception tasks for autonomous systems: the OCAPI approach. *International Journal on Information Science and Technology*, 3(2):140–163, January 1994. Also in Rapport de Recherche 2000, 1993, INRIA Sophia Antipolis.

[23] M. Thonnat and M.-H. Gandelin. An expert system for the automatic classification of zooplanktons from monocular images. In *9 th International Conference on Pattern Recognition ICPR9*, pages 114–118, November 1988.

[24] S. W. Tu, Y. Shahar, J. Dawes, J. Winkles, A. R. Puerta, and M. A. Musen. A problem-solving model for episodic skeletal-plan refinement. *Knowledge Aquisition*, 4:197–216, 1992.

[25] J. van den Elst, F. van Harmelen, G. Schreiber, and M. Thonnat. A functional specification of reusing software components. In *Sixth International Conference on Software Engineering and Knowledge Engineering*, pages 374–381. Knowledge Systems Institute, juin 1994.

[26] J. van den Elst, F. van Harmelen, and M. Thonnat. Modelling Software Components for Reuse. In *Seventh International Conference on Software Engineering and Knowledge Engineering*, pages 350–357. Knowledge Systems Institute, June 1995.

[27] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publisher, Inc, 1988.

[28] D.E. Wilkins. Hierarchical planning: Definition and implementation. Technical Note TN 370, SRI International, December 1985.

A Liste des fonctions disponibles dans LAMA

A.1 Fonctions élémentaires

- criterium (criteria objet)
- call(criteria set_of_selection)
- choose_one(set criteria)
- select(set criteria)
- sort (set criteria)
- collect_all(kb)
- connection(operator, sons)
- error(a b c)
- terminal(operator)
- prepare (operator)
- change-state ()
- execute(operator)
- preconditions_ok(operator)
- postconditions_ok(operator)
- constraints (request)
- activate(operator)
- desactivate(operator)
- activated?(op)
- asses-operator?(operator)
- asses-operator(operator)
- return(value)
- stop()

A.2 Fonctions complexes

- unify(operator)
- substitutions(elt request)
- create-new-request(request operator)
- update-request(request)

A.3 Fonctions d'accès aux données et aux structures

- type-of (objet)
- current-request()
- current-plan-element ()
- current-bc()
- push(stack element)
- pop(stack)
- top(stack)
- copy(stack)
- empty?(stack)
- map-push(stack set_of_selection)
- get-sons(operator)
- get-conditions(operator type valeur-test)
- get-attribute(champ objet)
- set-attribute(champ objet valeur)
- get-name(objet)
- get-functionality(objet)
- get-weight(operator)

- get-arguments-in(objet)
- get-arguments-out(objet)
- get-preconditions(operator)
- get-postconditions(operator)
- get-constraints(goal)
- get-criterium (operator type)
- set-criterium (operator type value)

Table des matières

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399