# Connaissances pour compiler des connaissances
## Le système Shal

**Jean-Luc Dormoy**

*DER-EDF IMA-TIEM*
*1, avenue du Général De Gaulle, 92141 Clamart Cedex*

RESUME. *Cet article présente l'état d'avancement du système Shal, qui est une base de connaissances pour compiler une base de connaissances. Après avoir motivé notre approche basée sur le bootstrapping de connaissances nous décrivons les ensembles de connaissances présents dans Shal : analyse du comportement d'une base de connaissances, découverte et vérification des contraintes sémantiques sur la mémoire de travail, reconnaissance de patterns comportementaux de règles, synthèse de traducteurs de textes informatiques, utilisation et expression déclaratives du contrôle. Nous donnons des indications sur les connaissances en développement : connaissances de programmation, connaissances sur la gestion de Shal par lui-même, connaissances pour revenir sur des déductions.*
*Nous tirons ensuite quelques leçons de notre expérience de construction d'un ensemble de connaissances qui doit être capable de s'utiliser lui-même.*

### 1. Pourquoi et comment compiler des règles ?

Le fait qu'un numéro spécial de RIA soit consacré à la compilation de règles prouve que des motivations suffisamment fortes poussent des chercheurs à explorer ce domaine. Pour quelles raisons ?

Tout d'abord, il y a un but d'efficacité. Les règles de production sont assez largement répandues comme support de la connaissance dans les systèmes experts, souvent de manière mixée à d'autres langages de représentation des connaissances (langages à objets, tableaux noirs, …). Il est donc normal de vouloir fournir à l'utilisateur des performances les meilleures possibles, par exemple pour les applications en temps réel. Associer les règles de production à d'autres langages ou moyens de représentation des connaissances peut entraîner une refonte des méthodes de compilation de règles. Ensuite, le problème de la compilation de règles apparaît

souvent comme étant lié à d'autres problèmes, par exemple la validation ou la vérification de bases de connaissances. On peut également envisager de revoir le problème de la compilation des règles sous le jour de moyens de calcul nouveaux, comme les ordinateurs parallèles. Enfin, une motivation implicite du chercheur est qu'il s'agit d'un problème difficile, et donc intéressant…

Etant ainsi motivé, le chercheur se demande comment répondre à ce problème. L'idée de la compilation est naturelle lorsqu'il s'agit d'efficacité. En effet, la conception selon laquelle un programme compilé est plus efficace qu'un programme interprété est largement répandue. Se référant à sa culture, le chercheur va donc se demander quel compilateur écrire, et comment. Le compilateur doit être un programme, qui remplace l'interpréteur communément appelé moteur d'inférence, et consister en la mise en œuvre d'un algorithme, qu'il s'agit de découvrir. A ce point de son raisonnement, le chercheur va, soit tenter d'améliorer des algorithmes existants, soit de trouver de nouvelles idées par analogie avec d'autres domaines.

Après être passé par ces divers stades de motivation, notre but actuel est différent. Comme on peut s'y attendre, les moyens d'y parvenir sont également différents.

Notre but à long terme est de fournir à la machine des méta-connaissances de divers ordres. Ces méta-connaissances sont bien des connaissances, elles ne diffèrent des connaissances "habituelles" que par le fait que leur domaine d'application est la connaissance. Parmi ces méta-connaissances, il y a les connaissances qui permettent d'utiliser des connaissances fournies par l'homme sur le support matériel dont nous disposons aujourd'hui, l'ordinateur[1]. Cela nécessite de disposer d'un langage, où les connaissances sont exprimées, et d'être capable de traiter ces connaissances, c'est-à-dire de remplir le fossé entre ce langage plus accessible à l'homme, et celui que l'ordinateur est capable de comprendre : des ordres élémentaires (instructions) exécutés en séquences. Ce sont les connaissances jetant un pont par-dessus ce fossé que nous appelons connaissances de compilation.

Il est bien évident que toute possibilité d'expression et d'utilisation de connaissances dépend de l'existence de ces connaissances de compilation. Il faut donc commencer par là. C'est pourquoi, dans la première phase de construction de notre système, nous nous intéressons avant tout à ces connaissances de compilation, et à celles qui peuvent faciliter leur expression et leur utilisation.

Par ailleurs, il faut bien choisir un langage de départ. Pour des raisons diverses, nous avons opté pour les règles de production. Non qu'il serait impossible de se lancer dans un projet similaire en partant d'un autre langage. En fait, nous connaissons assez bien l'utilisation des règles de production pour avoir déjà réalisé un moteur d'inférence. Par ailleurs, nous estimons que les langages à règles de production, même s'il est actuellement convenu de les décrier - avec parfois de bonnes raisons -, restent un des meilleurs outils[2] d'expression des connaissances. Enfin, le fait de

---

[1] Nous travaillons actuellement sur une machine séquentielle, un SPARC-2.

[2] Faute de mieux

disposer d'un moteur d'inférence bien connu est un avantage, puisque c'est lui qui va permettre d'interpréter les premières connaissances de compilation.

Notre but est donc radicalement différent de ceux exprimés plus haut (et peut-être indûment attribués à nos collègues). Nous ne cherchons pas à améliorer l'efficacité d'un langage de représentation des connaissances. Nous cherchons le moyen pratique de fournir à la machine de plus en plus de connaissances, en nous focalisant sur les connaissances d'utilisation des connaissances. Cependant, cette recherche est opportuniste. Nous partons d'un langage, les règles, mais sans nous interdire de le modifier si cela s'avère payant. Bien évidemment, l'efficacité joue un rôle, mais intervient ici non comme un concours de vitesse, mais comme un butoir. Les connaissances de compilation vont devoir être exécutées pour compiler des connaissances, et il est clair qu'un "programme" de la taille d'un compilateur écrit sous la forme d'une base de connaissances est gourmand en ressources. Nous le montrons dans la suite de cet article, un des obstacles majeurs à notre approche tient justement dans cette inefficacité : le compilateur-base-de-connaissances s'est avéré dans nos premières tentatives simplement incapable de compiler avec des ressources raisonnables (et même grandes vu les ordinateurs actuels) des bases de connaissances substantielles. Par ailleurs, l'écriture de grandes bases de connaissances, ce qui est notre cas, pose de nombreux problèmes de vérification, ou, dit plus simplement, d'erreurs. Donner des connaissances autres que celles strictement liées à la compilation peut donc s'avérer souhaitable, simplement pour faciliter leur acquisition.

Les moyens par lesquels nous entendons atteindre notre but sont également très différents de l'approche de "l'algorithme amélioré". Pour nous, l'utilisation des connaissances relève avant tout de la connaissance. La différence est énorme. L'approche "algorithmique" consiste à concevoir un programme général, qui réponde correctement à tous les cas pouvant se présenter. Sa mise en œuvre est extrêmement contrainte par les difficultés que nous avons à programmer. Il devra donc être "simple", c'est-à-dire compact, même s'il est astucieux, difficile à comprendre ou à prouver, et encore plus à concevoir. Devant donc avec peu de "mots" et de "concepts" couvrir une large gammes de possibilités, il sera incapable de tirer avantage des particularités des cas qu'il a à traiter. Par ailleurs, son extension sera très difficile. Les cas sont rares où il est possible de faire une modification majeure dans un programme déjà écrit. Tous ces problèmes sont bien connus en génie logiciel, et sont curieusement ignorés par les concepteurs d'applications informatiques ayant pour domaine d'application l'informatique elle-même. Enfin, et c'est là la différence majeure avec notre approche, ces algorithmes sont l'aboutissement et la forme condensée d'un grand nombre de connaissances, qui sont les connaissances "informatiques" de leurs concepteurs. Mais ces algorithmes ne constituent pas une représentation de ces connaissances et sont de manière évidente incapables d'appliquer les connaissances dont ils sont le résultat.

Notre approche pourrait tenir dans une formule simple : "Dès que l'on sait quelque chose, il faut l'utiliser". Le but de fournir des connaissances de compilation (et d'autres) à la machine en les représentant de manière aussi explicite que possible est justement d'aider à leur utilisation, et à l'acquisition de connaissances meilleures ou plus générales. Notre but est que, au fur et à mesure que notre système saura mieux

programmer, nous serons déchargés des détails de la programmation, et à terme complètement émancipés de ce goulet d'étranglement majeur. En définitive, notre approche est le *contraire* de l'approche algorithmique : nous ne cherchons pas à écrire un nouveau programme, nous cherchons *à ne plus en écrire du tout*.

## 2. Connaissances de compilation : vue d'ensemble

### 2.1. Les connaissances de compilation et leur acquisition

Outre le but lointain d'une base de méta-connaissances générale - dont nous sommes conscient qu'il ne pourra être réalisé par un individu seul, et aussi qu'il peut porter à discussion -, notre but à moyen terme est donc de disposer d'une base de connaissances qui soit capable de compiler des bases de connaissances, *entre autres elle-même*. Nous ne cherchons pas à déguiser un algorithme de compilation sous la forme d'une base de connaissances, même si au début cela peut y ressembler. Le moyen d'y parvenir est de construire des versions successives, encore exécutées par l'interpréteur - dont la charge devient de plus en plus faible - et qui sont utilisées pour traiter la version suivante.

Dans ce schéma général, une multitude d'options sont possibles. On peut par exemple vouloir réaliser le plus vite possible un compilateur complet, et le compiler. Un tel compilateur serait très proche d'un "algorithme déguisé". C'est la première approche que nous avons prise, qui s'est avérée stérile vu notre point de départ. On peut aussi retarder l'obtention d'un compilateur complet, en introduisant petit à petit des connaissances permettant de faire un "meilleur" compilateur. C'est la voie que nous suivons actuellement.

Ces "autres" connaissances sont de divers ordres. Il peut s'agir de connaissances ayant directement pour but d'améliorer l'interprétation et la compilation. Nous développons dans ce sens un corps de connaissances qui vise à mieux comprendre le comportement de la base de connaissances à compiler et à y trouver des propriétés spécifiques. Mais il peut s'agir aussi de connaissances qui *facilitent* l'acquisition des connaissances de compilation, sans y être directement impliquées. Par exemple, nous développons actuellement des connaissances pour traduire un texte informatique dans un autre. Dans le cours du développement de notre système, un certain nombre de "micro-langages" sont apparus, pour par exemple donner des propriétés de la mémoire de travail ou du contrôle de l'inférence. Le but de ces connaissances de traduction est de nous éviter d'avoir à représenter les "textes" exprimés naturellement dans ces langages dans la mémoire de travail. Il peut s'agir enfin de connaissances "informatiques", relativement indépendantes de la compilation de règles, mais ayant pour domaine l'écriture de programmes. En particulier, ces connaissances doivent être capables de construire et de manipuler des programmes et des structures de données.

S'il peut s'avérer souhaitable de disposer de connaissances ne concernant pas *stricto sensu* la compilation, mais qui aident à leur acquisition, il peut aussi être intéressant

de *modifier le langage de représentation des connaissances*. Ce point est très important. Les méta-connaissances, quelles qu'elles soient, travaillent sur la *forme* des connaissances. Pour un *contenu* donné, il est possible d'écrire des connaissances ayant des formes différentes. C'est aussi simple que de dire qu'il n'existe pas un seul programme qui résolve un problème donné. Certaines de ces formes seront sans doute plus faciles à traiter par les méta-connaissances, de la même manière que nous lisons plus facilement un programme bien écrit. Il est clair que le langage de représentation des connaissances contraint énormément ces formes. En pratique, plus celui-ci est expressif, général ou déclaratif, plus il est facile à des méta-connaissances de traiter des connaissances qui y sont représentées. Pour donner un exemple, il est sans doute plus facile de comprendre un programme Prolog que le "même" programme écrit en assembleur. Mais il y a une contrepartie. Plus le langage est "riche", plus il faut de connaissances pour le traiter. On est donc confronté à une situation paradoxale, où d'un côté la richesse du langage facilite son traitement, et de l'autre le complique.

En pratique, nous avons opéré des modifications pour l'instant mineures dans le langage de règles dont nous disposions; nous disons "mineures", car il y a toujours des règles, une mémoire de travail, un pattern matching, etc. La plus importante est le déplacement de la résolution de conflits vers la base de connaissances. Notre système fonctionne maintenant à divers niveaux méta, chaque niveau surveillant et agissant sur les niveaux inférieurs. Au sommet, il y a une stratégie fixe de résolution de conflits compilée par des connaissances de notre système.

## 2.2. Types de connaissances de Shal

Nous décrivons ici les connaissances que Shal a d'ores et déjà, et celles sur lesquelles nous travaillons actuellement. Nous les avons regroupées par grandes catégories, les unes déjà mises en œuvre, les autres en cours de réalisation.

### *2.2.1. Catégories de connaissances mises en œuvre dans Shal*

- *Contraintes sémantiques* : Ce sont des connaissances sur les propriétés de la mémoire de travail de la base de connaissances à compiler, avant et pendant l'inférence. Ce sont donc des connaissances sur les faits, et indirectement sur les règles, puisque des propriétés d'évolution de certaines classes de faits ou d'objets au cours de l'inférence sont représentées. Nous appelons ces connaissances *contraintes sémantiques*, par analogie avec les contraintes sémantiques sur la base de faits que considèrent certains travaux sur la vérification de bases de connaissances [HER 85], [AYE 87], [ROU 88], [CHA 90], [LAF 90]. Pour donner une idée de ces contraintes sémantiques, on peut par exemple dire qu'une certaine classe de faits est vide initialement, ou qu'elle ne peut que croître en cours d'inférence, ou qu'une relation est monovaluée dans un sens, ou symétrique, etc. Ces contraintes sémantiques, purement déclaratives, sont représentées dans la mémoire de travail de Shal. Le fait que Shal travaille à différents niveaux méta conduit à certaines difficultés, qui seront détaillées.

- *Connaissances de découverte et de vérification de contraintes sémantiques* : Dans les premières versions de Shal, c'était au concepteur de la base de connaissances soumise à Shal de fournir les contraintes sémantiques sur celle-ci. Cependant, cela est très lourd en pratique pour deux raisons. Premièrement, lorsque la base de connaissances est conséquente (et c'est le cas de Shal), les contraintes sont très nombreuses, d'où une source d'erreurs, d'oublis, et surtout de paresse. Deuxièmement, un grand nombre de contraintes peuvent être retrouvées (prouvées) en analysant les règles, ou soupçonnées de manière heuristique. Nous avons donc donné à Shal des connaissances pour trouver, soupçonner et vérifier les contraintes sémantiques. La vérification est très utile lorsqu'on change de version de Shal. Une ancienne contrainte sémantique peut s'avérer incorrecte dans une nouvelle version, et ces connaissances le découvriront le plus souvent. Elle est également indispensable pour juger des contraintes sémantiques devinées heuristiquement (cf. connaissances sur les défauts).

- *Connaissances d'analyse du comportement d'une base de connaissances* : Ces connaissances utilisent les précédentes. Elles analysent ce qui est communément appelé le graphe de précédence des règles, qui lie une règle R1 à une règle R2 lorsque un conséquent d'une règle R1 agit sur une prémisse d'une règle R2. Ces connaissances construisent ce graphe, suppriment des arcs, ou au contraire prouvent que certains arcs sont "obligatoires". D'autres connaissances fournissent au moteur les informations recueillies pour qu'il en tienne compte durant l'inférence, dans un but d'optimisation.

- *Connaissances de reconnaissance de patterns* : Ces connaissances reconnaissent certains patterns de règles ou de sous-ensemble de règles. Cela signifie que le comportement de ces règles obéissent à certains clichés comportementaux fournis au système (il ne les a pas trouvés). Cela aide à l'analyse du comportement de la base de connaissances, et rend possible de compiler ces patterns sous une forme optimisée.

- *Connaissances de compilation des patterns de règles* : Ce sont les connaissances qui transforment effectivement les patterns reconnus en programmes. Ces connaissances sont encore incomplètes. Elles ont besoin d'une connaissance globale, à la fois de la base de connaissances, mais aussi du moteur d'inférence. En effet, nous n'avons pas encore supprimé le moteur d'inférence, et il faut donc "glisser" du code dans celui-ci pour que l'interprétation des règles que l'on sait compiler se voie substituer l'exécution de leur forme compilée. Ce n'est pas un simple problème technique, il faut connaître au moins en partie les instructions du moteur d'inférence, ses structures de données et ce qu'elles représentent pour faire le pont. Nous travaillons à donner les connaissances à notre système pour qu'il accède à ces connaissances. Il est également évident que, pour compiler, il faut avoir des connaissances de programmation.

- *Connaissances de contrôle de l'inférence* : Ces connaissances travaillent essentiellement "au niveau méta". Ce sont elles qui font la résolution de conflit, en fonction de "buts" et de "plans" pour le moment fournis au système. Elles sont

encore relativement simples. La stratégie fixe de résolution de conflit au sommet des niveaux méta est synthétisée par une expertise.

- *Connaissances de synthèse de traducteurs de "textes" informatique* : Nous avons maintenant un certain nombre de "micro-langages" qui s'ajoutent aux règles et aux faits, en particulier les contraintes sémantiques et les plans. De plus, tous les faits déduits par Shal sont une représentation d'objets complexes que l'on aimerait pouvoir décrire de manière plus pratique. Cela est aussi vrai pour les règles : leur syntaxe actuelle conduit à écrire des règles ayant des dizaines d'éléments (prémisses, conséquents, variables). On aimerait donc une syntaxe plus compacte. Il est également souhaitable que Shal ait rapidement accès au code du moteur d'inférence. Actuellement, tout cela est représenté manuellement en base de faits, ce qui est pénible. On aimerait donc disposer de nombreux "micro-langages" pour donner du "sucre syntaxique", aujourd'hui indispensable[1]. Enfin, notre but est de supprimer le moteur d'inférence, dont une partie importante est consacrée à l'analyse syntaxique de la base de connaissances.

  Plutôt que d'écrire manuellement un traducteur pour chaque micro-langage, nous avons préféré donner des connaissances pour synthétiser ces traducteurs à partir d'une spécification aussi simple que possible[2]. Dans un proche avenir, ces connaissances permettront même de remplacer les connaissances qui *écrivent* les programmes dans un fichier à partir de leur représentation en mémoire de travail.

### *2.2.2. Catégories de connaissances en développement*

- *Connaissances de manipulation de programmes* : Les programmes sont construits par petites transformations successives par des règles. Ces connaissances permettent de manipuler et de "mettre en ordre" les "proto-programmes" partiellement représentés dans la mémoire de travail de Shal. Ces connaissances sont encore peu développées dans Shal, car nous avions jusqu'à présent joué le "court terme" en donnant des règles algorithmiques d'écriture de programmes spécifiques à chaque problème de synthèse. Ces connaissances sont de manière évidente complexes. En particulier, il s'avère nécessaire non seulement de fournir des connaissances qui "conçoivent un programme", mais aussi d'accompagner cette conception du "sens" des instructions synthétisées. Nous travaillons actuellement à fournir ces connaissances avec un degré de généralité suffisant, afin qu'elles s'appliquent aux cas de synthèse de programme dont nous avons besoin.

- *Connaissances de construction de structures de données* : Une partie importante des connaissances "informatiques" est concernée par les structures de données. Par exemple, Shal pourra déduire à un moment donné qu'il faut manipuler un ensemble, avec certaines caractéristiques (par exemple : on y fait des recherches

---

[1] Encore que "micro" soit un préfixe mal adapté, puisque nous incluons dans nos micro-langages les langages de programmation du moteur d'inférence - actuellement Pascal et C.

[2] D'où de nouveaux micro-langages pour cette spécification.

fréquentes, il peut être grand, on y ajoute des éléments mais n'en supprime pas, etc). Il faut savoir quelle structure de données efficace introduire pour représenter cet ensemble dans le code compilé. Cette expertise comporte deux volets. Tout d'abord, il faut connaître les différents types de structures de données utilisées en informatique. Cette expertise est largement répandue et bien connue, aussi espérons-nous avoir peu de problèmes à la mettre en œuvre. Deuxièmement, il faut savoir reconnaître les propriétés dynamiques (en fonction du programme où elles servent) des ensembles de données manipulés par un programme. Cela, en particulier, nécessite de connaître le "sens" des instructions du programme en cours de conception par rapport à sa spécification initiale.

- *Connaissances utilisant des programmes existants* : Pour accéder au code du moteur d'inférence, Shal doit avoir connaissance de la signification des structures de données et ce qu'elles représentent. Par ailleurs, nous aimerions pouvoir utiliser dans les niveaux méta des méta-informations, comme "la règle Ri ne s'est pas déclenchée …" ou "l'instanciation de la variable X de la règle Rj est …". Or, ces informations sont disponibles quelque part dans l'interpréteur, ou, si elles ne le sont pas, on peut y introduire des morceaux de code simples qui la conservent. De plus, il faut ajouter des sortes de démons qui transfèrent ces informations de l'interpréteur vers la mémoire de travail aux moments adéquats. Plutôt que de transformer l'interpréteur nous-même, nous aimerions fournir à Shal les connaissances qui lui permettent de le faire. Il faut donc que Shal sache utiliser des structures de données et des programmes existants, éventuellement y insérer de nouvelles instructions et structures de données, et traduire l'information d'une structure de données d'un programme dans une mémoire de travail. Ce dernier point ressemble beaucoup à la traduction de textes informatiques précédemment citée.

- *Connaissances pour la gestion des versions de Shal* : Ces connaissances interviennent à un niveau très différent des autres. Plus généralement, il faudrait un corps de connaissances qui gère une interface "intelligente" pour le développement de Shal lui-même. Nous avons placé actuellement cet objectif à la fin de notre agenda personnel. Cependant, Shal s'appuie dans des aspects de plus en plus nombreux de ses activités sur des déductions obtenues de façon heuristique. C'est par exemple le cas de la découverte des contraintes sémantiques, de la traduction de textes informatiques, ou de la reconnaissance de patterns de règles. Or, une mauvaise déduction heuristique peut avoir des conséquences catastrophiques, puisqu'elle peut en définitive changer les programmes ou le comportement de Shal. Il faut donc qu'il puisse vérifier sur des exemples que ses déductions sont correctes, et savoir les remettre en cause si elles ne le sont pas. Il est évidemment nécessaire que Shal sache quelles connaissances sont potentiellement sujettes à cette remise en cause. Nous construisons actuellement des mécanismes de défaut (via des connaissances) relativement simples, et seulement pour les déductions potentiellement douteuses qui sont faciles à confronter à des contre-exemples. Lorsqu'une connaissance par défaut est contredite, on refait les déductions qui ont permis de la déduire, et les déductions qui l'utilisent. Savoir quelles connaissances sont impliquées dans ces déductions relève du modèle d'expertises ci-dessous. Nous ne mettons pas en

œuvre de mécanisme de maintenance de la vérité genre TMS, qui s'avérerait inutilisable vu la taille déjà atteinte par Shal. Nos mécanismes sont à la fois plus grossiers, car moins précis dans le maintien de l'historique des déductions, et plus "malins", car ils maintiennent l'historique implicitement en termes de connaissances utilisées, et non en termes purement factuels.

- *Modèles d'expertises* : Les règles sont maintenant regroupées en paquets de règles appelés expertises. On essaie de faire en sorte que ces expertises correspondent à quelque chose de conceptuellement signifiant. Shal tente alors, grâce au résultat de son analyse du comportement, de construire un modèle de ces expertises, en terme d'entrées-sorties. Le modèle a la forme d'une règle (mais n'est pas une règle correcte du système) qui possède en prémisse les types de faits constituant "l'entrée" de l'expertise, et en conséquent ceux qui constituent sa sortie. C'est une vision où une expertise est considérée comme un composant physique, les faits qu'elle manipule comme les fluides qui le traversent, et le modèle comme une modélisation de son comportement. Ces modèles-règles permettent de construire un modèle grossier des capacités de déduction du système. Ils sont utilisés dans les défauts, et dans l'avenir permettront peut-être à Shal de construire lui-même ses plans de résolution de problèmes.

### *2.2.3. Relations entre catégories de connaissances*

On peut voir les relations entre ces ensembles de connaissances de deux façons. Comme tout système, Shal exécute des connaissances pour atteindre un but. Il y a donc une certaine "logique" des enchaînements. Par exemple, pour compiler une expertise, Shal utilise d'abord les connaissances de découverte et de vérification des contraintes sémantiques, puis utilise ces déductions dans l'analyse du comportement de la base de connaissances et la reconnaissance de patterns. Les connaissances de compilation des patterns en utilisent les résultats, et "appellent" les connaissances "informatiques". Tout cela est contrôlé par les connaissances de contrôle.

Mais cet aspect "linéaire" de l'utilisation de ces connaissances disparait dès que l'on considère que Shal est systématiquement utilisé sur lui-même dans sa construction. Plus précisément, on se sert de l'ancienne version de Shal pour traiter la nouvelle version. La raison d'existence de ces connaissances est justement de permettre une amélioration de Shal. Par exemple, les connaissances de programmation vont bien servir à compiler partiellement les versions de Shal, mais elles, et toutes les autres, vont surtout servir à améliorer, par exemple, les connaissances de programmation, à faire des choses plus complexes, pour en définitive s'émanciper de la programmation. Pour résumer, la raison de l'existence d'un ensemble de connaissances n'est pas seulement de fournir les informations à l'ensemble de connaissances suivant dans l'exécution, elle est de permettre d'augmenter le système tout entier. C'est la problématique exprimée par Jacques Pitrat [PIT 84, 85, 86, 87, 88, 89 & 90] sous le nom de *problème de l'amorçage*, ou de *bootstrapping*.

## 2.3. Où en sommes-nous?

Clairement, ce qui précède montre que Shal n'est pas terminé, y compris sous le "simple" aspect de la compilation de bases de règles. Par ailleurs, ces différents ensembles peuvent paraître à première vue disparates. Ce problème se règlera dans l'avenir, mais il est normal qu'il se manifeste dans l'état actuel de développement.

Le sentiment extrêmement frustrant que l'on a lorsque l'on suit notre approche est que, pour résoudre un problème, il faut les résoudre tous. Prenons un exemple "d'école". Pour faire un compilateur, il faut donner des connaissances. Or, on fait des erreurs dans ces connaissances. Il faut donc donner des connaissances pour trouver les erreurs. Mais qui va les compiler ? Qui va trouver leurs erreurs ? Les connaissances de compilation dont on dispose à un moment donné ne sont pas nécessairement capables de compiler les connaissances de debugging. Il faut donc d'autres connaissances de ce type, etc. On est pour le moins très vite confronté à un problème de taille. Cet exemple ne vaut que pour les aspects de compilation et de debugging. D'autres types de connaissances s'avèrent rapidement nécessaires en pratique : les connaissances "syntaxiques", "de programmation", "sur les défauts", etc.

Notre expérience montre qu'il faut adopter dans notre approche une attitude d'humilité: il faut tenter de résoudre beaucoup de problèmes à la fois, *mais mal*. En effet, si un sous-problème est "parfaitement" résolu par un ensemble de connaissances, on ne disposera probablement pas des connaissances pour le traiter. A la limite, si l'on avait des connaissances "parfaites" pour tous les problèmes (ce que nous ne prétendons pas avoir …), nous ne disposerions que d'un moteur d'inférence, de toute évidence incapable de les exécuter. *Il faut passer par une sorte d'évolution du système.*

Le processus de développement d'un tel système doit être en permanence guidé par la question récursive (et récurrente) suivante : *De quelles connaissances ai-je besoin pour réaliser l'étape suivante ? (*) Elles constitueront l'étape suivante. De quelles connaissances ai-je besoin pour traiter ces connaissances ? Aller en (*).* Lorsque les connaissances pour réaliser l'étape suivante sont déjà disponibles, on connaît l'étape suivante. Par exemple, un de nos buts actuels est de compiler les patterns de règles (nous l'avions déjà fait partiellement, mais de manière non satisfaisante). Or, dans l'écriture des connaissances pour ce faire, nous avions des problèmes de contrôle très pénibles. Nous avons donc introduit les connaissances de contrôle, et modifié le système pour qu'il fonctionne à différents méta-niveaux. Cela nous a conduit à introduire un nouveau micro-langage, celui des "plans". Par ailleurs, la taille de Shal s'accroissant, nous ne pouvions plus vivre sans que Shal ait des connaissances de vérification de ses connaissances. Cela impliquait une utilisation plus intensive des contraintes sémantiques. Or, nous ne disposions pas de "découvreur", et il fallait donc les entrer manuellement. Ces deux raisons nous ont incité à fournir au système des connaissances de traduction de textes informatiques, avec les avantages décrits. Mais cela implique de disposer de connaissances de manipulation de programmes plus poussées, etc. En résumé, pour compiler les patterns de règles, nous donnons des connaissances "syntaxiques" et "de programmation", des "défauts" et des "modèles de règles". Lorsque nous les aurons de manière suffisamment solide - mais

sans les rendre trop sophistiquées, sinon nous ne nous en sortirons jamais -, la compilation de patterns de règles sera gagnée. Et nous pensons que la disparition complète du moteur d'inférence ne sera pas loin.

## 3. Introduction

Cet article présente l'état d'avancement du système Shal, qui est une base de connaissances pour compiler une base de connaissances. La construction de Shal n'est pas terminée, et son aspect inachevé y est patent. Nous décrivons dans le paragraphe 4 le détail des connaissances que Shal possède actuellement, et celles qui sont en cours de mise en œuvre. Dans le paragraphe 5, nous montrons les avantages et les difficultés de la méthode du bootstrapping de connaissances que nous avons adoptée pour construire Shal. Le paragraphe 6 conclut.

## 4. Les connaissances de Shal

Nous présentons ici les connaissances que Shal possède à ce jour, et celles qui sont en cours de développement. Nous donnons une description "linéaire" de ces connaissances, c'est-à-dire que nous montrons comment elles sont utilisées par d'autres connaissances pour aboutir à un but du système. Cependant, l'approche réflexive de la construction de Shal fournit des types de motivation entièrement différents : certaines connaissances sont là, non pour servir directement dans les processus de déduction qui compilent une base de connaissances, par exemple, mais au contraire pour aider à la construction de Shal. L'ordre chronologique de développement de ces connaissances est donc très différent de celui adopté ici. En fait, nous devons avouer que nous avons été poussés par la nécessité, et que nous n'avons pas eu jusqu'à un passé récent de plan de développement bien défini. Nous pensons que cette attitude est inhérente à notre approche, du moins dans la phase d'exploration où nous sommes.

### 4.1. Les connaissances de contrôle

#### *4.1.1. Motivation*

Le langage de départ de Shal est celui de Boojum [DOR 87], où la résolution de conflits est élémentaire. La stratégie de Boojum est de sélectionner la première règle déclenchable (première dans l'ordre où elles sont fournies) et de saturer les déclenchements de cette règle. La raison pour laquelle nous avions adopté cette stratégie simple est qu'il n'y a pas de "bonne" stratégie de résolution de conflits indépendante de la base de connaissances à laquelle elle s'applique. Sa simplicité est aussi un avantage pour sa mise en œuvre informatique et pour sa compréhension par le concepteur de bases de règles.

Cependant, il est bien connu que l'existence d'une stratégie fixe de résolution de conflits pose de graves problèmes au concepteur de bases de règles. Celui-ci est contraint de la connaître dans le détail, et souvent de "tricher", de la "tromper", en introduisant par exemple des "faits de contrôle" pour que les règles s'appliquent dans l'ordre qu'il désire.

Pendant le début de la conception de Shal, nous avons vécu avec ces problèmes. Puis, la simple survie s'est avérée impossible pour deux raisons. Premièrement, "flaguer" toutes les règles d'un grand système s'avère vite incompréhensible. Deuxièmement, ce qui est vrai pour le concepteur l'est encore plus pour Shal. Shal tente en effet de "comprendre" la base de connaissances qui lui est fournie. Les connaissances pour retrouver le contrôle implicite exprimé dans les faits de contrôle seraient, si nous avions tenté de les écrire, sérieusement sophistiquées[1] ! Nous avons donc choisi une voie plus simple : donner explicitement le contrôle. Cela a le triple avantage de simplifier la vie du concepteur, de simplifier celle de l'interpréteur, qui ne sera plus encombré par les faits de contrôle, et de simplifier les connaissances analysant le comportement d'une base de connaissances. En fait, ces dernières connaissances disposent maintenant d'une meilleure information qu'auparavant.

### *4.1.2. Buts et plans*

Nous avons choisi dans un premier temps un moyen d'expression du contrôle simple, mais en tentant de prévoir les extensions futures. Tout d'abord, les règles sont regroupées par le concepteur en *expertises*, qui ne sont rien d'autre que des paquets de règles. Ensuite, l'utilisateur fournit des *buts* potentiels que le système peut avoir à résoudre, et des *plans* pour résoudre ces buts. Les buts ont une forme purement propositionnelle (ils ne mentionnent pas de variables). Des exemples de buts sont CompilerBaseDeConnaissances, SynthétiserTraducteur, etc, mais aussi, comme nous allons le voir, Planifier. Les plans sont des successions d'actions conditionnelles à effectuer. A chaque but est associé un plan, actuellement unique. Les actions se résument actuellement à exécuter une expertise ou à atteindre un but. Les plans se matérialisent en une série de transitions entre actions. Dans un plan donné, une action A peut transiter vers d'autres actions A' de cinq manières différentes. Il y a tout d'abord les transitions inconditionnelles, A --> A'. Dans ce cas, l'exécution de l'action A doit être systématiquement suivie de l'exécution de l'action A'. Il y a ensuite quatre sortes de transitions conditionnelles, A d--> A' et A nd--> A' si A est une exécution d'expertise, et A a--> A' et A na--> A' si A est une résolution de but. Si on a une transition A d--> A' (resp. A a--> A'), alors l'exécution de l'action A doit être suivie de l'exécution de l'action A' si au moins une règle de l'expertise exécutée dans A s'est déclenchée (resp. si le but a été atteint). A nd--> A' (resp. A na--> A') a l'effet inverse: A' sera exécutée si aucune règle de l'expertise exécutée dans A ne s'est déclenchée (resp. si le but n'a pas été atteint). Enfin, l'exécution d'une expertise (qui regroupe plusieurs règles) suit actuellement de manière interne la stratégie de résolution de conflits de Boojum.

---

[1] En fait, nous avions commencé à écrire de telles connaissances, et bien vite abandonné.

Les "plans" sont donc très simples. Premièrement, ils sont fournis par le concepteur, et non trouvés par le système. Ils ressemblent actuellement plutôt à un *langage de commande*. Malgré tout, ils permettent d'exprimer le contrôle plus facilement, et d'introduire de nouveaux éléments de contrôle, comme des contraintes. Certaines expertises contiennent en effet maintenant des règles sans conséquent. Exécuter une telle expertise a pour but de savoir si la partie gauche d'une de ces règles est satisfaite. Ces expertises agissent bien comme des contraintes que l'on peut vérifier. Ensuite, il est possible d'introduire des circuits dans les plans, c'est-à-dire ... des boucles. Il est aussi possible (et même nécessaire, cf. infra) d'écrire des expertises pour chaque but potentiel du système qui regardent si le but est satisfait. De telles expertises fournissent l'expression des conditions de satisfaction des buts, ou, dit autrement, des conditions d'arrêt. Enfin, on peut écrire des expertises qui regardent si le plan se déroule correctement, qui peuvent donc récupérer des erreurs. Enfin, nous avons récemment introduit le parallélisme entre les actions, c'est-à-dire qu'une action peut être suivie de plusieurs actions, et que plusieurs actions peuvent être suivies d'une action unique. Cela ne change rien quant à la logique des plans, mais apporte une source de connaissances supplémentaire pour la compréhension du comportement et pour le debugging, car des actions en parallèles sont réputées commutatives.

### 4.1.3. Méta-contrôle de l'exécution des plans

Venons-en maintenant à la manière dont sont exécutés les plans. Les plans sont exécutés par des expertises spéciales, des *expertises d'exécution de plans*. Ces expertises tentent de satisfaire un but, qui est de satisfaire le but courant du système. Ce but spécial s'appelle Planifier. Ces expertises sont donc exécutées selon le plan associé au but Planifier (ce plan s'appelle PlanPourPlanifier). Qui exécute ce plan ? Justement les expertises d'exécution de plans. Nous sommes partis dans les tours infinies de niveaux méta de H. Dreyfus …

Pour résoudre ce problème, l'architecture du système a été modifiée et est maintenant la suivante. Le système ne possède toujours qu'un unique ensemble d'expertises, mais ces expertises pourront être utilisées à n'importe quel niveau méta. Par contre, il n'y a plus une seule mémoire de travail (base de faits), mais *une mémoire de travail par niveau méta "actif"*. Dans la mémoire de travail "du bas", disons celle de niveau 0, on met les faits ordinairement mis dans la mémoire de travail, c'est-à-dire ceux qui représentent les éléments du problème posé (par exemple la représentation de la base de connaissances à compiler). Dans la mémoire de travail de niveau 1, on met le but assigné au niveau de base, et tous les plans. Dans les mémoires de travail de niveau supérieur, on met le but du niveau juste inférieur, et également tous les plans. Il est clair que l'on retrouve très rapidement des buts similaires.

Prenons un exemple (cf. Figure 1). Supposons que le but du niveau de base soit CompilerBaseDeConnaissances. Ce but est exprimé au niveau 1, et nous voudrions que les expertises d'exécution de plans s'exécutent au niveau 1 (c'est-à-dire sur cette mémoire de travail de niveau 1) pour qu'elles retrouvent le PlanPourCompilerBaseDeConnaissances, et l'exécutent. Pour cela, il faut mettre au

niveau 2 le but Planifier, de manière à ce que les expertises d'exécution de plans s'exécutent au niveau 2 pour exécuter ces mêmes expertises au niveau 1 selon le plan PlanPourPlanifier. Attention ! Ce sont bien les mêmes expertises d'exécution de plans qui s'exécutent aux niveaux 1 et 2, selon le même plan PlanPourPlanifier, mais elles n'exécutent pas le même plan. Au niveau 1, le plan exécuté est PlanPourCompilerBaseDeConnaissances, et au niveau 2 PlanPourPlanifier. Comment maintenant les expertises d'exécution de plan vont-elles être exécutées au niveau 2 ? On pourrait les faire exécuter par "elles-mêmes", prises au niveau 3, avec exactement le même but qu'au niveau 2. Mais il est clair que cela est inutile, car ces expertises vont faire la même chose aux niveaux 2 et 3. Par ailleurs, il faut arrêter l'ascension dans les niveaux méta. Cela est possible en *procéduralisant l'expertise d'exécution de plans appliquée au PlanPourPlanifier*. Il suffit donc de disposer d'un programme qui "soit" l'exécution de ces expertises pour ce plan, et de le laisser s'exécuter au niveau méta ad hoc.

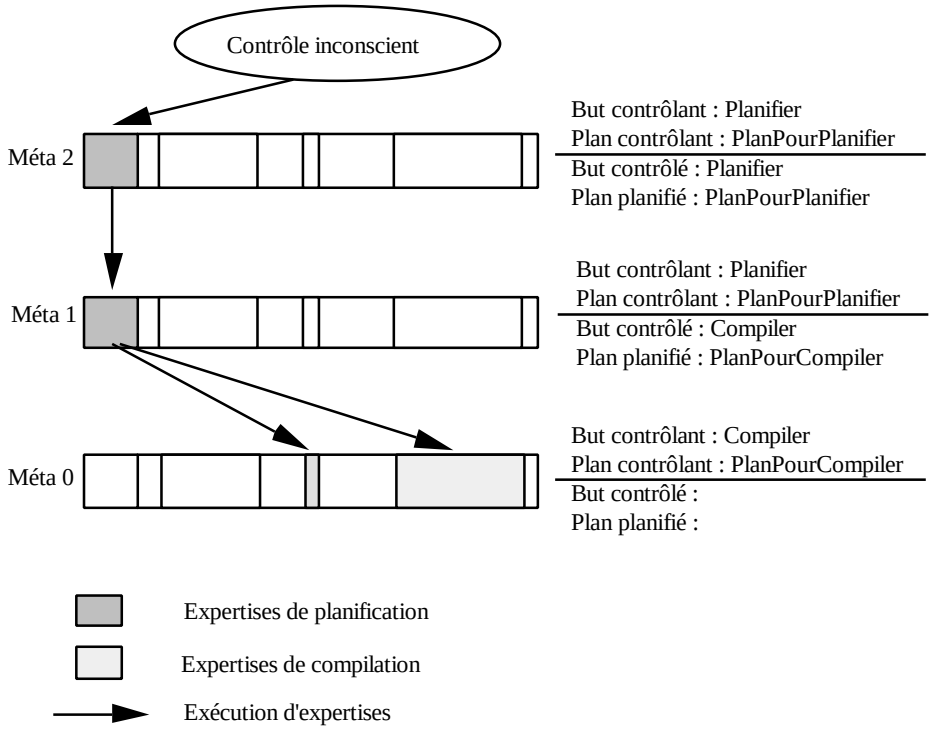



*Figure 1 : Le contrôle dans Shal*

Cette planification procéduralisée est effectivement un programme, mais nous ne l'avons pas écrit. Il est synthétisé par des expertises, dites de "synthèse du contrôle procédural", qui ne font que traduire en programme procédural la suite d'actions que donnerait les expertises d'exécution de plans lorsqu'elles sont appliquées au but Planifier en suivant le plan PlanPourPlanifier.

Ce qui précède montre que l'on peut mettre cette expertise procéduralisée, appelons-là *contrôle inconscient*, au niveau 2. Cependant, nous la mettons quelquefois au niveau 3. Cela peut sembler inutile, puisqu'elle y effectue à ce niveau exactement les mêmes actions que les expertises d'exécution de plans qu'elle contrôle au niveau 2. Cependant, des problèmes peuvent se poser lorsque l'on modifie les expertises d'exécution de plans et conséquemment celles de synthèse du contrôle procédural. Nous ne rentrerons pas plus loin dans cette discussion.

Lorsque l'on exécute un plan pour atteindre un but, il faut savoir si le but est atteint. Pour chaque plan ou but, une expertise est fournie, qui se déclenche si le but est atteint. Ainsi, l'expertise de succès du but CompilerBaseDeConnaissances est une simple expertise d'écriture de message de succès, mais qui ne serait pas exécutée s'il y avait eu un échec. Pour le but Planifier, l'expertise de succès regarde si l'expertise qui vient d'être exécutée (au niveau inférieur) était l'expertise déterminant si le but en cours est satisfait, et si elle s'est déclenchée (la planification est un succès si ce que l'on a planifié est un succès).

Nous avons décrit comment les mémoires de travail de chaque niveau méta sont initialisées (buts et plans pour les niveaux supérieurs). Au cours de l'inférence, ces mémoires de travail changent, et nous allons examiner ce qu'elles contiennent aux niveaux supérieurs. Elles contiennent tout d'abord la trace d'exécution des expertises du niveau inférieur. Celle-ci consiste en la liste (ou le graphe sans circuit) des actions exécutées. Sont également présentes des informations permettant de savoir quel plan est en cours d'exécution, où on en est dans le plan, les sous-buts en examen, etc. De plus, des informations sont nécessaires pour savoir quelle transition appliquer ensuite. Pour cela, il faut avoir accès à certaines informations relatives à l'exécution des expertises du niveau inférieur, et les représenter dans la mémoire de travail. Nous avons choisi dans la version actuelle de Shal de modifier le moteur pour mémoriser ces informations et les transférer dans la mémoire de travail du niveau méta ad hoc. Elles consistent, pour chaque expertise exécutée, en le nombre de déclenchement de règles de cet expertise dans l'appel (le fait qu'un sous-but est atteint est directement géré par les expertises, puisque cela n'a jamais été programmé dans le moteur). Dans l'avenir, nous aimerions introduire d'autres "méta-prédicats" similaires, mais sans avoir à les programmer.

#### *4.1.4. Avantages potentiels du méta-contrôle de Shal*

L'avantage de notre architecture est qu'elle est extensible. Actuellement, tout but est associé à un plan unique, toute action d'un plan doit être suivie de manière non ambiguë par une ou plusieurs actions selon des conditions de transition très grossières, etc. Toutes ces conditions sont restrictives. Cependant, nous n'aurions pas de difficultés de principe à étendre ces capacités. Par exemple, on pourrait associer plusieurs plans à un but, en ajoutant des connaissances de sélection de plan selon le contexte. On pourrait imaginer d'autres types de choix de transitions d'actions, par exemple basés sur des expertises. Nous n'en avons pas eu à ce jour un besoin criant, mais ces extensions sont réalisables.

Un autre problème se pose, concernant les "méta-prédicats". Actuellement, un niveau méta ne connaît des niveaux inférieurs que la trace des expertises exécutées au niveau juste inférieur, et le nombre de déclenchements de règles à chacune de ces exécutions. Il serait souhaitable qu'il en connaisse plus. Il pourrait par exemple connaître des informations connues du niveau inférieur, comme des morceaux de sa mémoire de travail. Il pourrait aussi connaître d'autres éléments du niveau inférieur, et inconnus de lui, comme les expertises, les instanciations, les essais ayant échoué, etc. Toutes ces informations sont quelque part, dans la base de faits ou dans la structure de données du moteur, ou sont relativement faciles à retrouver ou à conserver. Cependant, deux problèmes se posent. Premièrement, il est impossible à un niveau méta de "tout" connaître des niveaux inférieurs. En effet, ce "tout" est trop grand, voire sans limite. Il faut deuxièmement que le niveau méta puisse trouver les informations dont il a besoin. Pour cela, il doit savoir *où* aller chercher ces informations, et *comment*. Or, aucune de ces deux catégories de connaissances ne sont encore complètement disponibles à Shal, et les rendre disponibles dépend de notre travail sur les connaissances de manipulation de programmes et de structures de données. Dans un avenir que nous espérons proche, nous pourrons donc utiliser des "méta-prédicats" dans les expertises de contrôle, sans avoir à les *programmer*. Enfin, ce qui vient d'être dit pour les relations entre un niveau méta et le niveau juste inférieur pourrait être étendu à tous les niveaux inférieurs. Par exemple, les connaissances de planification du niveau 2 pourraient tirer avantage de la connaissance qu'elles planifient la planification de la compilation. Mais nous n'en sommes pas encore là.

Le contrôle que nous avons décrit n'utilise parmi les connaissances de Shal aux niveaux supérieurs que les connaissances liées aux plans. Formellement, toutes les connaissances sont pourtant disponibles à tous les niveaux, mais Shal ne sait rien en faire. Nous ne désespérons pas qu'un jour Shal détermine lui-même, au moins en partie, les plans. Cette tâche peut impliquer la résolution d'autres problèmes, qui peuvent prendre la forme de la résolution d'un problème de contraintes, de l'écriture et l'exécution d'un programme pour obtenir des résultats intermédiaires, etc. Dans ce cas, *toutes* les méta-connaissances seront potentiellement utiles à *tous* les niveaux méta.

Enfin, le contrôle inconscient est actuellement à un niveau décidé à l'avance (niveau 2 ou 3), et n'en bouge pas. Nous donnerons la possibilité à Shal de faire "monter dynamiquement" le contrôle inconscient lorsqu'il aura certaines capacités supplémentaires. Lorsqu'un plan s'exécute, certaines vérifications sont faites, qui vérifient que le plan donne les résultats attendus. Actuellement, si ce n'est pas le cas, Shal s'arrête tout simplement, en fournissant un message d'échec de la planification. Dans l'avenir, Shal aura certaines connaissances justifiant l'emploi d'un plan, et il est alors envisageable que, lorsqu'un plan échoue, il se demande pourquoi, et éventuellement apporte certaines actions correctrices, ou change de plan. Cela peut aussi s'envisager lorsque plusieurs plans concurrents sont disponibles, et que le plan essayé n'est pas satisfaisant. Dans ce cas, un autre type d'action sera exécuté, qui consistera à *monter* d'un niveau méta (au lieu de descendre) pour résoudre ce problème. Tout cela est similaire à ce qui se passe dans SOAR [LAI 87]. Si le

niveau méta où l'on veut monter est occupé par le contrôle inconscient, il faudra le "pousser" d'un cran vers le haut, installer une mémoire de travail équivalente à son action passée (c'est-à-dire "faire comme si" ce programme avait été un ensemble d'expertises), et continuer au nouveau niveau méta. Ces mécanismes ne sont pas mis en œuvre à ce jour, car le besoin n'en a pas encore été pressant.

#### *4.1.5. Efficacité du méta-contrôle*

Une croyance est largement répandue selon laquelle "le méta, c'est inefficace". Nous devons avouer que nous l'avons partagée, au moins sous certains aspects. Aussi avons-nous été très surpris par les résultats du méta-contrôle de Shal.

Dans Shal, le "méta" ne fait pas perdre de temps, *il en fait le plus souvent gagner*. Pourquoi ? Dans nos premières expériences de mise au point du méta-contrôle, nous avions pris pour exemple des bases de connaissances au niveau 0 très petites. Le résultat était rageant : pour chaque déclenchement de règle du niveau 0 (c'est-à-dire, le travail "utile"), il fallait 10 déclenchements au niveau méta, et 100 au niveau méta-méta. Le résultat était donc une dégradation des performances d'un facteur 100 ! Puis nous avons pris des exemples plus substantiels (Shal lui-même). Le rapport était alors très différent. Pour 1000 déclenchements de règles du niveau de base, il n'y avait plus que 10 déclenchements au niveau méta et 100 au niveau méta-méta. En définitive, on ne "perdait" que 10% du nombre d'inférences. La raison de la différence avec les exemples simples est que, lorsqu'une expertise du niveau de base est exécutée, c'est pour faire quelque chose de substantiel. Dans les exemples simples, l'exécution d'une expertise du niveau de base conduisait à 0 ou 1 déclenchement de règle, dans l'exemple complexe, le même type d'exécution conduisait à des dizaines ou des centaines de déclenchements. La proportion de temps perdu par le "méta" devenait donc faible.

D'un autre côté, ce temps perdu est largement compensé par un autre phénomène. Au lieu d'avoir à chercher à chaque cycle la règle à examiner parmi *toutes* les règles, le moteur d'inférence est d'emblée focalisé sur l'expertise à exécuter. D'autres tâches du moteur en sont également allégées. Il s'avère que ce gain de temps compense largement celui perdu par le "méta". Nous avons eu des cas où l'exécution a été trois fois plus rapide avec le "méta" que sans.

#### *4.1.6. "Philosophie" du méta-contrôle*

Pourquoi avons-nous introduit ce type de méta-contrôle ? Beaucoup de travaux ont déjà été consacrés au méta-contrôle, depuis TEIREISIAS [DAV 82] jusqu'à MACISTE [PIT 84-90] en passant par SNARK [VIA 85], [LAU 86] et PRODIGY [MIN 89]. Ce que nous n'avons pas voulu reproduire, c'est le contrôle procédural par "appel de programme", même si actuellement ce que nous avons fait y ressemble. Shal a une liberté potentielle de son comportement. Au lieu de contenir des ordres du type "faire çà", il commence par se dire "quel est mon but ?" puis "comment vais-je y

aboutir ?". La différence avec l'appel de programme sera nette lorsque le système sera capable de trouver lui-même les plans.

## 4.2. Les contraintes sémantiques

### *4.2.1. Motivation*

Une idée initiale de notre travail était de faire une base de connaissances pour "comprendre le comportement d'une base de connaissances". Il est facile de constater qu'une grande part des actions d'un moteur d'inférence sont inutiles ou vaines. Il manque au moteur deux sources de connaissances : des connaissances sur la base de connaissances utilisée, et des connaissances pour utiliser ces connaissances. C'est ce premier type de connaissances que nous décrivons ici, sous la forme qu'elles ont prises dans Shal, et que nous avons baptisées *contraintes sémantiques*.

Les contraintes sémantiques expriment des propriétés de la mémoire de travail. Dans certains travaux sur la vérification des bases de connaissances, le terme de contraintes sémantiques est également employé (en fait, nous le leur empruntons), et recouvre des propriétés de la base de faits initiale d'une base de règles, essentiellement des *incompatibilités de faits*. Notre notion de contrainte sémantique recouvre des choses différentes. D'un côté, les incompatibilités de faits initiaux ne nous ont pas été particulièrement utiles pour le moment, et de l'autre nous faisons porter les contraintes sémantiques, non seulement sur la base de faits initiale, mais aussi sur toutes les bases de faits courantes par lesquelles le système passe durant l'inférence. Ces informations vont être essentielles pour analyser la base de connaissances en vue de sa compilation.

### *4.2.2. Idée des contraintes sémantiques*

Considérons l'exemple très simple de la base de connaissances suivante, constituée de la seule règle GrandPere :

```
REGLE GrandPere
SI  PereDe(X) = (Y)
    PereDe(Y) = (Z)
ALORS
    GrandPereDe(X) = (Z)
```

Que savons-nous qui n'est pas dit dans cette règle ? Tout d'abord, on peut introduire une classe Homme, qui sera l'ensemble des individus mentionnés dans la mémoire de travail qui ont ou sont un père. Les faits de la forme (.,PereDe,.) et (.,GrandPereDe,.)

portent à gauche et à droite sur des éléments de Homme. On peut aussi imposer que la base de faits initiale ne contient aucun grand-père. La classe des faits mentionnant la relation PereDe est constante en fonction du temps pour toute base de faits initiale, la classe correspondante pour GrandPereDe est croissante (en particulier aucun grand-père n'est "tué"). Cette dernière classe est également croissante en fonction de la base de faits initiale, et ce pour tout instant. Cela signifie que, plus il y a de pères dans la base de faits initiale, plus il y aura de grand-pères déduits. De plus, PereDe est monovaluée dans le sens ad hoc (un individu a au plus un père), et un individu a au plus deux grand-pères. Enfin, les relations PereDe et GrandPereDe sont, par exemple, antiréflexives (un individu ne peut être son propre père) et antisymétriques (deux individus ne peuvent être mutuellement pères l'un de l'autre).

Pour traduire ces propriétés, nous introduisons des notions de classes, d'évolution des classes durant l'inférence et en fonction de la base de faits initiale, et de propriétés des relations. Tout d'abord, nous introduisons rapidement le langage de la mémoire de travail de Shal, pour fixer les idées.

### *4.2.3. Le langage de la mémoire de travail de Shal*

La mémoire de travail de Shal à un moment donné de l'inférence est un ensemble de faits. Un fait est un couple

o ' s

où o et s sont des objets. s est appelé le statut de o, et tout objet a exactement un statut.

Un objet peut être "atomique", et dans ce cas désigné par un identificateur, un nombre, ou être un triplet d'objets. La définition des objets est donc récursive, ce qui permet "d'emboîter les triplets". Voici quelques exemples de faits :

```
PressionPrimaire '255
(LouisXIII PereDe LouisXIV) 'VRAI
(PereDe Element RelationDeParente) 'VRAI
(JeanDupont SaitQue (LouisXIII PereDe LouisXIV)) 'FAUX
PileDesAppels '(Exp121 ListeAppel (Exp089 ListeAppel
                                        ( … ListeAppel NIL)) … )
```

*o* et *s* sont, dans chacun de ces faits :

```
o / PressionPrimaire                                  s / 255
o / (LouisXIII PereDe LouisXIV)                       s / VRAI
o / (PereDe Element RelationDeParente)                s / VRAI
o / (JeanDupont SaitQue (LouisXIII PereDe LouisXIV))  s / FAUX
o / PileDesAppels                          s / (Exp211 ListeAppel
                                                 (Exp089 ListeAppel
                                                      (…
                                                          ListeAppel NIL
                                                  )) … )
```

Rien n'est structuré dans la mémoire de travail de Shal. En particulier, il n'y a pas de notion *d'objet* comme étant une entité possédant certaines caractéristiques (attributs) ayant des valeurs données ou à déterminer. Les contraintes sémantiques pallient à ce vide structurel.

Les mots utilisés ici n'ont aucune valeur "sémantique" particulière (en particulier "VRAI" et "FAUX"). Par ailleurs, tous les objets d'un triplet sont "à égalité", aucun ne se distingue des autres par sa position. On peut leur donner diverses interprétations. Ainsi, PereDe est une relation binaire - ou un attribut - dans le second fait, SaitQue aussi, à moins de le considérer comme un opérateur modal, et ListeAppel n'est là que pour "typer les doublets" dans l'emboîtement de triplets qui représente une liste. PressionPrimaire et PileDesAppels sont des "cases mémoire", comme des variables informatiques. Ce sont là quelques exemples de représentation utilisant le langage de la mémoire de travail.

Il est à noter que les "relations" ne privilégient pas un sens sur un autre. Ainsi, il est inutile d'introduire la relation inverse d'une relation, comme FilsDe, par exemple. La spécification que nous donnons ici des faits ne préjuge en rien de leur représentation informatique (quels pointeurs, quelles structures ensemblistes, de listes, d'arbres, etc). De fait, Boojum, le système dont nous sommes partis pour construire Shal, avait déjà une certaine autonomie et des connaissances pour gérer les *index* sur la mémoire de travail, c'est-à-dire les liens entre objets et entre faits.

Enfin, les règles peuvent comporter des variables correspondant à n'importe quel élément de la mémoire de travail.

Dans ce qui suit, nous prendrons en compte une version simplifiée de ce langage. En particulier, nous ne considérerons que des classes de faits o ' s dont les objets o sont des triplets et les statuts s un des deux statuts VRAI et INEXISTANT. Enfin, nous écrirons indifféremment

(LouisXIII PereDe LouisXIV)

ou

LouisXIII PereDe LouisXIV

le fait

(LouisXIII PereDe LouisXIV) 'VRAI

#### *4.2.4. Les classes*

Les classes sont pour nous des ensembles dépendant du moment dans l'inférence et de la base de faits initiale (nous dirons en résumé le temps et … l'espace). Plus formellement, une classe est une application

$$c : BF \times N \rightarrow Ens$$
$$(bfi,n) \rightarrow c(bfi,n)$$

qui, à toute base de faits initiale et à tout instant n associe un ensemble. On distingue les classes d'objets et de faits.

Dans l'exemple précédent de la règle GrandPere, les objets sur lesquels portent les relations PereDe et GrandPereDe constituent la classe Homme, les faits ayant PereDe ou GrandPereDe comme relation constituent les classes de faits notées (.,PereDe,.) et (.,GrandPereDe,.). Si, par exemple, on considère la base de faits initiale BFI constituée des faits

LouisXIII PereDe LouisXIV
LouisXIV PereDe LouisDeFrance
LouisDeFrance PereDe LouisDeBourgogne
LouisDeBourgognePereDe LouisXV

alors

C(BFI,0) = ∅
C(BFI,1) = { LouisXIII GrandPereDe LouisDeFrance }
C(BFI,2) = { LouisXIII GrandPereDe LouisDeFrance,
LouisXIV GrandPereDe LouisDeBourgogne }
C(BFI,n) = { LouisXIII GrandPereDe LouisDeFrance,
LouisXIV GrandPereDe LouisDeBourgogne,
LouisDeFrance GrandPereDe LouisXV } pour n = 3

avec C = (.,GrandPereDe,.). La valeur de C(BFI,1) et C(BFI,2) dépend de l'ordre dans lesquelles sont faites les inférences, ce qui n'aura pas d'importance dans l'expression des contraintes sémantiques.

Pourquoi introduire les classes de cette manière ? En général, on entend par classe une représentation d'une catégorie d'objets *du monde extérieur*. Les objets instances d'une classe sont des représentations de ces objets. Cela présuppose que le système qui utilise ces notions manipule des objets de ce monde extérieur. Or, *le "monde" de Shal est constitué des bases de connaissances qu'on lui donne à traiter*. Donc, les objets que Shal manipule sont *déjà* des représentations, et il ne considère pas ce qu'ils représentent. Autrement dit, Shal travaille sur des "objets informatiques", des choses dans la machine, ce qui permet de donner une définition plus précise de la représentation des connaissances sur ces objets. Shal travaillant sur des bases de connaissances, qui comme tout programme ont un input et un état interne au cours de leur exécution, il est naturel de définir les classes comme nous l'avons fait.

### *4.2.5. Expression des contraintes sémantiques*

Les propriétés sur les classes dépendent a priori du temps et de l'espace. Nous utilisons pour les représenter des formules prédicatives en général quantifiées sur ces deux paramètres.

Nous montrons ici comment exprimer les contraintes sémantiques en nous appuyant sur l'exemple.

- *Instances d'une classe*

Supposons qu'en plus de ce qui est mentionné dans l'exemple, on sache que le fait

(LouisXIII PereDe LouisXIV)

est toujours en base de faits (donc, pour toute base de faits initiale, et à tout moment de l'inférence). Autrement dit, ce fait est une *instance* de la classe des faits (.,PereDe,.).

Si l'on veut traduire cela dans un langage pseudo-prédicatif en n'utilisant que les symboles de la théorie (naïve) des ensembles, on peut écrire :

$\forall$ bfi, $\forall$ n, (LouisXIII PereDe LouisXIV) $\in$ (.,PereDe,.)(bfi,n)

Cependant, nous n'avons pas adopté cette représentation, afin de simplifier le traitement par Shal. Plutôt que de considérer des variables représentant un point dans l'espace et un instant dans le temps, nous considérons toute la *zone spatio-temporelle* où la contrainte est valide. Une zone spatio-temporelle est une partie du produit cartésien BF x N. Ainsi, la contrainte précédente s'écrira

EstUn((LouisXIII PereDe LouisXIV) , (.,PereDe,.), EspaceTemps)

où EspaceTemps est tout BF x N.

EstUn est donc un prédicat, dont le premier argument est un des éléments potentiels d'une classe (objet ou fait), le second argument une classe, et le troisième argument la zone spatio-temporelle pour laquelle le premier argument est effectivement instance du second.

- *Projections d'une classe de faits*

Nous introduisons ici certaines fonctions sur les classes, à savoir les projections des classes de faits selon les directions 1, 2 et 3 des triplets. Ainsi, la projection de la classe (.,GrandPereDe,.) sur la première direction est l'ensemble des objets qui "sont grand-pères" d'un autre objet. Formellement :

$\pi_1$(.,GrandPereDe,.) : BF x N --> Ens
(bfi,n) --> {x | $\exists$ y, (x GrandPereDe y) $\in$ (.,GrandPereDe,.)(bfi,n)}

On note les projections $\pi_1$, $\pi_2$, $\pi_3$.

- *Inclusions de classes*

Prenons maintenant la première contrainte, qui stipule que PereDe porte sur des hommes. Conformément à ce que nous avons fait pour les instances, nous définissons le prédicat InclusClasse portant sur deux classes et dont le dernier argument est une zone spatio-temporelle où l'inclusion est valide :

InclusClasse($\pi_1$(.,PereDe,.) , Homme , EspaceTemps)

On a de même

InclusClasse($\pi_3$(.,PereDe,.) , Homme , EspaceTemps)
InclusClasse($\pi_1$(.,GrandPereDe,.) , Homme , EspaceTemps)
InclusClasse($\pi_3$(.,GrandPereDe,.) , Homme , EspaceTemps)

On peut aussi traduire la contrainte qui interdit d'avoir des grand-pères dans la base de faits initiale :

InclusClasse((.,GrandPereDe,.) , $\varnothing$ , EspaceTempsInitial)

où ø est la classe constante valant toujours l'ensemble vide, et EspaceTempsInitial la "tranche" d'espace-temps à n=0, BF x {0}.

- *Propriétés de monotonie des classes*

Les propriétés de monotonie des classes en fonction du temps et de l'espace sont traduites à l'aide de nouveaux prédicats, MonotoneTemporel, et MonotoneSpatial :

MonotoneTemporel((.,PereDe,.) , 0 , EspaceTemps)
MonotoneTemporel((.,GrandPereDe,.) , + , EspaceTemps)
MonotoneSpatial((.,GrandPereDe,.) , + , EspaceTemps)

Le premier argument de ces prédicats est une classe. Le second argument représente un sens de croissance. Le troisième argument représente la zone pour laquelle cette classe a ce sens de croissance, avec le sens suivant :

MonotoneTemporel(c, s, z) ssi
$\forall$ (bfi,$n_1$) $\in$ z, $\forall$ (bfi,$n_2$) $\in$ z, n1 = n2 $\Rightarrow$ c(bfi,$n_1$) Rel(s) c(bfi,$n_2$)
MonotoneSpatial(c, s, z)ssi
$\forall$ (bfi$_1$,n) $\in$ z, $\forall$ (bfi$_2$,n) $\in$ z, bfi$_1$ Inclus bfi$_2$ $\Rightarrow$
c(bfi$_1$,n) Rel(s) c(bfi$_2$,n)

où Rel(s) est respectivement Inclus, Egal ou Contient selon que s vaut +, 0 ou -.

Nous avons regroupé les propriétés de monotonie en fonction du temps et de l'espace. Pourtant, ces deux notions sont bien distinctes. La monotonie en fonction de l'espace s'apparente plutôt à la "monotonie" (ou plutôt la "non-monotonie") dans les

nombreux travaux en IA sur le raisonnement non-monotone. La monotonie en fonction du temps correspond plutôt à la caractérisation des "états intermédiaires" au cours de l'inférence par lesquels passent les déductions du système.

- *Relations issues de classes de faits*

Considérons maintenant les propriétés des relations PereDe et GrandPereDe. Une relation en théorie des ensembles est une partie du produit cartésien des domaines des arguments de la relation. Nous introduisons une nouvelle fonction, qui associe à toute classe de faits une "relation spatio-temporelle", c'est-à-dire une fonction qui, à tout point de l'espace-temps, associe une relation (binaire) au sens ensembliste. Puisque nous avons des faits constitués de triplets, il y a six manières (autant que de permutations de {1,2,3}) d'associer une relation spatio-temporelle à une classe de faits, et donc six telles fonctions, notées $r_{12}$, $r_{23}$, $r_{31}$, $r_{21}$, $r_{32}$, $r_{13}$. Par exemple :

$$\forall \text{ bfi}, \forall \text{ n}, \quad r_{13}(.,\text{PereDe},.)(\text{bfi},\text{n}) = \{(x,y) \mid (x \text{ PereDe } y) \in (.,\text{PereDe},.)(\text{bfi},\text{n})\}$$

De manière générale, si c est une classe de faits, $r_{ij}(c)(\text{bfi},\text{n})$ est la relation

$$r_{ij}(c)(\text{bfi},\text{n}) = \{(x,y) \mid \exists \text{ R}, \ \text{perm}_{ikj}(x \text{ R } y) \in c(\text{bfi},\text{n})\}$$

où $\text{perm}_{ikj}$ est la permutation $(1,2,3) \rightarrow (i,k,j)$ et $\text{perm}_{ikj}(x \text{ R } y)$ est le triplet obtenu en faisant agir cette permutation sur (x,R,y). Il est clair qu'à $r_{ji}$ correspond l'inverse de $r_{ij}$.

Cette définition est un peu compliquée. Elle généralise et précise ce que l'on entend habituellement par "la relation PereDe", par exemple. Enfin, puisque nos "relations spatio-temporelles" ont la forme générale d'une fonction de l'espace et du temps, on peut les considérer comme des classes (et donc leur appliquer les prédicats définis pour les classes). Ainsi, on peut exprimer qu'une relation est incluse dans une autre.

- *Propriétés relationnelles*

Nous pouvons maintenant introduire le prédicat ProprieteRelationnelle sur les relations spatio-temporelles. Par exemple :

ProprieteRelationnelle($r_{13}(.,\text{PereDe},.)$ , Injection , EspaceTemps)

signifie que, pour tous bfi et n, $r_{13}(.,\text{PereDe},.)(\text{bfi},\text{n})$ est une relation (ordinaire) injective. La forme générale du prédicat ProprieteRelationnelle est donc :

ProprieteRelationnelle(rst, proprel, z)

où *rst* est une relation spatio-temporelle, *proprel* une propriété des relations ordinaires et *z* une zone spatio-temporelle. Son sens est clair.

Les propriétés relationnelles considérées (l'argument *proprel*) sont Fonction, Application[1], Injection, Surjection, Reflexive, Symetrique, Transitive, AntiReflexive, AntiSymetrique, …, en fait toutes les propriétés dont on pourra avoir besoin. Par exemple, rappelons qu'une relation R sur ExF est injective ssi

$$\forall x_1 \in E, \forall x_2 \in E, \forall y \in F, \quad x_1 \, R \, y \text{ et } x_2 \, R \, y \Rightarrow x_1 = x_2$$

Dans l'exemple, on a donc aussi :

ProprieteRelationnelle($r_{13}$(.,PereDe,.) , AntiReflexive, EspaceTemps)
ProprieteRelationnelle($r_{13}$(.,PereDe,.) , Antisymetrique, EspaceTemps)
ProprieteRelationnelle($r_{13}$(.,GrandPereDe,.),AntiReflexive, EspaceTemps)
ProprieteRelationnelle($r_{13}$(.,GrandPereDe,.),Antisymetrique, EspaceTemps)

Il ne reste plus que la contrainte "un individu a au plus 2 grand-pères" à traduire. Pour cela, on considère les cardinaux maximaux et minimaux d'une relation :

CardinalMaximal($r_{13}$(.,GrandPereDe,.) , 2 , EspaceTemps)

De manière générale, on a

CardinalMaximal(rst , k , z) ssi
$\forall$ bfi, $\forall$ n, $\forall$ x, # {y| (x,y) $\in$ rst(bfi,n)} = k

où # est le cardinal d'un ensemble. CardinalMinimal se définit de manière similaire.

Il est clair que ces notions de cardinalité maximale et minimale recouvrent les notions de fonction, application, etc. Par exemple, sont équivalents

ProprieteRelationnelle(rst, Fonction, z)
et
CardinalMaximal(rst , 1 , z)

### *4.2.6. Représentation des contraintes sémantiques*

- *Une représentation possible*

Le langage de la mémoire de travail de Shal n'est pas un langage prédicatif, et il faut y *représenter* les formules prédicatives exprimant les contraintes sémantiques. Nous

---

[1] Avec pour Application un sens différent du sens usuel. Pour nous, une relation est une application ssi la relation inverse est surjective (et non cela, plus le fait qu'elle est une fonction). Autrement dit, une application associe *au moins* une image à tout élément de l'ensemble de départ (et non *exactement* une).

ne décrivons pas ici quelle représentation nous avons choisie, mais quelles difficultés elle pose.

Dans ce qui suit, nous appelons niveau de base, ou niveau représenté, la base de connaissances dont nous voulons traduire les contraintes sémantiques, et niveau méta, ou niveau représentant, la mémoire de travail où nous représentons les contraintes sémantiques du niveau de base.

Le fait de disposer d'éléments de langage pour construire des objets par emboîtement de triplets et de relations entre objets devrait faciliter les choses. Cependant, les contraintes sémantiques portent sur des éléments du langage de la mémoire de travail lui-même, les objets et les faits, et il va donc falloir *représenter aussi* ces éléments. Prenons un exemple, celui de la contrainte sémantique

EstUn((LouisXIII PereDe LouisXIV) , (.,PereDe,.), EspaceTemps)

Il faut représenter la classe (.,PereDe,.) et le fait (LouisXIII PereDe LouisXIV). Ce sont des éléments du ou sur le niveau représenté, et non du niveau représentant. En particulier, il n'y a aucune raison pour que le fait (LouisXIII PereDe LouisXIV) soit présent dans la base de faits où la contrainte sémantique est représentée.

Un moyen simple pour réaliser cela est d'utiliser des objets distincts de ceux utilisés dans le système à représenter, soit F34567 pour (LouisXIII PereDe LouisXIV) et ClasseFaitsPereDe pour (.,PereDe,.). Ces objets sont caractérisés par les faits :

F34567 Element RepresentationDeFait
F34567 PremierObjetFait LouisXIII
F34567 SecondObjetFait PereDe
F34567 TroisièmeObjetFait LouisXIV

ClasseFaitsPereDe Element ClasseDeFaits
ClasseFaitsPereDe PremiereComposanteClasse ObjetShal
ClasseFaitsPereDe SecondeComposanteClasse PereDe
ClasseFaitsPereDe TroisiemeComposanteClasse ObjetShal

RepresentationDeFait et ClasseDeFaits représentent des ensembles, au sens ordinaire du terme. RepresentationDeFait est l'ensemble des objets représentant un fait (potentiel) du niveau de base, et ClasseDeFaits est l'ensemble des objets représentant des classes de faits. Element est la relation d'appartenance ordinaire entre ensembles (naïfs).

ObjetShal est la classe des objets du système étudié, c'est-à-dire de tous les objets de sa mémoire de travail. C'est bien une classe, c'est-à-dire un ensemble dépendant de la base de faits initiale et du moment de l'inférence.

Par ailleurs, LouisXIII, PereDe, etc, sont des objets du système étudié. Ils sont donc implicitement représentés ici par eux-mêmes.

Maintenant, la contrainte sémantique va pouvoir s'exprimer (en utilisant les triplets emboîtés) :

(F34567 EstUn ClasseFaitsPereDe) VraiDansLaZone EspaceTemps

- *Ambiguïtés*

Supposons maintenant que le niveau représenté mentionne pour une raison quelconque l'objet ObjetShal. Comment alors faire la distinction dans la description de ClasseFaitsPereDe entre la *classe* ObjetShal et *l'objet* ObjetShal ? C'est impossible, parce qu'on a représenté par un même objet deux choses totalement distinctes, à savoir un objet du niveau de base et une classe se référant au niveau de base. C'est le même type d'ambiguïté que dans les phrases *Paris est la capitale de la France* et *Paris est un mot de cinq lettres*.

- *Comment adopter une représentation non ambiguë*

Une solution à ce problème consiste à introduire autant d'objets qu'il y a de concepts. ObjetShal restera la représentation de la *classe* ObjetShal, mais nous introduisons un nouvel objet au niveau représentant, disons ObjetShalRepresente, pour représenter l'objet ObjetShal du niveau de base. Il faut faire de même avec les autres objets du niveau de base. On a donc :

F34567 Element RepresentationDeFait
F34567 PremierObjetFait LouisXIIIRepresente
F34567 SecondObjetFait PereDeRepresente
F34567 TroisièmeObjetFait LouisXIVRepresente

ClasseFaitsPereDe Element ClasseDeFaits
ClasseFaitsPereDe PremiereComposanteClasse ObjetShal
ClasseFaitsPereDe SecondeComposanteClasse PereDeRepresente
ClasseFaitsPereDe TroisiemeComposanteClasse ObjetShal

Il faut aussi dire que ces nouveaux objets représentent des objets du niveau de base, soit par exemple

LouisXIIIRepresente Element RepresentationDObjet
LouisXIIIRepresente Represente LouisXIII
ou
ObjetShalRepresente Element RepresentationDObjet
ObjetShalRepresente Represente ObjetShal

RepresentationDObjet est l'ensemble des objets représentant des objets du niveau de base.

- *Synonymes et homonymes*

En obligeant à représenter tout objet de base par un objet distinct au niveau méta, on aboutit à un paradoxe apparent : on est obligé de mentionner quel objet on représente, sans le représenter. En effet, Shal “connaît” certains “mots”, et peut utiliser ses connaissances lorsqu'il repère ces mots dans la base de connaissances sur laquelle il travaille. Cela n'est qu'un problème apparent, il revient à représenter un même objet par deux objets distincts. On aura donc des *synonymes* dans la représentation. Par ailleurs, un autre paradoxe, tout aussi apparent, concerne ObjetShal dans notre exemple : c'est tantôt un des synonymes de *l'objet* ObjetShal, tantôt la *classe* ObjetShal. On a donc des *homonymes*.

Ces paradoxes apparents sont inévitables (à moins que l'on décide de ne jamais représenter ce que les objets représentent, ce qui est très restrictif). Ils ne sont cependant pas gênants. En effet, le contexte, ici les faits où ils sont utilisés, permet de lever l'ambiguïté sur le concept dont on parle lorsque l'on rencontre un objet ayant des homonymes. Ainsi, l'objet ObjetShal représente bien la classe ObjetShal dans le fait

ClasseFaitsPereDe PremiereComposanteClasse ObjetShal

car l'objet ObjetShal n'est pas la représentation d'un objet de base. Par contre, le ObjetShal du fait

ObjetShalRepresente Represente ObjetShal

est bien l'objet de base ObjetShal dès que l'on adopte la convention que l'objet de droite de la relation Represente est un objet de base.

Homonymes et synonymes sont donc inhérents à une représentation *dès que l'on veut pouvoir représenter la représentation*. Dans la représentation proposée ci-dessus, leur usage est minimal. Cependant, elle conduit à multiplier les objets représentant d'autres objets ou des concepts. Pour des raisons d'économie et d'efficacité, il est donc intéressant d'utiliser intensivement les homonymes et les synonymes. C'est ce que nous avons fait dans notre représentation, dont on pourra trouver une description détaillée dans [DOR 90a].

- *Représentation de la représentation*

Le problème de la représentation de la représentation est inhérent à notre approche, puisque Shal s'applique réflexivement à lui-même. Nous détaillons ici les contraintes auxquelles doit obéir une représentation pour permettre une réflexivité complète.

Le problème de la représentation de la représentation se pose de manière double dans notre approche. Premièrement, nous voulons donner à tout niveau méta de Shal la capacité d'examiner les niveaux méta inférieurs. Cela nécessite de pouvoir représenter dans une mémoire de travail de niveau k des informations sur les niveaux k-1, k-2, …, et cela sans ambiguïté. Deuxièmement, une représentation des éléments relatifs à une base de connaissances (faits, règles, contraintes sémantiques, …) revient à créer une sorte de "langage". Ainsi, dans l'exemple ci-dessus, nous

introduisons les "mots" ClasseDeFaits, RepresentationDeFait, Element, etc. Ces mots relèvent de l'expression des contraintes sémantiques, qui prennent aussi en compte les règles qui utilisent ces "mots", c'est-à-dire les contraintes sémantiques elles-mêmes. Tout élément du langage doit donc avoir la capacité d'exprimer les connaissances sur les représentations des éléments du langage. Dans le cas des contraintes sémantiques, cela signifie que l'on doit pouvoir exprimer les contraintes sémantiques des contraintes sémantiques.

Prenons le premier problème. Si on le formalise un tant soit peu, une représentation est une fonction r qui, à tout élément du langage à représenter, associe des termes du langage représentant :

$r : L \rightarrow L'$

Evidemment, le langage représentant est inclus dans le langage représenté. Dans Shal, L est tout le langage de Shal, L' le langage de la mémoire de travail. On a vu que r est en fait une multi-fonction (existence de synonymes) et n'est pas injective (homonymes).

Représenter la représentation consiste à composer r avec elle-même, c'est-à-dire à considérer r o r. Supposons par exemple que Shal travaille sur la base de connaissances "GrandPere". Shal possède donc dans sa mémoire de travail à un certain niveau méta n le fait

(F34567 EstUn ClasseFaitsPereDe) VraiDansLaZone EspaceTemps

qui représente cette contrainte sémantique. Maintenant, il est possible que Shal travaillant au niveau n+1 (par exemple pour contrôler ou observer les activités du niveau n) doive connaître les contraintes sémantiques connues du niveau n. Dans ce cas, il va falloir représenter la représentation de la contrainte sémantique, c'est-à-dire représenter le fait ci-dessus. On aura donc dans la mémoire de travail de niveau n+1 les faits

F45678 Element RepresentationDeFait
F45678 PremierObjetFait T56789
F45678 SecondObjetFait VraiDansLaZoneRepresente
F45678 TroisièmeObjetFait EspaceTempsRepresente

(F45678 EstUn Fait) VraiDansLaZone EspaceTemps

où T56789 est la représentation du triplet (F34567 EstUn ClasseFaitsPereDe) et Fait la classe des faits du niveau de base. Ici, on suppose que la contrainte sémantique est toujours connue du niveau de base.

Il est clair dans cet exemple qu'il n'y a pas de collision entre les informations du niveau n+1 et la représentation des informations du niveau de base.

La règle générale pour qu'il en soit ainsi est la suivante. Plaçons-nous au niveau méta n, et considérons une entité e du langage à représenter. Considérons $r^0$(l'identité), r, $r^2$=r o r, … , $r^n$ appliquées à e. La présence de e au niveau k se traduit par la présence de $r^{n-k}$(e) au niveau n. La règle est que, si e est une entité présente au niveau k, alors $r^{n-k}$(e) doit être reconnu sans ambiguïté au niveau n. Plus précisément, cela signifie que (i) e doit bien être reconnu comme étant présent au niveau k, (ii) que $r^{n-k}$(e) ne doit pas contenir d'autre information que le fait que e est présent au niveau k, et (iii) que la présence de $r^{n-k'}$(e') pour d'autres k' et e' ne doit pas interférer avec la reconnaissance de $r^{n-k}$(e). Ici "reconnaissance" signifie à la fois reconnaissance d'expressions bien formées par un analyseur syntaxique (lors de la lecture de faits sur un fichier) et reconnaissance par les programmes accédant à ces informations, c'est-à-dire les règles.

En pratique, cette règle sert plus comme guide dans la définition de r que pour prouver qu'une r donnée est correcte. En fait, il serait fastidieux et difficile de complètement prouver cette proposition.

Prenons maintenant le second problème, sous l'angle de l'expression des contraintes sémantiques sur les contraintes sémantiques. Cela est en fait relativement simple. Par exemple, la classe de faits (.,VraiDansLaZone,.) a pour projections respectives sur les directions 1 et 3 la classe des propriétés dépendant du temps et de l'espace, et la classe des zones spatio-temporelles. Représenter cette contrainte sémantique est immédiat dès que l'on peut représenter les $\pi_i$ et les inclusions de classe. On pourra dire aussi que cette classe (.,VraiDansLaZone,.) est croissante en fonction du temps et de la base de faits s'il s'avère que les connaissances qui l'utilisent ne suppriment pas de contraintes sémantiques, etc.

La représentation d'un langage dans lui-même a des avantages bien connus. Ainsi, A-M. Duclos [DUC 90] a fait un système qui vérifie qu'une base de faits vérifie bien les contraintes sémantiques (nous avons inclus dans Shal un tel système). Lorsque les contraintes sémantiques sur les contraintes sémantiques sont fournies au système, celui-ci vérifiera aussi les contraintes sémantiques "ordinaires". En fait, il ira même jusqu'à vérifier les contraintes sémantiques sur les contraintes sémantiques, puisque ce sont aussi des contraintes sémantiques "ordinaires". Ces “$CS^2$” servent alors comme programme et comme objet. Et il n'est pas impossible que le système trouve des erreurs, soit qu'une contrainte sémantique "programme" correcte contredise une autre contrainte "objet" incorrecte, soit l'inverse, soit même que les deux soient incorrectes. Randall Davis avait déjà montré dans TEIREISIAS [DAV 82] les avantages de cette approche.

### *4.2.7. Contraintes sémantiques sur la connaissance des contraintes sémantiques*

Nous mentionnons ici un dernier type de contrainte sémantique, encore embryonnaire, sur la connaissance des contraintes sémantiques.

Les contraintes sémantiques sont de nature locale. Pourtant, on voudrait parfois pouvoir exprimer des affirmations de nature globale. Par exemple, on peut dire qu'un certain fait est élément d'une classe dans une certaine zone spatio-temporelle. Mais on ne peut pas dire que l'on connait *tous* les faits d'une classe dans une zone. Pour cela, nous avons ajouté aux contraintes sémantiques une notion d'*ensemble de base connu en extension*. Un tel ensemble, représenté au niveau méta, est comme son nom l'indique un ensemble d'objets du niveau de base, et a la propriété que tous ses éléments sont représentés au niveau méta par la relation Element. Pour un tel ensemble E, le fait qu'un x ne soit pas élément de E est équivalent à l'absence du fait affirmant que x est dans E.

### 4.3. Connaissances de découverte et de vérification de contraintes sémantiques

Comme on l'a vu, les contraintes sémantiques peuvent être nombreuses, et elles sont représentées en base de faits de manière assez complexe. Dans les débuts de Shal, les contraintes sémantiques étaient fournies manuellement au système. Très vite, il s'est avéré impossible de fournir un ensemble de contraintes correctes et, sinon complet, du moins suffisant pour que Shal puisse en tirer des conclusions intéressantes.

Nous avons donc défini un ensemble de connaissances, à la fois pour vérifier des contraintes sémantiques en les confrontant à la base de connaissances, et pour trouver les contraintes sémantiques. Nous avons ajouté à cela des connaissances pour trouver des erreurs, plus ou moins basées également sur les CS.

#### *4.3.1. Comment découvrir les contraintes sémantiques ?*

Il y a deux types de connaissances pour trouver les contraintes sémantiques (sûres et heuristiques), et deux types de matériaux sur lesquels travaillent ces connaissances, la base de connaissances elle-même (règles, expertises et plans) et des exemples de base de faits pour chaque but potentiel.

Les contraintes sémantiques portant sur le comportement dans le temps et dans l'espace des classes d'objets manipulés par une base de connaissances nécessitent avant tout la donnée de la base de connaissances. En effet, il sera facile d'établir, par exemple, qu'une classe de faits est croissante en fonction du temps si aucun fait de cette classe n'est susceptible d'être éliminé par une règle participant à un plan donné, ou qu'elle est constante si aucun fait la matchant n'est déductible.

Cependant, ces connaissances "sûres" ne sont pas suffisantes pour tirer des informations intéressantes. Par exemple, il est impossible en considérant un ensemble d'expertises de déduire qu'une classe de faits doit être initialement vide (c’est-à-dire que tous ses faits sont déduits par le système). Pourtant, il peut y avoir de sérieuses indications de cette propriété. S'il s'avère que certaines règles n'utilisent en prémisse que des classes de faits constantes, ces faits sont obligatoirement présents en base de faits initiale, et les faits qu'elles permettent de déduire en sont probablement absents. Ce raisonnement peut alors être effectué de manière

récurrente, et on obtient ainsi un ensemble de classes de faits susceptibles d'être vides initialement. Il est intéressant de supposer alors que c'est bien le cas.

Cependant, ces conclusions ne sont pas "sûres", elles reposent en quelque sorte sur des raisonnements par défaut. Il faut donc être capable de les confronter avec d'autres "réalités", et de revenir sur elles si nécessaire.

Pour cela, un moyen simple est de confronter ces informations à des exemples, en l'occurrence ici à des bases de faits initiales "typiques".

Nous avons donc défini un ensemble de connaissances qui confronte une base de faits à un ensemble de contraintes sémantiques. Cet ensemble peut fournir deux types d'informations. Il peut trouver une contradiction dans des contraintes sémantiques établies de manière heuristique ou non. Mais il peut aussi s'avérer, particulièrement lorsque la base de faits a été composée manuellement ou est issue d'un programme nouveau, que la base de faits elle-même possède des erreurs. Cet ensemble de connaissances n'est pas capable actuellement de décider entre ces deux alternatives, et c'est au concepteur de trancher en fonction des contradictions qu'elle lui fournit. En effet, même lorsque la contrainte sémantique a été prouvée par des moyens "sûrs", il se peut qu'elle soit incorrecte, simplement parce qu'une expertise est incorrecte. C'est actuellement le concepteur qui corrige les erreurs.

Par ailleurs, certains types de contraintes sémantiques sont difficiles à déceler dans un ensemble d'expertises, par exemple celles qui portent sur la monovaluation des relations. Il est au contraire beaucoup plus facile de les induire à partir de bases de faits (initiales ou non). Nous avons donc également défini un ensemble de connaissances qui effectue ce travail. Pour les propriétés affirmant une borne supérieure sur l'image ou l'image inverse d'une relation, ou celles portant sur la symétrie, etc, d'une relation, c'est relativement simple : il suffit que l'échantillon soit suffisamment représentatif. Par contre, les propriétés affirmant une borne inférieure sont moins simples, puisque par exemple affirmer qu'une relation R sur AxB est surjective nécessite de connaître l'image B de R.

Or, les connaissances qui permettent de "typer" les classes de faits ou les relations travaillent essentiellement à partir de la donnée des expertises : c'est à travers les jointures entre variables des règles que sont apparentes les intersections entre les domaines des relations. Cela montre qu'en définitive les connaissances de découverte et de vérification des contraintes sémantiques sont très imbriquées. Le tableau de la figure 2 le matérialise.

### *4.3.2. Catégories de connaissances*

Chaque case du tableau de la figure 2 représente un ensemble de connaissances traitant les contraintes sémantiques du type de la colonne, en partant de la donnée des expertises ou d'exemples de base de faits. "H" signifie heuristique, "S" sûr et "V" vérification. En fait, chaque case devrait comporter un "V", car tout ensemble de connaissances peut vérifier le type de contrainte qu'elle déduit. En effet, les

contraintes sémantiques sont conservées d'une version de Shal à une autre, et certaines d'entre elles peuvent évidemment être invalidées par les modifications du système. Un "V" dans le tableau signifie une vérification de contraintes déduites par d'autres connaissances. Enfin, les trois flèches indiquent que, par exemple, les connaissances établissant des surjections à partir d'une base de faits initiale se sert des contraintes sur les classes établies à partir de la donnée des expertises.

Nos connaissances de découverte et de vérification des contraintes sémantiques, non contentes d'être potentiellement incorrectes de par leur nature heuristique, sont loin d'être complètes. En pratique, la construction de cet ensemble de connaissances se fait de manière opportuniste. Lorsque nous constatons que Shal a été incapable de déduire quelque chose d'important qu'il aurait pu déduire en disposant des contraintes sémantiques ad hoc, nous ajoutons les connaissances pour qu'il le fasse ensuite.

Enfin, les connaissances de vérification contiennent maintenant certaines heuristiques. Il s'agit là plus généralement du problème du debugging. Lorsque nous avons commis une erreur dans une modification de Shal, nous fournissons - en essayant de nous tenir à cette discipline - des connaissances capables de trouver cette erreur. Par exemple, si on a une CS affirmant qu'une relation R porte sur Ax(B≈C), s'il existe une relation fonctionnelle et injective S entre B et B', et si B' est inclus dans C, alors R est probablement "louche". En pratique, on a confondu B et B' lorsque l'on a écrit les règles mentionnant R. Cette heuristique a été rajoutée au système parce que nous avions commis ce type d'erreur une fois. Lorsque nous l'avons appliquée au système, elle a trouvé dix autres erreurs potentielles, dont une était vraiment une erreur jusqu'alors passée inaperçue.

| | "Typage" des relations | Fonctions, injections, symétries, etc | Surjections, applications | Comportement dans le temps et l'espace | Propriétés initiales |
|---|---|---|---|---|---|
| Expertises, plans | H | V | | S | H |
| Exemples de bases de faits | V, H | H | H | | V |

*Figure 2 : Connaissances de découverte et de vérification des CS*

### 4.4. Connaissances d'analyse du comportement d'une base de connaissances

Le nom donné à ces connaissances est relativement impropre, puisque les contraintes sémantiques portant sur le comportement dans le temps des classes de faits

participent à la description du comportement. On s'intéresse ici, non à la manière dont les données évoluent, mais à la manière dont les expertises sont utilisées. Cependant, les contraintes sémantiques sont utilisées dans ce but.

Ces connaissances tentent d'établir des propriétés des enchaînements entre expertises lorsqu'un certain plan est appliqué. Elles analysent essentiellement ce qu'on appelle le graphe de précédence des règles. Il s'agit du graphe qui relie un conséquent C d'une règle R1 à une prémisse P d'une règle R2 si les faits déduits par C peuvent ajouter ou au contraire supprimer une instanciation de P. On étudie également les graphes similaires entre règles et entre expertises. Nous avons pris l'habitude d'appeler les arcs de ces graphes récursions, bien que cela soit impropre. La justification de cette appellation est que ce sont surtout les circuits de ces graphes qui sont intéressants - d'où le nom de récursion -, en particulier en ce qu'ils sont une source essentielle d'inefficacité.

Actuellement, ces connaissances visent essentiellement à deux buts : établir qu'une récursion est *fictive*, ou au contraire qu'elle est *quasi-obligatoire*.

### *4.4.1. Récursions fictives*

Voici un exemple de récursion fictive :

R1
SI … (N) > 1 …
ALORS P(X) <-- (2 * (N) - 1)

R2
SI … P(U) = (V) … (V) < 1 …
ALORS …

(N), (X), etc, sont des variables. Le conséquent P(X) <-- (2 * (N) - 1) signifie que l'on remplace les éventuelles valeurs de P(X) par (2 * (N) - 1). Ne sont mentionnés que les prémisses et les conséquents qui nous intéressent dans cet exemple.

Syntaxiquement, il y a une récursion entre le conséquent P(X) <-- (2 * (N) - 1) et la prémisse P(U) = (V). Cependant, on voit que la valeur affectée à P(X) dans le conséquent de R1 est nécessairement un nombre plus grand que 1. Or, l'instanciation de (V) dans R2 doit être un nombre plus petit que 1. R1 ne peut donc agir sur R2, du moins via cette récursion. La récursion est *fictive*, une "illusion syntaxique".

Il est clair que les patterns de récursion fictive sont innombrables. Aussi, n'avons nous pas défini une règle par pattern, mais un ensemble de connaissances qui propagent des conditions nécessaires pour qu'une récursion soit possible. Ainsi, dans l'exemple, le raisonnement serait :

"Si P(X) = (2*(N) - 1) peut instancier P(U) = (V), alors (X) doit instancier (U) et (2*(N) - 1), (V). Or, (N) est plus grand que 1, et (2*(N) - 1) est donc aussi plus

grand que 1. Donc (V) doit être plus grand que 1. Par ailleurs, la prémisse (V) < 1 implique que (V) doit être plus petit que 1, et la récursion ne peut donc avoir lieu."

On peut définir un très grand ensemble de règles pour effectuer cette propagation. Ainsi, Marc Porcheron [POR 90] avait un tel ensemble de quelques 500 règles. Nous avons écrit dans les débuts de Shal une telle expertise d'environ 200 règles, puis nous avons provisoirement laissé de côté l'extension de cette expertise. Nous avons laissé dans Shal les connaissances qui permettent à Shal de déduire ses propres récursions fictives (il y a en particulier peu d'aspects numériques dans Shal). Notons à nouveau qu'elles utilisent intensivement les contraintes sémantiques.

### *4.4.2. Récursions quasi-obligatoires*

Les récursions "quasi-obligatoires" sont plus complexes. Elles signifient que, lorsque la règle R1 se déclenche, la règle R2 va presque certainement se déclencher aussi. Un exemple est donné par les deux règles :

```
REGLE EnleveValeurPossible
SI  ElementPossible(x) = (X)
    NombreDeConnections
                ((x) DansLaContrainte ((X) Contrainte (Y))) = 0
ALORS
    ElementPossible(x) = (X) 'Ote

REGLE MaintientLeNombreDeConnections
SI  ElementPossible(x) = (X) 'Ote
    ElementPossible(y) = (Y)
    EstDansLImageDe(x) =
                ((y) DansLaContrainte ((Y) Contrainte (X)))
    NombreDeConnections
                ((y) DansLaContrainte ((Y) Contrainte (X))) = (N)
ALORS
    NombreDeConnections
                ((y) DansLaContrainte ((Y) Contrainte (X))) <-- ((N) - 1)
```

La syntaxe est approximative. Si cela n'est pas clair au lecteur, mentionnons que ces deux règles effectuent ce qu'on appelle la *consistence d'arcs* dans un réseau de contraintes. ElementPossible(x) = (X) affirme que x est peut-être dans le domaine de la variable X, et NombreDeConnections((x) DansLaContrainte ((X) Contrainte (Y))) est le nombre de y éléments possibles de Y liés à X dans la contrainte (X,Y).

Il y a bien une récursion entre R1 et R2, via ElementPossible(x) = (X) 'Ote, et entre R2 et R1, via le nombre de connections. En fait, celle de R2 vers R1 est quasi-obligatoire, dans un sens que nous allons définir. Une condition nécessaire et

suffisante pour que cette seconde récursion ait lieu est que (N) soit instancié par 1. Si c'est le cas, toutes les prémisses de R1 sont satisfaites, car les faits instanciant les prémisses de R2 instancient aussi celles de R1, et ne sont pas tués par le déclenchement de R2. Cela est relativement facile à voir dès que l'on a effectué les propagations dont il était question précédemment. De plus, on sait exactement avec quelles instanciations R1 va se déclencher. Le comportement de R1 et R2 peut donc être (avec encore un peu de travail) résumé dans la série d'actions :

1 - Déclencher R1,
2 - Pour chaque x ôté de X, regarder si R2 est déclenchable
3 - Si, dans la dernière action, (N) était instancié par 1, déclencher R1 avec $x_{R1}/y_{R2}$, $X_{R1}/Y_{R2}$, $Y_{R1}/X_{R2}$. Aller en 2.

Construire cette séquence nécessite, outre les connaissances de propagation déjà mentionnées, des connaissances qui regardent les conséquences de cette propagation lorsque la récursion est a priori possible, c'est-à-dire qui déterminent les prémisses de la règle sur laquelle agit la récursion qui sont satisfaites parce que la première règle est déclenchée.

### *4.4.3. Utilisation des récursions fictives et quasi-obligatoires*

Déterminer les récursions fictives ou quasi-obligatoires a plusieurs buts. Tout d'abord, fournir au moteur ces renseignements lui permet d'optimiser son fonctionnement sur la base de connaissances. En effet, connaître les récursions fictives permet d'éviter de "réveiller" des règles lorsque cela est inutile. Nos expériences montrent qu'il y a en fait proportionnellement assez peu de ces récursions fictives "sensibles". Dans la version de Shal où nous avions mis en œuvre ces connaissances (Shal a beaucoup grossi depuis), Shal se découvrait environ 1500 récursions fictives sur un total de 2000 récursions. Sur ces 1500 récursions, 15 récursions seulement étaient essentielles à une amélioration de l'efficacité. Lorsque l'on supprimait l'accès du moteur à ces 15 informations (il connaissait donc 1485 récursions fictives), l'amélioration relative d'efficacité n'était que de 5 à 10 % environ. Par contre, lui fournir ces 15 informations supplémentaires permettait un facteur de gain de 2.

En ce qui concerne les récursions quasi-obligatoires, nous n'avons pas donné à ce jour au moteur la capacité de les exploiter. En effet, cela aurait exigé des modifications substantielles de son code, ce que nous nous refusons à faire. Nous préférons attendre que Shal ait des connaissances suffisantes, en particulier de compilation et de programmation, pour les exploiter dans une "véritable" compilation de la base de connaissances. L'avantage potentiel est évident : des étapes entières du pattern-matching sont évitées, et une version compilée des expertises prenant ces informations en compte sera bien meilleure. Notre travail actuel vise à donner les connaissances nécessaires pour ce faire.

Malgré tout, la connaissance des propriétés des récursions est essentielle, même si elles ne permettent pas d'améliorer directement l'efficacité du moteur. En effet, Shal

peut exploiter cette connaissance dans d'autres buts. C'est ce nous allons voir en particulier dans la partie suivante sur les patterns d'expertises.

## 4.5. Connaissances de reconnaissance de patterns

### *4.5.1. Origine des patterns de règles*

Nous avons modifié le moteur afin qu'il fournisse un diagnostic sur son fonctionnement lorsqu'il interprète une base de connaissances. C'est un morceau de programme écrit manuellement, et pas une expertise, puisque Shal n'a pas encore les connaissances pour accéder au code du moteur.

Ce diagnostic fournit une trace partielle du déclenchement des règles et des expertises : nombre de déclenchements, nombre d'examens, nombre d'examens infructueux, temps et vitesse d'exécution, etc. Nous cherchions deux choses à travers cette information : les examens inutiles, et les expertises les plus grosses consommatrices de ressources. Pister les examens inutiles permet d'indiquer comment améliorer l'analyse du comportement de la base de connaissances. Idéalement, il ne devrait pas y avoir d'examen inutile. Faire ressortir les expertises gourmandes en ressources permet de focaliser le travail d'extension des connaissances de Shal vers ce qui peut améliorer le plus rapidement son comportement. Nous avons défini pour cela une fonction simple, que nous appelons l'accélération, à savoir le rapport de la vitesse moyenne d'une expertise (nombre de déclenchements / temps passé à l'examen) par le temps passé à l'examen, soit #décl/(temps)$^2$. A priori, les expertises les plus intéressantes sont celles qui vont le plus lentement. Cependant, une expertise très lente, mais très peu utilisée, n'est pas gênante. L'accélération fournit le bon compromis entre les deux critères vitesse (faible) et temps (important). En pratique, l'accélération va de 1 à 50000, ce qui distingue nettement les expertises entre elles.

Cela nous a permis de découvrir que les expertises gourmandes suivaient presque toujours un pattern parmi quelques-uns, que nous avons baptisés "démon", "case of non récursif" et "case of récursif".

### *4.5.2. Règles démons*

Le démon est constitué d'un ensemble de règles ou d'expertises qui se déclenchent systématiquement lorsque d'autres règles sont déclenchées. Typiquement, une règle "transitivant" une relation est de ce type :

```
REGLE Transitive
SI      R(X) = (Y)    R(Y) = (Z)
ALORS   R(X) = (Z)
```

Dès qu'une instance nouvelle de la relation R est déduite quelque part, cette règle est réexaminée, pourvu que sa priorité l'y oblige. De plus, le fonctionnement actuel du

moteur est relativement défavorable dans ce cas. Il faudrait donc transformer ce type de règle en un petit morceau de code qui agisse comme un véritable démon.

### *4.5.3. Case of non récursif*

Un "case of non récursif" est un paquet de règles (en général présent dans une même expertise) dont chaque règle est un cas particulier d'un traitement général sur une même catégorie d'objets. Par exemple, une expertise de Shal construit les mots-clé (ou patterns) de chaque prémisse ou conséquent de la base de connaissances étudiée. Le mot-clé de la prémisse P(X) = (Y) est (P . = .). Les règles de cette expertise ont la forme (à la syntaxe près) :

```
REGLE MotCleDeType
SI  (T) est une prémisse ou un conséquent
    (T) a telles caractéristiques
ALORS
    (T) a le mot-clé (MC)
            (construit à partir des caractéristiques de (T)
                    mentionnées en prémisse).
```

Un traitement adéquat de ce type d'expertise est très différent de celui effectué par notre moteur. Il faudrait faire une boucle sur toutes les prémisses et tous les conséquents de la base de connaissances étudiée, pour chacune ou chacun retrouver ses caractéristiques (certaines caractéristiques sont communes à plusieurs règles), tester par un arbre de décision quelle(s) règle(s) du paquet est matchée par ces caractéristiques, et la (les) déclencher. Ce travail est actuellement effectué par le moteur autant de fois qu'il y a de règles de ce type.

### *4.5.4. Case of récursif*

Un "case of récursif" a une forme similaire, sauf qu'en plus certaines règles du paquet possèdent un conséquent matchant une prémisse de chaque règle du paquet. Les règles d'un tel paquet ont deux formes :

```
REGLE TypeArret
SI  P(X) = (Y)     …
ALORS
        …

REGLE TypeRecursion
SI  P(X) = (Y)     …
ALORS
    P(U) = (V)
        …
```

Les règles de la seconde forme offrent une "définition récursive" de P, celles de la première forme fixent les conditions d'arrêt de cette récursion. Un exemple de tel pattern est constitué des règles de propagation des conditions nécessaires d'une récursion entre règles dont il était question plus haut. Pour mettre à l'étude une récursion entre règles, des expertises installent des faits qui représentent la phrase "Je pose un problème : la récursion i entre R1 et R2 est-elle possible ? Etudier ce problème". Cela prend la forme d'un fait du genre StatutPB(PB) = AResoudre, où (PB) est la description du problème. Des règles entament la propagation en établissant des conditions nécessaires pour que la réponse au problème soit positive :

```
SI      StatutPB(PB) = AResoudre      …
ALORS
        ConditionNecessaire(PB) = (COND)      …
```

En l'occurrence, (COND) pourra représenter quelque chose du genre "la variable X de la règle R2 doit être instanciée par l'instanciation de la variable Y de la règle R1". Puis d'autres règles propagent les conditions nécessaires :

```
SI      StatutPB(PB) = AResoudre
        ConditionNecessaire(PB) = (COND)      …
ALORS
        ConditionNecessaire(PB) = (COND')     …
```

Si une condition nécessaire est violée (ici, récursions fictives) , des règles l'établiront:

```
SI      StatutPB(PB) = AResoudre
        ConditionNecessaire(PB) = (COND)      …
ALORS
        StatutPB(PB) <-- Resolu
        ValeurVeritePB(PB) = Faux
```

Les règles qui établissent qu'un problème est résolu et celles qui propagent les conditions nécessaires forment exactement un pattern de "case of récursif", les prémisses ou conséquents de la forme "ConditionNecessaire(PB) = (COND)" jouant le rôle de "P(X) = (Y)".

En pratique, ce sont les expertises de cette forme qui ont une accélération de l'ordre de grandeur de 1 (elles consomment jusqu'à 2/3 du temps dans Shal). Leur traitement devrait être très différent de celui effectué par le moteur actuel, qui est, dès qu'une condition nécessaire est établie, de réexaminer toutes les règles du paquet pour déterminer lesquelles sont déclenchables sur celle-ci. En examinant les règles de notre exemple, on s'aperçoit qu'elles traitent plusieurs problèmes (instanciations de la variable (PB)) à la fois, mais en parallèle. Par ailleurs, les prémisses portant sur les conditions nécessaires de ces problèmes (variable (COND)) peuvent être largement factorisées. Enfin, toutes les règles ne portent pas sur le même type de condition nécessaire, et une analyse fine (toujours par les mêmes méthodes de propagation) permet de déterminer quelles règles du paquet sont candidates lorsqu'une de ses

règles se déclenche. Tout cela permettrait de remplacer ce paquet de règles par une unique procédure très optimisée par rapport au moteur actuel.

#### *4.5.5. Reconnaissance des patterns*

Nous avons donc fourni à Shal des connaissances pour reconnaître ces différents patterns (le plus important - car le plus gourmand - étant celui de "case of récursif"). Nous ne décrivons pas en détail ici comment fonctionnent ces connaissances. Cependant, il faut noter que le travail effectué par les connaissances éliminant les récursions fictives est crucial. En effet, il y a en général beaucoup plus qu'une récursion principale entre les règles d'un case of récursif. Or, la plupart sont éliminées comme fictives, ce qui permet de dégager le pattern de sa gangue. En fait, l'élimination de ces "récursions secondaires" n'améliore en soi que très peu l'efficacité, mais elle est essentielle à la reconnaissance des patterns.

Depuis que nous avons mis au point ces connaissances, Shal a beaucoup grossi, et il s'est avéré que les expertises ayant ces patterns se sont multipliées, et que ce sont toujours les plus gourmandes. Les connaissances de reconnaissance de Shal se sont révélées robustes, puisqu'elles fonctionnent toujours correctement après que Shal ait quintuplé de volume.

**Remarque :** Le lecteur attentif aura pu noter que nos cas de "mauvais fonctionnement" pourraient être résolus en utilisant un algorithme de pattern-matching ad hoc. Nous ne parlons ici que des cas défavorables au moteur; or, cet autre algorithme pourrait bien être très inefficace là où notre moteur est satisfaisant. Le problème est donc de savoir quand appliquer tel ou tel algorithme. En fait, cette formulation est un peu simpliste, car, dès que l'on commence à décortiquer une base de connaissances, on s'aperçoit que l'on peut faire beaucoup mieux que n'importe quel algorithme général. *Il n'y a pas d'algorithme miracle*.

### 4.6. Connaissances de compilation des patterns de règles

Lorsque les patterns décrits dans le paragraphe précédent sont reconnus, on veut les compiler. Nous pensions lorsque nous avions commencé à mettre en œuvre ces connaissances que la reconnaissance serait délicate, et la compilation aisée. C'est évidemment le contraire qui s'est produit.

Compiler ces patterns signifie traduire chacun d'entre eux dans un programme. Cependant, le moteur d'inférence est encore nécessaire, pour plusieurs raisons. Tout d'abord, toutes les règles ne sont pas classées dans un pattern, et c'est encore au moteur de les interpréter. Deuxièmement, le moteur fait plus qu'exécuter des règles, en particulier il gère la représentation interne de la mémoire de travail pour améliorer le pattern-matching [DOR 87 & 89]. Il a encore d'autres fonctions (entrées-sorties, trace, etc). La solution consiste donc à synthétiser les morceaux de programme correspondant aux patterns, à les "glisser" dans le code du moteur d'inférence, et à

faire en sorte que le fonctionnement normal du moteur soit "dérivé" vers ces programmes lorsque c'est nécessaire. De multiples contraintes pèsent sur la forme de ces programmes; en particulier, il n'est pas possible (sauf à changer radicalement le moteur) de modifier sa gestion de la mémoire de travail. De plus, il faut avoir une certaine idée de ce comment le code du moteur met en œuvre son mécanisme, afin de glisser convenablement les appels aux programmes synthétisés. Et puis, il faut autant que possible partager les variables, etc.

Or, Shal n'a pas encore les connaissances suffisantes pour faire cela. Nous pensons qu'à terme, il est plus rentable de les lui donner d'abord, plutôt que de se lancer dans des modifications manuelles de fonds dans le code du moteur. Cela s'ajoute à - et justifie - notre a priori de fournir des connaissances de programmation plutôt que de programmer nous-même.

Nous avions malgré ces difficultés commencé à fournir des connaissances pour compiler les patterns. Or, d'autres difficultés sont apparues. En effet, s'il était relativement facile d'exprimer des connaissances pour dresser le canevas général de l'algorithme à synthétiser, il fallait en plus des connaissances de programmation pour effectivement transformer ce canevas en programme. C'est de manière évidente un gros morceau, sur lequel nous travaillons actuellement. Nous avions en effet donné certaines connaissances très spécifiques au problème de la compilation des patterns. Nous avons arrêté pour atteindre une plus grande généralité de ces connaissances.

Enfin, cette compilation de patterns a fait apparaître de manière récurrente bien d'autres problèmes, pour la résolution desquels nous avons justement dû fournir des ensembles de connaissances décrits ici : contrôle, micro-langages, etc. Ces problèmes n'ont pas pour origine la difficulté à exprimer des connaissances dont le contenu est de participer à la compilation de règles, mais la "méta-difficulté" qui consiste à rendre Shal capable de traiter ces connaissances de compilation et à nous rendre capable de les fournir. Ainsi, leur contrôle était un problème, bien que ce ne soit pas le contrôle en soi qui compile.

Nous nous sommes donc heurté (une nouvelle fois) à la difficulté de fonds du bootstrapping de connaissances : il ne suffit pas de savoir définir un ensemble de connaissances K1, il faut aussi savoir définir l'ensemble de connaissances K2 qui permettra au système d'utiliser K1.

### 4.7. Connaissances de synthèse de traducteurs de "textes" informatique

#### *4.7.1. But de ces connaissances*

Comme nous l'avons mentionné, nous disposons maintenant de nombreux "micro-langages" s'ajoutant aux règles et aux faits : contraintes sémantiques, plans, etc. Or, les "textes" que nous fournissons à Shal dans ces micro-langages doivent l'être dans la syntaxe de la mémoire de travail. Il faut *représenter* ces connaissances sous forme de faits. Cela est très lourd.

Pour résoudre ce problème, nous avons défini un ensemble de connaissances capables de *synthétiser un programme qui traduise un texte écrit dans une syntaxe S1 dans un texte équivalent écrit dans une syntaxe S2.* Ces connaissances partent d'une spécification des syntaxes S1 et S2, et d'une spécification commune à ces syntaxes de ce que contiennent "conceptuellement" les textes écrits en S1 ou S2.

Pour fixer les idées, supposons que nous ayons défini une syntaxe pratique et agréable pour fournir à Shal les buts et les plans. Cette syntaxe sera S1. Nous voudrions que ces connaissances soient représentées dans la mémoire de travail de Shal, selon donc une autre syntaxe S2. Nous allons donc fournir à Shal une définition de S1 et S2, plus une définition "conceptuelle" de ce qu'est un plan. Nous demandons alors à Shal de nous écrire un programme qui traduise tout texte en S1 dans le texte équivalent en S2.

Une remarque préliminaire : cela est différent de ce qui se fait dans la "théorie des compilateurs". De nombreux chercheurs ont travaillé depuis longtemps pour synthétiser automatiquement des analyseurs syntaxiques, voire pour synthétiser également des traducteurs basés sur la syntaxe. Il s'est avéré que nous n'avons pas trouvé dans cette source de connaissances ce dont nous avions besoin. Nous avons donc dû innover un peu (du moins le croyons-nous).

Par ailleurs, nous ne prétendons pas atteindre à une perfection formelle ou algorithmique. Les connaissances qui synthétisent le traducteur contiennent de nombreuses heuristiques, dont nous osons clamer que nous ignorons la portée. Nous ne savons pas non plus exactement quelle classe de langages est à la portée de notre synthétiseur de traducteurs. Par contre, il fonctionne pour une relativement large gamme de langages, précisément ceux dont nous avions besoin. Enfin, il est clair que notre synthétiseur ne peut fonctionner que si les contenus des textes participant à la traduction sont suffisamment proches "conceptuellement".

Ces connaissances sont maintenant bien constituées, et représentent un volume important. Aussi, pour l'équilibre de cet article, est-il impossible d'en fournir une description détaillée. Nous allons donc seulement donner une idée de ce qu'elles font.

### *4.7.2. Idée de la spécification*

Soient donc deux syntaxes S1 et S2 représentant des aspects d'un même objet. Pour travailler, Shal doit connaître la *syntaxe* de ces textes et avoir une idée de leur *contenu.* La syntaxe est classiquement spécifiée par une bnf[1]. Le contenu est spécifié par ce que nous avons appelé une *grammaire conceptuelle*.

[1] Pour une définition - et beaucoup d'autres choses - sur les bnf, le lecteur peut se reporter à [AHO 86].

Une grammaire conceptuelle utilise des concepts, des attributs, des fonctions, et permet de représenter des notions comme la dimension ou la portée d'un concept. La dimension d'un concept représente ce qui fait qu'une instance du concept est définie de manière unique. Par exemple, une règle de Shal (instance du concept Règle) est parfaitement définie par la donnée de l'ensemble de ses prémisses, de ses actions et de l'expertise où elle apparaît. La portée est une généralisation du *scope* et participe avec la dimension à la définition des instances d'un concept. Donnons trois exemples. Une variable d'une règle de Shal est locale à une règle. Autrement dit, deux variables ayant le même identificateur dans deux règles différentes sont distinctes. On dira que la portée d'une variable est la règle où elle est mentionnée. Un exemple plus complexe est fourni par les variables des langages de programmation comme Pascal ou C. Ici, une variable est définie par la première procédure emboîtante où cette variable est déclarée. C'est sa portée. Un dernier exemple est fourni -encore- par les variables d'une expression prédicative, comme :

$$\forall x\ P(x) \Rightarrow \exists y\ Q(x,y)$$

Le "x" de Q(x,y) est celui de "$\forall$ x". Sa portée est la première expression prédicative emboîtante la mentionnant dans un identificateur.

### *4.7.3. Idée des connaissances de traduction*

Certains liens entre syntaxe et contenu, c'est-à-dire entre bnf et grammaire conceptuelle, sont fournis. Cependant, ces liens ne sont pas complets, car cela reviendrait dans la pratique à une spécification aussi complexe que le traducteur à synthétiser. Shal a donc des connaissances pour compléter ces liens, ou si l'on veut "comprendre" comment le contenu conceptuel est syntaxiquement représenté. Une partie de ces connaissances est heuristique. Elles s'appuient évidemment sur les connaissances contenues dans la grammaire conceptuelle.

Une fois les liens complétés, Shal synthétise un analyseur syntaxique, des "actions sémantiques" glissées dans le corps de l'analyseur syntaxique, et qui construisent les instances des concepts lus, et un décompilateur. Il est à noter que les parties syntaxiques de ces programmes ne respectent pas toujours les canons des analyseurs LL ou LR.

L'écriture de ces connaissances a été rendue possible par notre constatation que les textes informatiques obéissent à deux grands types de syntaxes: les syntaxes "emboîtantes" et les syntaxes "relationnelles". Les syntaxes "emboîtantes" sont celles des langages de programmation. La relation entre un concept et ses composantes se traduit par un emboîtement syntaxique. Par exemple, une fonction C emboîte ses déclarations et ses instructions. Les syntaxes "relationnelles" sont désordonnées, et font le lien entre concepts par des mots réservés. C'est par exemple le cas d'une base de faits, ou en général d'un ensemble de données. Les deux types de syntaxe peuvent être mélangés. Nous n'avons pas rencontré d'exemple de syntaxe (pour autant que l'on se restreigne aux textes informatiques) radicalement différente.

L’auteur serait reconnaissant au lecteur ayant vécu une telle rencontre de le lui signaler.

### *4.7.4. Utilisation de ces connaissances*

Nous avons tenté de fournir ces "connaissances sur les textes informatiques" de manière assez générale. Notre travail n'est pas tout à fait terminé, mais nous pensons déjà qu'elles seront utilisables dans de nombreux cas. Par exemple, nous avons fait ou ferons synthétiser à Shal les traducteurs suivants :

(i) Plans et buts --> Base de faits
(ii) Règles et expertises Shal --> Base de faits
(iii) Base de faits BF --> Représentation de BF en base de faits
(iv) Contraintes sémantiques --> Base de faits
(v) Base de faits --> Contraintes sémantiques
(vi) Pascal et C --> Base de faits
(vii) Représentation de programme en BF --> Pascal ou C
(viii) bnf --> Base de faits
(ix) Définition conceptuelle, liens avec bnf --> Base de faits

Nous n'avons pas écrit à ce jour de programmes équivalents à (i), (iv), (v), (vi), (viii) et (ix). (i) nous servira à fournir les plans plus facilement, ainsi que (iv) lorsque nous devons fournir une contrainte sémantique manuellement (normalement, elles sont découvertes par le système, mais par des connaissances incomplètes). (v) nous servira à examiner les contraintes sémantiques découvertes par Shal. (viii) et (ix) servent à entrer facilement la spécification de la syntaxe et du contenu conceptuel d’un texte. Il y a déjà des programmes équivalents à (ii) et (iii), que nous avons écrits manuellement à partir du moteur, puisque Shal travaille sur une représentation en faits de la base de connaissances à étudier. (vi) nous servira à fournir à Shal l'accès aux instructions du moteur. (vii) sert à supprimer les connaissances écrivant les programmes conçus par Shal dans un langage de programmation existant. Il est à noter que ces connaissances sont utilisées dans la synthèse de traducteurs.

Enfin, nous avons l'intention d'améliorer sérieusement la syntaxe des règles. En effet, les règles utilisent actuellement la représentation en faits des règles, ou des contraintes sémantiques, etc. Cela fait que les règles sont assez lourdes à écrire, maintenir, et relire (les règles de 50 lignes ne sont pas exceptionnelles). Il serait plus simple pour le concepteur d'utiliser des langages plus concis. Nous fournirons donc à Shal une syntaxe étendue des règles, et nous lui ferons synthétiser un traducteur-préprocesseur qui transforme des règles dans la syntaxe étendue dans l'ancienne syntaxe. Les mécanismes d'inférence resteront inchangés, mais nous pensons que ce “sucre syntaxique” nous fournira un surcroît de productivité substantiel.

### 4.8. Connaissances de manipulation de programmes, de structures de données, et utilisant des programmes existants

Ces connaissances sont encore très peu développées dans Shal. Celles que nous avons mises en œuvre sont capables de traiter les proto-programmes issus du synthétiseur de traducteur de textes informatiques. Mais elles restent très spécifiques à ce problème. Les augmenter sérieusement constitue la prochaine tâche de notre agenda personnel.

En particulier, plusieurs thésards aux Etudes et Recherches d'EDF travaillent sur la synthèse de programme par des bases de connaissances, avec succès. Nous espérons pourvoir utiliser le résultat de leur travail.

### 4.9. Modèles d'expertises

Ces connaissances, en cours de mise en œuvre, construisent de manière encore assez simple un modèle de chaque expertise. Grâce au résultat de l'analyse du comportement de la base de connaissances étudiée, les classes de faits d'entrée et de sortie de chaque expertise sont établies, ainsi que leur évolution lorsque l'expertise est exécutée. Lorsque deux classes de faits C1 et C2 en entrée (resp. en sortie) se subsument, i.e. sont telles qu'il existe une classe C déjà rencontrée telle que C1 et C2 sont incluses dans C, alors C est substituée à C1 et C2 comme entrée (resp. sortie) de l'expertise. On construit ensuite la pseudo-règle dont les prémisses correspondent aux entrées et les conséquents aux sorties. Il faut établir les jointures entre classes en conservant celles des règles de l'expertise. Il s'agit en fait d'un processus de généralisation, encore élémentaire.

Le rôle attendu de ces modèles d'expertises est double. Premièrement, les connaissances "de défaut", qui remettent en cause un fait lorsqu'il s'est avéré incorrectement déduit, pourront s'aider de ces modèles pour retrouver comment ce fait a été déduit. Deuxièmement, nous espérons développer un ensemble de connaissances pour vérifier les plans fournis au système, et les retrouver au moins en partie.

### 4.10. Connaissances pour la gestion des versions de Shal

Il s'agit à ce jour essentiellement de la mise en œuvre des mécanismes de remise en cause d’un fait lorsqu'il s'avère avoir été incorrectement déduit. On a vu qu'un tel phénomène est "normal" dans Shal, non seulement à cause des erreurs dans les connaissances dues au concepteur, mais aussi à cause de la nature heuristique de nombre de ses connaissances. Ces connaissances savent actuellement remettre en cause un fait lorsqu'il a été déduit dans la même "session". Cela signifie par exemple que, si nous lançons Shal de manière conjointe sur les buts de découvrir et vérifier les contraintes sémantiques à partir de la base de connaissances et d'une base de faits exemple, et si une contrainte déduite à partir de la base de connaissances s'avère

incorrecte au regard de la base de faits, alors la contrainte sera éliminée (si le concepteur le décide).

Mais il nous faudrait beaucoup plus. En pratique, nous modifions en permanence Shal, et il est impossible de complètement le réanalyser (grâce à lui-même) à chaque modification. Aussi, nous n'exécutons Shal que sur certains buts et sur une partie de sa base de connaissances en fonction des modifications opérées. Par ailleurs, nous conservons (manuellement) certains résultats de certaines exécutions (comme les contraintes sémantiques), que nous réutilisons ensuite. Lorsqu'un fait incorrectement déduit est à supprimer, nous n'avons pas d'autre alternative que de refaire toutes les déductions antérieures manuellement. Certaines d'entre elles prenant jusqu'à 10 minutes de temps CPU (2'30 simplement pour charger la représentation de Shal dans la mémoire de travail), cela est très pénible.

La solution tient évidemment dans le fait de rendre Shal incrémental, au sens où il gère au moins en partie son propre développement. Nous avons donc ajouté comme tâche à notre agenda personnel de fournir des connaissances à Shal pour ce faire. Cela n'est pas simple, au moins en ce que cela implique plusieurs catégories de connaissances. Il faut évidemment que Shal connaisse la "logique" de son propre développement, par exemple ce pourquoi on le fait travailler sur tel but, comment s'enchaînent les buts, quand et comment il est nécessaire ou inutile de refaire un travail déjà effectué, etc. Mais il faut aussi fournir des connaissances pour *stocker intelligemment* la base de connaissances, ses représentations et ses déductions sur disque, et pour *puiser* dans cette base de données les informations dont on a besoin pour une tâche à effectuer. Tout cela est loin d'être résolu.

## 5. Les leçons de Shal

### 5.1. Bénéfices accumulés

Les avantages du bootstrapping de connaissances ont été expliqués dans les publications de Jacques Pitrat. Ils tiennent dans la formule *Dès qu'une connaissance est fournie au système, il peut l'utiliser*. Au début de la construction du système, cet avantage se fait très peu sentir. Il a trop peu de connaissances dans trop peu de domaines pour que celles-ci trouvent une opportunité de s'appliquer utilement. Cependant, après un certain temps, les connaissances fonctionnent avec une certaine synergie, qui libère le concepteur du système de tâches de plus en plus nombreuses. C'est vraiment la formule des intérêts composés : au début, on ne gagne pas grand-chose, mais sur le long terme la propriété exponentielle du gain se fait sentir.

Nous pensons que nous sommes à la charnière de ces deux périodes. Jusqu'à un passé récent, les connaissances de Shal n'étaient que d'une aide limitée dans son développement. Deux domaines de connaissances ont maintenant apporté des intérêts substantiels, ou sont en passe de le faire : la découverte d'erreurs et la traduction de textes informatiques.

Les connaissances qui tournent autour des contraintes sémantiques fournissent les matériaux pour mettre en œuvre des connaissances pour trouver des erreurs. Nous l'avons dit, nous essayons maintenant de nous tenir à la discipline qui consiste, à chaque fois que nous trouvons nous-même un nouveau type d'erreurs, à ajouter les connaissances pour que le système découvre lui-même ce type d'erreur à sa prochaine occurrence. Shal trouve effectivement maintenant de nombreuses erreurs, ce qui nous fait gagner un temps précieux.

La traduction de textes informatiques a fait avancer le système de manière encore plus visible. A l'origine, nous nous sommes lancé dans la mise en œuvre de ces connaissances essentiellement pour ne plus avoir à représenter manuellement dans la base de faits les contraintes sémantiques et les plans. En fait, nous avons développé en parallèle les connaissances pour découvrir les contraintes sémantiques, et nous pensons sérieusement à celles qui trouvent les plans. Ces connaissances de traduction de textes informatiques auraient donc fort bien pu être mort-nées *si elles n'avaient un champ d'application suffisamment général pour servir à des objectifs initialement non prévus* : représentation en mémoire de travail du code du moteur, amélioration de la syntaxe des règles, suppression de la partie du moteur qui lit les règles et les faits, sortie sous forme sympathique des déductions du système, etc.

Dans d'autres domaines, comme la compilation des connaissances de Shal, nous sommes à ce jour moins avancés. L'expérience nous a en effet montré qu'une option "à court terme", qui consisterait à mettre en œuvre un "compilateur minimal" pseudo-algorithmique était une impasse (nous revenons sur ce problème ensuite). Nous ne pouvons donc pas éviter d'attaquer les problèmes de fonds de manipulation de programmes et de structure de données. C'est de toute évidence un problème difficile et ambitieux, sur lequel nous sommes plusieurs à travailler. Mais il est sûr que les dividendes de ce travail seront élevés, sans préjuger de la forme qu'ils prendront.

### 5.2. Difficultés

Cependant, si elle promet des bénéfices importants, la méthode du bootstrapping de connaissances présente de nombreuses difficultés, dont la plupart nous semblent inédites dans le développement d'un système d'IA.

#### *5.2.1. Ne pas attaquer les sous-problèmes séparément*

La première difficulté consiste en une frustration ressentie en permanence : on aurait besoin des connaissances que nous sommes en train de mettre en œuvre ou que nous mettrons en œuvre plus tard pour les mettre en œuvre plus facilement. Il en est des exemples évidents; ainsi, il serait agréable de disposer d'un compilateur de règles efficace pour le mettre au point. Mais il est facile ici de se faire une raison. Par contre, il est plus difficile d'admettre *qu'il est presque impossible de prendre un sous-problème séparément*. Ainsi, Shal commence à être relativement important (environ 800 règles regroupées en 150 expertises, une quinzaine de plans, quelques centaines de relations - avec quelques milliers de contraintes sémantiques -, une dizaine de

langages, et 22000 lignes de code du moteur, plus quelques programmes annexes). Chacun peut se rendre compte qu'il devient pénible de gérer cet ensemble disparate. Nous nous disons chaque jour qu'il serait vraiment agréable de disposer d'un système pour gérer tout cela. Mais la construction de ce système *dépend de ce que nous faisons actuellement*. Par exemple, les connaissances de gestion du développement de Shal utiliseront à coup sûr les connaissances de programmation.

Une stratégie pour circonvenir à ces ennuis est de travailler sur plusieurs sous-problèmes à la fois, et d'y apporter en priorité les solutions partielles qui facilitent notre travail. C'est pourquoi nous développons les connaissances autour des contraintes sémantiques, sur les modèles de règles, etc, en parallèle avec les connaissances de programmation. Mais il est psychologiquement et culturellement difficile de se convaincre qu'il *faut* résoudre un problème *mal*.

### *5.2.2. Fournir des connaissances générales*

Cependant, mal résoudre un problème ne signifie pas nécessairement fournir des connaissances trop spécifiques. Cela peut être intéressant; c'est ce que nous avons par exemple fait pour les expertises qui synthétisent le contrôle procédural. Nous en avions un besoin criant, et ces expertises étaient peu importantes (3 expertises, 11 règles). L'investissement valait donc la peine. Mais il y a une contrepartie : presque à chaque extension des capacités du langage de plans, nous avons dû modifier cette expertise.

Lorsque le problème est moins criant et/ou que l'investissement est plus lourd, il est nécessaire d'approfondir les connaissances fournies au système. On tombe alors dans le problème bien connu - et qui n'est pas propre à notre approche - d'acquisition de connaissances générales. Ainsi, nous travaillons encore sur les connaissances de traduction pour mettre en œuvre des connaissances générales et indépendantes qui "placent" des actions sémantiques dans l'analyseur syntaxique. En effet, "glisser" de nouvelles fonctions dans un programme existant est un problème que nous rencontrons sous de multiples autres aspects : auto-programmation des méta-prédicats, remplacement de parties de code du moteur comme l'analyseur syntaxique, par exemple.

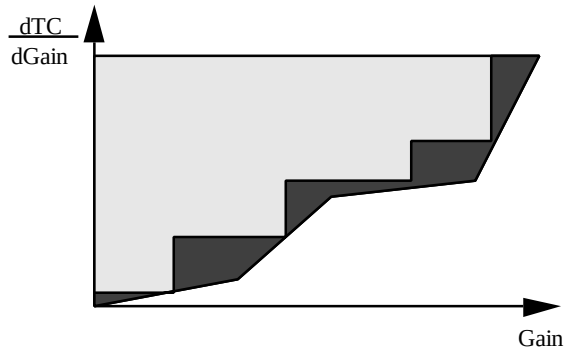
### *5.2.3. Ne pas tenter de griller les étapes*

Nos premières tentatives de compilation de règles par une base de connaissances ont consisté à reprendre avec Jean-Yves Lucas le travail de sa thèse sur le système KLAXON [LUC 89]. Lucas avait, à partir du système SIREN qui synthétise des programmes pour résoudre des problèmes combinatoires, construit KLAXON qui synthétisait une procédure par règle, en traitant le pattern-matching comme un problème combinatoire. Nous avons modifié la base de connaissances de KLAXON pour adapter le langage de règles qu'elle prenait en compte à celui dans lequel elle était exprimée, Boojum. Il nous est resté 400 règles, effectivement capables de complètement compiler des règles. Cependant, comme le souligne Lucas dans sa

thèse, cette base de connaissances n'avait pas les connaissances pour gérer les index sur la mémoire de travail qui étaient mises en œuvre implicitement dans le moteur de Boojum. Cela faisait que, dès que la mémoire de travail devenait importante, une base de connaissances était plus efficace lorsque interprétée par Boojum que sa forme compilée par KLAXON. Or, si l'on veut compiler KLAXON à l'aide de lui-même, il faut le représenter en mémoire de travail, ce qui fournit de manière évident une base de faits initiale importante (environ 200 faits par règle, soit 80000 faits initiaux). Il est quand même gênant de compiler un compilateur pour obtenir un compilateur moins efficace.

Une possibilité pour résoudre ce problème serait de fournir au système les connaissances qui lui manquent pour gérer correctement les index, disons 400 autres règles. Mais on aurait alors un système de 800 règles qui devrait compiler la représentation de 800 règles. Nous avons estimé que, compte tenu de la mise au point, cela nécessiterait plusieurs dizaines d'heures de Cray. Cela ne nous a pas paru raisonnable.

La raison de cet échec est que nous avons voulu faire une étape de bootstrap trop grande et trop directe. Pour schématiser ce point de vue, on peut considérer le graphique de la figure 3.




*Gain = gain dû à l'ajout de connaissances* *TC = temps de conception*

*Figure 3 : Le tunnel du bootstrapping*

Ce graphique idéal représente en abscisse le gain obtenu grâce aux capacités cognitives du système, et en ordonnée le gain de productivité retiré des connaissances déjà mises en œuvre (on peut considérer aussi les ressources consommées par le traitement du système par lui-même). On a représenté deux options extrêmes de bootstrapping. La première consiste en un "grand saut", la seconde en une série de "petits sauts". Les temps de mise en œuvre respectifs sont représentés par les aires hachurées (on n'a pas hachuré l'aire commune sous la courbe). Il est clair que les "petits sauts" sont bien moins coûteux. Pourquoi ? Simplement parce que le saut n+1 tire avantage des connaissances présentes au saut n. Le but est donc de rendre l'aire foncée la plus petite possible. C'est ce que nous appelons le "tunnel du bootstrap". En pratique, un "grand saut" peut être tellement exigeant qu'il est tout

simplement impossible vu le système existant. C'était le cas de notre premier compilateur.

### *5.2.4. Fournir des connaissances pour utiliser les connaissances que l'on fournit*

C'est une autre forme du conseil qui consiste à ne pas traiter un sous-problème en profondeur indépendamment des autres.

Après l'échec de notre premier compilateur, nous avons repris la construction de Shal à zéro en procédant cette fois par petites étapes. Une des premières catégories de connaissances que nous avons fournies concernait l'élimination des récursions fictives entre règles. Or, la forme des règles qui effectuent la propagation des conditions nécessaires pour qu'une récursion puisse avoir lieu est (cf. paragraphe 4.5.4.) :

```
SI      StatutPB(PB) = AResoudre
        ConditionNecessaire(PB) = (COND)      …
ALORS
        ConditionNecessaire(PB) = (COND')     …

SI      StatutPB(PB) = AResoudre
        ConditionNecessaire(PB) = (COND)      …
ALORS
        StatutPB(PB) <-- Resolu
        ValeurVeritePB(PB) = Faux
```

Ces règles sont tout à fait récursives, et il n'y a évidemment pas possibilité d'éliminer ces récursions. Autrement dit, leur contenu est de traiter un problème P, et le traitement de leur forme exige de savoir traiter le problème ¬P !

De manière générale, lorsque l'on fournit un ensemble de connaissances K1, on sait quel va être son contenu (ce qu'il est sensé faire). Par contre sa forme est largement imprévisible. Pour traiter cette forme, il faut en général ajouter des connaissances K2. Le processus peut continuer, jusqu'à avoir des connaissances K1, K2, …, Kn, qui sont capables toutes ensemble de se traiter elles-mêmes. En pratique, il est très difficile de savoir quels K2, …, Kn seront nécessaires lorsque notre intention initiale est seulement d'ajouter K1. De plus, l'ensemble K1, …, Kn peut atteindre une taille trop grande pour être simplement traitable par le système existant (cf. tunnel du bootstrap). Nous n'avons pas trouvé d'autre moyen pour résoudre ce problème que la méthode essais-erreurs.

## 6. Conclusion (provisoire)

Cet article a présenté l'état de développement du système Shal, qui doit par ses connaissances être capable de compiler des bases de connaissances. Nous avons montré que cela nécessitait, si l'on voulait réellement exprimer ces connaissances en

rupture avec une approche pseudo-algorithmique, des connaissances dans des domaines qui ne concernent pas directement la compilation de règles. Shal sait déjà trouver des propriétés intéressantes de la base de connaissances à compiler, y trouver des erreurs, traiter les problèmes syntaxiques, utiliser un contrôle fourni de manière déclarative, et remettre en cause certaines de ses déductions. Dans la prochaine étape, Shal aura des connaissances de programmation, des connaissances pour se gérer sur l'échelle de temps de sa propre construction, et pour trouver lui-même son contrôle.

Shal ne sait pas encore compiler des règles. Lorsqu'il le saura, il saura en même temps faire beaucoup d'autres choses.

**Remerciements**

Nous travaillons seul sur Shal, mais nous prenons beaucoup d'idées à d'autres. Tout d'abord, la perspective du bootstrapping de connaissances a été affirmée de manière systématique par Jacques Pitrat. Jean-Yves Lucas a été un des premiers à faire un compilateur de règles écrit en règles. Les connaissances sur les contraintes sémantiques, leur découverte et la découverte d'erreurs qu'elles permettent doivent énormément à Philippe Lafon, qui prépare une thèse sur la validation-vérification des bases de connaissances. Les "contraintes sémantiques sur les contraintes sémantiques" étaient déjà présentes dans la thèse d'Anne-Marie Duclos. Enfin, les connaissances de programmation sont déjà largement développées dans les systèmes de synthèse de programme par base de connaissances de Bruno Ginoux, Jean-Philippe Lagrange et Akim Boudaoud, qui préparent tous trois une thèse sur ce sujet. Nous remercions enfin le referee anonyme pour ses remarques et ses critiques sur la première version de cet article.

**Références**

[AHO 86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: principles, techniques and tools.* Addison-Wesley, 1986.

[AYE 87] Marc Ayel. *Détection d'incohérences dans les bases de connaissances : le système SACCO.* Thèse d'Etat de l'Université de Savoie, 1987.

[CHA 90] Evelyne Charles. *Méthodes logiques pour la détection d'incohérences et autres anomalies dans les bases de connaissances : le système MELODIA.* Thèse de l'Université Paris 6, octobre 1990.

[DAV 82] Doug Lenat & Randall Davis. *Knowledge-based systems in Artificial Intelligence.* Mc Graw-Hill, 1982.

[DOR 87] Jean-Luc Dormoy. *Résolution qualitative : complétude, interprétation physique et contrôle. Mise en œuvre dans un langage à base de règles : Boojum.* Thèse de l'Université Paris 6, décembre 1987.

[DOR 89] Jean-Luc Dormoy. *Amélioration de l'efficacité du pattern-matching dans le langage à base de règles Boojum.* Convention IA, janvier 1989. Hermès, éditeur.

[DOR 90a] Jean-Luc Dormoy. *Représentation et utilisation des connaissances : contraintes sémantiques sur une base de faits Shal.* Note EDF HI-21/7185, novembre 1990.

[DOR 90b] Jean-Luc Dormoy. *Predicting the Behavior of a Knowledge Base.* Cognitiva'90, November 1990, Madrid (Spain). Also to be published by Elsevier.

[DOR 91] Jean-Luc Dormoy. *Meta-knowledge, autonomy and (artificial) evolution: some lessons learnt so far.* First European Conference on Artificial Life, ECAL'91, December 1991, Paris (France). Also to be published by MIT Press.

[DUC 90] Anne-Marie Duclos. *Méthodologies de représentation des connaissances appliquées à l'automatique.* Thèse de L'Institut National Polytechnique de Lorraine, Septembre 1990.

[FOU 87] Jean-Marc Fouet. *Utilisation des connaissances pour améliorer l'utilisation des connaissances : la Machine GOSSEYN.* Thèse d'Etat de l'Université Paris 6, septembre 1987.

[HER 85] Jean-François Héry. *Convergence commutative d'une base de connaissances : les logiciels LRC appliqués à une maquette d'aide à la conduite d'une centrale nucléaire.* Thèse de 3ème cycle de l'Ecole Centrale des Arts et Manufactures, 1985.

[LAF 90] Philippe Lafon. *A Descriptive Model of Predicates for Verifying Production Systems.* ECAI Workshop on Validation and Verification of Knowledge-based Systems, August 1990, Stockholm (Sweden).

[LAI 87] Laird, Newell & Rosenbloom. *SOAR: an architecture for general intelligence*. Artificial Intelligence 33, 1987, pp. 1-64.

[LAU 86] Jean-Louis Laurière. *Un langage déclaratif : SNARK.* Technique et Science Informatique, mars 1986.

[LUC 89] Jean-Yves Lucas. *Génération automatique de programmes par règles et compilation de bases de règles. Application à un système expert de diagnostic de signaux courants de Foucault.* Thèse de l'Université Paris 6, février 1989.

[MAE 87] *Meta-Level Architectures and Reflection.* Springer-Verlag, 1987. Maes & Nardi, Editors.

[MAZ 90] Philippe Mazas. *Acquisition de connaissances de conception : le système SYSIFE.* Thèse de l'Université Paris 6, juillet 1990.

[MIN 89] S. Minton, J.G. Carbonell, C.A. Knoblock, D.R. Kuokka, O. Etzioni & Y. Gil. *Explanation-based learning: a problem-solving perspective*. Artificial Intelligence 40, 1989, pp. 63-118.

[PAR 88] Yannick Parchemal. *SEPIAR : un système à base de connaissances qui apprend à utiliser efficacement une expertise.* Thèse de l'Université Paris 6, décembre 1988.

[PIT 84] Jacques Pitrat. *MACISTE, un système qui utilise des connaissances pour utiliser des connaissances.* Colloque d'Intelligence Artificielle du Groupe de Recherche C.F. Picard, septembre 1984, Aix en Provence.

[PIT 85] Jacques Pitrat. *MACISTE, ou comment utiliser un ordinateur sans utiliser de programme.* Colloque d'Intelligence Artificielle du Groupe de Recherche C.F. Picard, septembre 1985, Toulouse.

[PIT 86] Jacques Pitrat. *Le bootstrap sur l'utilisation des connaissances.* Colloque d'Intelligence Artificielle du LAFORIA, septembre 1986, Strasbourg.

[PIT 87] Jacques Pitrat. *La déprocéduralisation.* Colloque d'Intelligence Artificielle du LAFORIA, septembre 1987, Cæn.

[PIT 88] Jacques Pitrat. *Ceci n'est pas un article.* Colloque Franco-Espagnol d'Intelligence Artificielle de la "Méta-Connaissance", septembre 1988, Areny de Mar (Espagne).

[PIT 89] Jacques Pitrat. *Qu'est-ce qu'un individu au royaume de la connaissance?* Colloque d'Intelligence Artificielle sur la Méta-Connaissance, septembre 1989, Le Mans.

[PIT 90] Jacques Pitrat. *Métaconnaissance, futur de l'Intelligence Artificielle.* Hermès, 1990.

[POR 90] Marc Porcheron. *Utilisation de méta-connaissances pour la compilation de règles de production.* Thèse de l'Université Paris 6, janvier 1990.

[ROU 88] Marie-Christine Rousset. *Sur la cohérence et la validation des base de connaissances : le système COVADIS.* Thèse d'Etat de l'Université Paris-Sud, 1988.

[STA 90] Thèse de l'Université Paris 6, 1990.

[VIA 85] Michèle Vialatte. *Description et applications du moteur d'inférence SNARK.* Thèse de l'Université Paris 6, mai 1985.