Réutilisation et générateurs de systèmes à base de connaissances : le *framework* BLOCKS

Sabine Moisan

INRIA Sophia Antipolis
2004 route des Lucioles- BP 93
F-06902 Sophia Antipolis
Sabine.Moisan@sophia.inria.fr

RÉSUMÉ. *Nous nous intéressons au problème de la réutilisation dans la conception de générateurs de systèmes à base de connaissances (GSBC). Etant donnée la durée de vie de ces logiciels, leurs concepteurs ont des besoins de réutilisabilité cruciaux. Pour répondre à ces besoins, nous avons tout d'abord utilisé notre expérience pour faire une analyse du domaine. De cette analyse, nous avons abstrait des concepts et des mécanismes de raisonnement qui nous ont amené à définir* BLOCKS, *un « framework » offrant des composants logiciels réutilisables et adaptables pour le concepteur. Enfin, nous avons utilisé les composants précédents pour réaliser des GSBC (dont certains par réingénierie de GSBC préexistants), ce qui nous a permis par un processus itératif de valider et d'améliorer le framework. Les conclusions de cette expérience sont que les techniques de réutilisabilité s'appliquent bien aux générateurs de systèmes à base de connaissances, permettent un gain en temps et améliorent la lisibilité des logiciels construits.*

ABSTRACT. *We are interested in the problem of re-use during the design of knowledge-based system generators (KBSG). The designers of these software products have major reuse needs. To address their needs, we have first conducted a domain analysis based on our past experience. This analysis has highlighted abstract concepts and reasoning mechanisms, which lead us to the definition of* BLOCKS, *a framework composed of reusable and adaptable software components, and dedicated to KBSG designers. Finally, we have used the components to build several KBSG (some of them by round-trip re-engineering), in order to validate and to improve the framework. We conclude that the re-use techniques are well fitted to knowledge-based system generators, and that they allow us a significant gain in time, as well as they improve sofware readability.*

MOTS-CLÉS : *framework, composants réutilisables et adaptables, systèmes à base de connaissances, ontologies de tâche.*

KEYWORDS: *framework, reusable and adaptable components, knowledge-based systems, task ontology.*

1. Introduction

Nous nous intéressons à des logiciels particuliers, les générateurs de systèmes à base de connaissances (GSBC), et aux problèmes qui se posent à leurs concepteurs lors de leur conception ou de leurs modifications. Au cours de nos travaux en intelligence artificielle, nous avons réalisé plusieurs générateurs de systèmes à base de connaissances différents. Ces logiciels ont une durée de vie assez longue, couramment de 10 ans ou plus, et subissent donc des évolutions. Dans nos expériences pratiques en tant que concepteurs de générateurs de systèmes à base de connaissances, nous nous sommes trouvés confrontés à un réel besoin de réutilisabilité. En effet, les modifications demandées par nos applications ont entraîné une redondance de développement et des difficultés de maintenance. Pour remédier à ces problèmes et pour offrir une meilleure compréhension des comportements implantés par un GSBC, nous proposons une solution dans la lignée des bibliothèques de composants [SZY 98] et des architectures génériques, aussi appelées « *frameworks* » [JOH 88].

Dans cette introduction, après une brève définition des générateurs de systèmes à base de connaissances, nous allons examiner les problèmes de réutilisation qui se posent pour ces logiciels et la nature de réutilisabilité qui nous intéresse.

1.1. *La réutilisation dans les générateurs de systèmes à base de connaissances*

Les systèmes dits *à base de connaissances* (ou SBC) sont qualifiés ainsi car ce sont des logiciels qui traitent des problèmes complexes en se servant de connaissances décrites de manière déclarative. Leur architecture comporte trois parties principales : une base de connaissances, remplie par un expert d'un domaine particulier, qui stocke une compilation de son expertise sur ce domaine ; une base de faits, contenant les objets représentant le problème particulier d'un utilisateur final ; et enfin un moteur, c'est-à-dire un logiciel écrit par un concepteur, chargé de réaliser des inférences pour traiter le problème à l'aide des connaissances fournies. Ces trois rôles (expert, utilisateur, concepteur) correspondent à trois types d'activités différentes, nécessitant en particulier des outils différents.

La notion de générateur de SBC a émergé vers la fin des années 80. Les générateurs sont issus d'un effort d'abstraction à partir des premiers systèmes-experts développés. Un générateur permet de réutiliser des éléments communs (au moins pour une gamme d'applications ou un type de problèmes), comme un moteur d'inférence général (dont la stratégie peut être modulable), des modes de représentation des connaissances (tels que *frames*, règles ou logiques) ainsi que des interfaces (comme des éditeurs d'acquisition de la connaissance) ou des modules d'aide à la réalisation de bases de connaissances. Un générateur est construit une fois pour toutes, indépendamment de toute expertise particulière à une application. Ensuite, il peut être utilisé par des experts, pour écrire des bases de connaissances et construire ainsi autant de systèmes à base de connaissances que d'applications à traiter. Les systèmes résultants peuvent être aussi divers qu'un système de diagnostic des maladies du foie, de classification de pollens ou de configuration d'architecture d'ordinateurs.

Au cours du processus de génération d'un SBC, l'expert construit ou utilise un modèle des connaissances du domaine et une méthode de résolution de problème adaptée. Ensuite, grâce aux outils fournis par un générateur de SBC, il aboutit à un système à base de connaissances opérationnel qui peut résoudre des problèmes dans ce domaine. Selon les domaines et les types de résolution de problème, les outils d'un GSBC donné peuvent rendre cette démarche plus ou moins facile.

C'est pourquoi, pour des raisons d'efficacité et de facilité d'utilisation, les générateurs sont souvent spécialisés pour une tâche particulière. L'organisation de la connaissance ou le contrôle du raisonnement que leurs moteurs implantent permettent de mieux modéliser cette tâche. Une tâche est définie dans [CHA 93] comme « un type de résolution de problème ... dont on peut parler de manière abstraite ». Par exemple, le diagnostic est une tâche car c'est un type de résolution de problème dont le but est d'identifier les causes d'un ensemble de dysfonctionnements observés dans un système. On peut ainsi parler d'un système de diagnostic indépendamment des méthodes algorithmiques adoptées pour implanter ce système.

1.2. *Besoins de réutilisabilité pour le concepteur*

Le concepteur d'un GSBC est un informaticien qui a pour rôle d'implanter un comportement de résolution de problème, en fonction des besoins d'experts et d'utilisateurs finaux (voir figure 1). Or, d'une part, le comportement demandé à un SBC

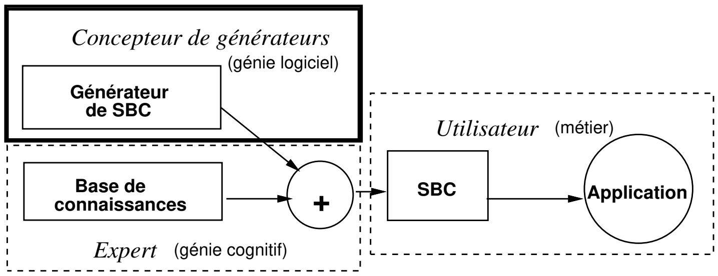

Figure 1. *Un générateur de SBC construit par un concepteur et utilisé par des experts, génère différents systèmes à base de connaissances, pour des applications relevant du métier des utilisateurs*

peut varier et évoluer selon les besoins d'une application particulière, l'évolution de la connaissance sur un domaine ou les exigences de certains utilisateurs. De plus, pour une même tâche, on peut envisager des moteurs plus ou moins sophistiqués ou des moteurs de comportements identiques mais dans des langages ou sur des machines différents. D'autre part, l'expert, qui souvent n'est pas un informaticien, va exprimer ses besoins en terme de comportements attendus du SBC final, à un niveau d'abstraction correspondant à la tâche qu'il souhaite réaliser, par exemple le diagnostic d'un dysfonctionnement, la classification d'un objet dans une taxonomie, la conception d'un artefact, etc.

Le comportement d'un SBC, qui dépend largement du moteur d'inférence, est difficile à comprendre, tant pour l'expert qui écrit une base de connaissances et veut utiliser au mieux les capacités du moteur qui lui est fourni, que pour le concepteur qui doit modifier ces capacités. En effet, pour implanter les moteurs et les structures de données pour la représentation des connaissances, le concepteur n'a souvent à sa disposition que des langages de programmation d'un niveau d'abstraction assez bas par rapport au niveau de raisonnement de la tâche de l'expert. Ceci conduit à des codes où la stratégie du moteur est noyée dans des détails d'implantation. De plus, les problèmes d'implantation conduisent souvent à figer cette stratégie, rendant les générateurs difficiles à étendre. Toute évolution de spécifications, de besoins ou de contraintes nécessite des reconfigurations du moteur, voire une réécriture complète. Ces modifications sont délicates et coûteuses, et peuvent avoir des effets de bord imprévus. Les évolutions sont cependant inévitables sur la durée de vie d'un générateur, qui est par définition conçu pour servir pour de nombreuses applications et sur une longue période.

Les GSBC sont conçus pour être réutilisables par les experts construisant les SBC finaux, et la réutilisation dans ce sens est en effet assez courante : c'est-à-dire qu'un expert peut facilement modifier et faire évoluer le contenu d'une base de connaissances, ou écrire une nouvelle base dans un autre domaine avec le même moteur. L'intelligence artificielle propose des outils pour aider à concevoir un modèle conceptuel d'une résolution appliquée à un domaine, du point de vue du *génie cognitif*. Ces outils permettent à un expert de s'exprimer en termes de son domaine d'application et de son raisonnement applicatif plutôt qu'en termes d'implantation informatique. Nous étudions aussi ce type d'outils (mentionnés en figure 2) mais nous ne les développerons pas dans cet article qui est consacré à l'aide au concepteur. Pour le concepteur, une fois le modèle cognitif défini, il s'agit de concevoir un modèle conceptuel de la résolution d'une tâche abstraite, du point de vue du *génie logiciel*, et ensuite de l'implanter sous forme d'un logiciel (un moteur). Le modèle conceptuel d'un moteur correspond aux concepts et au comportement de ce logiciel. Les concepteurs ont donc des besoins qui ne sont pas les mêmes que ceux des experts. En particulier, dans ce cas, les détails d'implantation doivent leur être exposés. Les GSBC et leurs moteurs sont des logiciels de taille importante et, du point de vue du génie logiciel, ils souffrent des mêmes difficultés que d'autres logiciels pour les maintenir et les faire évoluer [SOM 92]. Lors des évolutions, les modifications à réaliser dans un GSBC portent souvent sur des variantes de comportement, de fonctionnalités ou d'implantation, la tâche restant fondamentalement la même (par exemple le système final doit toujours réaliser une conception ou un diagnostic). Le concepteur a donc besoin, d'une part, de comprendre facilement le comportement d'un GSBC donné et, d'autre part, de réutiliser du code et des architectures, pour faire évoluer ce logiciel ou pour en créer une variante. Pour cela, les techniques de réutilisabilité semblent bien adaptées. Nous proposons une solution sous forme d'architecture générique – ou *framework* – offrant aux concepteurs des bibliothèques de composants réutilisables.

1.3. *Travaux similaires*

Le besoin de faciliter le travail de réécriture de codes à un niveau d'abstraction élevé et de fournir des environnements de développement uniformes a donné lieu à différents travaux en génie logiciel et en intelligence artificielle. Ces travaux diffèrent par la nature de la réutilisabilité qu'ils offrent.

En génie logiciel, l'objectif est la réutilisation de modèles de conception, de codes et d'architectures logicielles. Les systèmes proposés se veulent opérationnels, comme la plate-forme ITHACA [MEY 93] qui fournit un environnement de développement pour des applications à objet et guide le concepteur pour la réutilisation d'un ensemble de composants stockés par la plate-forme. Depuis MVC [KRA 88], beaucoup de *frameworks* ont aussi été proposés pour la réalisation d'interfaces graphiques, comme HotDraw [BEC 94] pour l'édition de graphes, ou ET++ [WEI 88] et Qt [DAL 99], qui sont distribués et utilisés. Mais les *frameworks* sont appliqués dans bien d'autres domaines. Pour les systèmes concurrents, on peut citer Actalk [BRI 99]), qui permet d'expérimenter et de comparer des modèles de langages à objets concurrents. Dans le domaine des réseaux, Conduits+ [HüN 95] permet l'implantation de protocoles de communication. Des *frameworks* existent aussi pour la compilation, comme RTL [JOH 91] ou Choices [CAM 93].

En revanche, en intelligence artificielle, les travaux – développés principalement dans la communauté « acquisition des connaissances » – ont pour but principal non pas la réutilisation de codes, mais la réutilisation de connaissances. Des systèmes comme PROTÉGÉ-II [MUS 93], COMMET [STE 91] ou SPARK [KIN 91] proposent une réutilisation des méthodes abstraites de résolution de problème et d'ontologies[1]. À un niveau encore plus conceptuel, les bibliothèques du système COMMONKADS proposent, pour l'acquisition des connaissances, des modèles réutilisables d'ontologies et de méthodes de résolution de problèmes à composer, sous forme de descriptions formelles [SCH 94].

Notre approche globale, décrite en section 4, permet les deux types de réutilisation : réutilisation de modèles et de méthodes de résolution de problème (en limitant la gamme des tâches abordées), mais aussi réutilisation de classes, de code et d'architectures de conception.

Les objectifs des travaux sur DSTM [TRI 99] ou TASK [TAL 96a], qui sont dédiés à la construction de SBC, sont simulaires aux nôtres. Ils prennent en compte aussi

1. Une ontologie définit un vocabulaire (lié aux concepts manipulés dans un domaine ou pour une activité), le sens des mots du vocabulaire, leur usage et les relations entre eux, de manière consensuelle pour un groupe de personnes travaillant dans ce domaine, indépendamment d'une application particulière.

bien la modélisation de la tâche que l'exécution effective du système. Cependant, les techniques utilisées et les priorités choisies sont différentes. DSTM a comme objectif le *prototypage*, afin de permettre rapidement de valider et d'affiner un modèle conceptuel cognitif avant implantation. Un noyau d'opérationnalisation permet de construire un prototype et offre, dans une certaine mesure, des capacités d'extension des primitives de modélisation et des mécanismes qui les manipulent. Dans notre approche, le *framework* BLOCKS constitue la partie dédiée à l'implantation des moteurs. Il est donc complémentaire de DSTM. De son côté, TASK propose aux ingénieurs de la connaissance un ensemble de langages pour les différentes étapes de création d'un SBC, avec en particulier un langage formel de spécification et un langage d'opérationalisation (pour une implantation en Lisp). Contrairement à BLOCKS, TASK ne permet pas la définition de méthodes de résolution génériques ni les extensions du contrôle, des mécanismes et des structures proposées par le langage d'opérationalisation [TAL 96b].

En complément des outils dédiés aux experts, le but de BLOCKS est d'assister les concepteurs face aux problèmes de réutilisabilité de logiciel auxquels ils sont confrontés lors de la conception de GSBC. Pour cela, nous proposons (section 2) une solution qui se base sur les notions de *framework* et de composants réutilisables et adaptables. L'identification de ces composants a été faite (section 3) et les *frameworks* qui en découlent ont été intégrés avec divers outils destinés aux concepteurs dans une plateforme logicielle décrite brièvement section 4. La conception des composants a abouti à la réalisation de BLOCKS (section 5). Enfin, des applications nous ont permis de mesurer les avantages de l'utilisation de *frameworks* pour la conception de GSBC et les capacités d'adaptation des composants, comme décrit section 6. En conclusion, nous dégageons les bénéfices apportés par cette approche et nous envisageons comment et dans quelles limites le processus itératif d'identification/conception/adaptation de composants va se poursuivre dans le cadre de nos applications.

2. Solution proposée

Pour répondre aux deux objectifs de facilité de conception et de compréhension, nous proposons une solution sous forme de *framework* [BOS 99] qui offre des primitives opérationnelles flexibles et adaptables ainsi que des mécanismes de manipulation de ces primitives.

Pour faciliter la conception de moteurs, le *framework* fournit au concepteur un ensemble de classes qui coopèrent et qui permettent la réutilisation de conceptions de moteurs existants. Il permet de représenter de façon uniforme des concepts abstraits interagissants, dont des variantes concrètes vont exister dans le GSBC développé. Ces concepts doivent être, d'une part, suffisamment abstraits pour être réutilisables dans différentes situations et, d'autre part, offrir des « points de flexibilité » pour que le concepteur puisse facilement les étendre et les adapter à des besoins particuliers.

Pour améliorer la lisibilité des codes des GSBC et la compréhension des comportements qu'ils implantent, les composants primitifs du *framework* fournissent une représentation qui, tout en étant opérationnelle, se situe à un niveau d'abstraction proche du niveau du dialogue entre le concepteur et l'expert. Le *framework* assure ainsi une correspondance entre l'expression des fonctionnalités comportementales abstraites demandées par l'expert et le code opérationnel correspondant fourni par le concepteur.

Le problème majeur est de trouver un équilibre entre la généralité et la réelle applicabilité des composants fournis. Plus un composant est général, plus il est réutilisable, mais plus il peut nécessiter d'effort d'adaptation pour l'être concrètement ; inversement, plus il est spécifique, moins il sera possible de le réutiliser dans différentes tâches. Une trop grande généralité nuit aussi à l'efficacité nécessaire lors de la réalisation d'applications réelles. Notre réponse à ce problème tient dans la limitation de la gamme de tâches visées. Nous n'envisageons pas de couvrir tous les GSBC possibles, ni toutes les tâches. Nous nous limitons à deux ou trois tâches particulières (complémentaires dans le cadre de nos applications). Ces tâches sont, premièrement, le pilotage de programmes (qui consiste à trouver, à partir d'un ensemble de programmes de traitement préexistants, le meilleur enchaînement possible de ces programmes, qui soit adapté aux données et à l'objectif de traitement d'un problème posé) et, deuxièmement, l'interprétation de haut niveau de séquences de données (des images, par exemple). La gamme de tâches visées étant ainsi restreinte, les composants réutilisables du *framework* que partagent ces tâches sont, en contrepartie, facilement applicables. De plus, l'approche objet facilite l'adaptation du degré de généralité des composants. Malgré la restriction des tâches, ce travail nécessite cependant l'abstraction de concepts issus de tâches distinctes et les composants doivent refléter cette généralité.

Grâce à un tel *framework*, les concepteurs peuvent écrire un moteur en réutilisant et en adaptant les composants proposés, sans réécriture de code massive. Ceci améliore la sûreté de programmation, mais aussi la lisibilité des GSBC et leur flexibilité. En effet, le code résultant est plus facile à comprendre et à modifier que du code brut. De plus, la coopération entre plusieurs systèmes à base de connaissances construits à partir des mêmes primitives est simplifiée. Ce type de coopération est parfois nécessaire dans des applications complexes et les protocoles de communication à mettre en place peuvent s'appuyer sur l'existence de composants communs. Enfin, le lien entre la phase de modélisation et la phase opérationnelle est facilité car les niveaux d'abstraction sont plus proches entre le modèle des experts et la description de sa réalisation opérationnelle.

L'élaboration du *framework* a commencé par une analyse de domaine pour permettre l'identification de ses composants.

3. Analyse de domaine

Nous avons entrepris cette analyse de domaine, en vue d'une réingénierie de l'existant, guidée par les besoins de nos applications. Cette analyse s'est appuyée sur nos réalisations passées (trois codes de GSBC respectivement en conception, classification et pilotage de programmes, et leurs variantes), ainsi que sur des articles sur divers autres GSBC. Comme nous l'avons dit, cette analyse ne visait pas l'exhaustivité, mais s'est limitée à quelques tâches. L'objectif de cette analyse était de trouver des abstractions qui recouvrent le mieux possible les concepts des différents GSBC visés et de déterminer quels changements ou variantes peuvent être nécessaires, c'est-à-dire quels sont les points de flexibilité à prendre en compte.

Une abstraction des principes et des architectures de ces différents générateurs a conduit à l'identification et à la modélisation de chaque concept important d'un SBC par une classe (abstraite), pour constituer une bibliothèque de classes réutilisables et extensibles, adaptée à la construction de différents moteurs de SBC. L'interface de ces classes a été conçue la plus complète possible, pour couvrir le plus de besoins sans que le concepteur ait à la modifier. Mais chaque point de flexibilité prévu (*hot spot* [PRE 94]) permet une adaptation afin qu'à une même interface abstraite puissent correspondre différents comportements à un niveau plus fin.

3.1. *Démarche*

L'analyse a procédé en différentes étapes et de manière cyclique. En effet, nous avons effectué plusieurs itérations, grâce à des retours sur les GSBC existants.

La première étape a été une modélisation abstraite de différents générateurs dédiés à des tâches différentes (en particulier pour le pilotage [MAR 99b, ELS 96]), grâce à des outils comme la méthode KADS [WIE 92] ou le formalisme UML [MUL 97]. Ces modèles se sont enrichis et affinés au fur et à mesure de l'évolution du *framework*. La méthode KADS a été utilisée tout d'abord pour des SBC réalisés dans des domaines d'application spécifiques. Nous l'avons ensuite utilisée de façon moins habituelle pour modéliser des moteurs, dédiés principalement à la tâche de pilotage, sans domaine d'application particulier. KADS a été très utile pour la partie modélisation du comportement dynamique des moteurs et la définition des ontologies. Cette étude préliminaire a aidé ensuite à dégager les modèles UML des classes permettant l'implantation de moteurs.

La deuxième étape a utilisé cette modélisation pour dégager des modèles fins à la fois des comportements de résolution de problème et des connaissances appliquées dans la réalisation des tâches. Cette étape essentielle a permis, de façon indépendante de l'implantation :

– d'identifier les concepts intervenant dans chaque tâche, et de dégager des concepts communs, comme les concepts d'état, d'arbre d'états, ou de base de règles ;

– d'identifier les rôles de chaque concept dans l'activité de résolution de problème. Par exemple, les règles de production jouent un rôle dynamique, car elles influent sur le comportement du SBC, alors que l'arbre d'états joue plutôt un rôle statique de stockage d'informations consultables sur le raisonnement. Ces rôles peuvent se raffiner selon la tâche. Ainsi, les règles d'initialisation des paramètres des programmes en pilotage interviennent dans la sous-tâche d'exécution du pilotage ;

– de catégoriser les comportements de base et la connaissance utilisée par ces comportements. Par exemple, un comportement courant dans un SBC est le changement d'état. Ce comportement utilise principalement les concepts d'état, d'arbre d'états et de transition entre états ;

– d'identifier les types d'organisation de la connaissance nécessaires selon les tâches et des structures organisationnelles abstraites pour ces types. Par exemple, en classification, il est utile de disposer d'une hiérarchie de prototypes d'objets.

Enfin, un modèle du contrôle des tâches a permis d'identifier les connaissances de contrôle, afin de gérer l'enchaînement des comportements de base pour obtenir un comportement complexe.

Cette analyse a ensuite mené à la conception et à l'implantation de classes agencées en un *framework* permettant de construire des moteurs. La conception du *framework* s'est basée principalement sur des diagrammes de classes UML déduits de l'analyse. Pour certaines classes dont le comportement est suffisamment dynamique et intéressant (comme, par exemple, `Rule`), des scénarios et des diagrammes d'états et d'activité ont aussi été décrits.

3.2. *Résultats de l'analyse*

Nous avons ainsi pu définir un ensemble (une bibliothèque) de classes qui représentent les concepts importants et un ensemble de comportements de base (ou *instruction de raisonnement*) traduisant – généralement par des méthodes associées aux classes – des étapes assez élémentaires de la résolution de problème dans des moteurs de SBC. Les collaborations nécessaires entre ces classes ont aussi été mises à jour. Ces classes sont des généralisations de concepts similaires dans les GSBC initiaux, avec éventuellement des paramétrages, des découpages ou la mise en place de classes intermédiaires pour le partage de comportements.

Dès le début, nous avons dégagé différentes parties dans la bibliothèque : des concepts généraux très abstraits, utiles dans de nombreux SBC, quels que soient la tâche et le domaine, et d'autres plus spécifiques de telle ou telle tâche, correspondant à des ontologies de tâche particulières[2].

À tous ces concepts sont associées des opérations, elles aussi générales ou spécifiques. Par exemple, l'appariement entre un motif et des faits ou le choix d'un élément dans un ensemble sont des opérations générales, alors que l'insertion d'un programme à exécuter dans un plan en cours de construction – en connectant les flots de données entre les programmes du plan – est une opération spécifique du pilotage de programmes.

2. Une ontologie de tâche [CHA 98] correspond à la terminologie utilisée pour cette tâche et sa résolution, quel que soit le domaine d'application (par exemple : cause, dysfonctionnement pour la tâche de diagnostic), les termes sont ensuite spécialisés selon les domaines d'application par les experts (par exemple, en médecine une douleur est une sorte de dysfonctionnement).

Pour construire un nouveau moteur, un simple catalogue de classes ne suffit pas. Pour implanter une méthode particulière de résolution de problème, le choix des classes utiles dans la bibliothèque n'est qu'une première étape. Il faut ensuite agencer correctement ces classes et les appels de leurs méthodes pour obtenir le comportement souhaité. Or, les concepts sont en interaction dans l'ontologie et ces interactions peuvent aussi se retrouver dans les classes de la bibliothèque. Ainsi au niveau général, il existe des interactions entre un état et les attributs des objets dont il mémorise les modifications. De même, en classification, il existe une relation entre les classes filles d'une même classe, qui doivent former un recouvrement de leur classe mère. De plus, lorsqu'on se limite à des tâches particulières, comme dans notre cas, une partie du protocole de contrôle dépend des tâches et peut être fournie par le *framework*, mais ce protocole doit être adaptable en certains points à identifier soigneusement. Par exemple, dans les tâches de pilotage comme de classification, on retrouve la notion de graphes, sous forme de graphes d'opérateurs de pilotage et de taxonomie de classes, respectivement. Mais ce qui est fait au moment de la traversée d'un arc dans un graphe dépend de la tâche : décomposition d'un opérateur en sous-opérateurs en pilotage, ou affinement de la classe d'appartenance de l'objet à classer, en classification. Pour permettre ces adaptations, nous avons utilisé différentes techniques, comme des redéfinitions de méthodes (par exemple, la méthode `traverse` de la classe `Edge`, montrée figure 3, pour le cas précédent), des classes et des méthodes génériques, etc. ; ce point est traité plus en détail en section 5.2. En intégrant ces interactions et ces protocoles de contrôle dans la bibliothèque, nous avons abouti à un *framework*, qui permet de réutiliser aussi la conception abstraite d'un moteur. Le concepteur peut donc bénéficier via le *framework* d'un guide architectural pour la structuration de son code.

Les classes de BLOCKS sont à la fois opérationnelles et proches du modèle conceptuel de l'expert, car l'expert et le concepteur y retrouvent les concepts de l'ontologie de la tâche qui leur sont familiers (c'est en des termes identiques ou proches qu'ils communiquent). La correspondance entre le modèle conceptuel de l'expert et le SBC opérationnel est ainsi facilitée. Il est aussi plus simple de comprendre, modifier, maintenir et expliquer le comportement d'un SBC final.

L'approche utilisée s'apparente à celle des *component frameworks* en génie logiciel. Comme décrit dans [NIE 95a], cette approche sépare l'ingénierie des composants, c'est-à-dire l'évolution du *framework*, du développement des applications (les GSBC dans notre cas) à partir des composants. Les composants sont conçus à partir de l'existant (dans notre cas, les modèles de certaines tâches et des générateurs de SBC) et affinés au fur et à mesure des nouveaux besoins applicatifs.

Les GSBC étant des logiciels complexes qui font intervenir différents outils, faciliter la construction de générateurs signifie faciliter la construction de moteurs, d'outils de vérification ou d'apprentissage, d'interfaces d'édition, etc. Donc, même si le moteur est un élément important, il n'est pas seul. Chacun des autres outils peut, lui aussi, potentiellement nécessiter un *framework* ; néanmoins ces autres outils dépendent beaucoup de la stratégie de raisonnement adoptée par le moteur. D'où l'idée de regrouper tous ces outils dans une même plate-forme logicielle et de commencer par les *frameworks* nécessaires à la conception des moteurs.

4. Une plate-forme logicielle pour les concepteurs et les experts

La plate-forme logicielle LAMA [MOI 98] fournit un environnement unifié pour construire non seulement des bases de connaissances expertes, mais aussi des variantes de moteurs et des outils annexes. En cela, LAMA est comparable aux environnements de programmation en génie logiciel. La plate-forme offre des outils communs que dif-

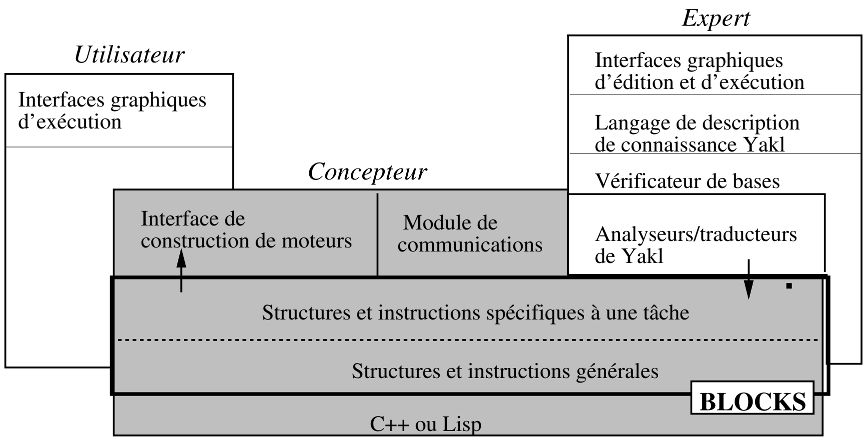

Figure 2. *Architecture de* LAMA *et outils offerts aux concepteurs, aux experts et aux utilisateurs finaux*

férents générateurs peuvent partager et que le concepteur peut adapter à la tâche de l'expert (voir figure 2). Ces outils sont complémentaires mais indépendants, de façon à pouvoir modifier, voire ajouter ou retirer un outil sans modifier le reste. Le concepteur a accès au *framework* BLOCKS fournissant des composants réutilisables pour implanter des moteurs, ainsi qu'à des outils de construction de moteurs et à un module de communication. L'expert qui écrit des bases de connaissances a accès au langage de description de connaissances YAKL, à ses traducteurs, à des outils de vérification de base de connaissances et aux interfaces graphiques associées. Enfin, l'utilisateur final qui soumet un problème au SBC final a accès à des interfaces graphiques d'exécution. Des outils spécifiques supplémentaires peuvent être ajoutés pour les utilisateurs, mais ils ne font pas partie de la plate-forme. Les outils de vérification, le langage YAKL et les interfaces graphiques sont adaptables par le concepteur, pour se rapprocher du modèle conceptuel de l'expert et des mécanismes de raisonnement du moteur. La plate-forme permet de construire différents générateurs (en restant dans la gamme définie en section 2), qui, chacun à son tour, permettra de construire des SBC pour des applications différentes.

La plate-forme regroupe des *frameworks* adaptés à la mise au point des outils nécessaires pour la construction des SBC. Actuellement, notre effort porte sur trois types de *frameworks* : pour les moteurs, pour les interfaces de construction et d'édition de bases de connaissances et pour les outils de vérification de bases de connaissances [MAR 99a]. Les *frameworks* peuvent partager certaines classes, mais ne les exploitent pas de la même façon. Par exemple, la vérification s'appuie systématiquement sur les interactions entre concepts afin d'assurer que des interactions définies dans le modèle se retrouvent bien dans les SBC où existent les instanciations concrètes des classes correspondantes. Le moteur, quant à lui, exploite les relations de façon plus dynamique et opportuniste, au fur et à mesure du raisonnement.

Le principal outil que le concepteur peut avoir à remanier est le moteur du générateur de SBC, qui implante une stratégie de raisonnement particulière conditionnant souvent les autres outils (interfaces, vérificateurs, etc.). C'est pourquoi nos efforts se sont d'abord portés sur le *framework* BLOCKS.

5. Blocks et le concepteur

BLOCKS (Basic Library Of Components for Knowledge-based Systems) est destiné au concepteur afin de faciliter son travail de construction, de modification ou de mise au point de moteurs de SBC. C'est une bibliothèque de classes, donc orientée-objet, qui offre des composants adaptables et réutilisables. Elle contient des structures élémentaires (classes génériques) et des instructions de raisonnement. Les classes représentent les concepts généraux des SBC et les concepts intervenant dans des tâches spécifiques. Les instructions sont des fonctions ou des méthodes attachées à ces classes, dont les combinaisons permettent de décrire différents mécanismes de raisonnement. Ces instructions correspondent à des étapes de raisonnement, souvent abstraites (elles sont alors génériques et/ou paramétrables par des critères). Elles ont pour objectif d'être suffisamment primitives, d'une part, et abstraites, d'autre part, pour implanter différentes stratégies de moteurs, tout en restant à un niveau d'abstraction facile à appréhender pour le concepteur. Elles correspondent soit à des besoins généraux comme des sélections, des tris, des appariements, soit à des besoins plus spécifiques pour une tâche, comme par exemple, en pilotage, l'évaluation de la correction d'un résultat ou l'exécution d'un opérateur de pilotage.

Les classes et les instructions constituent l'interface que le concepteur peut réutiliser ou adapter. Ceci permet une bonne séparation entre le niveau de description d'un moteur et le niveau du langage de programmation sous-jacent. BLOCKS a été conçu en partant des modèles de classes et des modèles dynamiques en UML issus de l'analyse. Le passage entre les modèles UML et le code de la bibliothèque a été réalisé sans utiliser de générateur automatique de code. Cependant, la procédure suivie a été comparable et aussi systématique. Les classes implantées dans le framework correspondent exactement à celles modélisées en UML, avec les mêmes attributs et méthodes. L'implantation est donc conforme à la modélisation en terme de structures de classes et également en terme de comportement, pour les classes pour lesquelles un modèle dynamique a été décrit. L'implantation de chaque composant peut être optimisée selon le langage-cible ; les langages ciblés sont actuellement C++ [ROG 97] pour sa large utilisation et sa portabilité, et Lisp, pour ses facilités de maquettage. D'une bibliothèque, nous sommes ainsi passés à un *framework* orienté-objet [ROB 96] qui fournit un ensemble de classes, les interactions entre ces classes et un protocole de contrôle.

5.1. *Les frameworks*

En fait, conformément aux résultats de l'analyse du domaine décrite section 3, BLOCKS est divisé en différents *frameworks*, au nombre de trois à l'heure actuelle. Le premier est le *framework* support qui sert de fondation aux autres. Il se compose :

– d'un système de *frames* comprenant les classes `Frame` et `Slot` principalement. Les frames sont des structures de données qui décrivent des prototypes d'objets ou de situations à l'aide d'une collection d'attributs (ou « slots »). Les slots sont à leur tour décrits grâce à d'autres attributs appelés facettes. Les facettes les plus courantes sont *valeur, valeur par défaut* et *domaine de validité*, ainsi que divers démons, de calcul de valeur par exemple. Les frames peuvent être organisés en hiérarchies, comme les classes ;

– d'un système de règles de production (dont les classes `Rule` et `RuleBase`) qui sont des unités granulaires de connaissance qui s'utilisent dans le cas où l'on doit modéliser des règles de décision ou un savoir-faire ;

– d'un système de gestion d'états, qui sert à représenter l'évolution du raisonnement du moteur, travaillant sur les classes `State`, `StateDag` et `Transition`, et leurs méthodes.

– d'un gestionnaire de bases de faits et de bases de connaissances, avec les classes `KnowledgeBase` et `FactBase` ;

– de classes utilitaires, comme `Set`, `Stack`, etc.

Ensuite, deux *frameworks* sont spécialisés pour deux tâches particulières. Le *framework* dédié au pilotage de code est le plus avancé dans sa conception et dans sa réalisation. Il comporte les classes `SupervisionOperator` (dérivée de `Frame`), `Goal`, `Argument`, etc. Il existe aussi une ébauche de *framework* dédié à l'interprétation, et plus précisément à la classification, avec les classes `Prototype` (dérivée de `Frame`) ou `FuzzySlot` (dérivée de `Slot`).

Ces différents *frameworks* sont détaillés ci-après.

5.1.1. *Le framework support*

Ce *framework* introduit des classes abstraites, correspondant à des concepts communs à la plupart des SBC, quelle que soit la tâche visée, ou à des organisations générales de ces concepts. La figure 3 montre la hiérarchie de ses principales classes. Le contrôle commun de toutes les applications est factorisé par les interfaces de ces classes et leurs relations. Les concepts correspondent, par exemple, aux notions d'état,

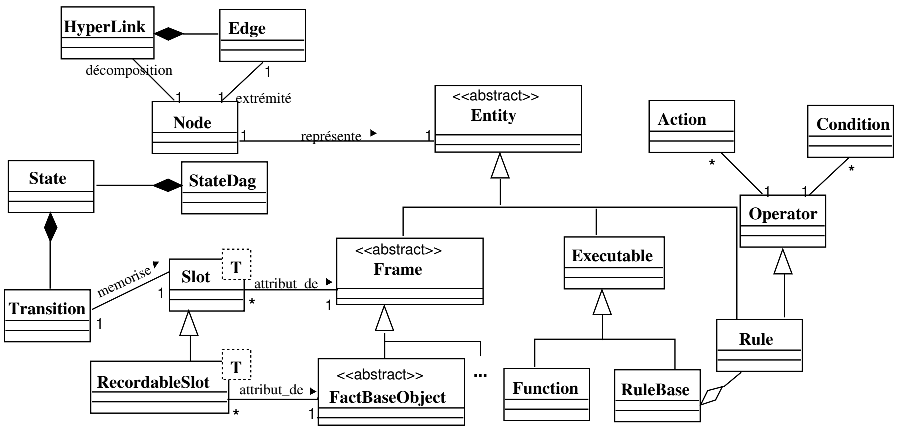

Figure 3. *Vue partielle des classes du framework support et de leurs relations, en notation UML. Au total, le framework support comporte une quarantaine de classes*

de règle de production ou de frame. Les organisations correspondent à des notions comme les hiérarchies ou les graphes de concepts. Les relations entre ces concepts sont assez simples et peu contraignantes. Par exemple, un état est en relation avec des objets du domaine ; l'état initial d'un arbre d'états correspond à la description des objets du domaine fournie par l'utilisateur final quand il pose son problème ; chaque état ensuite correspond aux modifications subies par ces objets pendant une étape de raisonnement du moteur.

Le *framework* support fournit une implantation par défaut de comportements simples, qui est modifiable et extensible. Ce sont les instructions générales associées aux concepts correspondant aux classes abstraites de haut niveau, comme `Frame` ou `Slot`. Par exemple, les accès aux slots en lecture/écriture font, en plus de l'affectation ou de la lecture usuelles, des vérifications d'appartenance au domaine de validité, s'il existe, ou un éventuel appel à un démon de calcul de valeur. Ce flot de contrôle est commun à tous les moteurs, mais il peut se redéfinir. Par exemple, le protocole d'accès aux slots permet la lecture de la valeur, l'écriture de la valeur, la lecture de la valeur par défaut et l'appel du démon. L'instruction `get_value` (accesseur en lecture) peut se redéfinir pour utiliser ce protocole dans un autre ordre que celui fourni par défaut.

De même, le concepteur peut enrichir ce protocole dans des sous-classes, par exemple l'accesseur en écriture (`set_value`) des slots peut aussi mémoriser le changement intervenu sur leur valeur, pour permettre des retours en arrière ; c'est le cas pour les slots de la sous-classe de `Frame` : `FactBaseObject`. Notons que la classe `FactBaseObject` reste abstraite (comme presque toutes celles du *framework* support) car les classes concrètes correspondront à un domaine d'application (comme `Voiture` dans la figure 4) et seront définies par l'expert, de manière transparente, comme sous-classes de `FactBaseObject`.

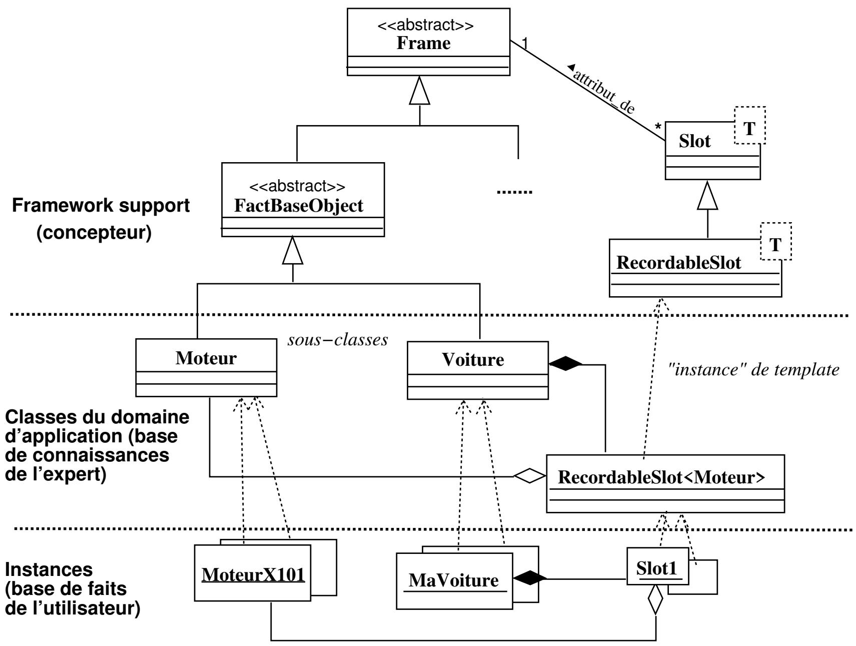

Figure 4. *Exemple d'utilisation du framework support pour la représentation des connaissances*

5.1.2. *Les frameworks spécifiques*

Actuellement, nous travaillons sur deux tâches : le pilotage de programmes et la classification. Les *framework*s spécifiques sont issus de l'analyse des modèles de ces tâches, à partir des outils existants. Les *framework*s offrent des composants réutilisables et adaptables, dans le cadre de l'ontologie de la tâche visée, qui servent de patrons pour définir des ontologies plus précises selon les besoins des experts et les domaines (comme le traitement d'images par exemple).

Ces *framework*s spécifiques enrichissent les concepts du *framework* support (par exemple, les hiérarchies de concepts deviennent des taxonomies de prototypes d'objets pour la classification). Les classes des *framework*s représentent les composants « naturels » des tâches de résolution de problème ainsi que leurs interactions. Ce sont souvent des spécialisations de classes du *framework* support comme `Frame` et `Rule`. En pilotage, par exemple, un concept important est celui d'opérateur qui représente les programmes, ou leurs agencements ; il correspond à la classe `SupervisionOperator` dérivée de `Frame`. Les interactions entre concepts sont renforcées, grâce à des mécanismes comme l'héritage, la composition ou le polymorphisme. La modélisation est plus strictement contrainte par la tâche. Ainsi, la classe `SupervisionOperator` a des arguments, qui sont des instances d'une sous-classe de `Slot`, possédant des attributs supplémentaires, comme un jugement porté sur la valeur de l'argument et un protocole d'accès plus riche (par exemple pour les accès au jugement).

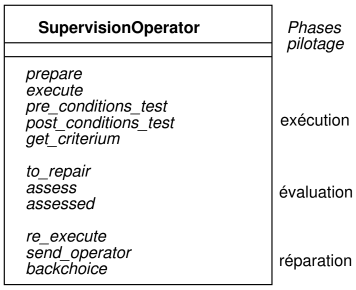

Figure 5. *La classe* `SupervisionOperator` *et ses méthodes significatives, qui sont des instructions de raisonnement pour un concepteur de moteurs de pilotage. Elles sont regroupées selon la phase du pilotage dans laquelle elles interviennent (voir figure 6 un exemple d'enchaînement de ces phases)*

La classe `SupervisionOperator` (figure 5) est l'abstraction-clef du *framework* de pilotage. Le protocole abstrait a pour rôle de permettre la construction de la plupart des stratégies de pilotage. Le pilotage consiste en un enchaînement de différentes phases : sélection d'un opérateur, exécution (ou décomposition) de cet opérateur, évaluation de ses résultats et réparation en cas de mauvais résultat. Les méthodes de la classe `SupervisionOperator` couvrent ces phases et permettent d'en implanter des variantes. Par exemple, pour faire des réparations plus ou moins sophistiquées, on peut se contenter de revenir au dernier point de choix chronologique où existait une alternative entre plusieurs opérateurs et refaire le choix (c'est l'option par défaut de la méthode `backchoice`), ou mémoriser en plus les choix déjà réalisés et les erreurs auxquelles ils ont conduit, ou encore revenir sur un point de choix précédent qui n'est pas forcément le dernier mais qui correspond à l'endroit où a pu apparaître le problème détecté. De telles variantes se réalisent par redéfinition de la méthode `backchoice`.

5.2. *Flexibilité*

Différents niveaux de flexibilité sont offerts au concepteur pour adapter une stratégie de moteur. Il peut tout d'abord utiliser le protocole fourni de différentes manières, ce qui aboutit à des variantes de moteurs. Par exemple, en pilotage, il peut décrire un moteur qui effectue un changement d'état (en utilisant l'instruction `change_state` qui fait partie du protocole du système de gestion d'états du *framework* support) après chaque exécution, ou seulement après les choix, ou même pas du tout.

BLOCKS offre des possibilités de réutilisabilité et d'extension au niveau des classes. Premièrement, les classes peuvent être réutilisées telles quelles. Pour des raisons de cohérence, les classes fournies ne peuvent pas être modifiées par un concepteur, mais elles peuvent être dérivées ou composées entre elles. Les classes abstraites, comme `FactBaseObject`, ou les classes génériques, comme `Slot`, permettent aussi de définir des classes du domaine par spécialisation et instanciation (voir un exemple figure 4). De plus, la définition dans des sous-classes de méthodes abstraites (*i.e.* qui n'ont pas de définition dans leur classe de base), peut permettre d'adapter le contrôle, de même que la redéfinition de méthodes génériques (voir un exemple en section 6.2). BLOCKS offre aussi des possibilités de réutilisabilité et de flexibilité au niveau des instructions. Les instructions sont souvent des fonctions génériques qui s'affinent par « instanciation » (voir section 6.2). De plus, elles sont aussi souvent paramétrables par des critères (qui sont des fonctions à signature prédéfinie). Cela permet d'affiner des stratégies de contrôle en modifiant les critères.

Toutes ces capacités d'adaptation peuvent être gérées soit manuellement par le concepteur, qui écrit directement le code correspondant, soit automatiquement par un méta-protocole objet [KIC 91] qui offre des mécanismes de haut niveau pour la transformation de code source. C'est une forme de pré-traitement intelligent d'un code qui permet d'adapter le comportement (et l'implantation) d'un langage à objets, en modifiant des mécanismes du langage, tels que l'appel de méthode, ce qui modifie le comportement des objets à l'exécution. Nous avons nous-même utilisé ce moyen dans le système de frames du *framework* support [CAV 98], en utilisant OpenC++ [CHI 95] afin de donner à des classes C++ un comportement de frames (avec slots, démons automatiques, etc.). Pour cela, une métaclasse `FrameClass` a été définie. Elle correspond au concept de frame en intelligence artificielle. La classe `Frame` et toutes ses classes dérivées sont des instances de cette métaclasse. La super-classe `Frame` du *framework* support fournit l'interface commune aux frames, qui sera héritée par ses classes dérivées. La métaclasse `FrameClass` impose la structure (par exemple, une frame est composée de slots) et le respect du protocole d'utilisation des frames (elle crée les constructeurs nécessaires en C++, par exemple). Elle ajoute à toutes ses instances (*i.e.* toutes les sous-classes de `Frame`) lors de leur création, ce qui est nécessaire pour donner aux instances de ces classes un comportement de frame.

5.3. *Avantages*

Pour le concepteur, grâce au niveau d'abstraction fourni par BLOCKS, la conformité entre l'implantation et le modèle conceptuel est facilitée. Les mêmes concepts abstraits et les mêmes instructions conceptuelles existent quel que soit le langage d'implantation ciblé. Le concepteur ne doit programmer que des extensions ou des affinements de composants de la bibliothèque. De plus, le choix d'une implantation utilisant un langage à objets (quel qu'il soit) offre un bon compromis entre l'efficacité d'exécution et une explicitation aisée de la connaissance.

Le problème le plus difficile est de déterminer la granularité de composants. Les composants doivent être de taille suffisante pour réaliser une fonctionnalité intéressante, mais assez petits pour que le concepteur puisse finement exprimer des variantes de comportements de moteurs. Une fois ces instructions munies d'une sémantique claire, le concepteur dispose d'un moyen d'expression proche du modèle de la tâche à implanter et adapté au niveau d'abstraction auquel s'expriment ses besoins. Cette approche permet une vision unifiée de différents moteurs et peut être une base commode de comparaison entre stratégies de résolution de problèmes. De plus, la construction d'un nouveau logiciel, à partir du *framework*, est plus rapide, plus facile et plus fiable que son développement à partir de zéro.

Enfin, pour l'expert, il n'y a pas de rupture non plus dans son processus de création de base de connaissances, car le language YAKL, adaptable par le concepteur à une tâche donnée, sert à la fois de langage de modélisation conceptuelle, de langage de description de base de connaissances et de support pour la vérification de la connaissance et la génération de code. L'expert n'a donc pas à entrer dans l'implantation.

La conception de BLOCKS s'est faite en plusieurs itérations, et a donné lieu à différentes réalisations de GSBC de pilotage qui sont détaillées ci-après.

6. Réalisations

La conception des *frameworks* s'est appuyée sur des GSBC initiaux, qui ont servi à la première étape d'abstraction des classes. Afin de valider la conception de BLOCKS, nous avons appliqué ses *frameworks* en cherchant tout d'abord à retrouver le comportement de GSBC pré-existants (« round-trip re-engineering »), puis à en obtenir des variantes et enfin à concevoir des moteurs aux comportements plus éloignés.

En particulier, pour la tâche de pilotage, nous avons modélisé et conçu trois moteurs notablement différents dans leur stratégie de construction d'un enchaînement de programmes. L'un de ces moteurs correspondait à l'un de nos trois GSBC de départ, les autres en sont plus ou moins éloignés. Ces moteurs ont été des tests pour les *frameworks* mais ils ont participé aussi à leur enrichissement. Ces essais ont permis de mesurer les capacités de réutilisation et d'adaptation des composants pour des moteurs différents ou des variantes du même moteur.

6.1. *Une première application :* Pegase

Nous sommes partis du GSBC OCAPI en pilotage de programmes [CLé 93] et d'une de ses variantes, pour construire le *framework* spécialisé pour cette tâche. Grâce à ce *framework* nous avons pu réaliser les fonctionnalités d'OCAPI, puis les enrichir, ce qui a donné PEGASE [VIN 97].

Dans PEGASE, comme dans OCAPI, la base de connaissances contient un seul graphe d'opérateurs, qui décrit toutes les combinaisons possibles, à partir d'un ensemble de programmes particulier. La résolution de problème consiste à trouver le meilleur chemin dans ce graphe, avec des retours en arrière possibles sur des choix. L'algorithme simplifié est décrit figure 6.

Les enrichissements par rapport à OCAPI ont porté sur les mécanismes de réparation en cas de mauvais résultats après exécution d'un programme. Des variantes

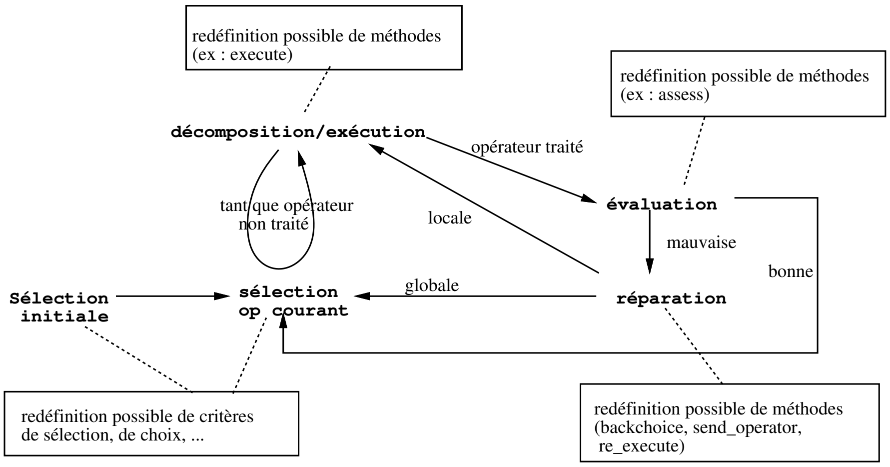

Figure 6. *Algorithme simplifié de* PEGASE*, avec les points de flexibilité possibles (indiqués par des rectangles) selon les différentes phases du pilotage identifiées figure 5*

simples de la stratégie du moteur ont aussi été facilement développées. OCAPI avait nécessité plusieurs années de développement. C'était un code d'environ 20 000 lignes de Lisp (hors interface graphique), peu lisible et donc difficilement modifiable. La première itération de BLOCKS, ainsi que la version 0 de PEGASE reproduisant le comportement d'OCAPI a pris environ deux ans. Le code de ce moteur (hors interface graphique) tient en deux pages ; il est écrit à un plus haut niveau d'abstraction et facile à comprendre. Ensuite, les variantes n'ont pris que quelques jours à quelques semaines de développement. La réutilisabilité offerte par BLOCKS a permis de diviser par un facteur quatre le temps de réalisation de variantes à partir d'un moteur initial. Outre l'avantage de la rapidité de réalisation, le *framework* définit la structure globale et le contrôle d'un moteur de pilotage générique. Le partage de structure et de contrôle ainsi réalisé simplifie la maintenance des moteurs et fait mieux ressortir les particularités de chacun. Ce dernier point facilite la comparaison de moteurs pour vérifier leur adéquation aux besoins d'un domaine. Diverses bases de connaissances ont été développées pour le moteur PEGASE [THO 99, SHE 99] et il a été facile d'adapter ces bases aux variantes grâce à l'architecture et aux structures communes fournies par BLOCKS.

6.2. *Extensions et adaptation pour un autre moteur :* **M*edIA***

Dans le cadre du pilotage de programmes pour l'imagerie médicale, les besoins identifiés ont fait apparaître des différences assez grandes dans la stratégie de pilotage avec ce dont PEGASE était capable. Ces besoins ont donné lieu à un nouveau GSBC MEDIA [CRU 99], dont le moteur a nécessité des adaptations et des extensions, qui sont résumées dans la figure 7. Pour la réalisation de MEDIA, la phase d'analyse des besoins et de conception des extensions à réaliser a été la plus longue. La réalisation à l'aide des *frameworks* s'est faite en deux mois environ et a réutilisé plus de 90 % des composants utilisés dans PEGASE. L'algorithme du moteur diffère de celui de PEGASE pour prendre en compte l'existence d'étapes « abstraites », qui correspondent non pas à un opérateur connu, mais à une fonctionnalité abstraite de traitement à atteindre. Pour une étape abstraite, le moteur doit réaliser une mise en correspondance supplémentaire afin de trouver les opérateurs concrets qui peuvent la résoudre.

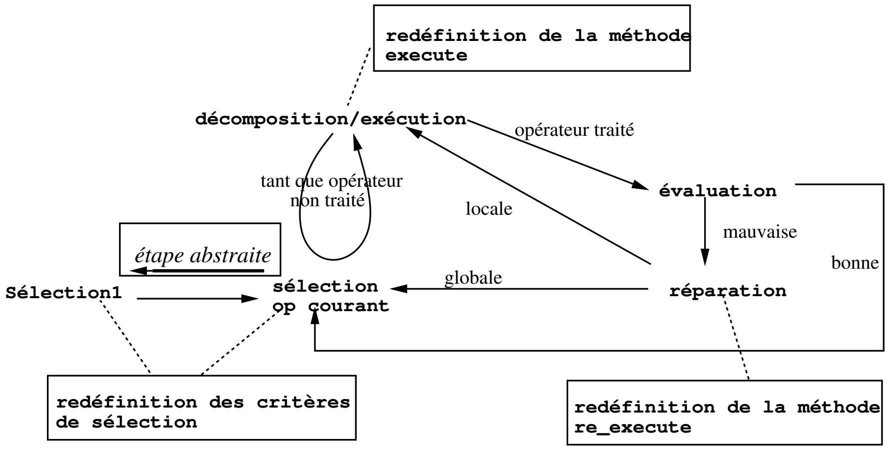

Figure 7. *Algorithme simplifié de* MEDIA *avec les points de flexibilité utilisés et les principales modifications effectuées (dans les rectangles)*

Voici quelques illustrations qui montrent comment les composants ont été réutilisés, étendus ou modifiés, pour la réalisation de MEDIA.

Concernant les critères passés en paramètres des instructions générales (de sélection, de tri, de choix, etc.), un critère spécifique a été écrit pour la sélection d'opérateurs concrets pour résoudre une étape abstraite. D'autres critères ont aussi été écrits pour enrichir l'étape de test des préconditions d'un opérateur, par exemple. Enfin, le critère de sélection initiale de PEGASE a été étendu (voir la figure 7). Pour implanter cette étape, il existe dans BLOCKS la fonction générique `select`, paramétrable par un critère. Cette fonction prend en entrée un ensemble d'objets de type X et un critère booléen sur des objets de type X, et retourne le sous-ensemble des objets de l'ensemble d'entrée qui vérifient le critère. Ce critère est une des heuristiques qui caractérisent un moteur. Pour MEDIA, le concepteur a tout d'abord écrit un critère spécifique à ce moteur pour sélectionner des opérateurs initiaux (X = `SupervisionOperator`). L'étape de sélection initiale appelle `select` avec l'ensemble des opérateurs de la base de connaissances courante et le critère précédent comme arguments.

Concernant les classes, une extension par sous-typage de classes a été nécessaire pour implanter des conditions « faibles », c'est-à-dire non systématiquement bloquantes, par dérivation de la classe `Condition`. La plupart des autres modifications ont consisté en des redéfinitions de méthodes, comme la méthode `re_execute` de la classe `SupervisionOperator`.

MEDIA est un exemple typique de moteur construit par comparaison et réutilisation de parties d'un autre moteur. Cette expérience nous a permis d'ajuster la granularité des composants de BLOCKS afin de pouvoir réutiliser ou modifier facilement les étapes importantes des moteurs.

6.3. *Utilisation de* BLOCKS

Les développements de moteurs ont jusqu'à présent été réalisés par des membres de notre équipe, ce qui nous a permis de valider et de faire évoluer les outils (dont BLOCKS). Mais, pour un concepteur extérieur, il est prévu que la plate-forme LAMA fournisse d'autres outils que BLOCKS pour construire un moteur (voir figure 2). Ces outils sont en cours de développement. Leur objectif est de permettre, d'une part, de décrire une stratégie de raisonnement de moteur à un niveau conceptuel, dans la lignée des outils existant en intelligence artificielle et, d'autre part, de faire la connexion avec l'implantation du moteur via le *framework* (nous envisageons aussi de faire certaines preuves de comportement du programme du moteur). Le premier type d'outil produira une description de la stratégie d'un moteur qui servira de spécification pour le concepteur et de documentation de référence sur le fonctionnement du moteur pour les experts qui l'utiliseront ensuite pour construire des bases de connaissances. Actuellement cette description se fait soit manuellement, soit en utilisant des outils externes à la plate-forme. Le second type d'outil sera une aide à la programmation et à l'utilisation du *framework*. En complément d'une documentation méthodologique, des outils graphiques permettront l'exploration des composants et leur composition [NIE 95b].

7. Conclusions

Au cours de la définition et de la réalisation des *frameworks* nous avons fait des allers et retours entre différents rôles : fournisseur de *framework* et utilisateurs – c'est-à-dire concepteur et même experts et utilisateurs finaux du SBC. Nous avons tout d'abord expérimenté la facilité de réingénierie d'un GSBC existant, avant d'en implanter des variantes et de construire un moteur nouveau. Nous avons validé l'approche sur des applications réelles en réalisant cinq bases de connaissances sur des applications différentes pour valider PEGASE, puis deux pour MEDIA. Ceci nous a permis de prendre en compte les besoins des utilisateurs tout au long du processus de développement de SBC.

L'impact de l'utilisation de *frameworks* pour la construction de GSBC s'est révélé important en gain de temps et en diminution de la quantité de code à écrire, aussi bien pour créer de nouveaux moteurs, que pour en reconfigurer. De plus, c'est un outil précieux pour comparer des GSBC entre eux. La réutilisabilité s'applique donc bien aux GSBC, comme à d'autres logiciels. Cette approche nous a permis un transfert de contrôle des codes des GSBC vers la bibliothèque, ce qui réduit la quantité de code à écrire et autorise aussi l'ajout de fonctionnalités dans les *frameworks* sans modification des réalisations existantes. La lisibilité des codes des GSBC a été ainsi améliorée, car les codes sont plus courts (par exemple seulement 2 pages pour PEGASE) et plus ciblés sur les spécificités d'un moteur, puisqu'une grande partie du code des classes et des mécanismes de base sont pris en charge par les *frameworks*. D'autre part, BLOCKS utilise des termes qui font partie de l'ontologie de la tâche de pilotage pour nommer les classes (comme `SupervisionOperator`) et les méthodes (comme `execute`). BLOCKS en fournit une implantation extensible et réutilisable qui sera partagée par tous les moteurs. La lisibilité et la compréhension d'un code de moteur par un concepteur est ainsi facilitée, de même que le dialogue entre expert et concepteur, lors de la création ou de la modification d'un moteur.

La plate-forme a tout d'abord été maquettée en Lisp. Nous avons fait évoluer les versions récentes en C++ et Java, ce qui permet une meilleure diffusion et une meilleure utilisation de la plate-forme et des moteurs qui en sont issus.

Un autre moteur de pilotage, utilisant un mode de planification différent, ainsi qu'un moteur pour la classification, sont en cours de développement. D'autres exemples de moteurs pour des SBC en interprétation vont nous amener à étendre les *frameworks* existants, à en créer d'autres, ou à définir plus précisément les limites de chacun d'eux.

Remerciements

L'auteur tient à remercier tous les doctorants qui ont travaillé autour de la plate-forme LAMA [ELS 96, VIN 97, CRU 99, MAR 99a] ainsi que J-P. Rigault pour sa participation à ces travaux et ses conseils pour l'écriture de cet article.

8. Bibliographie

[BEC 94] BECK K., JOHNSON R., « Patterns Generate Architectures », TOROKO M., PARESCHI R., Eds., *European Conference on Object-Oriented Programming, ECOOP'94*, vol. 821 de *LNCS*, Springer-Verlag, July 1994, p. 139–149.

[BOS 99] BOSCH J., MOLIN P., MATTSSON M.AND BENGTSSON P., FAYAD M. E., « Object-Oriented Frameworks: Problems & Experiences », M. FAYAD D. SCHMIDT R. J., Ed., *Building Application Frameworks: Object Oriented Foundations of Framework Design*, Chapitre 3, Wiley & Sons, 1999.

[BRI 99] BRIOT J.-P., « Actalk: A Framework for Object-Oriented Concurrent Programming - Design and Experience », BAHSOUN J.-P., BABA T., BRIOT J.-P., YONEZAWA A., Eds., *Object-Based Parallel and Distributed Computing II - Proceedings of the 2nd OBPDC'97 France-Japan Workshop*, Hermès Science Publications, Paris, 1999.

[CAM 93] CAMPBELL R. H., ISLAM N., RAILA D., MADANY P., « Designing and Implementing Choices: An Object-Oriented System in C++. », *Communications of the ACM*, vol. 36, n° 9, 1993, p. 117–126.

[CAV 98] CAVARROC J., MOISAN S., RIGAULT J.-P., « Simplifying an Extensible Class Library Interface with OpenC++ », CENTER FOR COMPUTATIONAL PHYSICS U. O. T., Ed., *OOPSLA'98, Worshop on Reflective Programming in C++ and Java*, October 1998.

[CHA 93] CHANDRASEKARAN B., JOHNSON T., « Generic Tasks and Task Structures: History, Critique and New Directions », DAVID J., KRIVINE J., R.SIMONS, Eds., *Second Generation Expert Systems*, p. 232-272, Springer-Verlag, 1993.

[CHA 98] CHANDRASEKARAN B., JOSEPHSON J., BENJAMINS V., « Ontology of Tasks and Methods », *Workshop on Applications of Ontologies and Problem-Solving Methods, ECAI'98*, August 1998, p. 31-43.

[CHI 95] CHIBA S., « A Metaobject Protocol for C++ », *Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'95*, vol. 30 de *SIGPLAN Notices*, ACM Press, October 1995, p. 285–299.

[CLé 93] CLÉMENT V., THONNAT M., « A Knowledge-based Approach to the Integration of Image Processing Procedures », *Computer Vision, Graphics and Image*, vol. 57, n° 2, 1993, p. 166-184.

[CRU 99] CRUBÉZY M., « Pilotage de programmes pour le traitement d'images médicales », Thèse de Doctorat, Université de Nice-Sophia Antipolis, février 1999.

[DAL 99] DALHEIMER M. K., *Programming with Qt*, O'Reilly, 1999.

[ELS 96] VAN DEN ELST J., « Modélisation de Connaissances pour le Pilotage de Programmes de Traitement d'Images », Thèse de Doctorat, Université de Nice-Sophia Antipolis, octobre 1996.

[HüN 95] HÜNI H., JOHNSON R. E., ENGEL R., « A Framework for Network Protocol Software », *Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'95*, vol. 30 de *SIGPLAN Notices*, ACM Press, October 1995, p. 358–369.

[JOH 88] JOHNSON R., FOOTE B., « Designing reusable classes », *Journal of Object-Oriented Programming*, vol. 1, n° 2, 1988, p. 22–35.

[JOH 91] JOHNSON R., MCCONNELL C., « The RTL System: A Framework for Code Optimization. », Technical report, department of computer science, 1991, University of Illinois at Urbana-Champaign.

[KIC 91] KICZALES G., DE RIVIÈRE J., BOBROW D., *The Art of the Meta-Object Protocol*, MIT Press, 1991.

[KIN 91] KINKLER G., BHOLA C., DALLEMAGNE G., MARQUES D., MCDERMOTT J., « Usable and Reusable Programming Constructs », *Knowledge Acquisition*, vol. 3, 1991, p. 117–135.

[KRA 88] KRASNER G., POPE S., « A Cookbook for Using the Model-View Controller user Interface Paradigme in Smalltak-80 », *Journal of Object-Oriented Programming*, vol. 1, n° 3, 1988, p. 26–49.

[MAR 99a] MARCOS M., « Verification and Validation of Program Supervision Systems », PhD thesis, Universitat Jaume I de Castelló (Espagne), February 1999.

[MAR 99b] MARCOS M., MOISAN S., DEL POBIL A. P., « Knowledge Modeling of Program Supervision Task and its Application to Knowledge Base Verification », *Applied Intelligence*, vol. 10, n° 2/3, 1999, p. 181-192.

[MEY 93] DE MEY V., NIERSTRASZ O., « The ITHACA Application Development Environment », TSICHRITZIS D., Ed., *Visual Objects*, Centre Universitaire d'Informatique, University of Geneva, July 1993, p. 267–280.

[MOI 98] MOISAN S., « Une plate-forme pour une programmation par composants de systèmes à base de connaissances », Habilitation à diriger les recherches, Université de Nice, avril 1998.

[MUL 97] MULLER P.-A., *Modélisation Objet avec UML*, Eyrolles, 1997.

[MUS 93] MUSEN M. A., TU S. W., ERIKSSON H., GENNARI J. H., PUERTA A. R., « PROTEGE-II: An Environment for Reusable Problem-Solving Methods and Domain Ontologies », *IJCAI*, Chambéry, August 1993.

[NIE 95a] NIERSTRASZ O., « Langages et modèles à objets », NAPOLI A., Ed., *Research Topics in Software Composition*, INRIA, octobre 1995, p. 193-204.

[NIE 95b] NIERSTRASZ O., TSICHRITZIS D., *Object Oriented Software Composition*, Prentice Hall International, 1995.

[PRE 94] PREE W., « Meta Patterns–A Means for Capturing the Essentials of Reusable Object-Oriented Design », TOROKO M., PARESCHI R., Eds., *European Conference on Object-Oriented Programming, ECOOP'94*, vol. 821 de *LNCS*, Springer-Verlag, July 1994.

[ROB 96] ROBERTS D., JOHNSON R., « Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks », *Pattern Languages of Programs*, sept 1996.

[ROG 97] ROGERS G., *Framework-based Software Development in C++*, Prentice Hall, 1997.

[SCH 94] SCHREIBER G., WIELINGA B., DE HOOG R., AKKERMANS H., VAN DE VELDE W., « CommonKADS: A Comprehensive Methodology for KBS D evelopment », *IEEE Expert*, vol. 9, n° 6, 1994, p. 28-37.

[SHE 99] SHEKHAR C., MOISAN S., VINCENT R., BURLINA P., CHELLAPPA R., « Knowledge-based Control of Vision Systems », *Image and Vision Computing*, vol. 17, n° 8, 1999, p. 667-683.

[SOM 92] SOMMERVILLE I., « *Software Engineering* », Chapitre 28, Addison-Wesley, Fourth Edition, 1992.

[STE 91] STEELS L., « Esprit Project CONSTRUCT - COMMET: A Componential Methodology for Knowledge Engineering », rapport, November 1991, AI LAb, University of Brussels.

[SZY 98] SZYPERSKI C., *Component Software - Beyond Object-Oriented Programming*, Addison Wesley, 1998.

[TAL 96a] TALON X., PIERRET-GOLBREICH C., « TASK: a framework for the different steps of a KBS construction », *Workshop on Knowledge Engineering and Modelling LAnguage (KEML'96)*, 1996.

[TAL 96b] TALON X., PIERRET-GOLBREICH C., « TASK: from the specification to the implementation. », *8th IEEE International Conference on Tools with Artificial Intelligence*, IEEE Computer Society Press, 1996, p. 80–88.

[THO 99] THONNAT M., MOISAN S., CRUBÉZY M., « Experience in Integrating Image Processing Programs », *International Conference on Computer Vision Systems*, Las Palmas, Canary Islands, January 1999.

[TRI 99] TRICHET F., TCHOUNIKINE P., « DSTM: a Framework to Operationalize and Refine a Problem-Solving Method modeled in terms of Tasks and Methods », *International Journal of Expert Systems With Applications (ESWA)*, vol. 16, n° 2, 1999, p. 105-120, Elsevier Science.

[VIN 97] VINCENT R., « Etude des mécanismes de gestion des erreurs dans les systèmes à base de connaissances de pilotage de programmes. », Thèse de Doctorat, Université de Nice-Sophia Antipolis, mars 1997.

[WEI 88] WEINAND A., GAMMA E., MARTY R., « ET++ –an object oriented application framework in C++ », *Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'88*, vol. 23 de *SIGPLAN Notices*, ACM Press, November 1988.

[WIE 92] WIELINGA B., SCHREIBER A., BREUKER J., « KADS: a Modelling Approach to Knowledge Engineering », *Knowledge Acquisition*, vol. 4, 1992, p. 5-53.

Article reçu le 2 novembre 1999
Version révisée le 16 juin 2000
Rédacteur responsable : Sabine Moisan

Sabine Moisan *est chargée de recherches à l'Inria Sophia Antipolis depuis 1983. Ses travaux de recherches se situent dans le domaine de l'intelligence artificielle et du génie logiciel, plus particulièrement pour la conception de générateurs de systèmes à base de connaissances. Elle a participé au développement de divers générateurs, comme* SMECI*, qui a été commercialisé par la société Ilog, ou* OCAPI *et* PEGASE *dédiés au pilotage de programmes. Elle a soutenu son habilitation à diriger les recherches en 1998, sur la plate-forme* LAMA *et le framework* BLOCKS*. Elle est actuellement responsable de l'axe « pilotage de programmes » au sein de l'équipe Orion.*