CHAPITRE 2

MODÉLISATION BASÉE SUR L'INTELLIGENCE ARTIFICELLE ET LES SYSTÈMES EXPERTS

2 Modélisation basée sur l'intelligence artificielle et les systèmes experts

2.1 Principes d'intelligence artificielle

Avant d'entrer dans le vif du sujet et de présenter les systèmes experts, une question semble s'imposer d'elle-même : qu'est-ce que l'intelligence artificielle (IA) ? La réponse à cette interrogation est assez complexe, car l'intelligence humaine est en soit plutôt difficile à appréhender, alors que dire de celle d'entités artificielles. En fait, selon [Russell et Norvig, 2003], il existe quatre grandes écoles de pensée, lorsqu'il vient le temps de définir la notion d'intelligence artificielle :

- L'étude des systèmes qui pensent comme des humains.
- L'étude des systèmes qui se comportent comme des humains.
- L'étude des systèmes qui pensent rationnellement. On dit qu'un système est rationnel s'il est capable d'effectuer sa tâche correctement, à partir de ce qu'il connaît.
- L'étude des systèmes qui se comportent rationnellement.

Chacune de ces écoles est associée à des domaines d'application particuliers de l'intelligence artificielle, notons : les algorithmes génétiques, la reconnaissance de la parole, les réseaux de neurones, la robotique, les systèmes experts, les systèmes multi-agents, la traduction de langages, ainsi que la vision et la perception. Pour notre part, nous allons principalement retenir la deuxième définition, puisque nous nous sommes intéressés à concevoir un système capable d'émuler le comportement d'opérateurs humains et d'automates machines.

Historiquement les ordinateurs ont toujours excellés à accomplir des tâches répétitives, telles de complexes calculs arithmétiques, ou le stockage et la recherche d'information à travers une source de données. Ces tâches ont comme dénominateur commun un algorithme. Il est donc possible de les accomplir, par le biais d'une suite d'instructions qui produisent la réponse attendue. Pour leur part, les humains ont de grandes aptitudes pour solutionner des problèmes qui impliquent des abstractions et des symboles, plutôt que des nombres. On peut penser à des problèmes tels, la planification d'activités ou la compréhension d'un texte. De ce point de vue, l'intelligence artificielle est donc la science qui permet aux ordinateurs de représenter et manipuler les symboles des

humains, dans le but de solutionner une problématique, difficilement concevable sous une forme algorithmique [Gonzalez et Dankel, 1993].

La section suivante a pour but d'introduire la notion de système expert, puisqu'il s'agit d'un des domaines de l'IA qui a été appliqué avec le plus de succès dans un contexte de prise de décision [Turban et Aronson, 2000].

2.2 Principes d'un système expert

Les systèmes experts (de l'Anglais « Experts Systems ») ont vu le jour d'un point de vue théorique, vers la fin des années cinquante. Ils se concrétisèrent véritablement à un niveau pratique, à partir des années soixante et soixante-dix. À ce titre, on peut penser à DENDRAL [Lindsay, 1980], un système d'interprétation permettant d'identifier des composants chimiques, ainsi qu'à MYCIN [Buchanan et Shortliffe, 1984] qui diagnostiquait et spécifiait des traitements pour les troubles sanguins. Une nouvelle approche a donc été introduite, pour développer des systèmes d'informations capables de solutionner des problèmes, issus d'un domaine bien précis. Ces systèmes se comportaient en partie comme un spécialiste humain, par exemple, un chimiste en ce qui concerne DENTRAL et un immunologiste dans le cas de MYCIN. Au cours des décennies suivantes, les systèmes experts firent l'objet d'un certain engouement, particulièrement vers la fin des années soixante-dix et au début des années quatre-vingt. Ils permirent de solutionner des problématiques reliées à de nombreux champs d'application, citons : la finance, les sciences naturelles et sociales, l'ingénierie, la conception, ainsi que la santé [Gonzalez et Dankel, 1993].

L'origine des systèmes experts découle d'observations des chercheurs en intelligence artificielle (IA). Ceux-ci constatèrent qu'il était avantageux d'axer leur méthode de résolution d'un problème, sur la connaissance du domaine d'application, plutôt que sur un éventail de connaissances générales, appliquées à plusieurs domaines. On considère souvent à tort, que l'appellation système expert est un synonyme d'intelligence artificielle. Cette perception erronée, s'explique du fait que les systèmes experts forment l'une des branches de l'IA qui a eu le plus de succès. Du moins en ce qui a trait aux applications pratiques.

Le docteur Edward Feigenbaum, un éminent chercheur de l'université Stanford qui est spécialisé en Intelligence artificielle, définit les systèmes expert comme suit :

> Des programmes intelligents qui utilisent la connaissance et des mécanismes d'inférences, pour solutionner des problèmes qui sont difficiles, au point de requérir une expertise particulière, lorsqu'ils sont considérés par des êtres humains. NDLR : traduit de l'Anglais [Feigenbaum, 1982]

Donc, l'appellation de ce type de système d'information, s'explique du fait que les connaissances et les méthodes, qui leur permettent de tirer des conclusions, s'apparentent à celles d'un expert du domaine d'application concerné. Par expert, on entend une personne qui possède des connaissances ou des habiletés très spécifiques, c'est-à-dire inconnues par le commun des mortels. À ce titre, on peut penser au savoir contenu dans un ouvrage spécialisé. Ainsi, dans le cadre de la modélisation et de la simulation de procédés industriels, nous nous intéressons aux systèmes experts, parce qu'ils permettent d'émuler la capacité qu'a un spécialiste humain à prendre des décisions.

Mentionnons que le fait de recourir à un système expert, tout comme n'importe quel domaine de l'IA, n'est pas sans risque, puisqu'il implique un niveau de complexité accru. Par exemple, il faut généralement impliquer des spécialistes en IA [Gonzalez et Dankel, 1993], pour travailler de concert avec l'équipe de modélisation ou encore, former les membres de cette dernière, pour qu'ils puissent intégrer efficacement ces technologies au modèle. C'est pourquoi il importe de déterminer si un problème requiert réellement un système expert. Nous allons donc mettre l'accent, tout au long du mémoire, sur le type de problématique qu'un système expert doit normalement adresser.

Il est notable de préciser que les termes *système à base de connaissances* ou *système expert à base de connaissances* sont souvent utilisés, au détriment de l'appellation *système expert*. En fait, pour certains auteurs, ces substantifs sont tous des synonymes. Cependant, d'autres affirment qu'il s'agit d'un abus de langage, puisqu'un *système expert* doit seulement comporter le savoir particulier d'un spécialiste humain, alors qu'un *système à base de connaissances* peut être associé à la fois à des connaissances générales et spécifiques d'un domaine particulier [Giarratano

et Riley, 1998]. Comme ce débat sémantique déborde du contexte de notre recherche et que de toute façon, nous considérons davantage une expertise spécifique, à savoir celle des spécialistes du cycle de vie des anodes, l'appellation système expert est utilisée.

À la base du système expert, se trouve l'utilisateur qui saisit un ensemble de faits, afin d'obtenir un avis ou une expertise. Le *moteur d'inférence* (de l'Anglais « Inference Engine ») compare les faits saisis, avec le savoir de la *base de connaissances* (de l'Anglais « Knowledge Base ») du système expert, dans le but d'établir des correspondances entre eux. Ainsi, les données de la base de connaissances permettent au moteur d'inférence de tirer les conclusions appropriées, en appliquant un processus de raisonnement, dont il sera question un peu plus loin. Les conclusions sont retournées à l'usager sous la forme d'une expertise [Giarratano et Riley, 1998]. La figure ci-dessous, illustre le fonctionnement élémentaire d'un système expert.

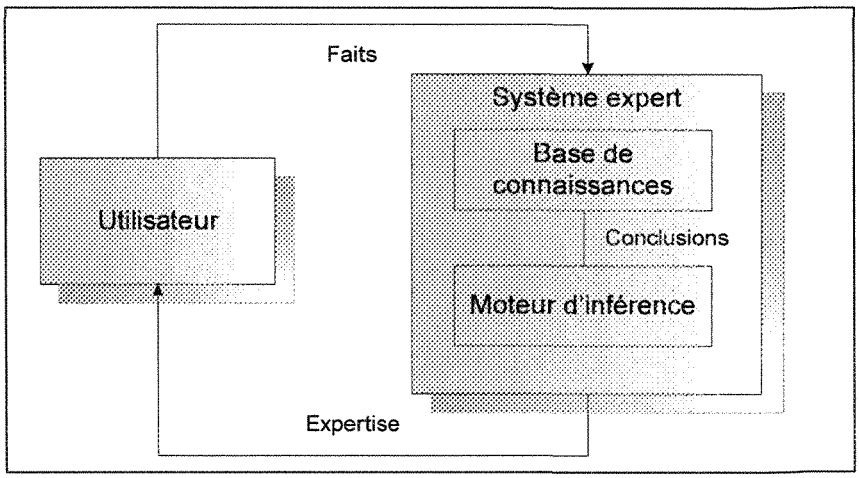

Figure 13 : représentation d'un système expert élémentaire

Au cours des sections suivantes, nous allons donc introduire avec plus de détails, la notion de système expert et leur architecture. Ces sections ont pour objectif d'éclaircir le rôle des systèmes experts, dans un contexte de modélisation de procédés industriels. Ainsi, suite à cette description, nous allons être en mesure de discuter, à partir de la section 3.4.2, de leur intégration avec notre modèle et des avantages qu'ils lui confèrent.

2.2.1 Extraction et représentation des connaissances

La base de connaissances est un élément clé du système expert. Elle contient des informations structurées qui représentent la connaissance spécifique au domaine d'application. L'acquisition des connaissances est effectuée par un spécialiste, appelé *ingénieur de la connaissance* (de l'Anglais « Knowledge Engineer »). C'est à partir d'entretiens exhaustifs, réalisés en compagnie d'un expert du domaine, que ce spécialiste peut extraire le savoir. Par la suite, il devient possible de l'encoder dans le système expert. Ce processus de conception est appelé *ingénierie des connaissances* (de l'Anglais « Knowledge Engineering ») [Gonzalez et Dankel, 1993].

Il existe plusieurs formalismes pour représenter le savoir au sein de la base de connaissances. Chacun d'eux ont leurs avantages et leurs inconvénients, qui doivent être pris en considération par l'ingénieur de la connaissance. Il est cependant notable de mentionner que les règles de type *si ... alors*, s'avèrent être l'une des représentations les plus courantes du savoir de la base de connaissances :

si <Prémisse, antécédent ou *partie gauche* (LHS)>
alors <Action, conséquence ou *partie droite* (RHS)>

Une règle, aussi appelée *règle de production*, est en quelque sorte une commande conditionnelle qui exécute une action dans certaines situations. Les systèmes experts, dont la base de connaissances est conçue à partir de règles, sont appelés des *systèmes à base de règles* ou *systèmes de production* (de l'Anglais « Rule-Based Systems » ou « Production Systems »). Ils permettent d'établir des conclusions selon des *prémisses*, c'est-à-dire la partie conditionnelle, située entre le *si* et le *alors* de la règle. Une prémisse est formée d'une ou plusieurs *conditions*, appelées aussi *motifs* (traduction de l'Anglais « Pattern »). Celles-ci sont généralement séparées par des opérateurs logiques (*et*, *ou*). Souvent, ces conditions contiennent des variables liées à des faits de la mémoire de travail. La prémisse est aussi appelée l'*antécédent*, la *partie gauche* de la règle ou *LHS*, de l'Anglais « Left Hand Side ». Quant aux actions exécutées lorsqu'elle est déclenchée, elles forment la *conséquence*, la *partie droite* ou *RHS* (« Right-Hand Side »).

Généralement, les actions de la RHS servent d'une part, à affirmer ou rétracter des faits et d'autre part, à signaler des informations à l'usager [Friedman-Hill, 2003] [Giarratano et Riley, 1998].

Concernant le procédé étudié, il est possible d'exprimer la connaissance à propos du routage des anodes à l'aide de règles. L'exemple qui suit nous montre une suite de règles génériques formées de plusieurs conditions. Elles permettent de déterminer si une *anode* quelconque (représentée par la variable *A*) traitée par un *poste* (variable *P*), peut être transférée vers un *convoyeur* (variable *C*), situé en aval de ce dernier. Le transfert se réalise une fois que le *temps de traitement requis* sur (*A*) est atteint et si le *poste* (*P*) est *actif* :

```
// Désactivation du poste si un arrêt est planifié.
(R1)  si
            Poste P est "actif" et
            Type d'action TA est un "arrêt" et
            Action planifiée de type TA survient actuellement sur P
      alors
            Rendre poste P "inactif"

// Activation du poste si un arrêt planifié est terminé.
(R2)  si
            Poste P est "inactif" et
            Type d'action TA est un "arrêt" AR et
            Action planifiée de type TA prend actuellement fin sur P
      alors
            Rendre poste P "actif"

// Vérification si l'anode A traitée par le poste P peut être transférée
// vers le convoyeur C.
(R3)  si
            Poste P est "actif" et
            Poste P est "en cours de traitement" et
            Poste P est en train de traiter anode A et
            Convoyeur C en aval du poste P n'a pas
                  atteint sa capacité maximale et
            Le temps de traitement écoulé au poste P sur anode A est
                  supérieur au temps de traitement requis au poste P
      alors
            Rendre le poste P en "arrêt de traitement"
            Retirer anode A du poste P
            Ajouter anode A au convoyeur C
            Mettre à zéro le temps de traitement écoulé au poste P

// Incrémentation du temps de traitement si le poste P est en train de
// traiter une entité quelconque.
(R4)  si
            Poste P est "actif" et
            Poste P est "en cours de traitement"
```

```
alors
      Incrémenter le temps de traitement écoulé de P
```

À ce titre, les règles et les autres représentations de l'expertise font brièvement l'objet de la section 2.5.1. Au cours de celle-ci, nous allons d'avantage discuter du concept de base de connaissances. De plus, à la section 3.4.2.1, nous allons traiter de la représentation de la connaissance à partir de règles de production.

2.2.2 Traitement par inférence des connaissances

Le *moteur d'inférence* est le second concept fondamental d'un système expert. C'est à partir de la connaissance du domaine d'application, que le moteur d'inférence effectue un raisonnement ou infère des conclusions, au même titre qu'un spécialiste humain infèrerait la solution d'un problème. Inférer consiste donc à conclure qu'un ensemble de faits est vrai, en vertu d'un premier ensemble de faits et d'une implication logique, tous deux étant initialement reconnus comme vrais [Gonzalez et Dankel, 1993]. Cette assertion correspond en réalité à une règle logique, appelée *modus ponens*. Elle consiste à déclarer : lorsque les affirmations **p** et $(\mathbf{p} \Rightarrow \mathbf{q})$ sont vraies, alors **q** est aussi vrai [Hu, 1989] [McAllister et Stanat, 1977]. Voici un exemple d'inférence associé à l'enchaînement des étapes d'une anode cuite : si l'entrepôt des anodes est vide (**p**) et qu'une anode cuite est attendue à la phase de scellement (**q**), nous inférons qu'une nouvelle anode sortie du four devra être acheminée directement au centre de scellement (**r**). Donc, si d'une part $(\mathbf{p} \wedge \mathbf{q})$ est vrai et que d'autre part $(\mathbf{p} \wedge \mathbf{q}) \Rightarrow \mathbf{r}$ l'est aussi, alors **r** est vrai.

Il existe plusieurs méthodes afin de mettre en œuvre le mécanisme d'inférence. Celles-ci dépendent généralement de la représentation des connaissances. Dans le cas d'un système à base de règles, les deux méthodes d'inférences les plus communes sont le *chaînage avant* (« Forward Chaining ») et le *chaînage arrière* (« Backward Chaining ») [Giarratano et Riley, 1998] [Gonzalez et Dankel, 1993]. D'ailleurs, lors de la section 2.7 qui aborde le rôle du moteur d'inférence, nous allons définir plus clairement les différents mécanismes d'inférences possibles.

2.3 Coquille de développement de systèmes experts

Une *coquille de développement* est un outil facilitant la conception de systèmes experts (de l'Anglais « Expert System Shell »). Plus précisément, elle est formée d'un moteur d'inférence, de composantes fonctionnelles d'un système expert typique et d'une base de connaissances totalement dépourvue de contenu [Friedman-Hill, 2003]. C'est durant les années soixante-dix qu'on élabora la première coquille de développement : EMYCIN, pour « Empty » MYCIN [Buchanan et Shortliffe, 1984]. Ses concepteurs décidèrent de dépouiller MYCIN de sa connaissance spécifique, à propos des troubles sanguins, afin de pouvoir résoudre des problématiques issues de différents domaines d'applications. Du même coup, on obtenait un outil générique de conception de systèmes experts.

Au cours de ce mémoire, nous allons principalement discuter de deux coquilles de développement de systèmes experts, à savoir le « C Language Integrated Production System » [CLIPS] et le « Java Expert System Shell » [Jess]. Elles sont dotées de fonctionnalités couvrant presque tous les besoins de l'ingénieur de la connaissance et du concepteur de systèmes expert, notons : l'affirmation, la rétraction et la modification de faits ou encore, la définition de nouvelles règles. Il est même possible de les intégrer à des applications C/C++ (CLIPS) et Java (Jess), puisqu'elles disposent toutes deux d'une interface de programmation (API) [Giarratano et Riley, 1998] [Jess 7.0 Manual]. D'ailleurs à la section 3.4.2, nous allons discuter d'avantage des principales fonctionnalités qui justifient leur usage dans le cadre de la simulation de processus.

2.4 Architecture d'un système expert

Les éléments d'un système expert typique sont décrits au sein de la Figure 14 [Friedman-Hill, 2003] [Luger et Stubblefield, 1998]. Notons que dans ce cas-ci, puisque le concept qui représente le savoir s'avère être des règles, la figure illustre un *système à base de règles* (« Rule-Based System ») ou *système de production* (« Production System »).

Globalement, son but est d'apparier les règles de la base de connaissances, aux faits de la mémoire de travail, afin d'inférer des conclusions. Lorsque toutes les conditions d'une règle sont

satisfaites par un ensemble de faits, on dit qu'elle devient *active*. Par la suite, une priorité est assignée à chacune des règles activées, afin de spécifier leur séquence d'exécution. Cet ensemble de règles ordonnées s'appelle l'*agenda des activations*. Le fonctionnement que nous venons de mentionner est celui d'un moteur d'inférence à *chaînage avant*. Normalement, celui-ci doit effectuer un ou plusieurs *cycles d'inférences*. Un cycle consiste à réaliser chacune des étapes que nous venons de décrire. Les sections suivantes ont pour but d'éclaircir le rôle des différentes composantes qui interviennent lors d'un cycle d'inférence.

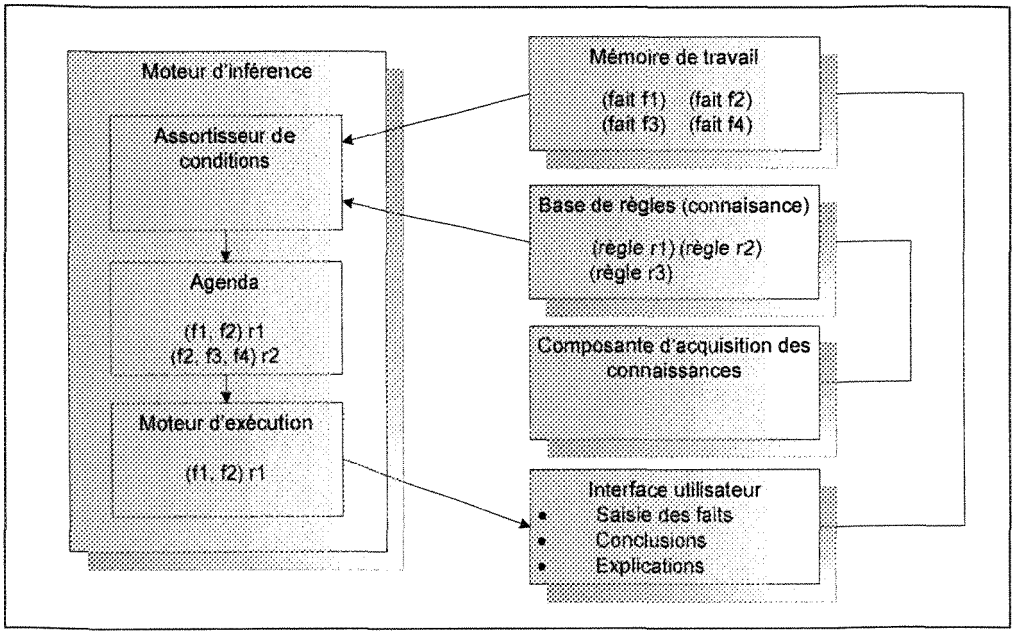

Figure 14 : architecture d'un système expert à base de règles

2.5 Base de connaissances

La *base de connaissances* (« Knowledge Base ») englobe tout le savoir inhérent au domaine d'application, afin de solutionner des problèmes spécifiques. Il existe trois catégories de connaissances [Krishnamoorthy et Rajeev, 1996] :

- Compilée : c'est-à-dire issue d'experts du domaine, de manuels spécialisés, d'études scientifiques, de spécifications reconnues, etc.

- Qualitative : basée sur des méthodes empiriques, des théories approximatives, des modèles causals, des heuristiques, le bon sens, etc.
- Quantitative : appuyée sur des théories mathématiques, des techniques numériques, etc.

Les deux premières catégories de connaissances sont aussi divisées en sous-groupes : *déclarative* et *procédurale* [Turban et Aronson, 2000]. La connaissance déclarative adresse les propriétés physiques du domaine, elle permet donc de le décrire. La connaissance procédurale réfère aux approches de résolution employées dans le cadre d'un problème. C'est le domaine d'application qui détermine la nature exacte et les interactions possibles de la connaissance. Par exemple, la connaissance qui permet de déterminer l'état du système de manutention des anodes, tel les postes arrêtés, les convoyeurs à pleine capacités, est dite déclarative. Quant à celle impliquant des prises de décisions, comme les actions qui permettent de combler un manque d'anode à la phase de scellement, elle est dite procédurale.

La *connaissance heuristique* est une catégorie spéciale, parfois conjointe aux trois précédentes. Elle est basée sur une ou plusieurs heuristiques, par exemple une règle empirique généralement reconnue [Gonzalez et Dankel, 1993]. Cette connaissance s'avère fondamentale, puisque généralement les problèmes soumis aux systèmes experts n'ont pas de solutions algorithmiques connues. De toute façon, l'usage d'un système expert devrait être reconsidéré, lorsque l'on sait qu'un problème a une solution exacte, telle un algorithme [Friedman-Hill, 2003].

2.5.1 Formalismes de représentations de la connaissance

Quelque que soit la catégorie de la connaissance impliquée dans la résolution d'une problématique, il prime avant tout de la représenter sous une forme compréhensible par un programme informatique. Il existe plusieurs formalismes de représentation. Parmi les plus courants notons [Gonzalez et Dankel, 1993] [Krishnamoorthy et Rajeev, 1996] : la *logique prédicative*, les *réseaux sémantiques*, les *cadres* (« Frames ») et les *règles de production*, aussi simplement appelées *règles*.

2.5.1.1 Logique prédicative

La *logique prédicative* ou *logique du premier ordre* est basée sur des faits, aussi nommés axiomes, c'est-à-dire des propositions qui sont soient vraies ou fausses [First-order logic]. Une proposition exprimant une relation est un prédicat. Il peut s'agir d'une relation entre plusieurs objets. Voici quelques exemples de prédicats :

```
<nom du prédicat>(<objet> +) :

        etat(four-R, actif)
        evenement(four-R, inactif, 245)
        temps-simule(245)
```

Une base de connaissances recourant à la logique prédicative contient des implications conditionnelles qui agissent à titre de méthodes d'inférences, citons la déduction, l'abduction et l'induction. Ces méthodes appliquées aux prédicats permettent de déduire des axiomes additionnels. Par exemple, à partir des prédicats ci-dessus et de la proposition suivante :

```
∀ P, ∀ T,
        [etat(P, actif) ∧ evenement(P, inactif, T) ∧ temps-simule(T)
        ⇒ etat(P, inactif)]
```

L'axiome qui suit est déduit :

```
etat(four-R, inactif)
```

Un des désavantages de la logique prédicative est qu'elle est entièrement basée sur des valeurs booléennes. Il n'y a donc aucun mécanisme pour gérer l'incertitude. Aussi, on constate qu'une fois qu'un axiome est déduit, il n'est plus possible de le modifier ou de le supprimer. La base de connaissances de tels systèmes a donc tendance à s'accroire énormément [Krishnamoorthy et Rajeev, 1996]. Certains systèmes experts qui s'appuient sur PROLOG [Clocksin et Mellish, 1987], une implémentation de la logique prédicative, résolvent ce problème en permettant la rétraction d'axiomes. Cependant, on ne peut plus véritablement parler de logique prédicative pure.

La logique prédicative, telle qu'implémentée par PROLOG, est implicitement déclarative. L'ingénieur de la connaissance n'a donc pas un droit de regard sur la méthodologie de recherche et d'appariements d'axiomes. En fait, on utilise un mécanisme d'inférence à *chaînage arrière*, dont il

est question à section 2.7.2. Or, dans certaines situations, celui-ci peut occasionner un temps d'exécution accru [Gonzalez et Dankel, 1993].

2.5.1.2 Les réseaux sémantiques

Les *réseaux sémantiques* furent introduits par [Quillian, 1968]. Un réseau sémantique est un graphe dirigé [Cormen et al., 1994] représentant la connaissance d'un domaine particulier. Chaque nœud du réseau sert à décrire les concepts ou les entités du domaine. Quant aux arêtes, elles illustrent les associations sémantiques qui interviennent entre les entités [Gonzalez et Dankel, 1993]. Les réseaux sémantiques servent d'avantage à exprimer la connaissance déclarative du domaine, afin de connaître ses propriétés physiques et sa taxonomie [Krishnamoorthy et Rajeev, 1996]. Il s'agit d'un inconvénient, puisqu'il est difficile de les utiliser dans un contexte où la connaissance procédurale est requise. Comme on l'a vu, ce type de connaissance est impliqué dans la résolution d'une problématique. Mentionnons que l'exploration d'un vaste réseau sémantique peut être assez longue, notamment lorsqu'il y a de nombreuses relations [Gonzalez et Dankel, 1993].

2.5.1.3 Les cadres et les daemons

Les *cadres*, de l'Anglais « frames » sont un autre formalisme de représentation de la connaissance. Ils furent introduits par [Minsky, 1974]. Ils regroupent et décrivent des faits, c'est-à-dire les entités d'un domaine, à partir d'attributs communs. Ces attributs descriptifs sont appelées les *fentes* (« slots »). On leur affecte une *valeur* ou un *remplisseur* (« filler »). Les fentes peuvent être subdivisées en *facettes* (« facets »). Ces dernières spécifient des informations comme : l'ensemble des valeurs possibles, le type d'une fente ou sa valeur par défaut [Durkin, 1994].

Également, les facettes référencent presque toujours des méthodes qui retournent ou affectent les valeurs des fentes. Ces méthodes sont appelées *daemons* [Gonzalez et Dankel, 1993]. Le rôle des daemons s'apparente à celui des règles d'un système de production, puisqu'il exprime à la fois le savoir déclaratif et procédural, en effectuant diverses manipulations à partir des

valeurs de fentes. Notons qu'on distingue deux types de cadres : les classes, décrivant les caractéristiques générales et les instances, c'est-à-dire un ensemble de cadres affiliés à une même classe. Ordinairement, les fentes de classes n'ont pas de valeur assignée, alors que celles des instances en ont. Parce que les instances héritent des attributs de leur classe ancêtre, il devient possible de construire une hiérarchie de cadres [Durkin, 1994].

Comme on peut le constater la notion de cadre [Krishnamoorthy et Rajeev, 1996], s'apparente à celle d'objet [Fowler, 2004] [Gilbert et McCarthy, 2001]. D'ailleurs, l'approche orientée objet est en grande partie basée sur les travaux effectués par des chercheurs en intelligence artificielle, durant les années soixante-dix [Champeaux et al, 1993].

Bien que la connaissance issue d'un système à base de cadres soit beaucoup plus structurée et organisée que celle des autres formalismes, son usage implique certains désavantages. Notamment, parce qu'il est difficile de décrire la connaissance heuristique à partir de cadres. En fait, les règles de production sont un mode de représentation beaucoup plus naturel, pour cette catégorie de connaissance [Gonzalez et Dankel, 1993].

2.5.1.4 Les objets et les règles de production

La connaissance basée sur les cadres implique que leurs fentes (attributs) et leurs daemons (méthodes) soient considérés comme deux concepts indépendants. Ainsi, une fente quelconque peut être manipulée par un daemon qui se trouve dans un autre cadre. L'interprétation d'un attribut spécifique peut donc différée d'un cadre ou d'un daemon à l'autre [Gonzalez et Dankel, 1993].

Par définition les *objets* ont une fonction beaucoup plus unificatrice que les cadres, puisqu'ils allient à la fois les *données* (attributs) et les *opérations* (méthodes), découlant d'un problème particulier [Müller, 1996]. En plus, le paradigme orienté objet comporte des caractéristiques intéressantes dans un contexte de représentation de connaissances. Il a été question de celles-ci à la section 1.7.1, à savoir : l'abstraction, l'encapsulation, l'héritage et le polymorphisme [Müller, 1996]. D'ailleurs, l'encapsulation permet d'éviter certaines incohérences associées aux cadres. En effet, les fentes d'un cadre peuvent être manipulées par n'importe quel daemon. Or, dans un

contexte de formalisation objet, l'encapsulation résout ce problème, car chaque attribut est référencé individuellement par des méthodes de leurs classes. Du même coup, tous les attributs ont une signification unique.

Toutefois, tout comme pour les cadres, le problème de la représentation de la connaissance heuristique demeure. C'est pourquoi la plupart des coquilles de systèmes experts ont une approche de représentation hybride, c'est-à-dire qu'ils recourent à la fois à des objets et des règles de production [Durkin, 1994]. Celles-ci construisent leurs conditions à partir des attributs de classes. Ces conditions sont appariées à des objets qui symbolisent les faits de la mémoire de travail. Les règles ont donc une structure similaire au pseudo-code suivant :

```
si
      poste.etat = "actif" et
      poste.nom = action.applique_a et
      action.type = "arrêt"
alors
      poste.etat = "désactiver"
```

La coquille de système expert CLIPS illustre bien cette approche hybride. Elle possède une extension appelée COOL (CLIPS Object-Oriented Language) [Giarratano, 2002] [Riley et al, 2005]. Elle permet d'implémenter des objets et de les référencer à partir de prémisses de règles. Quant à la coquille Jess [Jess 7.0 Manual], elle offre d'encore plus grandes possibilités d'interaction entre les règles et les objets. Nous en reparlerons d'ailleurs à la section 3.4.4.

Finalement, mentionnons qu'en ce qui concerne notre modèle, les règles sont un choix tout à fait naturel, pour représenter le fonctionnement du cycle de vie des anodes. Nous allons d'ailleurs revenir sur les forces de ce formalisme à la section 3.4.2.1. Puisque les règles de production font d'avantage l'objet de notre intérêt, le reste de la discussion traite principalement de ce formalisme.

2.5.2 Réseau d'inférence de règles

Un graphe s'avère particulièrement utile pour illustrer une base de connaissances formée de règles. En effet, une telle structure permet au spécialiste d'organiser la connaissance acquise, de rechercher les relations entre ses différentes sources, de vérifier sa cohérence et de la convertir

facilement sous une forme abstraite, reconnue par le système expert [Krishnamoorthy et Rajeev, 1996].

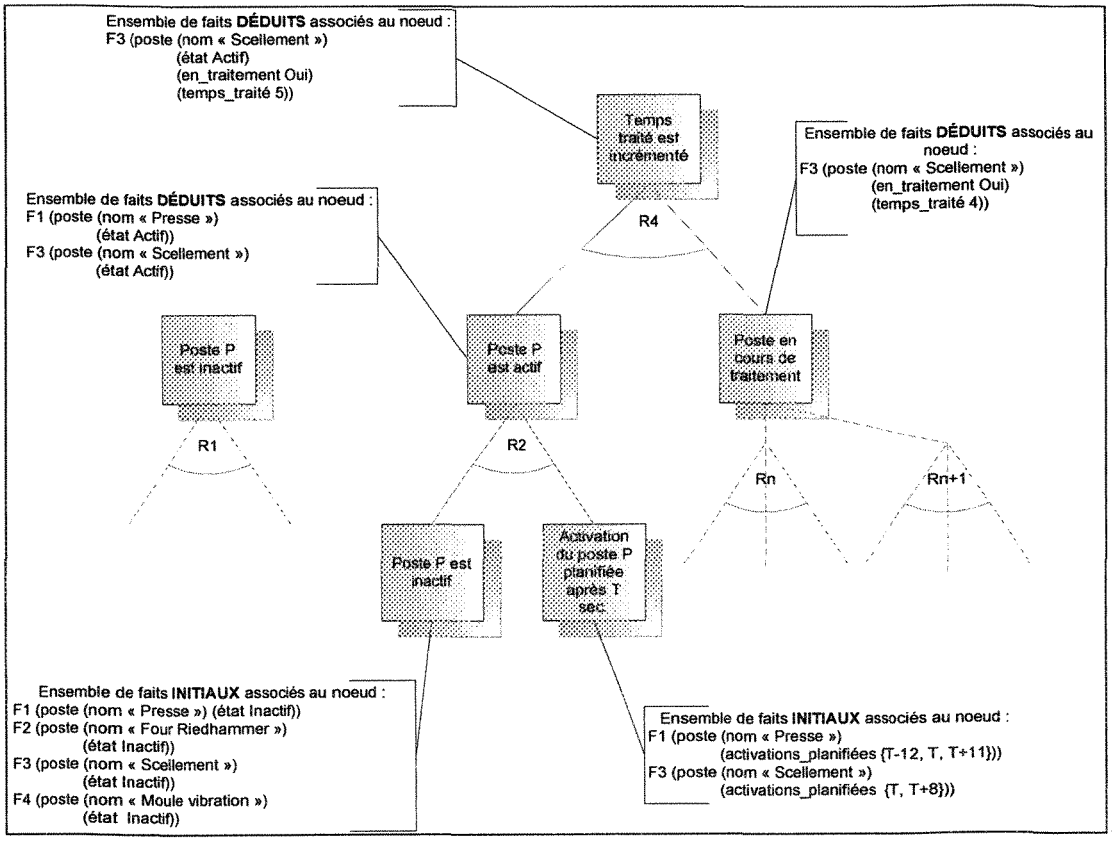

Figure 15 : réseau d'inférence d'une partie d'un problème

Puisque les conclusions tirées à partir des règles sont souvent des faits qui peuvent satisfaire les prémisses d'autres règles, il devient possible de visualiser ce graphe comme un réseau de règles et de faits interconnectés. On appelle cette structure un *réseau d'inférence* (de l'Anglais « Inference Network » ou « Inference Net ») [Gonzalez et Dankel, 1993]. Chaque nœud du graphe correspond à un paramètre associé à un ensemble de faits initiaux ou déduits durant le processus d'inférence. Ainsi, les nœuds peuvent d'une part, constituer la *partie gauche* ou *droite* d'une règle et d'autre part, permettre de référencer certains faits associés à des aspects ponctuels du problème. Quant aux jonctions entre les nœuds, elles servent à représenter les règles. La figure

ci-dessus nous montre le réseau d'inférence, illustrant une partie de l'exemple tiré de la section 2.2.1.

2.5.3 Acquisition et formalisation des règles

Bien que le réseau d'inférence soit une représentation tout à fait commode pour comprendre un problème et son domaine d'application, il ne correspond pas exactement à la représentation interne du système expert. Bien souvent une base de connaissances formée de règles est adjointe d'un *compilateur* [Friedman-Hill, 2003]. Celui-ci permet de restructurer la représentation des règles du système expert. Ainsi, le moteur d'inférence est en mesure de les traiter plus efficacement. C'est pourquoi la structure logicielle de la connaissance dépend en grande partie de l'algorithme du mécanisme d'inférence. L'algorithme de Rete [Forgy, 1982] constitue une des implémentations les plus courantes. Il nécessite que le compilateur construise une structure indexée, semblable à un graphe, appelée *réseau de Rete*. Chaque nœud entrant du réseau correspond à un motif d'une règle, à laquelle on peut associer des faits. Nous allons discuter de cet algorithme à la section 2.8.2.

Mentionnons qu'il existe différentes façons de formaliser des règles. L'une d'elle consiste à les programmer dans un langage spécifique. Grâce à ce programme source, le compilateur de la base de connaissances et un analyseur syntaxique sont en mesures de construire la structure en un réseau indexé, constituant la représentation interne des règles [Friedman-Hill, 2003]. Les coquilles de développement, dont il est question dans ce mémoire, à savoir CLIPS et Jess (voir la section 2.3), ont recours à la syntaxe LISP pour programmer les règles [Anderson et al, 1987]. L'acronyme LISP signifie « LISt Processing ». C'est un langage procédural qui, contrairement à PROLOG, est explicitement déclaratif. Il a été développé par John McCarthy durant les années 50. D'ailleurs, LISP et PROLOG ont été largement utilisés dans plusieurs domaines qui touchent l'intelligence artificielle [Luger et Stubblefield, 1998]. La structure de base du langage LISP est la liste, c'est-à-dire une séquence de symboles, séparés par des blancs et regroupés entre parenthèses :

```
(liste_de_symboles (voici une liste imbriquée de symboles en lisp))
(+ 2 4)
```

Dans le cas de CLIPS et de Jess, le premier symbole de la liste (la *tête*) a une signification particulière, lorsqu'il évoque un mot réservé reconnu par leur analyseur syntaxique. Ainsi, pour définir une règle quelconque, il suffit d'utiliser l'expression **defrule** et la syntaxe suivante :

```
(defrule <nom-de-la-regle> ["<Commentaire>"].
      <antecedent-premisse>*
=>
      <action-consequence>*
)
```

2.6 Mémoire de travail et les faits

La mémoire de travail (« Working Memory »), appelée aussi base de faits (« Facts Base »), contient un ensemble d'informations qui décrivent l'état ponctuel d'un système. D'une part, la mémoire de travail contient des faits assortis à des prémisses de règles et d'autre part, ces faits peuvent être affirmés, modifiés ou rétractés, par des actions imputables aux règles, c'est-à dire les conséquences. Afin de simplifier la recherche de faits, particulièrement lors des modifications et des rétractations, les systèmes experts modernes maintiennent des index [Friedman-Hill, 2003], similaires à ceux d'une base de données relationnelle [Elmasri et Navathe, 1994] [Gardarin, 2001].

2.6.1 Les valeurs des faits

Généralement la mémoire de travail permet de stocker des faits, dont les valeurs correspondent à des types élémentaires, tels des booléens, des chaînes de caractères, des nombre entiers ou réels. Certaines coquilles de systèmes experts offrent même la possibilité d'incorporer des objets complexes. C'est le cas de Jess, puisque sa mémoire de travail peut facilement référencer des instances de classes Java [Friedman-Hill, 2003]. Généralement, les outils de conception de systèmes experts, comme CLIPS et Jess, utilisent la syntaxe LISP pour programmer les faits de la mémoire de travail [Giarratano, 2002] [Jess 7.0 Manual]. Les valeurs de ces faits correspondent à l'ensemble des symboles qui suivent la *tête* d'une liste. Par exemple, la

tête du fait ci-dessous est `poste`. Les trois valeurs de ce fait sont respectivement de type : chaîne de caractères (`"Presse"`), symbole (`actif`) et nombre entier (`254`).

```
(poste "Presse" actif 254)
```

2.6.2 Les types de faits

Les coquilles de développement de systèmes experts modernes, comme Jess et CLIPS, mettent en œuvre deux types de faits : *non-ordonnés* et *ordonnés*. Les premiers sont formés de *fentes* (« slots »), c'est-à-dire les attributs qui décrivent le fait, par exemple : le nom, l'état, le temps traité, le temps de traitement, le temps d'opération, etc. [Giarratano, 2002] [Jess 7.0 Manual]. Ce type de fait ressemble à un enregistrement d'une table de base de données relationnelle [Elmasri et Navathe, 1994], où les fentes correspondent aux champs de la table. Afin de spécifier explicitement les attributs des faits non-ordonnés, CLIPS et Jess possèdent une instruction de définition de *modèles de faits* (« template »), à savoir le mot réservé **`deftemplate`** :

```
(deftemplate poste
      (slot nom (type STRING))
      (slot etat (type STRING)
      (slot temps_traite (default 0) (type INTEGER)))
```

Ainsi, on peut définir des faits non-ordonnés qui agissent comme des instances du modèle, par exemple :

```
(poste (nom "Presse") (etat actif) (temps_traite 254))
(poste (nom "Four R") (etat inactif) (temps_traite 254))
```

Quant aux faits ordonnés, ils sont utilisés lorsque l'information est dépourvue de toute forme de structure. En un mot, ils correspondent à une simple énumération de valeurs :

```
(poste "Presse" actif 254)
(actif inactif "arret planifie" "arret non-planifie")
```

Les coquilles CLIPS ou Jess utilisent le mot réservé **`assert`** [Giarratano, 2002] [Jess 7.0 Manual], pour signaler à leur analyseur syntaxique qu'un nouveau fait est affirmé, c'est-à-dire ajouté à la mémoire de travail. Éventuellement, les valeurs ou *fentes* des faits peuvent être assortis aux variables des conditions de règles (**`defrule`**) et contribuer à l'activation de ces dernières.

L'exemple ci-dessous montre l'affirmation de trois faits non-ordonnés `poste`. Le premier et le dernier satisfont l'unique condition de la règle `poste-actif`. Notons que la variable `?nom_poste` est liée à la *fente* `nom`. Cette *fente* est associée à des faits dont la tête est de type `poste`.

```
(assert (poste (nom "Presse") (etat actif) (temps_traite 254))
(assert (poste (nom "Moule") (etat inactif) (temps_traite 50))
(assert (poste (nom "Tour à pâte") (etat actif) (temps_traite 306))

(defrule poste-actif "Affiche à l'écran les postes actifs"
     (poste (nom ?nom_poste) (etat actif))
=>
     (printout t "Le poste " ?nom_poste " est actif." crlf)
)
```

On a comme sortie l'affichage suivant :

```
Le poste Presse est actif.
Le poste Tour à pâte est actif.
```

2.7 Moteur d'inférence

Le *moteur d'inférence* est la composante chargée d'interpréter le savoir de la base de connaissances. Il est divisé en trois parties : l'*assortisseur de condition*, l'*agenda des activations* et le *moteur d'exécution*. Il examine le contenu de la base de connaissances et les faits accumulés à propos d'un problème, afin de tirer des conclusions, établir des solutions ou obtenir des réponses. Toutefois, les mécanismes qui contribuent à atteindre ces résultats peuvent varier de façon significative. Notamment en raison des différents formalismes de la connaissance et des types de raisonnements pouvant être appliqués à celle-ci. Dans le cas d'un système expert formalisé à partir de règles, le fonctionnement du moteur d'inférence est basé sur deux catégories de raisonnements : le *chaînage avant* et le *chaînage arrière* [Gonzalez et Dankel, 1993].

2.7.1 Moteur à chaînage avant

Le *chaînage avant* est une approche de résolution de problème dite *de bas en haut*. Son appellation anglaise est « bottom-up ». En effet, elle consiste à d'abord s'appuyer sur des preuves de bas niveau, à savoir les faits, pour pouvoir établir des conclusions de niveau supérieur [Giarratano et Riley, 1998]. On dit aussi que c'est une approche *conduite par les données* (« data

driven »), car ce sont les faits qui mènent au résultat final. C'est pourquoi le *chaînage avant* convient d'avantage, lorsque la plupart des faits en présences sont nécessaires à la résolution de la problématique. On l'utilise aussi dans des situations où il existe plusieurs conclusions acceptables et dans le cadre de problèmes impliquant la synthèse de données, par exemple : le contrôle, la planification, la simulation, la surveillance, etc.

En pratique, ce type de raisonnement implique qu'il faut vérifier chaque règle de la base de connaissances, afin de déterminer si les faits observés satisfont pleinement leurs prémisses. Les règles ainsi satisfaites sont dites *actives* et doivent être exécutées par le moteur d'inférence, afin de déduire de nouveaux faits. À leur tour, ceux-ci pourront déclencher d'autres règles qui déduiront des faits additionnels et ainsi de suite. Normalement, le processus itératif que nous venons de décrire, se poursuit jusqu'à ce qu'il n'y ait plus aucune règle active. Ainsi, un moteur d'inférence à *chaînage avant* doit itérer plusieurs cycles. Ceux-ci impliquent les étapes qui suivent [Friedman-Hill, 2003] :

1. Assortiment :
   1.1. Comparer toutes les règles avec les faits de la *mémoire de travail*, grâce à l'*assortisseur de conditions*.
   1.2. Activer les règles dont toutes les conditions sont satisfaites par un ensemble de faits.
   1.3. Construire une liste non ordonnée, i.e. l'*ensemble des conflits*, à partir des règles activées durant ce cycle et les cycles antérieurs.
2. Résolution des conflits :
   2.1. Ordonner les règles conflictuelles à l'aide de la stratégie mise en œuvre par le *solutionneur de conflits*.
   2.2. Construire l'*agenda des activations*, i.e. l'ensemble des règles actives, ordonnées par la stratégie de résolution de conflits.
3. Exécution :
   3.1. Sélectionner la première règle active de l'*agenda*.
   3.2. Déclencher la règle sélectionnée.
   3.3. Apporter les changements à la *mémoire de travail* si l'exécution provoque des modifications, e.g. s'il y a des affirmations ou des rétractions de faits.
   3.4. Effectuer un autre cycle en allant à 1.1.

À première vue, de telles répétitions semblent impliquer des actions redondantes. On a qu'à penser à la première étape (assortiment) qui consiste à comparer les faits et les règles à chaque itération. Cependant, plusieurs systèmes experts utilisent des algorithmes pour éliminer cette

redondance. Notamment, en préservant les résultats issus de l'assortisseur de conditions (les étapes 1.1, 1.2 qui sont détaillées à partir de 2.8) et du solutionneur de conflits de l'agenda (les étapes 2.1, 2.2, dont il est question à la section 2.9).

Finalement, mentionnons que le mécanisme que nous venons de décrire, s'apparente à un parcours en largeur d'un graphe (de l'Anglais « Breadth-First Search ») [Shaffer, 1997] [Cormen et al., 1994], lorsqu'on l'applique au réseau d'inférence, illustrant la connaissance d'un processus particulier. C'est pourquoi le raisonnement avec *chaînage avant* convient d'avantage, pour des problèmes impliquant un réseau d'inférence assez large et peu profond [Giarratano et Riley, 1998].

2.7.2 Moteur à chaînage arrière

À l'inverse, le *chaînage arrière* est une approche de résolution *du haut vers le bas*. Elle implique un raisonnement à partir de structures de hauts niveaux, c'est-à-dire des hypothèses, vers des faits de bas niveaux qui supportent ces dernières. On dit aussi que le *chaînage arrière* est axé vers la *recherche d'objectifs* (« goal seeking ») [Giarratano et Riley, 1998]. C'est pourquoi, on la privilégie lorsqu'une quantité limitée d'hypothèse (ou d'objectifs) est initialement déterminée. Aussi, cette méthode d'inférence est fort appropriée si la problématique comporte un grand nombre de faits et si seulement une partie d'entre eux sont pertinents dans le cadre de la résolution. D'ailleurs, il arrive qu'un système expert avec *chaînage arrière* demande à l'usager de spécifier des faits absents de la mémoire de travail, afin de supporter les hypothèses qu'il a émises.

Pour toutes les raisons que nous venons d'énumérer, cette approche semble idéale pour les problèmes de diagnostic [Gonzalez et Dankel, 1993]. Par exemple, un spécialiste de la santé confronté à un malade doit se limiter à des symptômes spécifiques (les faits significatifs), plutôt qu'à l'état global du patient (tous les faits en présences), afin d'établir et valider un nombre minimal de causes (les hypothèses). D'autre part, on note qu'une méthode d'inférence avec *chaînage arrière* est similaire à un parcours en profondeur d'un graphe (de l'Anglais « Depth-First Search ») [Shaffer, 1997] [Cormen et al., 1994]. C'est pourquoi on l'utilise d'avantage lorsque le réseau d'inférence de la problématique est étroit et profond [Giarratano et Riley, 1998].

Tout comme un système expert basé sur un mécanisme d'inférence à *chaînage avant*, la base de connaissances des systèmes à *chaînage arrière* est formée de règles de type *si ... alors*. Cependant, leur mode de raisonnement est beaucoup plus complexe que la simple exécution des règles activées. En effet, dans un premier temps, il faut déterminer un ensemble d'objectifs (ou hypothèses) à atteindre. Puis, le moteur d'inférence va activement chercher à satisfaire les prémisses de certaines règles qui peuvent supporter ces objectifs, en considérant les faits en présences. Éventuellement, les règles ne pouvant être satisfaites permettront de déterminer des sous objectifs à atteindre [Friedman-Hill, 2003]. Les étapes suivantes décrivent de manière plus détaillée, le fonctionnement d'un moteur d'inférence à *chaînage arrière* [Gonzalez et Dankel, 1993]:

1. Construire une pile, (i.e. *l'ensemble des hypothèses)* composée des objectifs à atteindre. Ceux-ci sont supportés par des faits de la mémoire de travail.
2. Sélectionner le premier objectif de la pile, trouvez toutes les règles qui l'appuient (i.e. celles dont l'exécution permet d'affirmer des faits supportant l'objectif).
3. Pour chacune des règles qui valident l'objectif, examiner leurs prémisses :
   - 3.1. Si la prémisse d'une règle est satisfaite (i.e. toutes ses conditions sont assorties à des faits de la mémoire de travail), alors :
     - 3.1.1. Exécuter la règle satisfaite.
     - 3.1.2. Apporter les changements à la *mémoire de travail* si l'exécution provoque des modifications.
     - 3.1.3. Enlever le premier objectif de la pile et retourner à l'étape 2.
   - 3.2. Si la prémisse d'une règle n'est pas satisfaite (i.e. une ou plusieurs de ses conditions ne sont pas assorties à des faits de la mémoire de travail), alors :
     - 3.2.1. Rechercher une règle, dont l'exécution permet d'affirmer un fait capable de satisfaire la condition manquante.
     - 3.2.2. Si une telle règle existe, considérez que la condition assortie au fait manquant est un sous objectif à atteindre. Ajoutez-le à la pile et retournez à l'étape 2.
   - 3.3. Si l'étape 3.2 ne réussit pas à déterminer une règle, dont l'exécution affirme un fait satisfaisant la condition manquante, alors demandez à l'usager si ce fait est vrai.
     - 3.3.1. Si sa réponse est positive, affirmez le fait, considérez que la condition manquante est maintenant satisfaite et passez à la condition suivante de cette règle (i.e. retournez à l'étape 3.2.1).
     - 3.3.2. Si sa réponse est négative, considérez une autre règle (i.e. retournez à l'étape 3.1).

```
3.4.  Si toutes les règles qui peuvent appuyer l'objectif actuel ont
      été essayées sans succès, celui-ci demeure indéterminé. Enlevez
      l'objectif de la pile.
   3.4.1.  Passez à l'objectif suivant s'il y en a un (i.e. retournez
           à 2).
   3.4.2.  S'il n'y a plus d'objectif dans la file, le moteur
           d'inférence a complété son travail.
```

2.8 Assortiment de conditions

Les *moteurs d'inférences* des systèmes experts doivent mettre en œuvre un algorithme particulier, afin d'assortir les faits de la mémoire de travail, avec les conditions contenues dans chacune des règles. Cette tâche incombe à l'*assortisseur de condition* ou *assortisseur de motif* (traduction de l'Anglais « Pattern Matcher »). Par la suite, les règles dont toutes les conditions sont assorties vont être ajoutées à l'agenda des activations et éventuellement déclenchées. Cependant, l'implémentation de l'algorithme d'assortiment des motifs dépend du type de raisonnement imputable au moteur d'inférence. En ce qui concerne le modèle que nous avons conçu, il utilise un mode de raisonnement à *chaînage avant*, pour les raisons qui sont mentionnées à la section 3.4.2.2. La discussion qui suit est donc axée sur l'une de ses implémentations les plus connues, à savoir l'algorithme de Rete développé par le docteur Charles L. Forgy de l'université Carnegie Mellon [Forgy, 1982]. L'appellation Rete signifie *filet* en Latin ou « net » en Anglais, au sens d'un réseau (c'est-à-dire « Network »), puisqu'il consiste à construire un graphe, similaire au réseau d'inférence dont il a été question à la section 2.5.2.

2.8.1 Assortiment basé sur un algorithme naïf

Afin de mieux comprendre en quoi Rete est une solution appropriée au problème d'assortiment de conditions, considérons d'abord l'implémentation naïve, du raisonnement à *chaînage avant* que nous avons décrite à la section 2.7.1. Cette solution triviale consiste à conserver une liste de règles et vérifier que toutes leurs conditions sont assorties à des faits de la mémoire de travail. On peut ainsi former un ensemble d'activations pour toutes les règles assorties. Après avoir sélectionné et exécuté une règle de l'ensemble, on la retire de ce dernier et on passe

au cycle d'inférence suivant. Ceci implique une nouvelle tentative d'assortiment de conditions. Cette approche, où les règles « recherchent » des faits, est clairement inefficace. En effet, à chaque cycle, il faut revérifier les activations, car le déclenchement d'une règle peut avoir modifié la mémoire de travail [Friedman-Hill, 2003 et 2005] [Giarratano et Riley, 1998].

Maintenant considérons la règle suivante (`R1'`), une simplification de `R1` définie à la section 2.2.1, afin d'analyser quelle pourrait être la complexité de cet algorithme.

```
(R1') si
        Poste P est "actif" et
        Action planifiée de type "arrêt" survient actuellement sur P
     alors
        Rendre poste P "inactif"
```

Supposons qu'il y ait **p** faits décrivant les *postes* et **a** faits associés aux *actions planifiées*. En appliquant l'algorithme naïf à cette règle, dans un premier temps, on doit rechercher tous les *postes actifs* (où `P` représente l'un d'eux) et toutes les *actions planifiées* de type *arrêt* actuellement en vigueur. Dans un deuxième temps, à cause de la jointure découlant de la variable `P` des deux conditions, on estime les *arrêts* qui s'appliquent aux *postes actifs*. Nous pouvons donc affirmer que dans ce cas, l'algorithme naïf devrait avoir une performance $\mathbf{O((a+p)^2)}$. La complexité de la solution semble donc être dépendante d'une part, du nombre de conditions formant la prémisse (**2** dans `R1'`) et d'autre part, du nombre de faits qui leur sont associées (**a** et **p**). Néanmoins, il est plutôt difficile d'estimer la complexité exacte de cette implémentation, lorsqu'on applique à toutes les règles de la base de connaissances. Notamment parce que chacune d'elles n'ont pas le même nombre de motifs (conditions) et que ceux-ci ne sont pas nécessairement liés par des variables (`P` dans `R1'`) [Jess 7.0 Manual]. Par contre, nous pouvons affirmer qu'en moyenne, la performance de la solution naïve est de l'ordre de $\mathbf{O(R \cdot F^C)}$ où,

1. **R** = nombre de règles de la base de connaissances
2. **F** = nombre de faits de la mémoire de travail
3. **C** = nombre moyen de conditions par règle.

Le nombre moyen de conditions par règle cause donc un accroissement exponentiel de la complexité, au fur et à mesure que des faits sont ajoutés. Cet algorithme est d'autant plus inefficient, puisque que l'on doit l'appliquer à chaque cycle d'inférence.

2.8.2 Assortiment basé sur l'algorithme de Rete

L'amélioration du mécanisme d'assortiment de motifs est basée sur une constatation, à savoir que la plupart des systèmes experts ont une mémoire de travail avec un contenu relativement fixe. Ainsi, bien que des faits soient affirmés, modifiés ou rétractés, leur pourcentage demeure relativement négligeable [Friedman-Hill, 2003] [Giarratano et Riley, 1998]. L'implémentation que nous avons décrite, où les règles « recherchent » les faits, est donc inefficace, puisque l'ensemble des assortiments de motifs demeure sensiblement le même pour chaque cycle d'inférence.

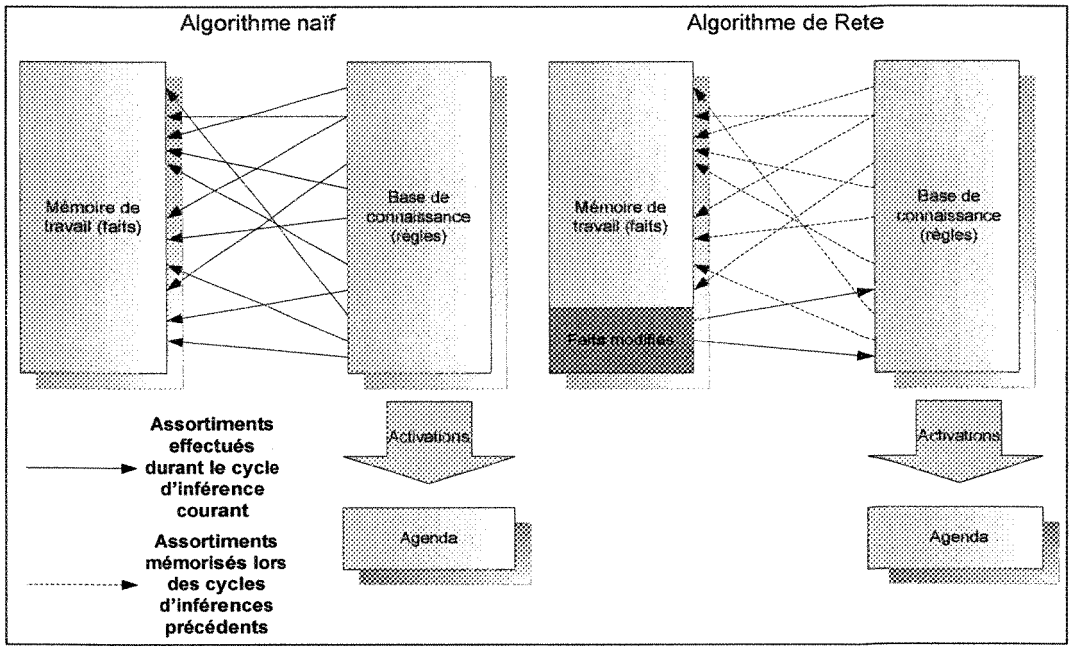

Figure 16 : l'algorithme naïf et de Rete pour l'assortiment des conditions

L'algorithme de Rete apparaît comme une amélioration, car il mémorise les assortiments des cycles précédents et les met à jour, uniquement lorsque des faits ont été changés. Cette solution semble donc moins coûteuse, car ce sont uniquement les faits modifiés qui « recherchent » les

règles. La figure ci-dessus illustre les deux algorithmes d'assortiment de motifs, dont il a été question jusqu'ici. Notons les faits modifiés dans la zone foncée et la faible proportion des assortiments devant être à nouveau vérifiées, c'est-à-dire les flèches pleines, lorsqu'on considère Rete.

La mémorisation des assortiments implique que chacune des règles connaissent les faits qui satisfont leurs conditions, en considérant les restrictions découlant des variables qui les lient. Donc, si un fait modifié était assorti à la troisième condition d'une règle, l'information provenant de l'assortiment des deux conditions précédentes doit être disponible, afin de compléter le processus d'appariement. Cette information indiquant l'ensemble des faits qui satisfont un groupe de conditions successives, en considérant les variables qui les lient, s'appelle une *association partielle* (de l'Anglais « Partial Match ») [Giarratano et Riley, 1998]. Par exemple, soit la règle `R1'` :

```
(R1') si
        Poste P est "actif" et
        Action planifiée de type "arrêt" survient actuellement sur P
     alors
        Rendre poste P "inactif"
```

En ayant recours à la syntaxe LISP, nous avons la règle suivante :

```
(R1') (defrule desactiver-poste

        ; Première condition de la règle pour poste.
        ; (n.b. ?fait_poste contient un fait associé à la condition.)
        ?fait_poste <-
              (poste (nom ?nom_poste) (etat "actif"))

        ; Deuxième condition de la règle pour action planifiée.
              (action_planifiee (type "arrêt")
                        (poste ?nom_poste) (debute_a ?t_actuel))
     =>
        ; Modification de le l'attribut état du fait poste.
        (modify ?fait_poste (etat "inactif"))
```

Supposons que la mémoire de travail contienne les faits suivants et que le temps de simulation actuel (variable `?t_actuel`) soit de 245 secondes :

```
F1 : (poste (nom "Presse") (etat "actif"))
F2 : (poste (nom "Tour à pâte") (etat "inactif"))
F3 : (poste (nom "Moule") (etat "actif"))
F4 : (poste (nom "Four R") (etat "actif"))
```

```
F5 : (action_planifiee
        (type "démarrage") (poste "Tour à pâte") (debute_a
     250))
F6 : (action_planifiee
        (type "arrêt") (poste "Presse") (debute_a 245))
F7 : (action_planifiee
        (type "arrêt") (poste "Moule") (debute_a 245))
F8 : (action_planifiee
        (type "arrêt") (poste "Scellement") (debute_a 245))
```

Dans ce cas-ci, les *associations partielles* correspondent aux ensembles de faits suivants :

1ère condition : {F1} et {F3}
1ère et 2ième condition : {F1, F6} et {F3, F7}

Notons qu'une *association partielle* de toutes les conditions d'une règle forme une *activation*, comme c'est le cas pour les ensembles suivants : {F1, F6} et {F3, F7}. Également, il existe un autre type d'information, appelée *association de condition* (en Anglais « Pattern Match »), qui doit être prise en compte par l'algorithme de Rete. Elle survient lorsqu'une condition de règle est satisfaite par un fait, peu importe les variables des autres motifs limitant le processus d'assortiment [Giarratano et Riley, 1998]. En considérant l'exemple précédent, les *associations de conditions* de `R1'` sont :

1ère condition : F1, F3 et F4
2ième condition : F6, F7 et F8

Mentionnons que ces faits se bornent à effectuer une association avec la 1ère et 2ième condition, sans se soucier de la variable `P` (`?nom_poste`) qui lie les deux motifs.

2.8.2.1 Mise en œuvre de l'algorithme de Rete

Comme on l'a mentionné, l'algorithme de Rete est mis en œuvre à l'aide d'un réseau de nœuds interconnectés, voir la Figure 17 plus bas. Chacun d'eux représentent généralement un ou plusieurs tests à effectuer. Ceux-ci résultent des diverses conditions de la partie gauche des règles. Chaque nœud du réseau a soit une ou deux *entrées* et une seule *sortie*. Les faits de la mémoire de travail qu'on affirme, modifie ou rétracte sont donc testés par certains nœuds du réseau. Lorsque le test d'un nœud est fructueux, les faits résultants sont envoyés au suivant. Une activation est

effectuée lorsqu'un ensemble de faits a entièrement traversé le réseau. Le nœud se trouvant à la toute fin du réseau s'appelle le *nœud terminal* [Friedman-Hill, 2003] [Jess 7.0 Manual].

La mise en œuvre de l'algorithme de Rete, s'effectue en deux étapes [Giarratano et Riley, 1998]. Dans un premier temps, lorsque des changements au sein de la mémoire de travail ont eu lieu, la vérification des *associations de conditions* mémorisées doit être faite, pour estimer celles qui sont toujours valides. Dans un deuxième temps, la comparaison des variables liant les conditions doit être effectuée, afin de déterminer les groupes de motifs dont les *associations partielles* demeurent effectives.

L'algorithme de Rete réalise la première étape grâce à un arbre appelé le *réseau de condition* (traduction de l'Anglais « Pattern Network »). À son sommet se trouve un ou plusieurs nœuds à *entrée unique*, appelés *nœuds de condition* (de l'Anglais « Pattern Node »). Ceux-ci permettent de filtrer les faits selon leur *tête* et d'envoyer au nœud suivant les faits résultants [Friedman-Hill, 2003]. Par exemple, dans le cas de `R1'`, les faits *poste* et *actions planifiées* empruntent des chemins totalement différents à travers le *réseau de condition*. En plus, les *nœuds de condition* de cet arbre permettent de représenter les contraintes individuelles d'attributs (« slots »), en autant qu'ils ne soient pas liés à d'autres motifs [Giarratano et Riley, 1998]. Cette représentation d'un motif non lié, s'explique du fait que le réseau mémorise les *associations de conditions* qui, comme on l'a vu, ne tiennent pas compte des liens entre les motifs. Ainsi, en considérant `R1'`, parce que l'attribut *état* de la première condition impose que sa valeur soit « *actif* », un nœud devrait être ajouté au *réseau de condition*. Ce nœud permet d'effectuer le test qui découle uniquement du premier motif.

Quant à la seconde étape, elle est accomplie à l'aide du *réseau de branchement* (de l'Anglais « Join Network »). Celui-ci est adjoint au *réseau de condition*. Donc, une fois que les conditions ont été appariées aux faits, la comparaison des variables qui les joignent doit être effectuée, pour s'assurer que celles utilisées dans d'autres motifs possèdent des valeurs cohérentes [Giarratano et Riley, 1998]. Le *réseau de branchement* est formé de nœuds à double

entrées appelés *nœuds de branchement* (traduction de l'Anglais « Join Node »). Généralement, ils testent une jointure entre deux conditions. De plus, ils sont utilisés pour mémoriser l'ensemble des faits provenant des deux *entrées*. Les nœuds utilisent deux unités de stockages pour préserver les faits entrants, à savoir la *mémoire alpha* pour l'arête gauche et la *bêta* pour l'arête droite.

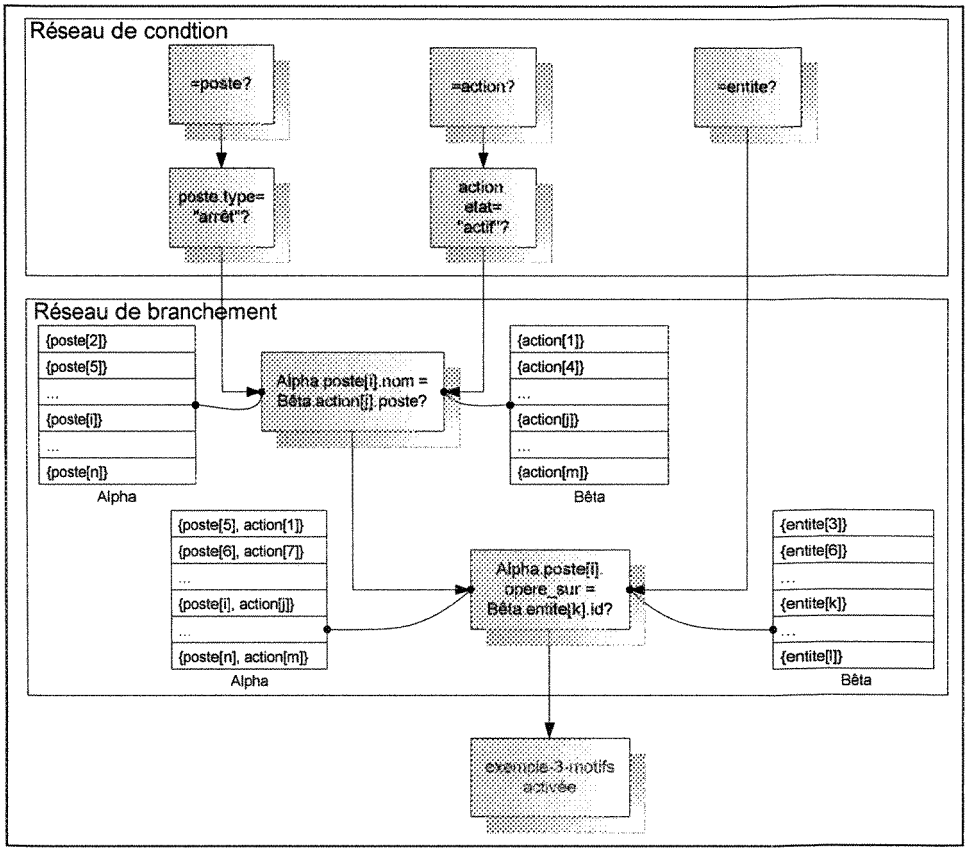

Figure 17 : réseau de Rete d'une règle

Généralement, on utilise une structure de donnée indexée, telle une table de hachage [Cormen et al, 1994], pour représenter les deux mémoires et ainsi faciliter la recherche des faits et des règles appariées [Friedman-Hill, 2003]. Le *réseau de branchement* est construit de manière à ce que *l'entrée* gauche d'un nœud reçoive des faits ayant des *têtes* identiques ou différentes. Quant à l'entrée de droite, elle obtient des faits qui ont tous la même *tête*. Les deux *entrées* ont des fonctions distinctes, puisque d'une part, les faits provenant de l'arête droite découlent d'un motif unique *m* (supposons le $n^{ième}$ d'une règle quelconque) et d'autre part, ceux issus de l'arête gauche

résultent des assortiments des *n-1* motifs qui précèdent *m*, autrement dit un intervalle de motifs *[1..n-1]* [Friedman-Hill, 2003]. La Figure 17 illustre le réseau de Rete de la règle suivante :

```
(defrule exemple-3-motifs
      (poste (nom ?nom_poste) (etat "actif") (opere_sur ?id_entite))
      (action (type "arrêt") (poste ?nom_poste))
      (entite (id ?id_entite))
=>
      )
```

Lorsqu'on analyse son *réseau de branchement*, on constate que *l'entrée* de droite du premier nœud dérive des assortiments du second motif (*n = 2*), tandis que celle de gauche découle du motif précédent (où l'on a *[1..n-1]* ≡ 1er motif). La même logique est applicable au niveau du second nœud, puisque son arête de droite reçoit les faits associés au troisième motif (*n = 3*), alors que celle de gauche résulte des associations des deux motifs précédents (où l'on a *[1..n-1]* ≡ 1er et 2^{ième} motif). C'est d'ailleurs pourquoi chaque cellule de la mémoire *alpha* de ce nœud est formée d'un couple de faits `poste` (1er motif) et `action` (2^{ième} motif), contrairement à *bêta* qui contient seulement des faits dont la *tête* est `entite`.

2.8.2.2 Rétraction et modification selon l'algorithme de Rete

Jusqu'ici nous avons vu comment l'algorithme de Rete met en œuvre l'assortiment de faits lors d'affirmations. Cependant, il est notable de mentionner que la rétraction utilise un principe similaire. En réalité, Rete ne propage pas des faits à travers le réseau mais plutôt des jetons. Un jeton est composé d'un ou plusieurs faits et d'une commande spécifiant l'action à exécuter : affirmation ou rétraction. Ainsi, lorsqu'un jeton avec la commande de rétraction parvient à un *nœud de branchement*, Rete recherche un fait identique au sein de la mémoire appropriée. Si la recherche est fructueuse, le fait est détruit. Puis, pour tous les faits associés, se trouvant dans la mémoire opposée, on crée des jetons avec la commande de suppression. Ceux-ci sont à leur tour envoyés à la *sortie* du *nœud de branchement*. Finalement, lorsqu'un *nœud terminal* du réseau de Rete reçoit un jeton avec la commande de suppression, l'activation correspondante est retirée de l'agenda [Friedman-Hill, 2003].

Concernant la commande de modification, bien qu'elle ne soit pas précisée dans la description originale de Rete [Forgy, 1982], la plupart des systèmes experts la mettent en œuvre en rétractant le fait, puis en changeant les valeurs d'attributs et en réaffirmant le fait modifié [Giarratano, 2002] [Jess 7.0 Manual].

2.8.2.3 Optimisation de l'algorithme de Rete

Il existe différentes méthodes pour optimiser l'algorithme de Rete. Deux des plus simples consistent à effectuer un *partage des nœuds* [Friedman-Hill, 2003] [Jess 7.0 Manual]. Le premier type de partage implique la mise en commun des nœuds du *réseau de condition*. Donc, en considérant la règle suivante :

```
(defrule exemple-2-motifs
      (poste (nom ?nom_poste) (etat "actif"))
      (action (type "arrêt") (poste ?nom_poste))
=>
      )
```

Ainsi que `exemple-3-motifs`, décrite à la section 2.8.2.1, on constate que quatre nœuds du *réseau de condition* peuvent être mis en commun : `=poste?`, `=action?` (pour la vérification des *têtes*), ainsi que `poste.etat="actif"?` et `action.type="arrêt"` (pour la vérification des valeurs d'attributs).

Cependant, en ce qui concerne cet exemple, on peut éviter encore plus de redondance, en mettant aussi en commun le *nœud de branchement* qui effectue le test suivant : `Alpha.poste[i].nom = Bêta.action[j].poste?`, pour les deux règles. La figure ci-dessous montre le réseau de Rete partageant à la fois les *nœuds de condition* et de *branchement*.

Puisque l'opération de jointure implique la comparaison de plusieurs conditions, les tests du *réseau de branchement* risquent d'être en plus grand nombre que ceux du *réseau de condition*. C'est pourquoi la jointure des variables de motifs compte pour une grande part du temps d'exécution de l'algorithme. Il s'avère donc fort avantageux de partager les *nœuds de branchement* autant que possible. Ainsi, en mettant en commun le premier *nœud de branchement* dans la figure

suivante, on doit s'attendre à ce que la performance de l'algorithme soit doublée [Friedman-Hill, 2003].

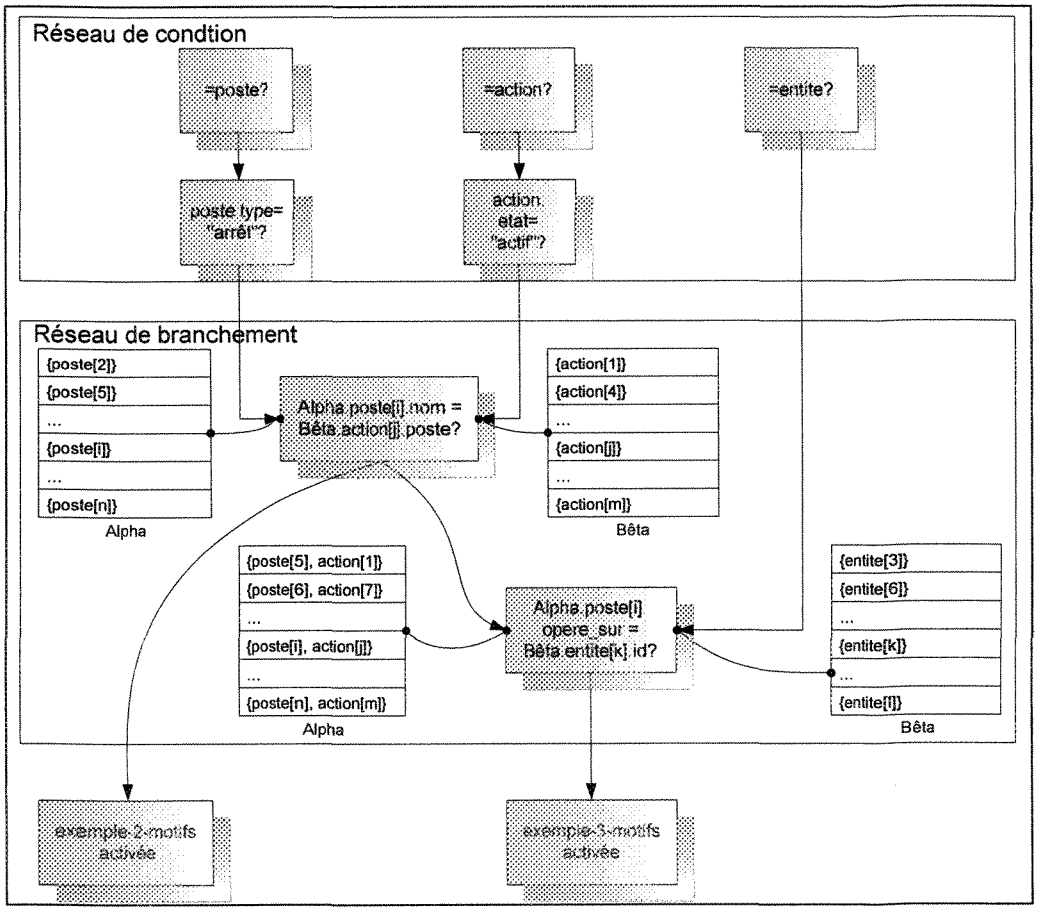

Figure 18 : partage des nœuds de condition et de branchement

2.8.2.4 Analyse de l'algorithme de Rete

À l'instar de la solution naïve, la complexité de l'algorithme de Rete, lors du premier cycle d'inférence est aussi $\mathbf{O(R \cdot F^C)}$. En effet, puisqu'il n'y a aucun résultat antérieur mémorisé, il faut effectuer les *associations de conditions*. Concernant les cycles d'inférences suivants, dans le pire des cas, c'est-à-dire lorsque que tous les faits changent et qu'il n'y a pas de nœuds partagés, la performance demeure sensiblement la même que l'algorithme naïf [Friedman-Hill, 2003].

L'amélioration la plus notable se situe au niveau du cas moyen, alors que peu de faits sont modifiés. Dans cette situation on s'attend à une complexité $\mathbf{O(R' \cdot F'^{C'})}$ où,

1. **R'** (nombre de règles associées aux faits modifiés) < **R** (nombre de règles)
2. **F'** (nombre de faits modifiés) < **F** (nombre de faits)
3. **C'** (nombre moyen de conditions des règles associées aux faits modifiés) < **C** (nombre moyen de conditions des règles).

Le principal désavantage de l'algorithme de Rete est qu'il utilise beaucoup de mémoire. En effet, la préservation de plusieurs milliers d'*associations partielles* et d'*associations de conditions* peut s'avérer coûteuse en espace de stockage. Toutefois, le compromis entre le temps d'exécution et la quantité de mémoire vaut souvent la peine. Mentionnons qu'il existe tout de même des approches pour optimiser l'utilisation de la mémoire. Notamment en réécrivant les règles, de manière à ce que Rete ait beaucoup moins d'*associations partielles* à effectuer [Giarratano et Riley, 1998] [Jess 7.0 Manual].

2.9 Agenda des activations

Au cours de chaque cycle d'inférence, la liste des règles actives, pouvant potentiellement être déclenchées, doit être maintenue. Cette liste de règles éligibles est appelée l'*ensemble des conflits*. Cependant, il importe d'établir leur ordre de déclenchement. Pour ce faire, une *stratégie* préétablie est utilisée, afin d'affecter une priorité d'exécution à chacune des règles de l'*ensemble des conflits*. Ce processus d'ordonnancement est appelé la *stratégie de résolution de conflits* (de l'Anglais « Conflict Resolution Strategy ») [Giarratano, 2002]. La liste des règles, issues du processus de résolution de conflits, constitue l'*agenda des activations* [Friedman-Hill, 2003]. Une application adjointe à un système expert, se termine une fois que son *agenda des activations* est vide.

Il peut arriver qu'à chaque cycle d'inférence, l'*agenda des activations* ne contienne pas plus d'une règle. La *stratégie de résolution de conflits* est alors superflue. Du même coup, recourir à un système expert s'avère tout aussi inutile. En effet, cette conjoncture implique qu'il est possible de

résoudre la problématique à l'aide d'un programme séquentiel, écrit dans un autre langage de troisième génération [Giarratano, 2002]. Il est donc fondamental de vérifier si le problème a une solution algorithmique séquentielle, avant même de considérer une approche basée sur un système expert.

2.9.1 Définition de l'importance des règles

La plupart des coquilles de développement permettent d'affecter à chaque règle une priorité dans l'*agenda des activations* [Giarratano, 2002] [Jess 7.0 Manual]. Pour ce faire, on assigne à une propriété particulière l'*importance* relative de la règle (de l'Anglais « Salience »). Plus la valeur de cette propriété est élevée, plus l'*importance* de la règle est accrue. Lorsque deux règles ont des *importances* différentes, la stratégie de résolution de conflits n'entre pas en ligne de compte. Cependant, en affectant des priorités distinctes à plusieurs règles, on risque de détériorer les performances du système expert, du moins en ce qui concerne la *stratégie de résolution de conflits*. De plus, lorsque l'ordre d'activation des règles est fortement établi, l'usage d'un système expert doit être reconsidéré, puisqu'une fois encore, le problème peut aussi bien être résolu par un programme séquentiel [Friedman-Hill, 2003].

2.9.2 Stratégie de résolution de conflits

Chaque coquille de développement de système expert a sa propre *stratégie de résolution de conflits*. Généralement, une *stratégie* efficace doit considérer la spécificité ou la complexité de chaque règle et le moment de leur activation. CLIPS [Giarratano, 2002] et Jess [Jess 7.0 Manual] utilisent par défaut une *stratégie de résolution de conflits* dite *en profondeur* (« Depth ») ou FIFO (« First In First Out »). Celle-ci signifie qu'à *importance* égale, les règles les plus récemment activées sont d'abord déclenchées. L'agenda agit donc comme une pile de règles. Les deux coquilles de développement mettent aussi en œuvre une *stratégie* dite *en largeur* (« Breadth ») ou LIFO (« Last In First Out »). Elle considère que l'agenda est une file, où les règles sont déclenchées dans leur ordre d'activation. CLIPS intègre également trois *stratégies* alternatives

[Giarratano, 2002]. Quant à Jess, il permet uniquement d'utiliser les stratégies FIFO en *profondeur* et LIFO en *largeur*. Toutefois, l'élaboration de *stratégies* personnalisées est possible.

L'approche *en profondeur* s'avère suffisante pour résoudre la plupart des problèmes. Certaines situations délicates peuvent toutefois survenir. C'est notamment le cas, lorsque toutes les règles déclenchées activent d'autres règles. Les activations les plus anciennes sont alors repoussées dans l'agenda et n'ont jamais l'occasion de se déclencher [Friedman-Hill, 2003]. À ce titre, nous allons discuter de la *stratégie de résolution de conflits* que nous avons opté à la section 3.4.4.3.

2.10 Moteur d'exécution

Il s'agit d'une composante qui est responsable de l'exécution des actions de la *partie droite* (RHS) d'une règle. Sous sa forme la plus simple, cette tâche revient à affirmer, retirer ou modifier des faits. Cependant, pour la plupart des systèmes experts modernes, le déclenchement d'une règle implique des actions beaucoup plus complexes. Ainsi, certaines coquilles permettent de définir les actions dans un langage de programmation structuré. Dans ce cas, le *moteur d'exécution* représente l'environnement dans lequel ce langage de programmation s'exécute. Pour d'autres systèmes, le *moteur d'exécution* agit comme un interpréteur de langage ou encore, comme intermédiaire responsable d'invoquer un programme compilé ou une librairie externe [Friedman-Hill, 2003].

2.11 Interface utilisateur

L'*interface utilisateur* permet d'échanger des informations, entre le système expert et l'usager. Pour ce faire, elle met à la disposition de l'utilisateur, un ensemble de moyens d'interactions tels : des menus, des boîtes de textes, le langage naturel ou un affichage graphique. Plus précisément, voici quelques fonctions qui lui sont propres [Gonzalez et Dankel, 1993] :

- Permettre à l'utilisateur de fournir au système expert des informations sur l'état du problème. Par exemple, le moteur d'inférence à *chaînage arrière* peut demander à l'usager de saisir des

données supplémentaires pour atteindre un objectif. Également, le système expert peut requérir un ensemble de faits initiaux, avant le commencement du processus d'inférence.

- Fournir des explications pour justifier certaines demandes d'informations faites par le système expert.
- Permettre à l'utilisateur de formuler des questions quant aux conclusions qui sont émises.
- Afficher les conclusions obtenues sous forme de texte, de graphiques, etc.
- Permettre à l'utilisateur de préserver ou d'imprimer les conclusions, une fois le processus d'inférence terminé.

2.12 Composante d'acquisition des connaissances

La *composante d'acquisition* est un outil qui assiste le spécialiste, lors l'élaboration de la base de connaissances. Comme on l'a mentionné, ce spécialiste interagit généralement avec l'expert du domaine, afin d'acquérir le savoir et de l'encoder sous une forme particulière. Citons en exemple, des règles rédigées selon la syntaxe LISP et utilisables par une coquille de développement comme CLIPS ou Jess. Sous sa forme la plus simple, cet outil agit comme un éditeur de la base de connaissances. Ainsi, il fournit au spécialiste une vue de la base de connaissances et un moyen d'y apporter des changements. Sous sa forme la plus complexe, la *composante d'acquisition* peut disposer d'un éventail de fonctionnalités [Gonzalez et Dankel, 1993], en voici quelques-unes :

- Localiser les problèmes associés aux éléments de la base de connaissances.
- Comparer les éléments qui forment la base de connaissances.
- Conserver les modifications qui ont été faites.
- Fournir des fonctionnalités d'éditions et de saisies avancées, etc.

Nous terminons ainsi le second chapitre. Au cours de celui-ci, nous avons vu en détail les fondements des systèmes experts et pourquoi ils font l'objet de notre intérêt. Nous allons maintenant présenter la problématique à laquelle nous avons appliqué notre stratégie de

modélisation. Nous allons également montrer l'architecture du modèle qui nous a permise de résoudre cette problématique.