Université de Montréal

Gestionnaire de connaissances pour systèmes hybrides objets-règles

par

Mustapha Es-salihe

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des Études Supérieures
en vue de l'obtention du grade de
Maître ès Sciences (M.Sc.)
en informatique

Décembre, 2000

Université de Montréal
Faculté des Études Supérieures

Ce mémoire intitulé :

Gestionnaire de connaissances pour systèmes hybrides objets-règles

présenté par :
Mustapha Es-salihe

a été évalué par un jury composé des personnes suivantes :

Président-rapporteur	: Nadia El-Mabrouk
Directeur de recherche	: Houari Sahraoui
Codirecteur de recherche	: Hakim Lounis
Membre du jury	: Peter Kropf

Mémoire accepté le :

SOMMAIRE

Les systèmes à base de connaissances (SBC) ont prouvé leur efficacité comme technologie pour la résolution de plusieurs types de problèmes dans différents domaines de l'industrie et de la gestion. Les SBC ont réussi à résoudre des problèmes complexes, en dérivant la solution à partir de l'application d'un corpus de connaissances, plutôt que de l'application d'algorithmes impératifs. Dans les années 80, la technologie des SBC a été largement appliquée pour résoudre plusieurs problèmes. Les exemples classiques de l'utilisation réussie de cette technologie sont la résolution de problèmes de diagnostic (par exemple, en médecine ou en ingénierie), la prestation d'assistance (par exemple, les applications de soutien informatique) et la construction/configuration (par exemple, la gestion de produits et de transports). Dans les années 90, plusieurs organisations ont identifié leurs connaissances collectives comme ressource très importante, et ont appliqué la technologie de SBC pour acquérir, structurer et exploiter ces connaissances de façon systématique.

Avec la large application de la technologie de SBC, il est devenu essentiel de garantir une bonne qualité et d'assurer un bon fonctionnement de ces systèmes. Les activités de vérification et de validation (V&V) constituent donc deux phases importantes durant la réalisation et l'exploitation de ces systèmes. La vérification est le processus qui permet de contrôler la conformité du système réalisé par rapport aux spécifications énoncées par ses utilisateurs, tandis que la validation permet de déterminer si le système est capable de trouver les conclusions attendues par ses utilisateurs.

Depuis la réalisation des premiers SBC, les approches utilisées étaient basées sur des logiques classiques (logique de propositions et logique de premier ordre). Ainsi les techniques de V&V ont été conçues pour ce type de formalisme. Il s'agit essentiellement de vérifier si une base de connaissances (BC), composée d'un ensemble de faits et d'un ensemble de règles, est exempte de certaines erreurs et anomalies (redondance, conflit, inutilité, etc.), et de valider cette BC sur un ensemble de cas de test.

L'émergence de la technologie objet (OO) et sa capacité de représenter naturellement et de structurer adéquatement les connaissances d'un domaine conduit à

une intégration des deux formalismes (objets et règles). Cette intégration va considérablement améliorer la qualité de SBC. Cependant elle rend plus complexe, comme nous allons le voir par la suite dans le détail du sujet, les tâches de vérification et de validation. Dans ce contexte (formalisme hybride objet-règle), notre objectif est de proposer une approche de vérification et de validation pour ce type de systèmes. Nous allons entre autres proposer une architecture pour un gestionnaire de connaissances comme cadre général pour la V&V, un modèle pour les éléments nécessaires à la V&V en terme de contraintes et des états du système et enfin un modèle global de V&V.

Mots clés : systèmes à base de connaissances, vérification, validation, objets, règles

Remerciements

Je tiens à remercier vivement le professeur Houari Sahraoui, mon directeur de recherche, pour m'avoir encadré dans ce sujet, pour sa disponibilité malgré ses nombreuses occupations, pour ses directives, ses conseils et ses critiques qui m'ont aidé à bien élaborer mon sujet.

Je tiens à remercier également le professeur Hakim Lounis, mon codirecteur de recherche, pour m'avoir impliqué dans le projet PReCI, suivi mon travail et pour m'avoir donné ses conseils tout au long de cette maîtrise.

Il me tiens à cœur de remercier plus profondément mes chers parents qui ont fait de moi ce que je suis aujourd'hui. Que dieu, le tout puissant, leur donne santé et longue vie.

Des remerciements particuliers à ma très chère sœur Najat pour son soutien moral et ses encouragements tout au long de ma recherche.

Mes remerciements aussi à mes chers amis, Hicham, Bachir, Adnane, Kamal et Abd Ellah pour leurs encouragements et pour le climat familial et chaleureux dans lequel on a vécu ensemble, tout le temps.

J'aimerais remercier tous les membres de l'équipe GLIC du CRIM pour leur participation et leur dialogue constructif. Un grand merci à Daphné Bélizaire, responsable du centre documentaire du CRIM, pour sa gentillesse et son aide au niveau de ma recherche bibliographique.

À mes chers parents, chers frères et sœurs

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

API	Application Programming Interface
BC	Base de connaissances
BF	Base de faits
BNF	Backus Naur Form
BR	Base de règles
EVA	Expert system Validation Associate
GO	Graphe d'occurrence
JNI	Java Native Interface
OCL	Object Constraint Language
OMG	Object Management Group
OO	Orienté Objet
SBC	Systèmes à base de connaissances
UML	Unified Modeling Language
V&V	Vérification et Validation

Chapitre 1

Introduction

1.1 Problématique

Durant les dernières années, il y a eu une explosion d'activités portant sur la vérification et la validation (V&V) de systèmes à base de connaissances. Par exemple, de nombreux ateliers au sein des conférences de l'association AAAI (American Association of Artificial Intelligence) ont pour thème principal la vérification et la validation. La conférence IJCAI (International Joint Conference on Artificial Intelligence) organise des ateliers sur la V&V depuis 1989. Il en est de même pour la conférence européenne sur l'intelligence artificielle ECAI (European Conference on Artificial Intelligence) qui a accueilli plusieurs ateliers sur la V&V. Cet intérêt s'explique par le besoin de tester un grand nombre de SBC qui ont été développés depuis les années 80. Avec le rôle de plus en plus élevé des systèmes intelligents dans des situations critiques, comme la médecine et la défense, il est devenu aussi primordial d'accorder une grande importance à la vérification et la validation de ces systèmes.

Typiquement, la vérification des SBC est basée sur le concept d'anomalie. Une anomalie peut être vue comme un usage abusif ou un usage inutile de la connaissance. Elle peut être considérée comme une erreur potentielle qui a besoin d'être vérifiée ou une situation voulue par l'expert. La validation permet de trouver les situations de disfonctionnement durant l'utilisation d'un SBC.

Jusqu'à présent, les travaux sur la vérification et la validation des SBC ont porté sur les systèmes classiques. Dans de tels systèmes, la base de connaissances est constituée d'un ensemble de faits logiques indépendants et d'un ensemble de règles formulées dans une logique de premier ordre ou logique de propositions. La vérification consiste à appliquer des techniques statiques ou dynamiques pour la recherche de certaines anomalies (redondance, conflit, inutilité, etc.), et la validation consiste à tester la BC sur un ensemble de cas de tests pertinents pour contrôler le comportement du système. Ces techniques utilisées pour la vérification et la validation sont relativement simples, compte tenu du fait que le formalisme de représentation est simple (ensemble de faits logiques). La vérification consiste donc à comparer les faits produits par les différentes règles pour détecter les différentes erreurs et

anomalies. La validation consiste, de son côté, à injecter un ensemble de faits initiaux en entrée du système et comparer les faits finaux avec ceux attendus.

En exploitant la technologie objet pour la représentation de connaissances statiques d'un domaine et les règles pour représenter les connaissances déductives (raisonnement), la BC devient plus complexe et le système de raisonnement se différencie de celui d'une logique classique. Les connaissances statiques sont définies par des classes (attributs et méthodes) qui représentent les entités du domaine d'application et les liens entre elles. Les conditions et les actions des règles sont formulées à l'aide des attributs et des méthodes de classes. Nous parlons alors de systèmes hybrides objets-règles pour la représentation des connaissances. Par la suite, et durant tout le mémoire, le terme hybride désigne cette intégration des objets avec les règles. Dans de tels systèmes, Le raisonnement se fait par transformation d'un état à un autre. Un état est composé de l'ensemble des instances de classes qui se trouvent dans la mémoire de travail (mémoire utilisée par le moteur d'inférence pour stocker les résultats de l'inférence au cours d'une session de raisonnement). Une autre particularité de ce raisonnement réside dans son caractère non monotone. Si dans une logique classique, on essaie toujours de ramener de nouvelles connaissances exprimées par de nouveaux faits qui deviendront vrais et sans remettre en cause les faits qui sont déjà vrais, dans le cas des systèmes hybrides, les actions des règles permettent de retirer des objets déjà existants, de modifier d'autres, et/ou d'ajouter de nouveaux. Ces deux particularités font que les techniques classiques de V&V des SBC ne fonctionnent pas. Notre objectif est de proposer une approche globale de V&V d'une BC hybride objet-règle.

1.2 Objectifs de la recherche

Nos principaux objectifs sont :

- La proposition d'une architecture pour un gestionnaire de connaissances comme cadre général pour la V&V. Cette architecture inclut en outre l'aspect de maintenance.
- L'élaboration d'un modèle rigoureux pour la représentation des contraintes et des états des objets utilisés par le système. Ces deux types d'informations sont nécessaires à la V&V.
- Enfin, l'aboutissement à un modèle de représentation de systèmes hybrides objet-règle basé sur l'utilisation des réseaux de Petri pour but de vérification et de validation.

1.3 Projet PReCI

Notre travail rentre dans le cadre du projet PReCI (Planification de la gestion des Ressources hydriques basée sur les Connaissances expertes et l'Inférence), impliquant le CRIM (Centre de Recherche Informatique de Montréal), l'Université de Montréal et l'UQAM, ainsi que la compagnie industrielle ALCAN. L'objectif de ce projet est de proposer une approche à base de connaissances pour apporter une solution plus flexible quant à l'automatisation du processus décisionnel pour la gestion des ressources hydriques. Le gestionnaire de connaissances est une partie de l'approche proposée. En plus de permettre aux experts du domaine de modifier et de mettre à jour la base de connaissances, ce gestionnaire doit mettre en œuvre des techniques de maintien de la cohérence et de la complétude de la BC ainsi que sa validation. Le formalisme utilisé pour bâtir le SBC est de type objet-règle. Après la phase d'acquisition de connaissances, ces dernières sont représentées par des classes suivant une conception objet à la UML (Unified Modelling Language), et des règles qui sont définies sur ces classes.

1.4 Organisation du mémoire

Le mémoire est organisé comme suit :

Le deuxième chapitre présente une description de la problématique de V&V des SBC. Il la compare avec la V&V en génie logiciel et introduit le problème de V&V des systèmes hybrides objets-règles.

Le troisième chapitre définit et détaille les différents aspects de V&V. Avec des exemples simples, nous essayons d'expliquer les différents éléments de V&V. Nous présentons aussi un ensemble de travaux sur la vérification et la validation. Comme plusieurs approches et techniques sont utilisées dans différents systèmes, le choix des travaux à présenter est dicté par le souci de couvrir ces différentes approches.

Le quatrième chapitre détaille l'architecture du gestionnaire de connaissances comme cadre général pour la V&V. Le formalisme adopté est étudié et présenté en détail, les différentes sources de connaissances utilisées sont décrites, et les différents modules du gestionnaire sont présentés.

Le cinquième chapitre présente notre modèle de V&V. Les techniques utilisées pour représenter les états et les contraintes du système y sont également détaillées. L'approche suivie pour effectuer la V&V y est décrite.

Le sixième chapitre présente la plate-forme choisie conformément au formalisme objet-règle, ainsi que les différents prototypes et les expérimentations réalisées dans le cadre du projet PReCI.

Chapitre 2

Description de la problématique

Comme tout autre logiciel, les systèmes à base de connaissances doivent être vérifiés et validés. Les bases de connaissances sont exprimées dans un style déclaratif qui sépare les connaissances du contrôle (l'inférence). Ce style et cette séparation augmentent les possibilités de vérification par rapport à l'approche de programmation standard (structurée ou objet) où les connaissances et le contrôle sont imbriqués. Par ailleurs, les progrès réalisés, tant au niveau du matériel qu'au niveau du logiciel, rend possible la construction de bases de connaissances de grande taille. Pour ce type de systèmes, la vérification et la validation sont nécessaires et même essentielles pour assurer un fonctionnement durable du système. Un tel problème n'est pas propre aux SBC mais se rencontre également en programmation classique où il est étudié dans le domaine de recherche qu'est le génie logiciel.

Si aujourd'hui, le nombre de SBC commercialisés n'est pas aussi important que le laisse supposer l'engouement soulevé par l'utilisation de ce type de logiciels, la principale raison en est l'absence des phases de vérification et de validation. En effet, comment commercialiser un produit sans garantie sur sa qualité, et sachant qu'il aura à résoudre des problèmes importants dans des domaines critiques (c'est à dire, médecine, industrie, défense, ...etc.)? Un SBC étant une abstraction de la réalité, on ne pourra jamais dire qu'il est totalement valide ou invalide. On doit néanmoins spécifier un niveau de performance acceptable par tous les intervenants dans le développement du logiciel [28].

Nous constatons alors la nécessité et l'importance de ces deux phases dans le cycle de développement et de réalisation de tout SBC.

2.1 V&V des SBC vs V&V en génie logiciel

En génie logiciel, le problème de test de programmes est étudié depuis de nombreuses années. Bien que de telles recherches aient donné des résultats fort intéressants et prometteurs, le problème est encore jugé très difficile par les chercheurs actuels du domaine[17]. La grande difficulté vient essentiellement de la diversité des erreurs possibles dans un logiciel : des erreurs peuvent exister au niveau même de la conception des

algorithmes implantés; d'autres erreurs peuvent être liées à la syntaxe du langage de programmation utilisé et à l'absence de contrôles syntaxiques suffisants. De plus, un logiciel est le résultat de l'intégration de plusieurs modules souvent conçus par des personnes ou des équipes différentes : il peut alors surgir des erreurs de structuration entre ces différents modules. Le test de logiciel consiste soit à examiner un programme (vérification statique), soit à l'exécuter (vérification dynamique) dans le but d'y trouver des erreurs.

Dans une approche génie logiciel, on est capable de définir préalablement les spécifications complètes d'un programme. Ces spécifications sont fort utiles pour vérifier et valider ces programmes; ces derniers doivent en être conformes. Dans une approche système à base de connaissances, il est difficile, voir impossible, d'avoir une spécification complète avant l'implantation d'une base de connaissances. Une telle base est le résultat d'un processus d'acquisition de connaissances auprès des experts, et ces connaissances sont sujettes, à tout moment, à des améliorations (modification, ajout et/ou suppression). La vérification et la validation des SBC consistent donc à déterminer la cohérence et la complétude interne de ces systèmes ainsi qu'à leur assurer un bon fonctionnement.

2.2 Vérification

La vérification des SBC est basée sur le concept d'anomalie. Toute anomalie qui engendre une situation d'incohérence ou d'incomplétude de la base doit être recherchée. La base de connaissances a souvent la syntaxe d'un ensemble de règles de productions; les anomalies « syntaxiques » des règles se résument à des erreurs de typage entre attributs ou à des simples fautes de frappe : Détecter de telles erreurs est souvent très facile et du ressort d'un bon analyseur syntaxique. Cependant, l'intérêt de la détection de ce type d'anomalies reste très limité : dépasser ce cadre nécessite alors l'accès à la sémantique du domaine d'expertise. Une connaissance sémantique liée au domaine est nécessaire pour décider des caractères de cohérence et de complétude d'une BC. Comme nous allons le voir par la suite, ces informations constituent le modèle de cohérence sur lequel nous nous basons pour vérifier une BC. De telles informations, liées au domaine d'expertise, ne sont généralement pas du ressort du programmeur de la BC, et ne peuvent être fournies que par un expert du domaine qui, seul, peut spécifier les contraintes sémantiques sur son domaine.

La vérification des SBC dépend du formalisme utilisé pour représenter les connaissances. Le formalisme le plus utilisé est celui des règles. Il est donc normal que la plupart des outils et des travaux de vérification portent sur les systèmes des règles.

Globalement, il s'agit de vérifier que la base de règles est exempte d'anomalies telles que la redondance, la contradiction, la circularité et l'incomplétude. Ces anomalies seront définies plus tard au cours du développement du sujet.

Comme point de départ de notre travail, nous nous intéressons, au niveau de la vérification, à l'étude des systèmes classiques qui étaient les plus souvent utilisés pour implanter des SBC. Dans de tels systèmes, la base de connaissances est constituée d'un ensemble de faits logiques indépendants et d'un ensemble de règles formulées dans une logique de premier ordre ou logique de propositions.

2.3 Validation

L'application à grande échelle des systèmes à base de connaissances dans plusieurs domaines d'activités impose des besoins forts en terme de qualité et d'efficacité. De par la nature non-conventionnelle de ces systèmes, les méthodes conventionnelles utilisées pour s'assurer de la qualité logicielle, ne peuvent pas s'appliquer. Certains aspects des SBC sont similaires à ceux des autres types de logiciels (comme les interfaces usagers), mais leur caractéristique exceptionnelle réside dans la capacité de résoudre des problèmes nécessitant un ensemble de connaissances plus larges. Cette caractéristique particulière des SBC les rend plus difficiles à évaluer et certifier.

L'activité de validation rentre dans le cadre de l'assurance qualité des SBC, il s'agit de démontrer qu'un SBC est capable de trouver les conclusions correctes. Si dans la vérification on s'intéresse à la correction des structures (faits, règles, attributs, etc), dans la validation, il s'agit de prouver que le système fonctionne comme prévu par l'expert. Ces deux activités sont donc complémentaires, en termes de la nature des erreurs révélées. En effet, la vérification a pour objectif la détection des erreurs commises lors de la codification de la base de connaissances (insertion de règles dupliquées, circulaires, etc), tandis que la validation est nécessaire pour la détection des erreurs de formulation de la base de connaissances (par exemple des connaissances inappropriées ou incorrectes).

Même si la vérification et la validation sont toutes les deux importantes pour assurer la fiabilité de tout SBC, il y a une grande différence entre les états de l'art de ces deux activités. Tandis que les outils de vérification automatique sont avancés, les travaux en validation ne sont pas abondants. La plupart d'entre eux concernent la définition d'un niveau de performance que doit respecter le SBC pour atteindre les besoins spécifiés. Typiquement, ceci

implique la comparaison des performances du système avec celles de l'expert sur la base d'un ensemble de cas dont la solution est préalablement donnée par l'expert.

Un autre problème concerne la façon d'effectuer la validation en l'absence d'un nombre suffisant de cas de test. La solution évidente dans cette situation consiste à créer des cas de test synthétiques et d'effectuer la validation à travers eux. Habituellement l'expert être chargé de la spécification des cas de test. Mais cette approche a des inconvénients et des limitations car les résultats de validation deviennent dépendants des cas fournis par l'expert. Le processus de validation devient aussi semi-automatique.

2.4 Problématique spécifique

La vérification est vue comme une boite blanche dans laquelle l'expert peut contrôler le processus de vérification en le portant sur la totalité ou une partie de la BC. Aussi il peut spécifier sur quoi la vérification peut être portée en précisant les types d'erreurs ou d'anomalies à chercher au cours du processus. Par contre la validation est vue comme une boîte noire; l'expert fournit des cas de test au système et obtient les conclusions de celui-ci, ces résultats vont être comparés pour trouver des défaillances dans le système.

Partant du choix d'un nouveau paradigme de représentation de connaissances basé sur l'intégration de la représentation en objets avec celle de règles pour former une BC plus structurée, nous nous intéressons particulièrement à la V&V de ce type de systèmes hybrides. Le grand apport de ce formalisme réside dans la structuration et la formulation des connaissances. Mais, comme nous allons le voir par la suite, l'intégration des techniques OO avec les systèmes de règles rend les techniques de V&V des systèmes classiques inutiles; la plupart des systèmes classiques ont utilisé des approches statiques, ces dernières sont inadéquates pour des règles contenant des appels de méthodes et des expressions plus complexes.

Dans un premier temps, nous allons étudier plus en détail les différents aspects de vérification et de validation des systèmes de règles. Au cours du prochain chapitre, nous allons donner les définitions des différentes erreurs et anomalies que nous devons rechercher pendant la V&V d'un SBC ainsi que les techniques utilisées pour les détecter. Avec des exemples simples, nous essayons d'expliquer les différents éléments de V&V. Nous présentons aussi quelques travaux de recherche dans le domaine.

Chapitre 3

Vérification et validation des systèmes de règles

En représentation de connaissances, nous trouvons des catégories syntaxiques de base du langage qui sont représentées par des symboles se rapportant explicitement au domaine du discours (ou domaine d'expertise). Nous trouvons aussi des constantes qui désignent des noms spécifiques d'objets, de personnes ou d'événements, des variables qui désignent des noms génériques pour des personnes, des objets ou des événements et des prédicats qui désignent des règles d'assemblage entre constantes et variables. Ces spécifications définissent une logique de propositions (pas de variables) ou une logique de premier ordre. Nous considérons ces deux formalismes comme systèmes ou approches classiques pour la représentation des connaissances. Certains langages de programmation (Prolog par exemple) sont basés sur une grammaire définie par une logique de premier ordre. Les systèmes de règles que nous allons étudier ont utilisé ce genre de grammaire pour construire leurs bases de connaissances.

Une base de connaissances regroupe généralement deux types de connaissances:

- Des connaissances dites factuelles, qui permettent de décrire le domaine d'application du système. Nous appellerons BF (base de faits) cette partie des connaissances
- Des connaissances déductives, grâce auxquelles la connaissance du domaine d'application et des connaissances caractérisant un problème particulier vont pouvoir être utilisées pour inférer de nouvelles connaissances. Nous appellerons BR (base de règles) cette partie de la connaissance. Cette partie est généralement représentée sous forme de règles de production.

Une base de connaissances (BC) est donc la réunion de BF et de BR : BC = BF $\cup$BR.

Une règle dans BR a la forme suivante :

$$L_1 \wedge L_2 \wedge \ldots \wedge L_n \rightarrow M$$

où L_i et M sont des littéraux.

Si la condition $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ de la règle est satisfaite, celle ci est déclenchée, et on peut dériver M. Un littéral M peut être dérivé à partir d'une base de connaissances BC si M est une conséquence logique de BC.

Dans une approche classique de représentation de connaissances, la base de connaissances utilise le style déclaratif pour exprimer les inférences; les différentes parties d'une inférence (conditions et actions) sont formulées sous forme de faits en utilisant une logique de prédicats. La base de faits, elle aussi, est décrite sous forme de prédicats. L'approche adoptée dans notre cas, qui sera détaillée par la suite dans le gestionnaire de connaissances, consiste à représenter la base de faits par une base d'objets. Celle-ci respecte le schéma de classes qui modélisent le domaine d'application, les inférences sont exprimées en utilisant les définitions de classes du domaine. En partant de ces deux paradigmes (déclaratif et objet), les aspects de vérification et de validation vont se baser sur ce type de représentation.

Nous nous intéressons, dans un premier temps, à la V&V des systèmes de règles. Nous détaillons les différents aspects de la V&V, les approches utilisées, et nous présentons quelques travaux de recherche qui ont porté sur la V&V de ces systèmes. Ces travaux ont, pour la plupart, donné naissance à des systèmes opérationnels que nous présentons en détail. Il faut noter qu'à cause de la non possibilité d'accès à ces outils, nous n'avons pas pu les tester et les expérimenter. Par contre, les articles que nous avons étudiés les décrivent en détail.

3.1 Vérification

Avant de détailler les différents types de vérification qui doivent être effectués sur une BC, nous allons décrire tout d'abord l'ensemble de spécifications en terme de cohérence et de complétude. Ces spécifications incluent des contraintes que la BC doit respecter durant la durée de vie du système.

3.1.1 Étude de la cohérence d'une base de connaissances

La vérification d'une base de connaissances suppose que soit donné (implicitement ou explicitement) un modèle conceptuel du monde « réel » qui servira à l'étude de la cohérence. C'est par référence à ce modèle que l'on va pouvoir vérifier la cohérence de la base de connaissances sans avoir à faire appel à un expert. Le concept de cohérence représentera en fait la compatibilité de la base de connaissances, censée modéliser une partie du monde « réel», avec la référence que constitue le modèle de cohérence. Ce dernier rassemble les définitions de cohérence statique que doit vérifier un fait, un ensemble de faits, une règle ou un ensemble de règles.

Cette idée n'est pas spécifique aux SBC. Dans le cas des bases de données, un système de gestion de bases de données doit assurer le maintien de la cohérence des données par rapport aux schémas (contrôles de type), et également entre elles (contrôle de redondance). Ceci est fait en se basant sur les contraintes d'intégrité que doit respecter la base de données.

Un modèle de cohérence exprime des propriétés et des liens sémantiques entre les entités de la base de connaissances. On peut y trouver par exemple des contraintes d'intégrité exprimant les incompatibilités sémantiques. Le modèle de cohérence peut contenir des spécifications dépendantes ou indépendantes du domaine d'application. Ainsi, considérer que les valeurs « grand » et « petit » pour un attribut sont incompatibles relève du domaine d'application. Par contre analyser la correction syntaxique d'une BC, l'absence de contradiction, de redondances et de cycles est une exigence totalement indépendante du domaine d'application étudié. De plus, il n'existe pas un modèle de cohérence universel. Selon le contexte d'utilisation d'une BC en particulier, on peut trouver des contraintes à respecter qui ne sont pas valables dans d'autres contextes.

Dans ce qui suit, nous allons dissocier les propriétés concernant les faits (dépendantes du domaine), des propriétés concernant les règles (indépendantes du domaine).

3.1.1.1 Propriétés des faits et des ensembles de faits

Les propriétés que nous allons décrire sont très fortement liées au domaine d'application. Leur puissance d'expression dépend plus ou moins du mode de représentation de la connaissance. Par exemple toutes ces propriétés peuvent s'exprimer sous la forme de « contraintes de cohérence » pour les quelles on peut faire un parallèle avec les contraintes d'intégrité d'une base de données.

Dans le cas d'une représentation objet, la description des classes, la hiérarchie et les liens entre classes, ainsi que les attributs de ces classes constituent déjà une bonne partie des propriétés qui doivent être vérifiées. Nous définissons en plus d'autres propriétés comme :

- Les contraintes de cohérence
- L'ensemble de valeurs autorisées pour un attribut
- Le degré d'un attribut ou son arité maximale

a) Les contraintes de cohérence

Analogues aux contraintes d'intégrité pour les bases de données, les contraintes de cohérence permettent de décrire de manière tout à fait générale des situations potentielles d'incohérence. La cohérence d'un ensemble de faits se définit alors par l'absence de telles situations.

Par exemple l'expression,

Si X est Humain Λ V est Ville Λ X non-résident-à V Λ X électeur-dans V Alors *incohérence*.

est une spécification d'un ensemble de faits incohérents, car pour être électeur dans une ville donnée il faut y être résident. La présence de ces faits ensemble permet de produire une situation d'incohérence. Cette forme a été utilisée dans des systèmes comme COVADIS[17] et EVA[8].

b) L'ensemble de valeurs autorisées pour un attribut

Pour chaque attribut, on définit l'ensemble de valeurs qu'il peut prendre. Cette spécification peut se faire soit sous la forme d'une liste exhaustive des valeurs autorisées (extension), soit par une fonction d'appartenance (intention). Par exemple, l'attribut «continent » de la classe « pays » peut avoir des valeurs dans l'ensemble{Afrique, Europe, Asie, Amérique, Australie}.

c) Le degré d'un attribut ou son arité maximale

C'est le nombre maximum de valeurs que peut prendre un attribut d'une classe donnée. Pour un attribut qui ne peut prendre qu'une seule valeur, on parle de contrainte de monovaluation. C'est le cas échéant pour les attributs des classes. Par exemple, l'attribut «a pour parents» de la classe «individu» peut avoir au maximum deux valeurs.

3.1.1.2 Propriétés des règles

Les propriétés d'incohérence pour un ensemble de règles sont moins fortes et moins absolues que les propriétés d'incohérence pour un ensemble de faits. Pour les ensembles de faits, les incohérences décrivent des interdits. Pour les règles, l'absence de redondance est une propriété que l'on peut exiger d'un ensemble de règles, et encore là, ce n'est pas systématique. Bien que des redondances soient détectées dans une base de connaissances, elles ne sont pas forcément supprimées. Elles peuvent être utiles d'un point de vue stratégique pour la résolution. Le système les déclare comme anomalies potentielles, et c'est à l'expert que revient la charge de décider de l'action convenable à effectuer sur la base de connaissances (supprimer, modifier, ou ne rien faire).

L'ensemble des propriétés suivantes est défini sur la base de règles :

a) La cohérence interne d'une règle

La cohérence interne d'une règle est vérifiée quand les constituants de ses prémisses et ses conclusions sont cohérents. En vérifiant la cohérence des instances possibles de

l'ensemble des faits contenus dans les prémisses et les conclusions d'une règle on peut juger la cohérence interne d'une règle. Par exemple, on donne la règle suivante,

Si X est canadien $\wedge$ âge(X) >= 18 alors état(X) $\leftarrow$ adulte.

Cette règle vérifie la cohérence interne si la classe « canadien » existe, si cette classe possède les attributs « âge » et « état », si l'attribut « âge » peut prendre des valeurs supérieures ou égales à 18 et si l'attribut « état » peut prendre entre autres la valeur « adulte ».

b) Règles redondantes

Dans une base de connaissances contenant n règles, R_1, R_2,...,R_n, la règle R_j est redondante si elle est une conséquence logique des règles R_1,...R_{j-1}, R_{j+1},...,R_n. C'est à dire que l'on peut avoir les conclusions de R_j en l'absence de celle ci.

Cette définition inclut les deux cas suivants :

- Règles subsumées : une règle R_1 est subsumée par une autre règle R_2 si sa partie prémisse implique celle de R_2 et les deux règles ont une conclusion en commun.

  Par exemple, si on a deux règles R_1 et R_2 telles que

  R_1 : Si X est professeur $\wedge$ X est titulaire alors X est membre-faculté

  R_2 : Si X est professeur alors X est membre-faculté.

  Alors R_1 est subsumée par R_2.

- Règles dupliquées : deux règles R1 et R2 sont dupliquées si elles ont les mêmes conditions et les mêmes conclusions. Remarquons que la duplication est un cas particulier de la subsumption.

  Par exemple, si on a deux règles R_1 et R_2 telles que

  R_1 : Si X est étudiant $\wedge$ X a-degré maîtrise alors X est étudiant-gradué.

  R_2 : Si Y a-degré maîtrise $\wedge$ Y est étudiant alors Y est étudiant-gradué.

  alors R_1 et R_2 sont dupliquées.

c) Règles inutiles

Une règle R dans une base de connaissances BC est inutile si sa partie conclusion n'est unifiable ni avec un littéral but ni avec un littéral de la partie prémisse d'une autre règle.

Par exemple,

R : Si X est enseignant $\wedge$ X est titulaire alors X est professeur

Si « X est professeur » n'est pas déclaré comme but final ou ne figure dans aucune condition des règles de la base de connaissance, cette règle est alors inutile.

d) Règles inatteignables

Une règle R est inatteignable si un littéral de sa partie prémisse n'est unifiable ni avec un littéral en entrée ni avec une conclusion d'une autre règle de la base de connaissances.

Par exemple,

R : Si X est étudiant $\wedge$ X a-degré maîtrise alors X est étudiant-gradué.

Si « X a-degré maîtrise » n'est pas donné comme un fait initial et ne figure pas dans la conclusion des autres règles de la base de connaissances, R est alors inatteignable.

e) Règles en conflit

Commençons tout d'abord par le cas particulier de deux règles. R_1 et R_2 sont en conflit si la partie prémisse de R_1 subsume la partie prémisse de R_2, et leurs conclusions infèrent une situation d'incohérence.

Par exemple,

R_1 : X inscrit-en Y $\wedge$ Y est cours-gradué alors X est étudiant-gradué.

R_2 : X inscrit-en Y $\wedge$ Y est cours-non-gardué alors X est étudiant-non-gradué.

Partant des faits initiaux {Kamal inscrit-en ift6330, Kamal inscrit-en ift3310, ift6330 est cours-gardué, ift3310 est cours-non-gardué }, on déduit que « Kamal est étudiant-gradué », et « kamal est étudiant-non-gradué » ce qui dénote une situation d'incohérence.

Cette anomalie peut être généralisée à un ensemble de règles. On parle alors d'un ensemble de règles contradictoires. Une chaîne de règles contradictoire est un ensemble ordonné de règles $R_1,\ldots,R_n$ tel que les conclusions d'une règle et une partie des prémisses de la règle suivante soient appariables et tel que les conclusions de la dernière règle R_n et les prémisses de la première règle R_1 infèrent une situation d'incohérence.

Par exemple,

R_1 : Si X est étudiant Λ X a-degré maîtrise alors X est étudiant-gradué.

R_2 : Si X est étudiant-gradué alors X est prof-assistant.

R_3 : Si X est prof-asssistant alors X est professeur.

On peut avoir en même temps « X est étudiant » et « X est professeur », ce qui représente une situation de conflit.

f) Règles circulaires

Une boucle de règles est un ensemble ordonné de règles $R_1,\ldots,R_n$ tel que les conclusions d'une règle et une partie des prémisses de la règle suivante soient appariables et tel que les conclusions de la dernière règle R_n et une partie des prémisses de la première règle R_1 soient appariables également. Par exemple,

R_1 : X a-degré Y alors X est étudiant-gradué

R_2 : X est étudiant-gradué alors X a-degré Y

R_1 et R_2 forment une boucle.

3.1.2 Étude de la complétude d'une base de connaissances

Une base de connaissances est incomplète s'il existe un fait initial pour lequel un but doit être conclu alors qu'il ne l'est pas. L'application de cette définition requiert une spécification détaillée et explicite de l'ensemble des buts finaux pour l'ensemble des faits initiaux. Tant que la base de connaissances en elle-même est décrite de cette façon, cette définition est difficile à mettre en pratique. Une façon de procéder est de considérer que, pour chaque ensemble de faits initiaux possibles, des conclusions doivent être obtenues. Par exemple (en logique de premier ordre),

R1 : Si p(a) alors r(a)

R2 : Si $\neg$ p(a) Λ q(a) alors $\neg$ r(a)

Pour les faits initiaux $\{\neg$ p(a), $\neg$ q(a)$\}$ et $\{\neg$ q(a)$\}$ on ne peut pas obtenir des conclusions ce qui implique une incomplétude dans la base de connaissances.

Avec cette considération, il est encore difficile d'effectuer une vérification totale de la complétude, en raison du nombre élevé des faits initiaux qui peuvent être utilisés en entrée. Une des approches utilisées [1,2] consiste à essayer de localiser certains symptômes qui indiquent une incomplétude potentielle dans la base de connaissances. Quelques symptômes connus sont la présence des littéraux inutiles et des valeurs manquantes.

a) Littéraux inutiles

Si un littéral est déclaré demandable dans l'ensemble de spécifications de la base de connaissances et qu'aucune règle ne l'utilise, ce littéral est inutile. Cette vérification nous permet de constater que des règles, qui utilisent ce littéral, sont probablement manquantes.

b) Valeurs manquantes

Si certaines des valeurs d'un attribut parmi l'ensemble de valeurs autorisées ne sont utilisées dans aucune règle de la base de connaissances, on peut supposer qu'il y a des règles manquantes qui doivent utiliser ces valeurs.

3.1.3 Approches de vérification

Nous allons maintenant voir les différents types de vérification de bases de connaissances, en se référant au modèle de cohérence et à la définition de la complétude d'une BC.

3.1.3.1 Vérification statique

Partant des spécifications énoncées dans le modèle de cohérence, il s'agit de vérifier chaque partie de la BC (BF et BR). Cette vérification ne tient pas compte de la puissance déductive de la base. Dans les sections précédentes, nous avons défini les différentes propriétés de cohérence statique que doivent vérifier les faits et les règles. Cette vérification est aisée pour les propriétés concernant les faits, les propriétés de redondance des règles et les conflits entre règles. Elle est cependant plus ardue pour la recherche de boucles de règles ou de chaînes contradictoires.

a) Vérification statique de la base de faits

Il s'agit de vérifier qu'un fait (ou un ensemble de faits) n'est pas en contradiction avec la définition de la cohérence.

Par exemple, (état-matrimonial = adulte) est un fait incohérent, la valeur « adulte» n'appartenant pas au domaine de valeurs de « état-matrimonial ».

En ce qui concerne la cohérence statique d'un ensemble de faits (BF), la vérification consiste à relever un ensemble de faits incohérents.

Exemple : (x taille grande) et (x taille petite) constituent un ensemble de faits incohérents si les valeurs « grande » et « petite » d'une taille sont deux valeurs contradictoires dans le modèle de cohérence.

b) Vérification statique de la base de règles BR

Les vérifications sont effectuées sans déclenchement de règles. Ainsi, si dans un ensemble de règles deux d'entre elles permettent de conclure que le même attribut peut avoir deux valeurs contradictoires, alors on peut conclure qu'elles sont statiquement incohérentes. Cependant, il faut noter qu'il est possible que ces deux règles ne soient jamais simultanément déclenchables par le moteur d'inférence.

Par exemple, si (x taille grande) alors (x taille petite) est statiquement incohérente.

Exemple pratique : Système EVA

Les premiers systèmes experts (MYCIN, ONCOIN) ont été vérifiés en utilisant des analyses statiques de leurs bases de connaissances (outils TEIRESIAS et RCP) [3,6]. L'outil qui représente le mieux cette approche est le système CHECK [4,5,6]. Après CHECK, d'autres outils ont été implantés en améliorant les travaux déjà réalisés; nous pouvons citer notamment les systèmes ARC [3], EVA [8,9,10,11], KB-Reducer [3], COVER [1], etc. Parni ces systèmes, nous prenons comme exemple le système EVA.

Le système EVA (Expert system Validation Associate) est un projet, lancé au milieu des années 80 par Lockheed Martin Corporation dans le but de développer un outil générique de vérification et de validation des systèmes experts. Il a été entièrement implanté en Prolog. EVA est un ensemble de modules intégrés. Les modules de vérification permettent la détection des anomalies découlant de la structure de la base de connaissances (vérification des structures, vérification logique, etc.).

EVA n'est lié à aucun langage de règles ou coquille (Shell). Un de ses points forts réside dans le fait qu'il offre un outil de traduction des bases de connaissances, crées à l'aide des langages et outils de développement des systèmes experts les plus connus (ART, OPS5, LES, etc.), vers une forme canonique interne basée sur la logique de prédicats. EVA supporte aussi un méta-langage qui peut être utilisé pour exprimer des contraintes sémantiques (modèle de cohérence). Pour chaque base de connaissances écrite dans un langage supporté par EVA, ce dernier lit et traduit cette base source et stocke le résultat dans sa base de données interne. Les fonctionnalités de EVA sont présentées dans la figure 3-1 [8].

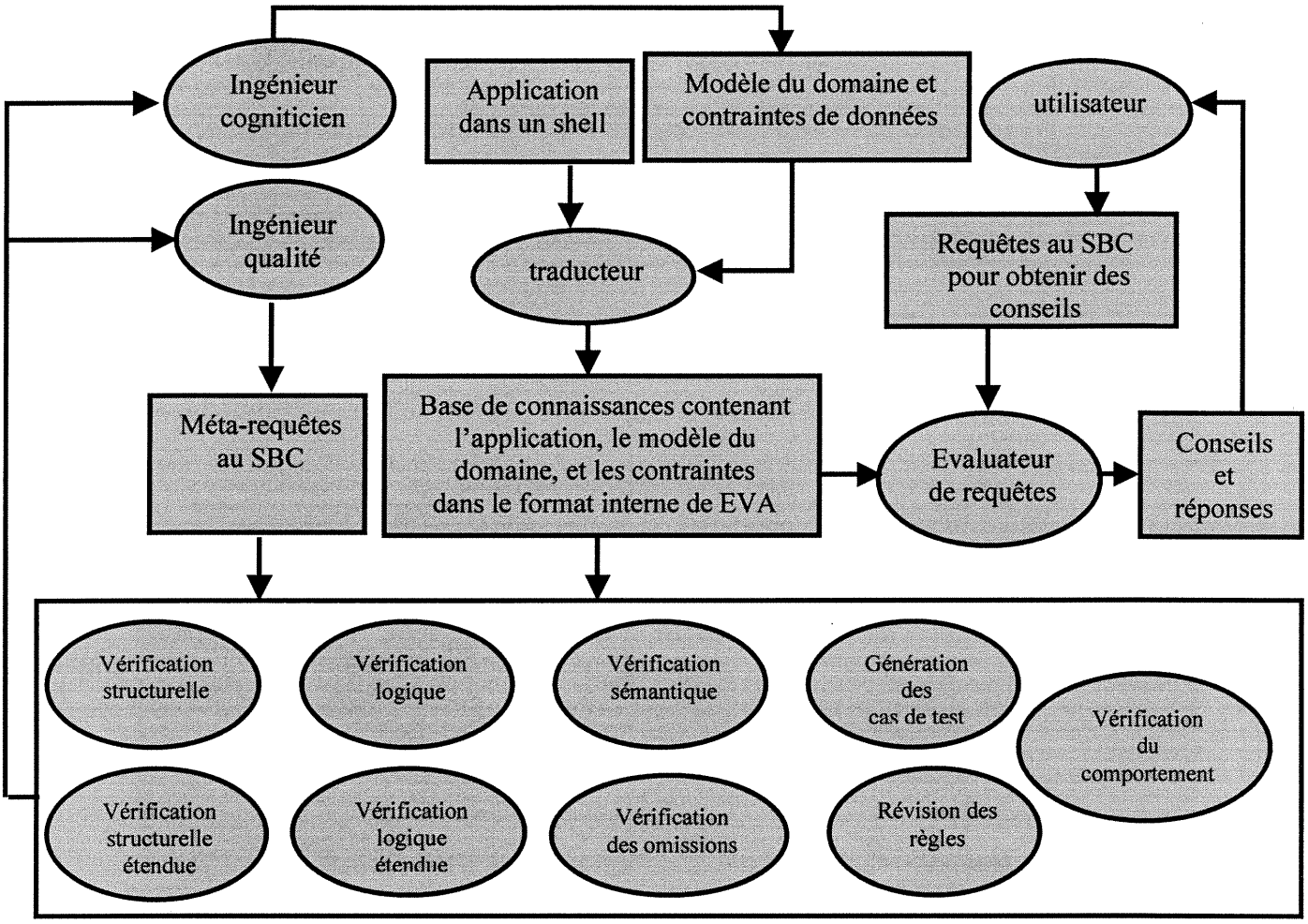


Figure 3-1: Architecture de EVA

Dans ce qui va suivre, nous présenterons les différents modules en rapport avec la vérification, à savoir la vérification structurelle, la vérification structurelle étendue, la vérification logique, la vérification logique étendue, la vérification sémantique et la vérification des omissions.

❑ Vérification structurelle

Elle permet de détecter les quatre types d'anomalies suivantes dans une BC : les règles inatteignables, les littéraux inaccessibles, la redondance et les cycles.

- Règles inatteignables : une règle est inatteignable si sa partie gauche LHS (Left Hand Side) n'est jamais satisfaite parce qu'elle contient un littéral qui ne peut pas correspondre à un fait initial ou à un littéral d'une partie droite RHS (Right Hand Side) d'une autre règle.
- Faits ou littéraux inaccessibles : un fait ou un littéral d'une partie RHS d'une règle est appelé littéral inaccessible s'il ne peut pas correspondre à un littéral d'une

partie LHS d'une autre règle. Un littéral inaccessible n'est jamais utilisé et doit être éliminé ou la règle courante doit être modifiée.

- Redondance : une règle est redondante si elle est dupliquée ou subsumée par une autre. Deux règles sont dupliquées si les clauses des conditions et des actions sont les mêmes respectivement, possiblement dans des ordres différents et avec des noms de variables différents. La subsumption est vérifiée lorsque les deux règles ont des conditions identiques et les actions d'une règle est un sous-ensemble des actions de l'autre.
- Cycles : les prédicats impliqués dans une base de connaissances peuvent être soit des prédicats primitifs soit des prédicats dérivés. Habituellement, un prédicat primitif est utilisé pour représenter un fait indépendant, tandis qu'un prédicat dérivé représente un fait dépendant d'autres faits. Un prédicat dérivé est déduit à partir des règles en terme de prédicats primitifs ou dérivés. Pour chaque prédicat dérivé, EVA calcule un méta prédicat «derive» qui donne le chemin de sa dérivation. Un chemin de dérivation est une liste de règles. S'il y a des duplications dans cette liste, alors il y a un chemin circulaire dans la BC.

  Par exemple,

  (1) parent(X,Y) → antécédent(X,Y).

  (2) parent(X,Y), antécédent(Y,Z) → antécédent (X,Z).

  derive(antécédent,[1]).

  derive(antécédent,[2,1]).

  derive(antécédent,[2,2]).

Le dernier calcul du méta-prédicat «derive» indique une boucle. Il est donc nécessaire de définir un point d'arrêt en modifiant la deuxième règle.

❑ Vérification structurelle étendue

Elle permet de détecter les mêmes anomalies que la vérification structurelle (règles inatteignables, redondance, cycles) en utilisant la relation de hiérarchie qui existe entre les entités statiques de la base de connaissances (relation *is-a* exprimée par la plupart des Shells). EVA utilise le méta-prédicat «synonymous» pour exprimer cette relation et essaie de trouver d'autres situations symptomatiques des anomalies.

Par exemple,

Règle 1 : marin(X) → plonger(X)

Règle 2 : sous-marin(X) → plonger(X)

Si on a « marin » et « sous-marin » déclarés comme synonymes alors ces deux règles sont dupliquées.

▫ Vérification logique

Une base de connaissances est incohérente si elle permet de dériver des conclusions contradictoires. La vérification logique a pour but de détecter les incohérences dans la BC.

Pour trouver des incohérences logiques, EVA utilise le prédicat «incompatible» comme but à atteindre. En procédant par chaînage arrière, une session de résolution de ce but est initiée. S'il y a une réponse pour ce but alors la BC est incohérente, sinon elle est cohérente.

Par exemple,

fait 1 : auto(c).

fait 2 : nouveau(c).

fait 3 : grand(c).

fait 4 : casse(c).

règle 1: auto(X), casse(X) $\rightarrow$ mauvais(X).

règle 2 : auto(X), nouveau(X), grand(X) $\rightarrow$ cher(X).

règle 3 : auto(Y), mauvais(Y) $\rightarrow$ $\neg$cher(Y).

Le but à atteindre est alors représenté par le prédicat incompatible(B, $\neg$B). Pour cet exemple, ça serait incompatible(cher(c), $\neg$ cher(c)). En procédant par chaînage arrière, on peut prouver à la fois cher(c) et $\neg$ cher(c), ce qui donne lieu à une situation d'incohérence.

▫ Vérification logique étendue

Comme dans la vérification structurelle étendue, ce module permet de trouver d'autres situations d'incohérence en utilisant la relation d'hiérarchie exprimée par le méta-prédicat «synonymous» combiné avec le méta prédicat «incompatible».

Par exemple,

règle 1 E $\wedge$ F $\rightarrow$ envoyer-sous-marin.

règle 2 : E $\wedge$ F $\rightarrow$ $\neg$envoyer-marin.

Sachant que « envoyer-sous-marin » et « envoyer-marin » sont synonymes, on peut déduire le fait incompatible(envoyer-sous-marin , $\neg$ envoyer-sous-marin). Ces deux règles sont donc en conflit.

Le méta prédicat «incompatible» n'est pas limité à une paire de littéraux complémentaires. Il peut aussi être utilisé pour représenter un ensemble de littéraux qui ne peuvent pas être vrais ensemble (par exemple incompatible(prof-adjoint(c), permanent(b))).

▫ Vérification sémantique

Elle a deux fonctions majeures : vérifier que les faits et les règles d'une base de connaissances ne violent pas les contraintes sémantiques et vérifier également que les contraintes sémantiques sont cohérentes entre elles. Il existe deux types de contraintes, les contraintes positives et les contraintes négatives. Les faits et les règles de la base de connaissances doivent satisfaire les premières, mais pas les dernières. Les contraintes négatives sont représentées par le méta prédicat «incompatible».

Par exemple,

incompatible(titre,membre-université,[(retraité,non-retraité),(retraité,étudiant),(retraité, employé)]) signifie qu'aucun *membre-université* ne peut avoir, pour le prédicat *titre*, les valeurs « retraité » et « non-retraité », « retraité » et « étudiant », ou « retraité » et « employé » respectivement en même temps.

Les contraintes positives sont spécifiées à l'aide de différents méta-prédicats. Ces derniers supportent les contraintes sémantiques à respecter lors de la formulation de la base de connaissances (valeurs minimums et maximums, valeurs possibles, hiérarchie, etc.).

La vérification consiste à prendre une contrainte négative ou la négation d'une contrainte positive comme but et essayer de la prouver. Si ce but est atteint, alors la BC viole les contraintes sémantiques. En examinant l'arbre de déduction, on peut trouver les faits et les règles qui ont contribué à la violation de cette contrainte. Le développeur peut les étudier et apporter les modifications nécessaires.

▫ Vérification des omissions

Elle permet de trouver s'il y a des faits ou des règles manquantes dans une base de connaissances. Pour effectuer ce traitement, on a besoin d'une spécification complète de toutes les entrées du système. Pour chaque entrée possible, la BC est complète si une sortie peut être produite à partir de cette entrée. Si aucune sortie n'est déduite pour certaines entrées, la BC peut avoir des règles ou faits manquants.

Puisqu'il y a un grand nombre d'entrées possibles, il est pratiquement impossible de tester la totalité des cas. On utilise donc des méthodes indirectes. Une de ces méthodes consiste en l'exploitation des contraintes de complémentarité. Par exemple, s'il y a une règle qui utilise un attribut avec une certaine valeur, alors il doit exister d'autres règles utilisant le même attribut avec le reste des valeurs possibles pour cet attribut. Sinon, il y a des règles manquantes.

En conclusion, EVA a essayé d'améliorer les techniques de vérification statique réalisées par les premiers systèmes (CHECK, TEIRESIAS, etc.) en les rendant indépendantes de la grammaire utilisée pour représenter les connaissances. Son méta-langage, utilisé pour représenter des contraintes sémantiques, rend ses techniques trop spécifiques et difficilement réutilisables.

3.1.3.2 Vérification dynamique

La vérification statique est basée sur l'examen des règles prises une à une (cohérence interne, prémisses inutiles, etc.) ou deux à deux (règles subsumées, règles en conflit, etc.). Le test de boucle ou de chaînes contradictoires à partir d'une règle donnée nécessite au maximum l'examen de toutes les règles. Ces traitements rendent la vérification statique très coûteuse particulièrement si la base est volumineuse.

Une approche de vérification dynamique, par contre, repose sur l'étude des chaînes de déduction. On distingue deux approches :

a) Approche exhaustive

Elle exploite le modèle de cohérence pour en tirer toutes les spécifications de toutes les situations incohérentes. Partant de ces spécifications d'incohérences, cette approche consiste à prouver qu'elles peuvent être atteintes ou non à partir d'un ensemble de faits cohérents. Si tel est le cas, la base de connaissances est incohérente, dans le cas contraire elle est cohérente. Le système INDE [17] et COVADIS [17] ont utilisé des techniques systématiques et exhaustives pour la vérification.

b) Approche heuristique

Elle s'appuie sur l'utilisation des méthodes de recherche non systématiques et donc moins coûteuses et accepte le risque d'incomplétude puisqu'elle utilise des heuristiques pour focaliser la recherche sur les conjectures d'incohérences les plus plausibles. En revanche, elle apporte une réponse (au moins partielle) là où la complexité combinatoire est un handicap pour l'approche exhaustive. C'est une approche suivie par SACCO [19]

En utilisant des heuristiques de sélection, on essaie d'extraire à partir du modèle de cohérence un ensemble de conjectures d'incohérences intéressantes. Ces dernières vont être utilisées comme entrée à la base de connaissances. À ce niveau, d'autres heuristiques peuvent être utilisées pour sélectionner une conjecture d'incohérence et essayer de la prouver.

Exemple pratique : Système SACCO

SACCO est un outil de vérification développé au sein du laboratoire LIA à l'Université de Savoie en France [18]. Les objectifs de la première version de ce système étaient :

- L'acquisition interactive des connaissances et du modèle conceptuel de la base de connaissances.
- Le contrôle statique des diverses connaissances tout au long de leur saisie.
- La vérification de la cohérence dynamique, à des moments choisis par l'expert.

Le système SACCO est destiné à traiter des bases de connaissances comportant des connaissances factuelles sous la forme de triplets (objet, attribut, valeur) et des connaissances déductives sous forme de règles de production comportant des variables.

Le principal apport de SACCO est la technique de vérification dynamique adoptée. C'est pourquoi nous l'avons classé comme exemple représentatif de cette approche.

▫ Formalisme de représentation de connaissances utilisé par SACCO

Les connaissances factuelles décrivant le domaine d'application ou un problème particulier sont décrites à l'aide des triples (objet, attribut, valeur) ou (objet, relation, objet).

Les connaissances déductives sont des règles de production sous la forme:

Si premisse1(x,y,...) Λ premisse2(x,y,...) Λ....Λ

Predicat(x,y)

Alors consequent1(x,y,...), consequent2(x,y,....). où x,y,... sont des variables.

Pour exprimer les contraintes sémantiques, SACCO utilise un modèle de contraintes appelé C-Modèle. Ce modèle définit les propriétés sur la base de faits et celles sur la base de règles. Au niveau des propriétés sur les faits, on retrouve les concepts de classes, objets instances de classes, attributs des classes, attributs hérités par les sous-classes, valeurs d'attributs, ensemble de valeurs autorisées pour chaque attribut, les degrés des attributs, ensemble de valeurs simultanément contradictoires, couples d'attributs exclusifs, et l'ensemble d'attributs clés. Au niveau des propriétés sur les règles, on retrouve les définitions de cohérence interne, redondance, conflit entre deux règles, règles ayant une prémisse inutile, boucles de règles, et les chaînes contradictoires.

▫ Vérification statique de la cohérence

Cette vérification se fait d'une façon interactive, lors de l'acquisition d'une connaissance (fait ou règle), le système déclenche les contrôles correspondants.

L'ajout d'un fait déclenche les contrôles suivants :

- Vérifier que le premier élément du triplet (objet) a une classe, sinon la définir
- Vérifier que le second élément est bien un attribut autorisé pour cette classe
- Vérifier que le degré de l'attribut n'est pas dépassé
- Vérifier que la valeur prise est autorisée pour l'attribut
- Vérifier que dans l'ensemble des valeurs prises par cet attribut ne figure pas un sous-ensemble de valeurs simultanément contradictoires
- Vérifier qu'aucune propriété d'exclusion entre deux attributs n'est violée par l'ajout de la nouvelle valeur
- Vérifier que la contrainte des ensembles d'attributs clés n'est pas violée
- Vérifier enfin, qu'aucune contrainte de cohérence ne devient déclenchable.

L'ajout d'une règle déclenche les contrôles suivants :

- Vérifier la propriété de cohérence interne
- Vérifier si son ajout engendre une redondance
- Vérifier qu'elle n'entre pas en conflit avec une autre règle
- Vérifier que toutes ses prémisses sont utiles
- Vérifier qu'il n'y a pas apparition de boucles ou de chaînes contradictoires.

Une règle ne vérifiant pas la propriété de cohérence interne est rejetée automatiquement par le système. Dans les autres cas, les anomalies sont signalées à l'expert qui jugera des modifications nécessaires. L'ajout d'une contrainte au C-Modèle déclenche la vérification de cette propriété sur la base de connaissances déjà acquise.

❑ Recherche d'incohérences dynamiques

Ceci se fait en deux étapes. Premièrement, on essaie de limiter le nombre de conjectures d'incohérences et deuxièmement on essaie de prouver celles retenues.

À partir du C-Modèle, on utilise des heuristiques pour définir un nombre limité de conjectures d'incohérences. En général, l'expert n'a qu'une idée imprécise des incohérences qu'il va essayer de découvrir. Quatre méthodes sont utilisées pour limiter le nombre de conjectures d'incohérence à étudier :

- Faire porter la construction des conjectures d'incohérence sur une partie du C-Modele seulement, ceci permet de ne pas prendre en compte les conjectures qui seraient issues de certaines propriétés. Par exemple, on exclut les propriétés concernant telle classe ou tel attribut.

- Limiter la complexité des conjectures pertinentes. Par exemple ne pas s'intéresser aux conjectures qui auraient plus de trois prémisses et plus de deux variables.
- Spécialiser le C-Modele en définissant des valeurs singulières pour les attributs. L'ensemble des valeurs non-autorisées n'est pas toujours défini et quand il l'est, il est souvent très grand. Donc il est nécessaire de définir un nombre restreint de valeurs significatives, dites valeurs singulières extérieures, c'est à dire un ensemble de valeurs non-autorisées proches sémantiquement de la frontière de l'ensemble de valeurs autorisées pour l'attribut. Il est nécessaire de définir également les valeurs singulières intérieures qui sont des valeurs autorisées proches sémantiquement de la frontière de l'ensemble des valeurs non-autorisées. La recherche d'incohérence va porter sur ces valeurs singulières extérieures ou intérieures. Le triplet (objet, attribut, valeur) où valeur appartient à l'ensemble des valeurs singulières extérieures est la conjecture définie sur ces valeurs, et (objet, non-attribut, valeur) où valeur appartient à l'ensemble des valeurs singulières intérieures est la conjecture définie sur ces valeurs.
- Focaliser la recherche des incohérences sur les conjectures utilisant certaines entités (classes, objets, attributs, valeurs, règles), cela est réalisé par la définition des coefficients de pertinence. Dans SACCO, on a la possibilité de définir un coefficient de pertinence pour une entité donnée (fait, règle, classe, objet, etc). L'expert définit un niveau de pertinence par lequel on va filtrer les entités à prendre en considération.

Une fois les conjectures pertinentes construites, le résolveur du système SACCO tente de les prouver en utilisant la base de règles en chaînage arrièreselon le cycle décrit dans la figure 3-2 [17].

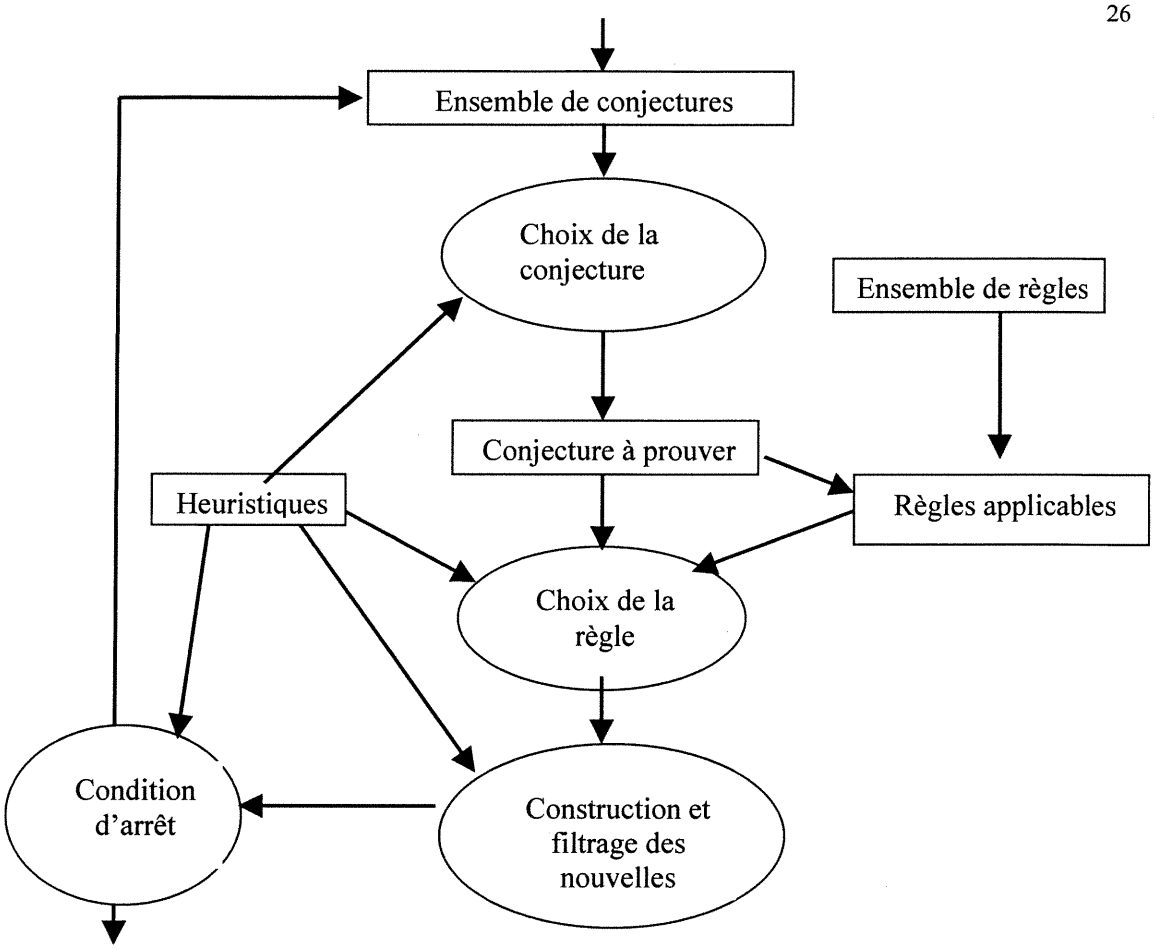


Figure 3-2: Fonctionnement de SACCO

La condition d'arrêt du résolveur est remplie dans trois cas :

- la conjecture déduite est instantiable sur la base de faits ou elle peut avoir une instantiation commune avec une spécification de base de faits initiaux. Dans ce cas une incohérence est détectée.
- l'ensemble des conjectures à prouver est vide. Il y a donc présomption de cohérence de la base.
- Les limites imposées au résolveur sont atteintes (temps d'exécution, nombre de conjectures testées,...). Il y a aussi dans ce cas présomption de cohérence.

Malgré le fait que SACCO utilise le formalisme objet pour structurer les connaissances sous forme de classes, la représentation des connaissances factuelles sous forme de faits indépendants (objet, attribut, valeur) engendre les mêmes vérifications effectuées par les autres systèmes classiques lors de l'ajout d'un fait. De plus, SACCO n'exploite pas complètement le formalisme objet car il est limité à la notion du triplet (Object Attribut Valeur) qu'il utilise pour former des prédicats des conditions et des actions des

règles. Dans notre cas, nous allons montrer qu'une base d'objets, utilisée pour représenter les connaissances factuelles, est beaucoup plus efficace et présente tous les avantages d'utilisation du formalisme objet. Nous allons voir aussi comment on peut formuler le raisonnement en terme d'objets, ces derniers forment le contenu de la mémoire de travail à un moment donné. La vérification dynamique dans SACCO est basée sur le C-Modèle défini et donc du formalisme utilisé pour représenter les contraintes de cohérence et les propriétés à vérifier.

3.2 Validation

La figure 3-3 présente l'architecture fonctionnelle typique d'un outil de validation.

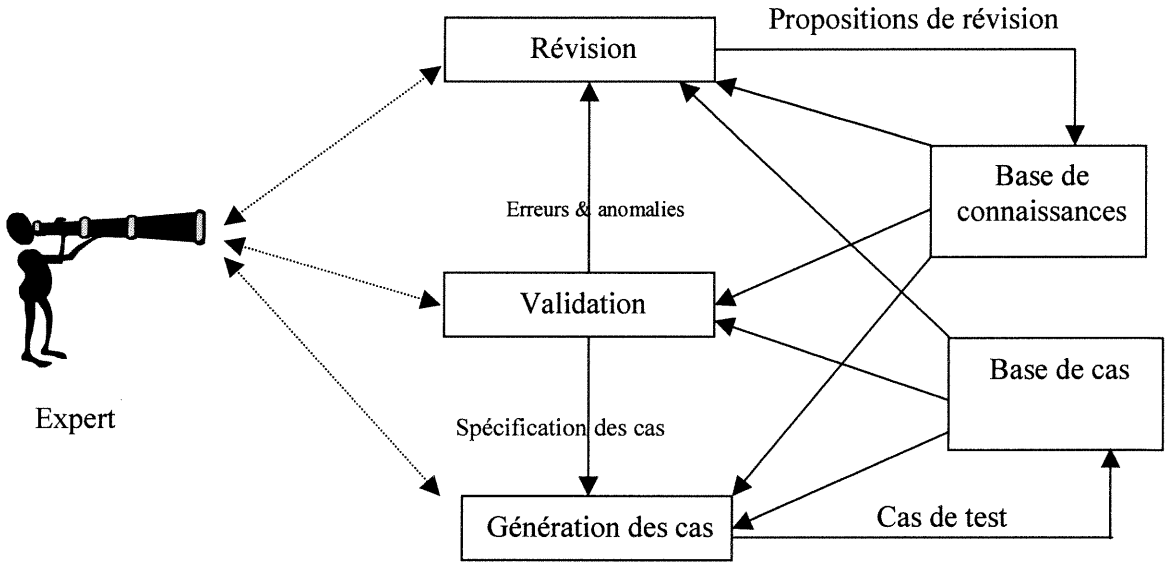


Figure 3-3 : Architecture d'un outil de validation

Un tel outil est un système constitué de trois parties majeures : le module de validation, le module de révision, et le module de génération des cas de test. Il opère en utilisant deux sources de connaissances : la base de connaissances et la base de cas. La partie validation doit intégrer un moteur d'inférence pour pouvoir exécuter les cas de test sur la base de connaissance. L'utilisateur contrôle le processus de façon interactive, en ayant un accès direct aux différents composants du système.

- Module de validation : il agit comme un testeur. Il prend la base de connaissances et l'ensemble des cas de test et, en utilisant le moteur d'inférence, démarre une session d'inférence en prenant chaque cas comme état initial. Ensuite, il vérifie les résultats obtenus et rapporte les anomalies et les erreurs au module de révision.

Finalement, il met à la disposition du module de génération des cas toute spécification pertinente pour la création des cas de test additionnels (par exemple, indique les parties de la BC non testées).

- Module de révision : il utilise la base de connaissances existante, les cas de test, et les erreurs et anomalies rapportées par le module de validation pour créer un ensemble de propositions de raffinement.
- Module de génération des cas : il utilise les spécifications des cas de test produites par le module de validation pour créer des cas de test additionnels dans l'optique d'atteindre un ensemble de cas le plus représentatif possible.

3.2.1 Différents types de validation

Dépendamment de la disponibilité d'une base de cas, on distingue deux types de validation. La validation statique a pour but d'utiliser les modules de validation et de génération des cas pour créer ou améliorer l'ensemble des cas de test disponibles. La validation dynamique a pour objectif d'utiliser les modules de validation et de révision avec un ensemble de cas de test pour mesurer et, si nécessaire, améliorer les performances de la base de connaissances.

En pratique, ces deux activités sont complémentaires et se chevauchent. Un scénario plausible de validation et de révision peut se présenter comme suit. Initialement, en l'absence d'un ensemble de cas de test adéquat, l'utilisateur lance une validation statique, suivie d'une validation dynamique. Si la validation dynamique révèle des erreurs dans le système, l'utilisateur effectue une révision. Si la validation dynamique indique que l'ensemble des cas de test est inadéquat, l'utilisateur peut démarrer une validation statique pour améliorer les cas de test, suivie encore d'une validation dynamique. Ceci peut continuer jusqu'à ce que le système et la base de cas soient jugés fiables.

Exemple pratique 1 : Système EVA

Dans la partie vérification, nous avons décrit les différents modules de vérification implémentés dans ce système. Nous nous intéressons maintenant aux autres modules qui concernent la validation des SBC.

▫ Vérification du comportement

L'objectif de cette vérification est de prouver que toute spécification des entrées possibles à un SBC et les sorties correspondantes produites par celui-ci est correcte (vérification fonctionnelle).

Par exemple, considérons la BC suivante pour vendre des pièces:

classe(partie).

slot(nom-partie, partie).

slot(crée-par, partie).

slot(pays, partie).

slot(coût, partie).

slot(peut-être-vendue, partie). /* oui ou non */

slot(partie-majeure, partie). /* oui ou non */

Règle1 : partie(P), coût(P,C), C>5000 → partie-majeure(P,oui).

Règle2 : partie(P), crée-par(P, M), M =abc → peut-être-vendue(P, non).

Règle3 : partie(P), partie-majeure(P, oui), pays(P, usa) → peut-être-vendue(P, oui).

Si on suppose que toute pièce qui coûte plus de 10000$ doit être fabriquée aux États-Unis (usa) pour être vendue, ceci est représenté par la règle suivante :

partie(P), coût(P,C), C>10000, pays(P,usa) →peut-être-vendue(P, oui).

Pour vérifier cette spécification, on introduit les prémisses de cette règle comme faits initiaux sur la base de connaissances, et on essaie de prouver sa conclusion. Si la conclusion est atteinte cette spécification est validée. Autrement, elle n'est pas validée.

▫ Révision des règles

La formulation initiale d'une règle peut être trop générale ou trop restrictive. Des cas de test spécifiques vont être choisis à partir de la base de cas concernant cette règle, l'expert va être interrogé par le système si cette règle est applicable à ces cas de test. Une réponse non indique que la règle est trop générale, et des règles plus spécifiques vont être proposées. Si la réponse est oui pour tous ces cas, ceci indique que la règle est trop restrictive. Le système va sélectionner encore d'autres cas de test en exploitant la hiérarchie entre les entités de la base de connaissances qui concernent cette règle.

Par exemple, considérons la Base de faits suivante:

est(étudiant, membre-université).

est(professeur, membre-université).

étudiant(sam).

statut(sam, inscrit).

étudiant(ted).

statut(ted, non-inscrit).

étudiant(ray).

statut(ray, inscrit).

professeur(sara).

statut(sara, retraité).

professeur(jim).

statut(jim,employé).

Si on a la règle suivante :

étudiant(X) → peut-prêter-livre(X).

Le système va présenter des instances de « étudiant» (en l'occurrence ici sam, ted, et ray) et interroger l'expert si la conclusion de cette règle est effectivement applicable à ces cas ou non. La réponse va être oui pour « sam » et « ray », et non pour « ted ». Puisqu'il y a une réponse non, la règle est trop générale, alors l'expert est amené à changer cette règle par : étudiant(X) Λ statut(X, inscrit) → peut-prêter-livre(X)

La règle est maintenant correcte pour tous les cas de test concernant le prédicat (entité) « étudiant ». Le système va ensuite exploiter les relations de cette entité avec les autres entités du système pour présenter de nouveaux cas de test pour cette règle. Il va donc présenter des instances du prédicat « professeur » (sara et jim) et l'expert va répondre par non pour « sara » et oui pour « jim ». La règle modifiée est donc trop restrictive puisqu'il y a une réponse oui. Ainsi, cette règle va être changée et donnera lieu aux deux règles suivantes :

étudiant(X) Λ statut(X, inscrit) → peut-prêter-livre(X).

professeur(X) Λ statut(X, employé) → peut-prêter-livre(X).

Puisque « étudiant » et « professeur » sont des sous classes de « membre-université», ces deux règles vont être généralisées par :

membre-université (X), classification(X, actif) → peut-prêter-livre(X).

étudiant(X) Λ statut(X, inscrit) → classification(X, actif).

professeur(X) Λ statut(X, employé) → classification(X, actif).

Le but de cet outil est d'assister l'expert dans la révision des règles. Puisqu'il s'agit d'un processus interactif, une bonne interface est donc requise.

▫ La génération des cas de test

Un système à base de connaissances peut être examiné avec une base de cas générée automatiquement pour évaluer son comportement, son efficacité, etc. Le test d'un SBC est toujours nécessaire parce qu'un futur utilisateur de ce système l'acceptera rarement sans l'exécuter sur des données et évaluer les résultats. Puisque la sélection manuelle des cas de test est une tâche fastidieuse, peut générer des erreurs et fait intervenir l'expert, l'utilisation d'un générateur automatique de cas de test s'avère fort utile.

Deux types de générateurs sont utilisés par EVA, l'un basé sur la structure de la BC et l'autre basé sur son fonctionnement. Le générateur basé sur la structure de la BC explore les relations entre les règles et les faits. Il représente ces relations par un graphe de connexion. Les règles et les faits sont les nœuds et un arc entre ces nœuds représente une correspondance entre un littéral dans la partie droite d'une règle et un littéral dans la partie gauche d'une autre règle. Un fait peut être considéré comme une règle sans partie gauche. L'idée est de générer un ensemble de cas de test à partir du graphe de connexion où chaque règle de la BC doit être déclenchée au moins une fois. L'algorithme suivant illustre la procédure à suivre :

1. Générer le graphe de connexion.
2. Générer le digramme de flux des règles à partir du graphe de connexion. Ce diagramme est un graphe orienté où les nœuds représentent les règles et les arcs les séquences d'exécution de ces règles.
3. Créer un ensemble de chemins dans le diagramme de flux tel que chaque nœud (règle) est couvert par au moins un chemin de cet ensemble.
4. Pour chaque chemin, générer des cas de test qui satisfont la conjonction de toutes les conditions dans le chemin. Chaque conjonction est analysée pour générer les dépendances entre variables. Les variables indépendantes prennent des valeurs aléatoires de leurs types de données appropriés. Les variables dépendantes prennent des valeurs telle que la conjonction est satisfaite.

Le générateur basé sur le fonctionnement du SBC fonctionne de la manière suivante. Lorsqu'on spécifie un SBC, on a besoin de spécifier les entrées et les sorties du système. Une spécification d'entrée-sortie inclut habituellement des données sur les entités, les attributs, les valeurs d'attributs, et les contraintes sur ces données. On peut donc générer automatiquement des cas de test qui satisfont ces données et contraintes.

Par exemple, considérons un SBC qui calcule le grade d'un étudiant à partir de ses réponses à un examen donné. L'entrée du système est constituée des réponses de l'étudiant et des réponses correctes de l'examen. Le schéma de données est le suivant :

étudiant(nom:chaîne, réponses (oui+non)*)

examen(examen-id:chaîne, nombre-de-questions: entier,réponses-correctes(oui+non)*)

où (oui+non)* est une liste de oui et non. Les contraintes de données sont :

longueur(réponses)=nombre-de-questions.

longueur(réponses-correctes)= nombre-de-questions.

En se basant sur ce schéma de données et les contraintes, on peut générer automatiquement des cas de test :

étudiant(john,[oui,oui]).

examen(examen1,2,[oui,non]).

Exemple pratique 2 : Système SYCOJET

SYCOJET est un outil de validation statique développé au sein du laboratoire LIA à l'université de Savoie en France [21]. Il concerne la génération des cas de test pour valider le comportement d'un SBC.

❑ Formalisation de la base de connaissances

Dans SYCOJET, une base de connaissances est représentée par trois éléments:

- Base de faits du domaine, BFd
- Base de règles, BR
- Base de faits des problèmes à résoudre par le système, BFp.

La représentation des connaissances est basée sur le formalisme objet. Les notions suivantes sont utilisées :

- Fait : c'est un triplet (objet, attribut, valeur)
- Pattern : c'est un triplet (objet, attribut, valeur) où l'objet est une variable classe ou une instance d'une classe, l'attribut est un attribut de la classe, et la valeur soit une variable, soit une constante.

Par exemple, (*Humain parent-de Peter) où « *Humain » est une variable de type classe « Humain ».

- Règle : elle prend la forme suivante.

IF prémisse ANDIF prémisse …

ANDIF prédicats

THEN conséquent.

« prémisse » et « conséquent » sont des patterns, « prédicats » portent sur les variables des patterns.

- Classes-problèmes, objets-problèmes, attributs-problèmes, valeurs-problèmes : C'est la partie statique de la base, définie par l'expert, qui constitue l'ensemble des éléments pouvant être utilisés comme point de départ pour résoudre les problèmes.
- Pattern-problème Pb : c'est un pattern où l'objet appartient à une classe-problème, l'attribut est un attribut-problème, et la valeur est une constante.
- Fait-problème : l'objet, l'attribut, et la valeur sont des entités-problèmes.
- Condition d'invalidité d'un problème (IVC) : c'est une conjonction de faits-problèmes ou de patterns-problèmes associée à des prédicats portant sur les variables des patterns. Si Pb est un problème réel alors il ne peut pas satisfaire IVC.

  Par exemple, (*Humain température X1) (*Humain vivant vrai) (X1<33 or X1>42)
- Problème réaliste (FBpb) : un ensemble de faits-problèmes, qui ne satisfont aucune IVC.

  Par exemple, (Peter température 37) (Peter vivant vrai)
- Spécification d'un problème réaliste (SFBpb) : un ensemble de patterns-problèmes tel que : $\forall\sigma$, une substitution, $\forall$ IVC une condition d'invalidité, SFBpb ne satisfait pas σ(IVC).

  Par exemple, (*Humain température 37) (*Humain vivant vrai)

❑ Calcul des problèmes réalistes en utilisant la technique des labels

L'objectif dans SYCOJET est de construire, d'une façon automatique, un ensemble de cas de test. Cela consiste en la compilation de la base de règles en construisant des labels pour chaque fait terminal. Pour ce faire, deux notions sont introduites:

- Environnement de déduction : un environnement de déduction d'un pattern P est le couple (PS, SR) où PS est une spécification d'un problème réaliste qui permet la déduction de P, et SR est l'ensemble de règles utilisées pour déduire P à partir de PS.
- Label de déduction : un label de déduction pour un pattern P est constitué de l'ensemble de ses environnements de déduction.

Pour construire le label de déduction d'un pattern P, on construit l'arbre de recherche de P, en chaînage arrière. La procédure est la suivante:

Un nœud N de l'arbre de recherche de P est constitué de cinq parties:

- N.P : l'ensemble des patterns à prouver
- N.PbP est l'ensemble des patterns-problèmes déjà utilisés dans le chemin entre la racine (contenant le seul pattern P à prouver) et N
- N.Pred : l'ensemble des prédicats
- N.R : l'ensemble des règles déjà utilisées entre la racine et N
- N.σ : est une substitution.

La racine de l'arbre de recherche de P est le nœud ({P,Φ,Φ,Φ,Φ,Φ), Φ est l'ensemble vide. Les successeurs de chaque nœud N de l'arbre de recherche sont construits en prenant un des patterns à prouver Pk. Pour chaque règle R telle qu'il existe une substitution σ entre le conséquent de R et Pk, le successeur N' est construit avec :

- $N'.P = \sigma\{N.P \setminus \{Pk\}\} \cup \sigma(Prem(R))$
- $N'.PbP = \sigma(N.PbP)$
- $N'.Pred = \sigma(N.Pred) \cup \sigma(Pred(R))$
- $N'.R = N.R \cup \{\sigma, R\}$
- $N'.\sigma = \sigma \circ N.\sigma$.

Si Pk, ou n'importe quelle substitution σ(Pk), est un pattern-problème alors le successeur N' est construit comme suit :

- $N'.P = \sigma\{N.P \setminus \{Pk\}\}$
- $N'.PbP = \sigma(N.PbP) \cup \sigma(PK)$
- $N'.Pred = \sigma(N.Pred)$
- $N'.R = N.R$
- $N'.\sigma = \sigma \circ N.\sigma$.

Construction des cas de test

Il est impossible d'exécuter chaque problème possible sur la base de connaissances. L'objectif est d'obtenir un ensemble de cas de test pertinents, c'est à dire des cas de test qui permettent de détecter le maximum d'erreurs. Pour cela, le calcul des labels de déduction se fait uniquement pour les patterns terminaux. Un pattern terminal est un pattern de la forme (objet, attribut, valeur) tel que :

- Il existe une règle dont (objet, attribut, valeur) est son conséquent
- Il n'existe aucune substitution σ tel que le pattern σ(objet, attribut, valeur) apparaît dans les prémisses d'une autre règle.

La construction des arbres de recherche pour tous les patterns terminaux fournit la spécification des tous les problèmes réalistes du SBC à tester. Cependant, il est difficile de tester l'ensemble des problèmes réalistes, on doit choisir les spécifications des problèmes réalistes les plus pertinentes à utiliser dans une session de test.

Tout d'abord, on choisit un pattern terminal, et on détermine la spécification du problème réaliste (PS) à utiliser avec le pattern retenu. Enfin, pour le PS retenu on doit choisir les valeurs à affecter aux variables. Pour ce dernier choix, la méthode la plus simple consiste à prendre des valeurs aléatoires parmi les valeurs autorisées des variables.

Plusieurs méthodes sont utilisées pour aider à faire le choix des patterns terminaux, des spécifications à retenir, et des valeurs à affecter aux variables. Les méthodes utilisées limitent le nombre de problèmes à proposer en renforçant leur qualité et leur pertinence :

- Critères de couvertures : des critères sont utilisés pour évaluer la qualité d'un cas de test par rapport aux composants de la base de connaissances (fait, règle, classe, etc.) qui sont utilisés pour l'exécuter, ces critères sont :
  - Couverture d'un fait. C'est le nombre de faits utilisés pour la résolution d'un problème sur le nombre total des faits. La couverture des faits-problèmes est également définie de la même façon.
  - Couverture d'une règle et la couverture des groupes de règles sont définies selon le même principe.
  - Couverture d'une classe, couverture d'un objet, et d'un attribut.

On calcule la contribution d'une spécification d'un problème dans un critère de couverture par le nombre d'éléments de la base de connaissances à couvrir et qui ne sont pas encore comptés dans l'ensemble des cas de test construits jusqu'a présent. De même, la contribution d'un pattern terminal dans un critère de couverture est le maximum de contribution des PS de ses labels.

- Valeurs singulières : en utilisant la notion des valeurs singulières internes ou externes, telles que vu dans SACCO au niveau de la section 3.1.3.2, des valeurs limites pour les tests sont proposées. Ces valeurs sont affectées aux variables de la spécification retenue pour un pattern terminal à tester.
- Heuristiques de construction : le testeur peut exprimer certaines préférences à propos du type de test qu'il veut faire : un test de valeur limite, un test de robustesse ou un test aléatoire. Il peut aussi demander de couvrir des composantes spécifiques de la base de connaissances (faits, règles, etc).

Pour le raffinement du choix des spécifications du problème, SYCOJET propose deux critères :

- Maximiser le nombre de cas de test c'est à dire des problèmes simples avec un grand nombre ou minimiser le nombre de cas de test c'est à dire des problèmes plus complexes avec un petit nombre
- Maximiser le nombre de faits dans chaque cas de test propose c'est à dire des problèmes plus complexes ou minimiser le nombre de faits c'est à dire des problèmes simples.

L'utilisateur spécifie le critère à retenir et SYCOJET l'utilise durant le choix des patterns terminaux et les spécifications des problèmes.

❑ Le T-Modele

Toutes les connaissances utilisées pour générer automatiquement des cas de test pertinents sont regroupées dans un modèle de test, appelé T-Modele, il est composé de:

- Domaine de chaque attribut
- Environnements de déduction
- Ensemble des valeurs singulières internes pour chaque attribut
- Ensemble des valeurs singulières externes pour chaque attribut
- Conditions d'invalidité.

3.3 Synthèse

Les premiers travaux de vérification des bases de connaissances (Check, EVA, RCP, etc.) ont concerné surtout la vérification statique de la cohérence et de la complétude en utilisant des méthodes syntaxiques (comparaison des faits des règles en se basant sur leurs syntaxes). Ces techniques ont été bien utilisées pour identifier les anomalies dans les bases de connaissances exprimées sous forme de règles de production. Malgré ce succès, la portée et l'applicabilité de ces méthodes a été limitée en raison d'un manque de définitions rigoureuses des anomalies détectées.

Nous notons que les systèmes étudiés ici couvrent les travaux réalisés jusqu'au début des années 90. les réalisations ultérieures, qui utilisent des systèmes classiques pour la représentation des connaissances, se sont beaucoup inspirées des travaux que nous avons détaillés ici. Nous citons le sous projet ViVa (Verification, Improvement and Validation of Knowlegde Based Systems) [29], il fait partie du grand projet ESPRIT. ViVa a été réalisé entre 1995 et 1997 en reprenant tous les travaux du laboratoire LIA (SACCO et SYCOJET)

de l'Université de Savoie (membre du consortium). Le but était de développer une méthodologie et un ensemble d'outils pour la V&V des SBC en suivant un cadre d'applications (framework) bien défini. Le cinquième colloque européen sur la V&V des SBC (EUROVAV'99) a eu lieu en 1999 à Oslo [27]. Les travaux présentés ont axé surtout sur l'importance de la V&V dans le cycle de développement du logiciel intelligent. Ils y ont présenté des méthodologies et des démarches à suivre pour l'assurance de la qualité des SBC. Plusieurs contributions ont montré les efforts déployés pour intégrer les tâches de V&V dans la réalisation des systèmes critiques (en médecine par exemple). Le formalisme de représentation adopté est resté toujours le même en utilisant la logique classique. Notre intérêt porte sur l'intégration totale des techniques OO (notion d'attributs et appels de méthodes) avec les systèmes de règles. C'est pourquoi ces systèmes ne représentent pas un grand intérêt pour les étudier avec plus de détail.

3.4 Choix et orientations

Jusqu'à maintenant, les travaux de V&V ont porté sur des logiques classiques non monotones; même avec l'utilisation de la notion (objet, attribut, valeur) inspiré du formalisme objet, les faits restent indépendants les uns des autres, et les règles sont définies par une logique de premier ordre monotone. Avec l'émergence des techniques objets, leur utilisation pour la représentation des connaissances devient de plus en plus cruciale. En se plaçant dans le contexte d'une base de connaissances hybride objet-règle, nous allons étudier les caractéristiques d'une telle représentation (notion d'états, non-monotonie, etc.) et présenter l'architecture d'un gestionnaire de connaissances comme cadre général pour la V&V. Par manque d'une architecture standard à suivre pour effectuer la V&V, nous proposons notre propre architecture qui intègre ces deux activités dans le gestionnaire de connaissances. Ce dernier constitue l'interface à travers laquelle l'expert peut contrôler le SBC. Dans le chapitre suivant, nous détaillons ces points :

- Modèle de cohérence : aucun modèle universel n'existe, il faut définir pour le domaine dans lequel nous voulons implanter un système à base de connaissances l'ensemble des contraintes et des propriétés à respecter dans la formulation de la base de connaissances, ainsi que dans toutes les étapes de développement et d'exploitation du système. Ces contraintes peuvent dériver soit du domaine d'expertise, soit être indépendantes du domaine. Les différents systèmes que nous avons étudiés ont proposé des extensions spécifiques à leurs besoins pour

représenter le modèle de cohérence. Dans un contexte objet-règle nous allons présenter un langage formel pour la définition des contraintes.

- Approche de vérification : si l'une des approches de vérification, statique ou dynamique, peut s'avérer utile et possible dans le cas d'une grammaire classique, dans le cas d'une grammaire hybride non monotone les choses sont différentes. La complexité de cette grammaire, comme nous allons le voir, nous impose une approche de vérification dynamique.
- Validation : il faut définir un scénario de validation de notre base de connaissances.

Chapitre 4

Architecture du gestionnaire de connaissances

L'acquisition des connaissances est une étape importante dans le processus de développement et d'implantation des systèmes à base de connaissances. Des interviews doivent être effectuées entre les concepteurs du système et les experts du domaine. Les documents de travail utilisés, les applications déjà existantes ou toute autre source d'information employée doivent être consultés. La transformation de ces connaissances acquises en une représentation claire et adéquate constitue un problème majeur dans la construction des SBC. Dans une approche système expert, deux types de connaissances sont nécessaires pour modéliser un domaine d'expertise : des connaissances factuelles qui décrivent la partie statique du domaine, et des connaissances déductives qui décrivent le processus de raisonnement, ces dernières sont représentées souvent par des règles de production.

Les techniques orienté objet fournissent des mécanismes standards et efficaces pour traiter les problèmes de représentation de connaissances, elles sont même devenues incontournables dans toutes les branches de l'informatique. Les mécanismes de base sont maintenant bien connus et maîtrisés. Ces techniques sont devenues nécessaires dans les outils de construction de systèmes experts pour deux raisons principales. Premièrement, elles facilitent la réalisation d'interfaces graphiques et les facilités de réutilisation. Deuxièmement, elles sont utilisées pour modéliser les connaissances statiques du domaine; au lieu de représenter la partie statique de la base de connaissances sous forme de faits indépendants, on regroupe les connaissances sous forme d'objets et de liens entre ces objets. Ce principe de réification, qui consiste à associer à un « quelque chose » du monde réel un objet informatique, justifie bien l'emploi d'un paradigme objet pour la représentation de connaissances. En génie logiciel, les techniques objet sont devenues les plus souvent utilisées, aussi bien durant les phases de conception que d'implantation. De nos jours, la technologie objet remplace les approches classiques de développement des logiciels. Dans la communauté de l'intelligence artificielle, le principe des frames a précédé la notion d'objets, et ceci depuis les années 70. Les frames sont proposés par Minsky qui a montré que lorsqu'on

est en train de résoudre des situations de problèmes, on pense en terme de structures stéréotypées d'informations appelées frames qui intègrent des slots et des actions (démons). Actuellement, et avec la généralisation, la réutilisation et l'extensibilité présentes dans les techniques objet, ces dernières doivent être étroitement couplées avec le développement des SBC.

Dans notre étude, nous avons opté pour un formalisme objet-règle. Ceci s'expliqué par la nature diversifiée des connaissances dont dispose un expert. La partie statique du domaine va être décrite dans un modèle objet qui représente aussi bien les entités du domaine (données et comportements) que les liens entre ces entités (base de faits = base d'objets). La partie déductive qui sera sous forme de règles de production est définie sur les objets (attributs, valeurs, opérations, liens). De plus, les contraintes et les propriétés de cohérence qui relèvent du domaine, et qui définissent des invariants sur la base d'objets et les règles de production, doivent être formalisées dans un modèle de cohérence. Comme résultat du processus d'acquisition des connaissances et de leur modélisation, ces trois parties (objets, règles, et contraintes) vont être produites. Les différents modules de notre SBC vont les utiliser conjointement pour la réalisation de différentes tâches (résolution de problèmes, simulation, maintenance de connaissances, vérification, etc.).

À travers le chapitre précédent, nous avons vu les différents aspects de vérification et de validation des bases de connaissances ainsi que des travaux qui ont été réalisés dans ce sens. À travers ce chapitre, nous allons présenter l'architecture d'un gestionnaire de connaissances comme cadre général pour la vérification et la validation. Premièrement nous allons étudier le formalisme objet-règle utilisé pour représenter les connaissances, ses caractéristiques, et ses particularités, et deuxièmement nous allons décrire les différents modules du gestionnaire. Un autre aspect que nous allons voir dans le gestionnaire de connaissances concerne la maintenance de la base de connaissances, chose qui n'as pas été présentée dans l'état de l'art. En pratique, il s'agit de la réalisation de l'ensemble des interfaces nécessaires pour présenter aux experts la base de connaissances et leur permettre sa manipulation pour différents objectifs (mise à jour, consultation, etc.).

La figure 4-1 présente l'architecture globale du gestionnaire de connaissances.

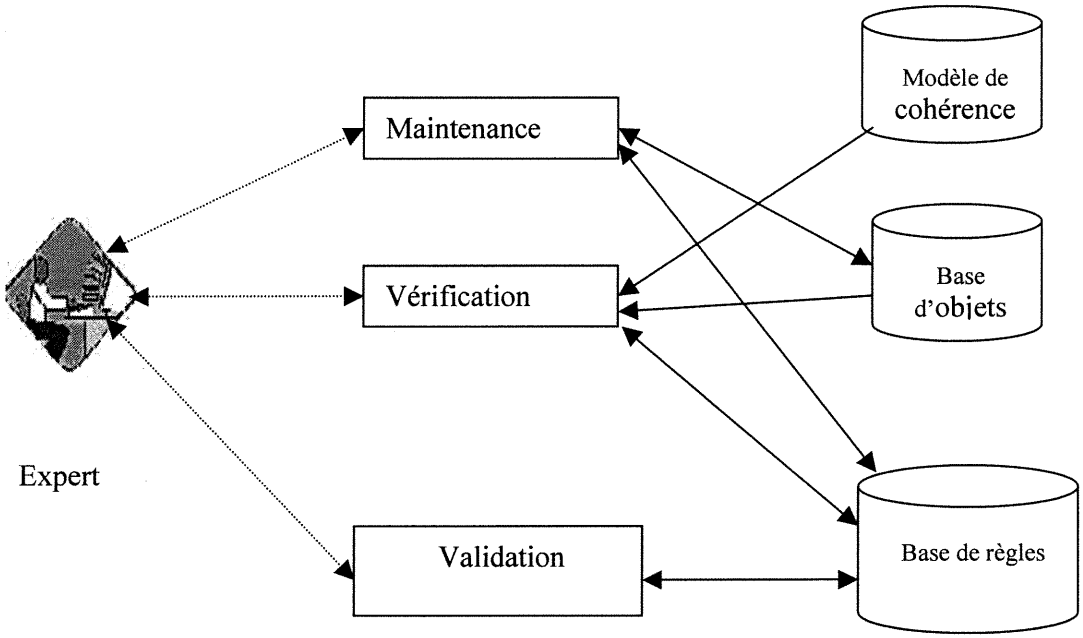


Figure 4-1 : architecture du gestionnaire de connaissances

4.1 Représentation de connaissances et de contraintes

4.1.1 La base d'objets

Si au début des développements des systèmes à base de connaissances, et grâce à la popularité du langage C++ et d'autres langages orienté objet (SmallTalk, Simula, etc.), plusieurs travaux de recherche et de développement de ces systèmes ont été réalisé par ce type de langages, ces langages ont été utilisés surtout dans la phase de réalisation (programmation). Actuellement, et avec le développement considérable des techniques orientées objet, une approche objet doit accompagner le développement des SBC dès les premières phases. Plusieurs principes fondamentaux de ces techniques peuvent être exploités et utilisés pour mieux concevoir de tels systèmes :

- Principe de classes : en orienté objet, la définition d'une classe regroupe l'abstraction des données/connaissances et les méthodes qui sont nécessaires pour décrire le contenu et le comportement d'une entité du monde réel, ceci permet aux techniques OO de modéliser adéquatement divers types de connaissances utilisées par les experts dans leurs domaines. En OO, le passage de messages est le mécanisme utilisé pour communiquer entre les objets, instances des classes, où les objets envoient des messages pour activer les méthodes des autres objets. Certaines méthodes sont à usage public, tandis que d'autres méthodes et la plupart des

attributs sont à usage privé. Plusieurs objets peuvent collaborer pour résoudre un problème à travers une séquence de messages échangés entre eux.

- Principe d'encapsulation : ceci signifie que chaque objet d'une classe donnée est indépendant des autres objets actifs dans le système. Dans un SBC, chaque objet est vu comme une boite noire. À travers les méthodes publiques de sa classe, le moteur d'inférence, les interfaces usagers, et les autres objets peuvent communiquer avec lui et vice versa. L'encapsulation fournit une solution directe au problème de la maintenance des connaissances. Puisque chaque objet est indépendant des autres objets et contient les connaissances et les fonctionnalités lui permettant de participer au processus d'un SBC, de nouveaux objets peuvent être ajoutés ou des objets existants peuvent être modifiées ou retirés avec un minimum d'effets sur la base de connaissances.
- Principe de polymorphisme : il permet de manipuler, de la même façon, des objets différents appartenant à une même hiérarchie de classes et ayant un comportement similaire. Chaque sous-classe hérite de la spécification des opérations de ses super classes, avec la possibilité de modifier localement le comportement de ces opérations afin de mieux prendre en compte les particularités liées à un niveau d'abstraction donné. Ceci facilite l'intercommunication entre objets, puisque on n'est pas obligé de connaître tous les détails d'implémentation. En conséquence, la réutilisation des objets est simple à effectuer.
- Le principe d'héritage et d'agrégation : dans leur vie quotidienne, les experts d'un domaine donné utilisent les principes de classification OO pour encoder les informations contenues dans le monde qui les entoure. Cet acquis va leur permettre de bien comprendre un modèle objet et par conséquent ils seront capables de transférer leurs expertises d'une façon plus complète et transparente. En OO, plusieurs mécanismes de classification existent. Premièrement, le principe d'héritage qui permet de définir de nouvelles classes à partir de classes déjà existantes (plus générales ou abstraites), ceci permet de construire un arbre de hiérarchie. L'agrégation est un autre mécanisme de classification qui permet de représenter des relations de type maître esclave entre les classes du domaine d'application. Typiquement une super classe est définie avec la combinaison de plusieurs type d'autres classes, par exemple une classe « automobile» est constituée de plusieurs parties qui sont elles mêmes des classes (moteur, roues, etc.). Les relations d'héritage et d'agrégation, qui sont à la base des mécanismes de

classification OO, représentent bien les méthodes employées par les experts et fournissent des modèles plus compréhensibles et plus naturels qui facilitent aussi l'acquisition et la représentation de connaissances.

Conceptuellement on représente le modèle objet dans un diagramme de classes qui décrit les différentes entités du monde réel (attributs et méthodes) ainsi que les relations entre ces entités. En utilisant une notation UML, le diagramme d'objets peut aussi être utilisé pour représenter l'état statique du domaine (objets et liens) à un instant donné. Dérivé du diagramme de classes qui représente le schéma général comme abstraction du monde réel, la base d'objets représente toutes les instances possibles des différentes classes. L'avantage de cette approche est la capacité de représenter les connaissances extensionnelles et intentionnelles d'un domaine. L'extension d'un concept est l'ensemble des objets dénotés par ce concept (base d'objets) alors que son intention détermine qu'est ce qu'il signifie, c'est à dire la définition des attributs et des comportements (diagramme de classes).

4.1.2 La base de règles

Si l'utilisation d'une approche OO pour la description du domaine d'expertise apporte un grand avantage dans la représentation de ce type de connaissances en permettant une bonne structuration des connaissances, la modélisation des connaissances qui apportent des jugements et décrivent le processus de raisonnement des experts nécessite un style déclaratif de représentation. Ces connaissances sont mieux représentées par un grand nombre de règles de déduction simples. Ces règles sont des assertions sous forme d'implications et représentent chacune un quantum de connaissances exprimées sous la forme : si PREMISSE(S) alors ACTION(S). Ces règles vont être utilisées par le moteur d'inférence pour guider le dialogue entre le système et l'utilisateur, et déduire des conclusions selon plusieurs modes de raisonnement (avant, arrière, etc.). Les avantages de ces règles de production sont multiples. Outre leur simplicité d'expression, liée à leur syntaxe restreinte, on peut faire figurer à leur actif leur liberté d'utilisation (la règle peut être manipulée dans différents sens, et dans différents buts), leur indépendance qui permet en particulier leur introduction "en vrac", leurs modifications sans remettre en cause l'ensemble du système, et leur aspect pédagogique dû au fait que ce sont des règles explicites.

Une approche objet-règle nécessite que le système de production soit intégré dans le formalisme objet adopté, les règles sont utilisées pour exprimer des connaissances faisant appel à plusieurs objets. Cette approche a plusieurs avantages:

- Raisonner en terme d'objets avec les règles est plus facile et naturel, les règles sont plus claires et facilement maintenables.
- La logique du domaine est séparée du reste du système, ce qui permet plus de modularité et de facilité de communication entre composants du système.
- La base de règles est mieux structurée. Un des mécanismes clé de la programmation par règles est sa nature non déterministe d'où la nécessité des mécanismes de structuration que peut apporter la technique objet.

La figure 4-2 montre l'interaction entre la base d'objets et la base de règles.

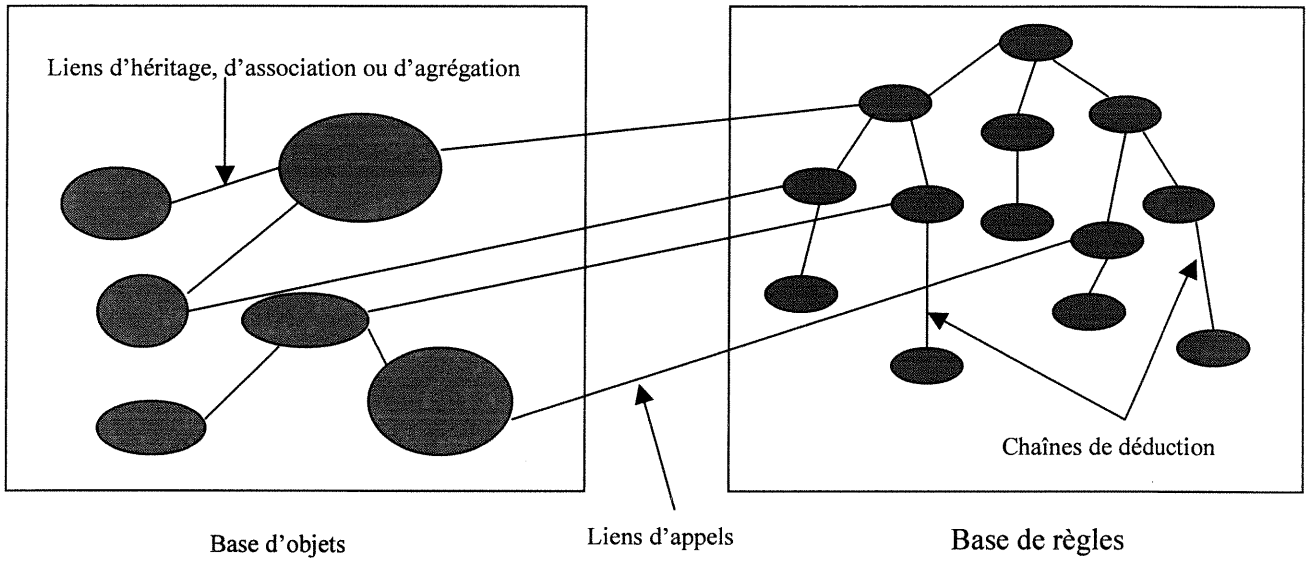


Figure 4-2 : Interaction entre objets et règles

Dans un domaine d'application donné des SBC, les règles peuvent avoir des liens et des enchaînements entre elles. Pour résoudre un problème particulier, plusieurs règles sont utilisées, soit en chaînage avant ou en chaînage arrière. Le déclenchement de ces règles se fait par l'examen et la comparaison que fait le moteur d'inférence entre les conditions des unes et les actions des autres. Cependant, d'autres règles peuvent être totalement indépendantes et aucun lien, direct ou indirect, n'existe entre elles. Ainsi le regroupement des règles liées permet de mieux structurer la base de règles. Ces règles peuvent être organisées par paquet. Les paquets permettent donc une organisation logique de la base de règles. L'expert aura ainsi une vue globale sur toute la base de règles ou une vue partielle par paquet. Au niveau du gestionnaire de connaissances, l'expert peut effectuer des vérifications et des validations sur un ou plusieurs paquets qui jugent candidats à des sources d'erreurs ou d'anomalies dans le système.

Dans un contexte objet-règle, une règle peut prendre la syntaxe suivante :

Regle : := Identifiant

{

Conditions

Actions

};

Conditions ::= Si

{

(Condition)+

}

Condition ::= Condition simple | Condition négative | Condition Existe

Condition simple ::= (<Variable> :) ? Condition classe

Condition négative ::= not Condition classe

Condition Existe : := existe Condition classe

Condition classe ::= <Nom classe> (Suite test)

Une condition simple permet de trouver tous les objets de la classe sur laquelle porte le test et qui vérifient cette condition. Une condition négative est vraie lorsqu'aucun objet de la classe ne vérifie la condition, et une condition «existe» est vraie lorsqu'il y a au moins un objet de la classe qui vérifie la condition.

Les conditions sont définies dans «Suite test», elles peuvent inclure des tests sur les attributs de la classe en utilisant les prédicats prédéfinis (==, !=, <, etc.) ou des résultats d'appels de méthodes de classes.

Actions ::= Alors

{

*(Action)**

}

Action ::= Assertion | Retrait | Modification

Assertion ::= assert <Nom classe> (Arguments)

Retrait ::= retract <Nom classe> (Arguments)

Modification ::= modify <Nom classe> (Arguments)

L'action « assert » ajoute un objet d'une classe donnée avec les paramètres nécessaires (arguments). De même, l'action « retrait» supprime un objet référencé par une

variable définie dans l'une des conditions de la règle, et l'action «modify » modifie un objet référencé par une variable d'une condition de la règle.

Ce formalisme de règles hybrides est générique. Plusieurs langages existent et définissent des règles selon cette grammaire avec quelques différences, mais le principe reste le même (définir des règles sur des classes). On cite par exemple OPS5, Rules, et Jrules.

Dans cette approche, le raisonnement se fait en terme d'objets et non en terme de faits indépendants qui sont vrais ou faux. Au départ, la mémoire de travail est initialisée par un ensemble d'objets (état initial comme état courant). Ensuite, les règles sont utilisées par le moteur d'inférence lors d'une session de résolution de problèmes. À ce moment là, des objets seront ajoutés, modifiés ou supprimés selon les règles déclenchées.

Un autre point très important concerne le caractère non monotone des règles utilisées. Trois types d'actions existent; assert, retract, et modify. Ces actions peuvent remettre en cause les connaissances acquises à un moment donné dans une session d'inférence (modify ou retract). Dans une logique classique de premier ordre, on essaie de ramener de nouvelles connaissances sans remettre en cause celles déjà acquises. Ce caractère non monotone des règles utilisées va influencer certains aspects de vérification et validation des SBC comme nous allons le voir par la suite.

4.1.3 Le modèle de cohérence

Dans la base d'objets, nous définissons la partie statique de la base de connaissances en termes d'objets et de liens entre objets. Dans la base de règles, on décrit la partie déductive de la base en termes de règles de production. Cependant ces deux parties ne sont pas suffisantes pour décrire tous les éléments de l'espace du problème étudié. Il faut disposer d'un moyen de définir des propriétés et des contraintes de cohérence sur le modèle objet et la base de règles. Il s'agit de l'ensemble de contraintes qui relèvent soit du domaine d'application, soit indépendamment du domaine.

Une base d'objets qui définit les états des objets doit respecter le schéma des classes. Ainsi, une partie du modèle de cohérence est définie par le diagramme de classes qui définit ce schéma, de même dans ce diagramme on définit l'ensemble de valeurs autorisées (fonction d'appartenance ou liste de valeurs). En plus du diagramme de classes, les propriétés et contraintes qui relèvent du domaine d'application doivent être formulées dans le modèle de cohérence. Nous distinguons entre des contraintes positives que la base de connaissances (base d'objets et base de règles) doivent respecter, et les contraintes négatives qui ne doivent

pas être vérifiées par la base de connaissances. Le tableau suivant résume les propriétés et les contraintes de cohérence que l'on peut définir sur une base de connaissances :

Modèle de cohérence	
Propriétés	***Contraintes***
• Diagramme de classes • Arité d'un attribut • Valeurs simultanément contradictoires pour un attribut • Attributs exclusifs d'une classe	Contraintes positives
	Contraintes négatives

Tableau 1 : Modèle de cohérence

Dans notre architecture, nous utilisons deux paradigmes: l'objet (notation UML) pour représenter la partie statique du système (diagramme de classes) et les règles d'inférence pour représenter la partie dynamique. L'un des deux paradigmes peut être utilisé pour décrire les contraintes sur les classes et les règles.

Des méta-règles peuvent être utilisées pour modéliser ces propriétés et contraintes. Par exemple, une contrainte négative portant sur la propriété d'exclusion mutuelle entre attributs d'une même classe peut se définir ainsi :

règle exclusion1
{
Si {
X: <Nom classe> (Attribut1.valeur == Attribut2.valeur)
}
Alors {
INCOHERENCE
}
}

Sachant que «Attribtut1» et «Attribut2 » ont les mêmes champs de valeurs mais ne doivent pas prendre des valeurs égales en même temps, cette méta règle permet de trouver les cas ou il y a violation de cette propriété de cohérence. Ce cas peut être présent aussi bien dans la base d'objets que dans la base de règles (conditions ou actions).

Le standard UML propose un langage de définition de contraintes, appelé OCL (Object Constraint Language). Il peut être utilisé pour enrichir les diagrammes de classes par des contraintes qui doivent être respectées dans tout le système (base d'objets et base de règles). Dans le chapitre suivant nous allons montrer comment on peut exploiter OCL pour définir notre modèle de cohérence et aussi les états des différentes classes du système.

4.2 Modules du gestionnaire de connaissances

4.2.1 Module de maintenance

La maintenance de la base de connaissances est une tâche délicate. Des événements comme le changement dans l'environnement du SBC, de nouvelles spécifications du problème, le déboguage de connaissances inadéquates et la personnalisation du système selon les préférences des utilisateurs et les pratiques d'utilisation, sont les principales raisons qui nécessitent la mise au point d'un outil de maintenance de la base de connaissances.

Dans ce module, l'expert pourra effectuer les traitements nécessaires pour mettre à jour la base de connaissances et le modèle de cohérence. Si pour des raisons de changement dans les spécifications du problème étudié, de nouvelles règles ou de nouveaux objets doivent être ajoutés ou modifiés, l'expert sera amené à effectuer ces changements sur la base. Ce module est utilisé aussi pour corriger la base de connaissances; après une session de vérification ou de validation, le gestionnaire de connaissances rapporte des anomalies ou des propositions de révision, l'expert décide des changements nécessaires et les effectue en utilisant les facilités d'édition de la base. Cette approche d'intervention manuelle de l'expert a l'avantage de le laisser seul responsable des actions à effectuer. Puisque il connaît bien le domaine, il est capable de fournir les bons changements à effectuer sur la base. Il a aussi la liberté de ne pas effectuer des changements proposés, par le système, car pour des raisons stratégiques de résolution de problèmes, des anomalies peuvent exister dans la base.

À ce niveau, les interfaces jouent un rôle très important. Il faut offrir à l'expert des interfaces de manipulation de la base assez conviviales qui reflètent bien le contenu et l'état des connaissances exprimées par celui-ci. Les outils suivants sont à la disposition de l'expert :

- Éditeur de règles : il permet de visualiser la base de règles et de la tenir à jour. Les possibilités de création, de modification, de consultation et de suppression sont offertes à l'expert. Cet outil est utilisé aussi pour apporter les modifications suite à des vérifications ou des validations effectuées dans les modules concernés.

- Éditeur d'objets : il sert à éditer le contenu de la base d'objets qui représente l'état courant des entités du domaine d'application. À l'aide de cet outil, l'expert pourra naviguer entre les objets qui existent ainsi que les différents relations entre ces objets (aggréagation, composition et héritage). L'expert pourra apporter toutes les modifications nécessaires à ces objets (valeurs d'attributs) suite à des changements. Évidemment, il pourra aussi ajouter de nouveaux objets ou en supprimer d'autres.

La figure 4-3 montre le fonctionnement du module de maintenance.

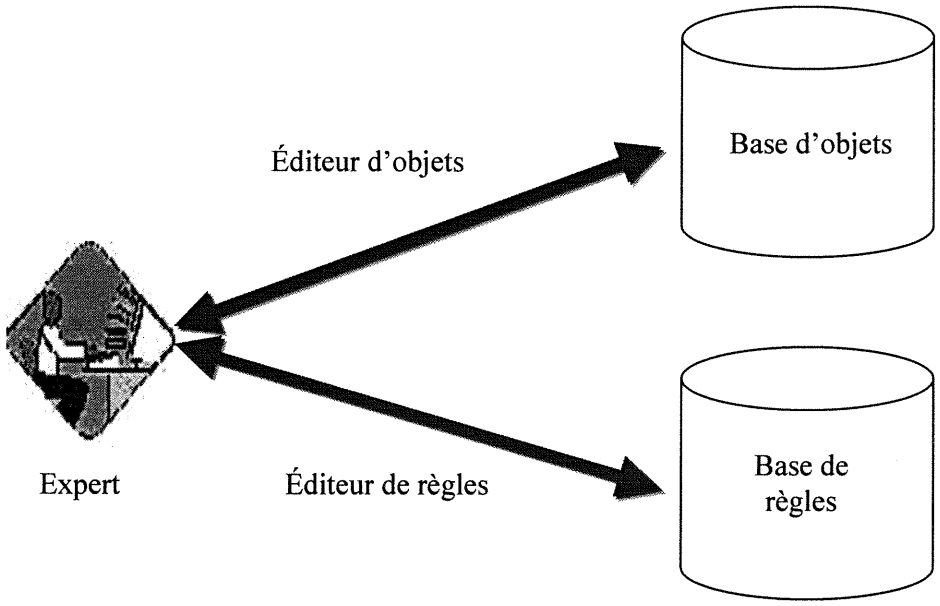


Figure 4-3 : module de maintenance

4.2.2 Module de vérification

Plusieurs techniques vues dans le chapitre précédent sont utilisées pour la vérification des bases de connaissances et la détection des anomalies. Elles nous fournissent différentes définitions de cohérence et de complétude, et leurs procédures de vérification associées. Ces techniques concernent en général la logique classique de premier ordre ou des règles de type (objet, attribut, valeur).

Dans un système hybride objet-règle on s'intéresse essentiellement à la vérification de la base de règles car les faits sont intégrés dans la base d'objets. Cette dernière respecte bien le schéma décrit par le diagramme de classes. Cependant, on peut prévoir des vérifications sur la base d'objets. Il s'agit de prouver que tous les objets respectent bien les propriétés et contraintes définies dans le modèle de cohérence.

En terme de type d'erreurs et d'anomalies, la figure 4-4 montre les différents aspects sur les-quels nous voulons porter la vérification d'une base de règles hybride.

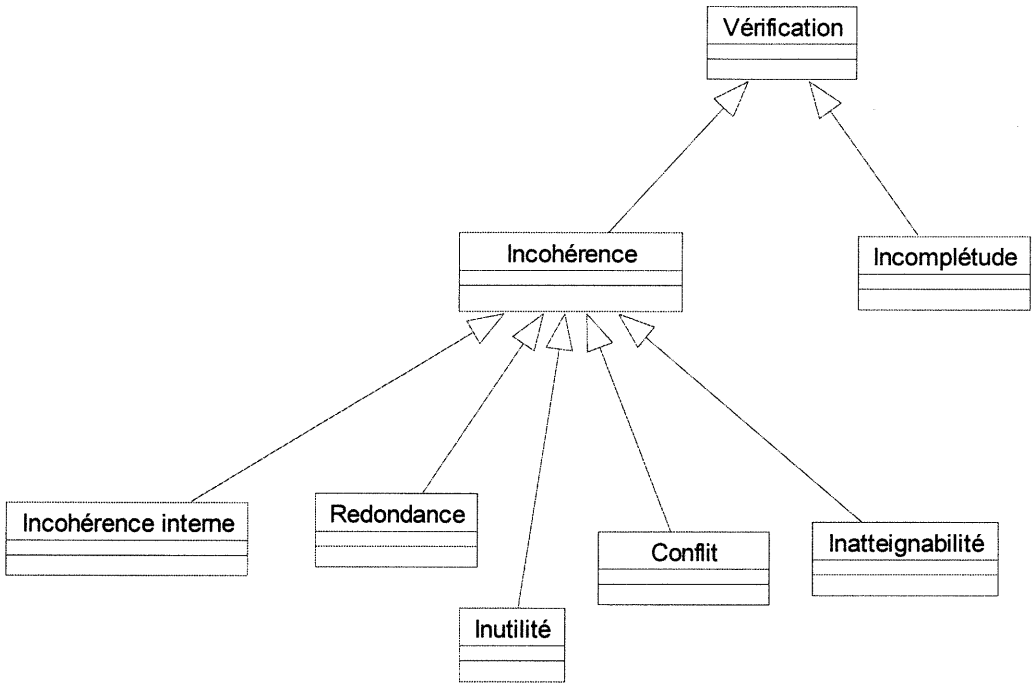


Figure 4-4 : Différents éléments de vérification

Si les premiers systèmes experts ont, pour la plupart, été vérifiés en utilisant une approche de vérification statique, la vérification dynamique a montré son efficacité dans les cas où elle a été utilisé (INDE, SACCO, etc.). En plus, dans cette approche nous exploitons la puissance déductive de la base ce qui permet de procéder à une vérification plus approfondie de la base.

Une des caractéristiques d'une grammaire objet-règle est que les règles, en plus des tests sur les attributs des classes, font appel aux méthodes de ces dernières. Une évaluation statique de ces règles ne permet pas de vérifier certains aspects; on peut bien trouver certaines conditions et/ou actions des règles statiquement différentes mais leur exécution permet d'aboutir à des situations dont leur présence peut induire des erreurs ou anomalies. Une vérification dynamique d'une base de règles hybride objet–règle est la plus adéquate. Contrairement aux règles exprimées par une logique de premier ordre, les règles hybrides filtrent des objets par leurs conditions, et comme elles sont non monotones elles ajoutent, modifient, et/ou retirent des objets par leurs actions. Donc le raisonnement se fait en termes d'objets instances de classes et non en terme de faits logiques, et la vérification des règles est basée sur la recherche des états dans lesquels la base est incohérente et/ou incomplète. Un état

est l'ensemble des objets qui la composent. Dans le prochain chapitre, nous allons montrer comment appliquer une technique de vérification dynamique basé sur la simulation, en utilisant les réseaux de Petri, pour rechercher les différentes situations d'incohérence et d'incomplétude.

4.2.3 Module de validation

Comme vu dans le chapitre précédent, le fonctionnement inadéquat d'un SBC est dû à une formulation inadéquate de règles par l'expert. L'outil de validation doit rechercher les situations de disfonctionnement du système. Dans la plupart des branches de l'informatique (génie logiciel, réseaux, etc.), la validation est basée sur l'utilisation d'un ensemble de cas de test pour évaluer les performances d'un système. Dans une approche système à base de connaissances, et après l'introduction en vrac de règles dans une BC par les experts, la confrontation de ces règles par des cas de test permet de valider le comportement de la base.

En optant pour une approche de vérification dynamique, basée sur la simulation de la base de règles, la vérification et la validation peuvent se faire dans le même scénario. Au niveau de la validation, il s'agit d'exploiter le résultat de simulation de la base pour trouver les relations de dépendance entre les règles. Ces relations constituent le fonctionnement global du système qui peut être inadéquat dans certaines situations. Une étude de ces relations permet de valider la base de règles.

Conclusion

Le gestionnaire de connaissances joue un rôle primordial pour un système à base de connaissances. En plus de permettre aux experts la modification et la mise à jour de la base de connaissances, il permet l'implantation des différentes techniques de maintien de la cohérence et de la complétude de la base ainsi qu'un fonctionnement adéquat du système dans les différentes sessions de résolution de problèmes. Il constitue un cadre global pour les différentes activités qui doivent accompagner le développement et l'exploitation d'un SBC. Les objectifs d'utilisation sont :

- Accroître la qualité du SBC par la détection de connaissances redondantes ou incomplètes.
- Diminuer les cycles d'ingénierie des connaissances par la détection des contradictions et des incohérences aux premières phases de développement.

- Aider les experts dans la formulation et la structuration de leurs connaissances pour une utilisation efficace du SBC.

Après avoir modélisé les connaissances en utilisant un paradigme objet-règle et décrit les spécifications fonctionnelles de notre gestionnaire de connaissances, il s'agit de définir les techniques qui vont être utilisées pour la vérification et la validation de la base de connaissances ainsi que les éléments nécessaires pour les effectuer. Le prochain chapitre propose un modèle de vérification et de validation et décrit comment ces dernières vont être effectuées.

Chapitre 5

Modèle de Vérification et de Validation

Un langage objet-règle offre la possibilité de définir des règles qui manipulent des instances de classes. Une base de connaissances implémentée avec ce langage est caractérisée par :

- La base de faits, qui représente les différentes instances de classes (appelée également base d'objets)
- Les conditions des règles qui sont exprimées sous forme de conjonctions de clauses (positives ou négatives). Chaque condition est définie sur une classe et contient des tests qui portent sur les attributs de celle-ci ou sur le résultat des appels à ses méthodes
- Les actions des règles qui sont non monotones. Elles sont principalement de trois types : « assert » pour ajouter des objets dans la mémoire de travail, « retract» pour retirer des objets de cette mémoire, et « modify » pour modifier les états des objets dans la mémoire de travail.
- L'inférence qui peut être en chaînage avant, chaînage arrière ou en chaînage mixte, selon le moteur d'inférence choisi.

Pour permettre la vérification, un système à base de connaissances doit contenir, en plus des règles et des faits, certaines déclarations explicites. Ces déclarations qui font partie de la spécification même de la base de connaissances doivent couvrir au moins les types suivants [2] :

1. Déclarations des littéraux qui représentent les conclusions finales.
2. Déclarations des littéraux qui sont des entrées.
3. Déclarations des types permis pour les littéraux.
4. Déclarations des valeurs permises pour les littéraux.
5. Déclarations des ensembles inadmissibles de littéraux.

Dans un cadre objet, les trois dernières spécifications sont définies par défaut dans le schéma des classes utilisées pour construire les règles. Le premier et le deuxième types de

déclarations sont des informations qui doivent être fournies explicitement pour prendre en considération les entrées et les sorties du système.

Dans ce chapitre, nous allons étudier les caractéristiques qui influencent la vérification et la validation d'un langage objet-règle. Nous définirons ensuite une façon de spécifier les différents états des objets et les contraintes sur ceux-ci en utilisant le langage UML. Ces deux spécifications nous permettent de connaître les entrées et les sorties du système. Enfin, nous proposons une approche de V&V.

5.1 Espace d'états, états déclencheurs et états résultants

Avant d'aborder l'analyse structurelle d'une base de règles hybride pour but de V&V, nous donnons, dans ce qui suit, des définitions des notions nécessaires à la recherche d'erreurs et d'anomalies :

- Espace d'états : C'est l'ensemble des états valides d'un système à base de connaissances. Chaque règle peut être vue comme un opérateur qui transforme un état (avant déclenchement) en un autre état (après déclenchement). Chaque règle spécifie un ensemble de conditions qui, lorsqu'elles sont satisfaites, rendent la règle éligible pour le déclenchement. Le nombre d'états que peuvent satisfaire les conditions d'une règle peut varier entre zéro à plusieurs. Puisque l'on raisonne en termes d'objets, un état est défini par l'ensemble des instances qui se trouvent dans la mémoire de travail.
- États déclencheurs : C'est l'ensemble D_r des états qui peuvent déclencher une règle r. Il est défini formellement par : $D_r = \{ s_i \mid s_i$ est un état valide et s_i satisfait les conditions de r$\}$
- États résultants : De la même façon, on définit l'ensemble R_r des états résultants après le déclenchement de r par : $R_r = \{ s_i \mid s_i$ est un état valide et s_i est obtenu par application de r sur un état de $D_r\}$

D'une manière générale, le calcul de D_r se fait en deux étapes :

1. Trouver l'ensemble des instances qui satisfont les clauses positives de r
2. Garder uniquement celles qui ne satisfont pas les clauses négatives de r.

Le calcul de R_r se fait comme suit :

1. Considérer $R_r = D_r$
2. Pour toute action « assert » qui définit une instance à ajouter, mettre cette dernière dans chaque état de R_r

3. Pour toute action « modify », modifier les instances de chaque état de R_r dont leur classe est celle concernée par cette action
4. Pour toute action retract enlever toutes les instances qui y correspondent dans R_r
5. Si nécessaire enlever les instances dupliquées.

Ces deux procédures sont plus ou moins faciles à mettre en œuvre, car nous travaillons sur une seule règle. Si nous voulons étudier la relation entre deux règles pour chercher les états qui, après avoir déclenché une règle r_1, peuvent déclencher une deuxième règle r_2, nous devons former alors les clauses A_{r1} qui caractérisent R_{r1} et qui sont calculées par application des actions de r_1 sur les clauses formant les conditions de celle-ci. L'ensemble des états qui peuvent satisfaire A_{r1} et les conditions de r_2 définissent une relation d'activation entre ces deux règles. Cette relation est vraie si cet ensemble n'est pas vide. Le processus qui permet de trouver une solution à un problème en utilisant un système de règles, peut se voir sous la forme d'une recherche d'un chemin (sous forme de relations d'activation entre les règles de la base utilisée dans l'inférence) qui à partir d'un état initial aboutit à l'état qui représente le but recherché. Un des objectifs de la V&V de la base est de trouver deschemins invalides par l'examen de ces relations d'activation entre les règles.

L'ensemble des états valides pour un SBC est constitué de tous les états initiaux possibles, les états intermédiaires qui découlent de l'application des règles sur ces états initiaux, et les états finaux qui sont les buts de l'inférence. Une façon de dériver l'ensemble des états déclencheurs d'une règle est de décomposer les clauses de ses différentes conditions en conjonctions d'égalités-inégalités (par exemple $a < 15.2 \Lambda b > 250 \Lambda c = vrai$), et de chercher en utilisant un résolveur de contraintes les états qui peuvent les satisfaire. Ceci suppose deux conditions :

1. Chaque attribut utilisé dans ces clauses peut prendre des valeurs à partir d'un ensemble fini
2. Les expressions utilisées, qui peuvent apparaître dans ces égalités-inégalités, sont linéaires, c'est à dire qu'elles n'impliquent pas des expressions non linéaires comme X * Y * Z ou X * Y / Z

Si nous nous plaçons dans le contexte d'un langage hybride qui utilise un langage de programmation OO comme langage hôte (utilisation libre des expressions de ce langage), ces deux conditions ne sont pas respectées. D'abord, ce langage utilise les attributs définis dans le langage hôte (Java, C++, etc.). Ces attributs peuvent être de type primitif ou des objets, et ils peuvent donc prendre probablement des valeurs d'un ensemble infini. Ensuite, ces langages hôtes permettent de définir des expressions complexes non linéaires qui peuvent être utilisées

dans les conditions ou les actions des règles de la BC. De plus, soit en raison de l'encapsulation qui interdit l'accès direct aux attributs, soit en raison d'un besoin d'informations qui doivent être calculées, il est obligatoire de passer par des appels de méthodes pour exprimer certaines conditions ou actions des règles. Nous constatons donc qu'une base de règle définie dans un langage hybride est d'une grande complexité. La spécification explicite des états des classes de la BC est donc nécessaire pour effectuer la V&V.

5.2 Système monotone vs non monotone

En plus de la complexité des langages hybrides, le fait que ces derniers supportent le retrait et la modification des faits (retract et modify) rend les techniques classiques inutilisables. En raison du caractère non monotone du langage, la sémantique des erreurs et des anomalies est elle aussi différente de celle relative à un langage classique monotone. Pour illustrer cette différence, prenons un exemple simple. Considérons les deux règles suivantes:

R1 : si A alors ajouter B

R2 : si A et C alors supprimer B

Dans le cadre d'une logique classique, on pourrait assimiler ces deux règles aux implications logiques :

$A \rightarrow B$

$A \wedge C \rightarrow \neg B$

Ceci mènerait à une incohérence en présence des faits initiaux A et C. Or, cette façon artificielle de représenter ces règles n'est pas correcte d'un point de vue sémantique. En effet, comme nous l'avons expliqué au début de ce chapitre; des règles exprimées à l'aide d'un langage objet-règle non monotone doivent être vues comme des opérateurs de transformation d'états. Un nouvel état ne doit pas garder nécessairement les faits déjà acquis, il peut les modifier ou même les retirer. Ainsi, nous constatons donc que la sémantique de certaines erreurs et anomalies peut être différente dans une base hybride.

5.3 Vérification statique vs dynamique

Comme nous l'avons vu dans le troisième chapitre, les premiers systèmes ont été vérifiés en utilisant des techniques statiques. Même s'il n'y avait pas de méthodes standards, chaque système utilisait des techniques qui se basent sur l'étude structurelle du langage utilisé pour la représentation des règles et essayait de détecter les erreurs et les anomalies. Plus

précisément, ces systèmes utilisaient souvent des règles définies par une logique de propositions ou de prédicats. La vérification consistait donc à comparer les faits logiques qui composent les règles et essayer de chercher les erreurs potentielles (redondance, inutilité, etc.). L'utilisation de ces techniques pour l'étude d'un langage hybride s'avère inappropriée. Imaginons une règle contenant des expressions complexes et des appels de méthodes, l'étude statique de ses clauses est très fastidieuse quand elle est possible. De plus, on ne peut pas vérifier statiquement des règles contenant des appels de méthodes pour des raisons liées au polymorphisme et à l'utilisation des librairies externes. Une étude du comportement dynamique de ces règles quant à elle représente une bonne alternative pour évaluer et corriger ces règles. Il s'agit en fait d'injecter des entrées valides au système et d'étudier les états intermédiaires et finaux pour chercher les erreurs et les anomalies potentielles.

5.4 Spécification des états et des contraintes d'un SBC hybride objet-règle

L'utilisation du formalisme objet pour la modélisation des connaissances statiques offre la possibilité de définir les états initiaux et finaux d'un SBC en utilisant le même formalisme. En UML, les diagrammes des états transitions proposent un modèle uniforme pour définir les états d'une classe ainsi que les transitions entre ces états. Ils sont devenus largement utilisés surtout dans les systèmes interactifs. Dans une approche purement objet, on peut faire la correspondance entre d'une part les états d'une classe et les valeurs de ses attributs, et d'autre part entre les transitions et les appels de ses méthodes. Dans une approche hybride objet-règle, la dynamique du système est supportée par les règles. Les méthodes sont souvent utilisées pour faire des calculs, affecter des valeurs aux attributs et obtenir les valeurs d'attributs. Ces méthodes sont appelées à partir des règles. Les transitions sont donc les règles. Il est donc nécessaire de définir une façon de représenter à la fois les états des objets et les règles qui serviront comme transitions entre ces états. Comme nous allons le voir plus tard, nous avons proposé une représentation par réseaux de Petri pour la modélisation des règles. Dans un premier temps, nous nous intéressons uniquement à la représentation des états des classes. Une bonne façon de représenter les états d'une classe est d'utiliser le design pattern ***State*** comme décrit dans le livre de Gamma et al. [21]. La structure de ce design pattern est représentée par la figure 5-1.

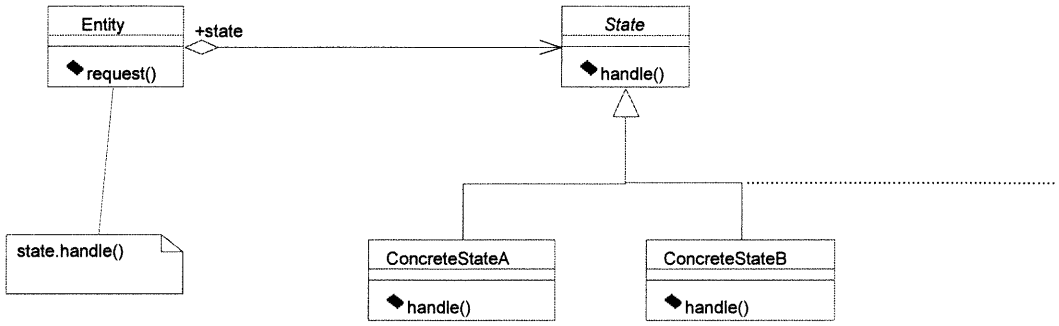


Figure 5-1 : Design pattern ***State***

Dans sa forme de base, ce modèle permet de définir pour chaque classe (Entity) autant d'états qu'elle peut en avoir. Ceci permet à la classe de changer de comportement selon son état courant. Les transitions sont effectuées explicitement par changement d' un état courant à un autre (variable state).

Pour des besoins de vérification, on veut savoir non seulement les différents états d'un objet mais également les caractéristiques de chaque état. Au niveau de ce design pattern, les états se différencient seulement par leur façon d'implémenter les différents comportements délégués par leur classe. Pour nous, il est important de définir un état en terme des valeurs des attributs de l'objet. Étant donné que les attributs sont définis au sein de la classe Entity, on peut étendre la définition de chaque état par des contraintes sur les attributs de cette classe.

Pour définir des contraintes sur un diagramme de classes, on peut utiliser OCL [22]. OCL est un langage qui permet de décrire des contraintes sur un modèle objet en utilisant des expressions déclaratives. D'une manière générale, une contrainte est une restriction qui porte sur les valeurs d'un ou plusieurs attributs d'un modèle objet. OCL est en fait une extension de UML.

Les contraintes peuvent être utilisées dans les cadres suivants [22] :

- Pré et post conditions : l'interface offerte par un objet est constituée par l'ensemble des opérations qui peuvent être effectuées par cet objet. Pour chaque opération, on peut définir des pré conditions. Une pré condition doit être vraie pour qu'une opération puisse s'exécuter. Les conditions d'acceptation des effets d'une opération sont spécifiées par des post conditions. Une post condition doit être vraie à la fin de l'exécution de l'opération pour que l'effet de celle-ci soit valide. OCL permet de définir les pré et post conditions sur les opérations d'un objet. Les pré conditions et les post conditions sont définies par des contraintes de façon déclarative pour spécifier les effets d'une opération sans avoir à définir son

implémentation. Ce principe de pré et post conditions est utilisé particulièrement dans la conception par contrats.

- Invariants : un invariant est une contrainte qui doit être toujours respectée par les instances d'une classe, un type ou une interface. Un invariant est décrit sous forme d'une expression qui doit être vraie si l'invariant est respecté. OCL offre un moyen puissant pour décrire des invariants.

D'une façon générale, les expressions OCL sont définies sur des objets et sur leurs propriétés. Chaque objet possède un type. OCL utilise les types suivants :

- Types prédéfinis : ils incluent les types primitifs (Integer, String, etc.) et les collections (Collection, Set, etc.). Ces dernières sont utilisées pour spécifier les résultats de navigation à travers les associations d'un diagramme de classes.
- Types définis par l'utilisateur : chaque classe, interface ou type défini dans un diagramme de classes est automatiquement considéré comme un type par OCL.

Chaque expression OCL est relative à un élément de modélisation (classe, interface, opération, etc.). L'élément dépend du type de la contrainte supportée par cette expression. Pour un invariant, l'élément est toujours une classe, une interface ou un type. Par exemple, un invariant sur l'âge d'un employé peut être représenté comme suit :

Employee ***Qui signifie que Employee est l'élément de cette contrainte***

Age > 18

Cet invariant signifie que cette contrainte doit être respectée par les instances de son éléement (constructeurs, méthodes qui changent l'attribut, etc.).

L'élément d'une pré ou post condition est toujours une opération. L'exemple de la figure 5-2 montre un invariant et deux post conditions sur une classe.

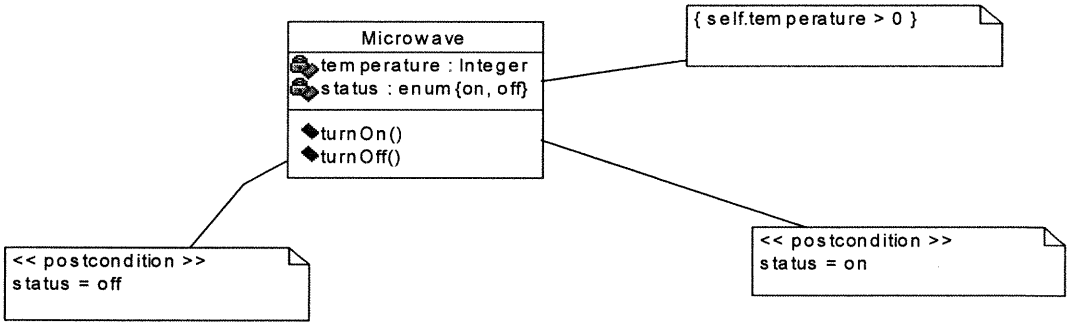


Figure 5-2 : Exemple de contraintes OCL

Afin d'utiliser le design pattern State pour la vérification, nous proposons d'étendre la définition de chaque état par des contraintes OCL sur la classe Entity. Ces contraintes

définissent des invariants sur les attributs de cette classe pour caractériser chaque état. Cette spécification offre les possibilités suivantes :

- Les états se différencient non seulement par leur comportement mais aussi par les valeurs des attributs. Ainsi, si un objet est dans un état donné, il doit respecter les contraints définies sur cet état.
- L'utilisation des invariants permet également de générer des cas de test pour la V&V dynamique. Les objets qui y vont être utilisés vont avoir des valeurs dérivées de la spécification de leurs états initiaux.
- L'utilisation des contraintes permet en outre de vérifier la cohérence interne et la complétude de la base de règles, comme nous allons le voir plus loin.

Il est important de souligner également qu'un ensemble de contraintes peut être défini au niveau de la classe Entity pour spécifier les invariants qui doivent être respectés quel que soit l'état. Elles sont obtenues à partir du domaine d'application du SBC. Ces contraintes constituent le modèle de cohérence. Ainsi donc, OCL offre un moyen uniforme d'exprimer le modèle de cohérence (voir 4.1.3) par des expressions au niveau de chaque classe (entité du domaine). Ces contraintes doivent être vérifiées par la base d'objets et celle des règles. L'avantage d'OCL réside dans le fait qu'il est un standard et ne dépend donc d'aucun langage spécifique. Contrairement à certains systèmes tels que ceux vus dans le chapitre 3 qui utilisent des techniques spécifiques à leurs langages d'implémentation (le prédicat *incompatible* dans EVA par exemple), OCL propose des expressions formelles pour la définition des contraintes. La structure étendue du design pattern ***State*** que nous proposons est définie dans la figure 5-3.

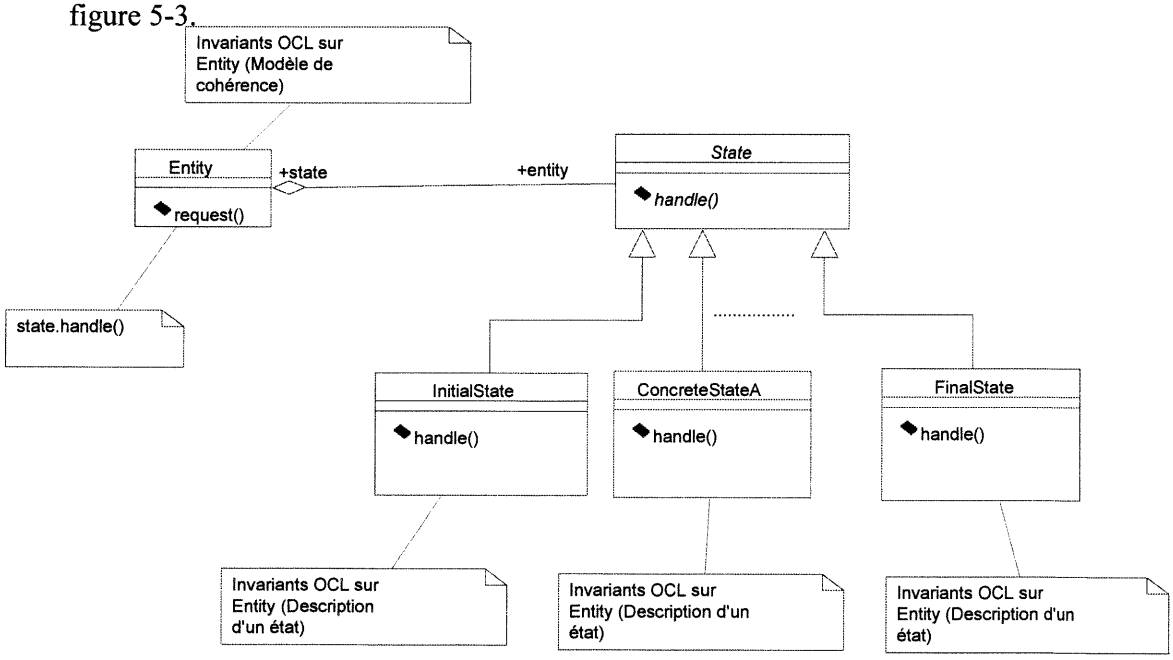


Figure 5-3 : Design pattern State avec des invariants OCL

5.5 Une approche de vérification et de validation

Dans le troisième chapitre, nous avons présenté plusieurs techniques de V&V des bases de connaissances. Les systèmes étudiés utilisent une approche classique pour la représentation des connaissances. Notre choix d'une nouvelle approche hybride objet-règle nous impose de définir une nouvelle technique pour la V&V des BC. Comme précisé plus haut, une évaluation statique des règles non monotones contenant des appels de méthodes pour en trouver des erreurs et des anomalies est très difficile, voir impossible. Nous proposons donc d'utiliser une approche dynamique basée sur la simulation de l'exécution de la BC. Pour permettre cette simulation, nous proposons d'utiliser les réseaux de Petri. Notre approche consiste à transformer la BC en un réseau de Petri coloré, générer le graphe d'occurrence associé et analyser le graphe d'occurrence pour vérifier et valider la BC.

Dans ce qui suit, nous introduisons les réseaux de Petri. Nous détaillons ensuite la transformation de la BC, et nous montrons comment nous pouvons exploiter les techniques d'analyse du graphe d'occurrence pour effectuer la V&V.

5.5.1 Réseaux de Petri

Les réseaux de Petri peuvent être vus comme un modèle mathématique permettant l'étude de différentes propriétés d'un système. Ils ont été définis principalement pour étudier les systèmes parallèles. Leur normalisation ainsi que leur utilisation fréquente ont permis d'étendre leur utilisation dans la modélisation et l'analyse d'une gamme large de systèmes. Ils ont été utilisés entre autres pour modéliser la dynamique des systèmes chimiques, juridiques et de communication, ainsi que les modèles de cerveaux et le calcul de propositions [20]. Dans le cadre du développement du logiciel, les réseaux de Petri sont utilisés aussi bien au niveau de la spécification d'un système, qu'au niveau de la V&V.

Le premier modèle de réseaux de Petri a été proposé par Carl Adam Petri, d'où leur nom est dérivé [20]. D'une manière générale, un réseau de Petri est un ensemble de places et de transitions entre les places. Des jetons sont utilisés dans les différentes places pour définir un marquage du réseau. Un marquage capte toutes les places avec leurs jetons correspondants à un moment donné. Une place peut avoir zéro ou plusieurs jetons, et les transitions permettent de changer le marquage courant du réseau en déplaçant, ajoutant et/ou retirant des jetons.

Formellement, un réseau de Petri de bas niveau est une structure C = (P,T,I,O). P = $\{p_1, p_2,\ldots,p_n\}$ est un ensemble fini de places avec n > 0. T = $\{t_1, t_2, \ldots, t_m\}$ est un ensemble

fini de transitions avec m >=0. L'ensemble de places et l'ensemble de transitions sont disjoints. I : T→ P^{∞} est une fonction d'entrée qui associe les transitions à des mutli-ensembles de places. Ainsi, pour chaque transition, on spécifie les places qui y entrent. O : T→ P^{∞} est une fonction de sortie qui associe les transitions à des multi-ensembles de places. Ainsi, pour chaque transition on spécifie les places qui en sortent. Un multi-ensemble est une généralisation d'un ensemble qui permet des occurrences multiples d'un élément. Une place p_i est une entrée d'une transition t_j si $p_i \in I(t_j)$, et p_i est une sortie d'une transition t_j si $p_i \in O(t_j)$. La multiplicité d'une place en entrée d'une transition est le nombre d'occurrences de cette place dans le multi-ensemble d'entrée de cette transition, notée #(p_i, $I(t_j)$). De même, La multiplicité d'une place en sortie d'une transition est le nombre d'occurrences de cette place dans le multi-ensemble de sortie de cette transition, notée #(p_i, $O(t_j)$) [23]. On donne un exemple d'un réseau de Petri en définissant C, P, T, $I(t_j)$ et $O(t_j)$ dans la figure 5-4.

$C = (P, T, I, O)$

$P = \{p_1, p_2, p_3, p_4, p_5\}$

$T = \{t_1, t_2, t_3, t_4\}$

$I(t_1) = \{p_1\}$ $\quad O(t_1) = \{p_2, p_3, p_5\}$

$I(t_2) = \{p_2, p_3, p_5\}$ $\quad O(t_2) = \{p_5\}$

$I(t_3) = \{p_3\}$ $\quad O(t_3) = \{p_4\}$

$I(t_4) = \{p_4\}$ $\quad O(t_4) = \{p_2, p_3\}$

Figure 5-4 : Structure d'un réseau de Petri de bas niveau

L'un des aspects les plus intéressants des réseaux de Petri est qu'il est extrêmement aisé de les visualiser. En effet on peut représenter un réseau de Petri comme un graphe biparti.

Les places sont représentées par des ronds : ◯

et les transitions par des rectangles : ▮

Des arcs orientés lient les places et les transitions. Un arc orienté qui lie une place avec une transition définit cette place comme entrée à la transition. Si cette place a une multiplicité supérieure à 1, on représente cela par un nombre sur l'arc. De même, un arc orienté qui lie une transition avec une place définit cette place comme sortie de la transition, et la multiplicité de cette place est représentée par un nombre sur l'arc. L'exemple du réseau définit dans la figure 5-4 peut être représenté par le graphe de la figure 5-5.

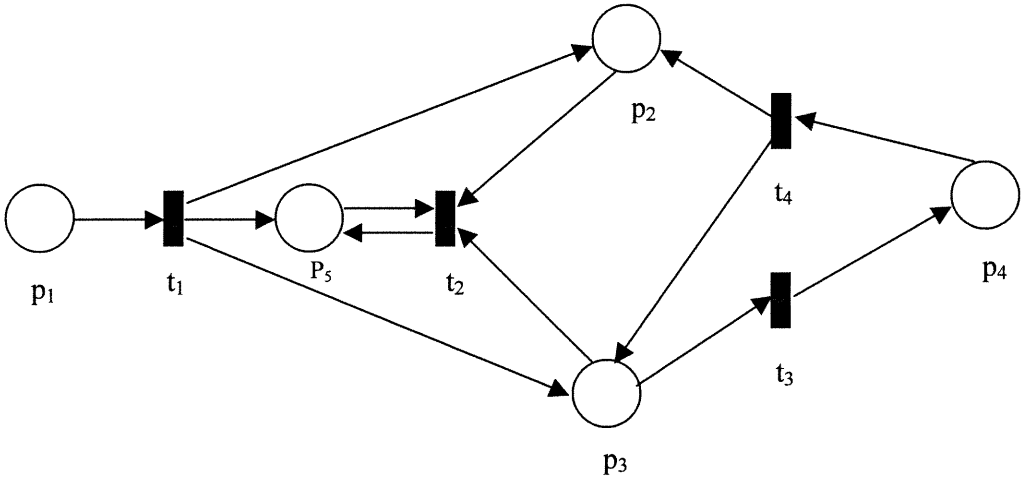


Figure 5-5 : Graphe d'un réseau de Petri de bas niveau

Un marquage M est une affectation de jetons aux places du réseau. Un jeton est un concept de base des réseaux de Petri (comme les places et les transitions). C'est une fonction de l'ensemble des places vers des entiers non négatifs. Un marquage μ d'un réseau de Petri donné peut être défini par un n-vecteur, $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$, où n est le nombre de places et où chaque $\mu_i \in N$, $i = 1,\ldots, n$. Le vecteur μ définit pour chaque place le nombre de ses jetons. La figure 5-6 montre le réseau de Petri de la figure 5-5 avec un marquage $\mu = (1,1,0,2,0)$.

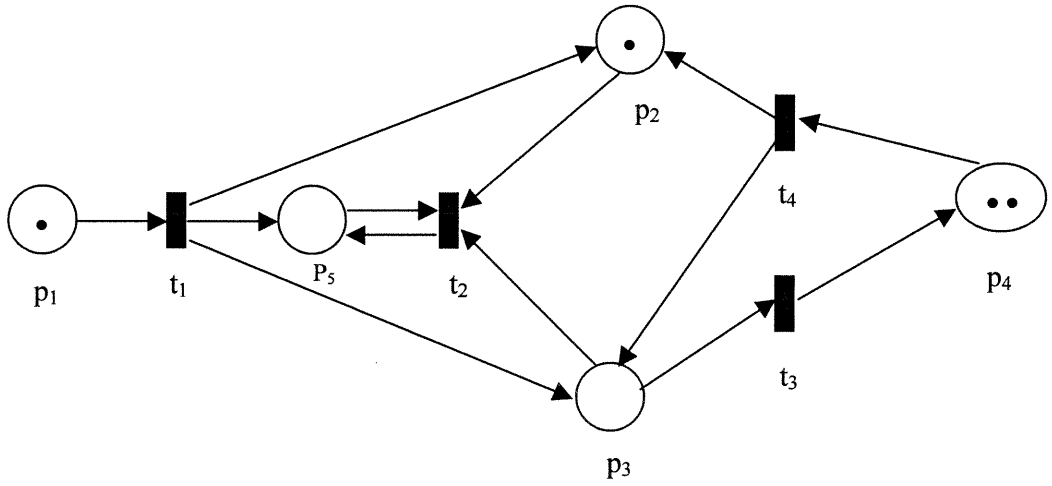


Figure 5-6 : Graphe d'un réseau de Petri de bas niveau marqué

Le nombre et la position des jetons peut changer durant l'exécution du réseau de Petri. Les jetons sont utilisés pour définir l'exécution du réseau. un réseau de Petri s'exécute par

déclenchement des transitions. Une transition se déclenche et a pour effet la suppression des jetons de ses places en entrée, la création de nouveaux jetons et leur distribution sur les places en sortie. Les jetons supprimés et ceux crées dépend de la multiplicité des places en entrée et en sortie. Ainsi, une transition $t_j \in T$ dans un réseau de Petri C = (P, T, I, O) avec un marquage μ est déclenchable si pour tout $p_i \in P$, $\mu(p_i) \geq \#(p_i, I(t_j))$. Le déclenchement de t_j génère un nouveau marquage μ' définie par : $\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$ pour tout $p_i \in P$.

Dans les premiers réseaux de Petri, il y avait un seul type de jetons ce qui signifie que l'état d'une place est décrite par un entier (nombre de jetons) ou dans certains cas par une valeur booléenne (absence ou présence d'un jeton). Plusieurs extensions ont été apportées aux réseaux de Petri pour aboutir à des réseaux de haut niveau. Le concept de type y a été introduit comme dans le cas des langages de programmation. Chaque jeton peut contenir plusieurs informations liées à son type. Le type est appelé couleur d'où l'appellation de réseaux de Petri colorés.

Formellement, un réseau de Petri coloré est une structure CPN = (Σ, P, T, A, N, C, G, E, I) [24] tel que :

- Σ est un ensemble non vide de types appelés l'ensemble coloré.
- P est l'ensemble de places.
- T est l'ensemble de transitions.
- A est l'ensemble des arcs tel que : $P \cap T = P \cap A = T \cap A = \phi$. P, T et A sont disjoints.
- N est une fonction nœud. Elle est définie de A vers $P \times T \cup T \times P$; c'est à dire chaque arc lie une place à une transition ou une transition à une place.
- C est une fonction de couleur. Elle est définie de P vers Σ, chaque place possède une couleur (type).
- G est une fonction garde. Elle est définie de T vers un ensemble d'expressions booléennes. Chaque transition peut être munie par des expressions booléennes qui doivent être évaluées à vrai pour pourvoir être déclenchée.
- E est une fonction expression d'un arc. Elle est définie de A vers un ensemble d'expressions. Chaque arc peut contenir des expressions dont l'évaluation donne un multi-ensemble ayant comme type celui de la place liée à cet arc.

- I est une fonction d'initialisation. Elle est définie de P vers des expressions sans variables. L'évaluation de ces expressions produit des jetons sur les places constituant ainsi le marquage initial du réseau.

Au niveau des réseaux de Petri colorés, les fonctions de couleur, les fonctions de garde, les fonctions d'expression des arcs et les fonctions d'initialisation sont définies à l'aide d'un langage d'inscription. Ce langage définit un ensemble de types, d'opérateurs et d'expressions qui permettent d'exprimer ces fonctions du réseau [24].

Après la construction du réseau de Petri, plusieurs techniques d'analyse peuvent être utilisées pour étudier les propriétés du réseau et tirer des conclusions sur le système modélisé. La simulation du système modélisé à travers le réseau est la technique la plus fréquemment utilisée, elle est similaire à l'exécution et au test d'un programme. Un bon simulateur d'un réseau est analogue à un bon testeur d'un programme. Il assiste l'utilisateur dans l'examen minutieux de certaines séquences d'exécution. La simulation est extrêmement utile pour la compréhension et l'analyse d'un réseau de Petri, en particulier durant la conception et la vérification de systèmes plus larges [24].

La génération des graphes d'occurrence (GO) est l'une des techniques d'analyse la plus utilisée. L'idée de base consiste en la construction d'un graphe qui contient un nœud pour chaque marquage obtenu par déclenchement d'une transition et un arc qui représente la transition déclenchée. Dans la section 5.5.3, nous allons donner l'exemple d'un graphe d'occurrence ainsi que l'algorithme que nous avons utilisé pour générer un tel graphe.

5.5.2 Modélisation d'une BC par un réseau de Petri

Les réseaux de Petri ont été employés également dans l'étude de systèmes à base de règles [20]. Leur capacité de capturer les propriétés statiques et dynamiques des systèmes les rend très appropriés pour la modélisation des bases de règles. Dans la suite de cette section, d'abord nous allons montrer comment nous pouvons modéliser une BC classique par un réseau de Petri de bas niveau. Ensuite, nous proposons une modélisation d'une BC hybride à l'aide d'un réseau de Petri coloré.

a. Modélisation d'un système de règles classique par un réseau de Petri

En logique classique, une des modélisations possibles d'une base de règles consiste à représenter chaque règle par une transition et les différents faits de la base par des places. Puisqu'un fait a seulement deux états (vrai ou faux), l'utilisation d'un réseau de Petri de bas

niveau est suffisante. Ainsi la présence ou non d'un jeton dans une place correspond à la validité du fait correspondant [20]. Par exemple, soit la règle :

$R1 : A \rightarrow B \wedge C$

En présence du fait A, on peut représenter cette règle comme dans la figure 5-7.

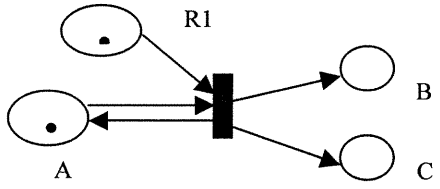


Figure 5-7 : Une règle en logique classique représentée par un réseau de Petri

L'arc de sortie vers la place A permet de garder la véracité du fait A. On remarquera également qu'une place R1 est ajoutée pour empêcher le re-déclenchement de la transition. Après son premier déclenchement, cette place perd son jeton sans le récupérer. Toute la base de règles est transformée ainsi. Deux matrices (faits et règles en lignes, transitions en colonnes) sont utilisées pour représenter les arcs en entrée et les arcs en sortie. La détection des erreurs est basée sur la présence de certaines séquences de transitions spécifiques (violation de contraintes, redondance, etc). En utilisant une représentation matricielle, cette détection se réduit à une résolution d'un ensemble d'équations linéaires [20].

b. Une proposition de modélisation d'un système hybride par un réseau de Petri

Le formalisme des réseaux de Petri de bas niveau n'est pas adéquat pour représenter une base de règles hybride objet-règle. Une telle base, comme nous l'avons expliqué dans la section 5.2, agit comme un ensemble d'opérateurs transformationnels des états du système et non comme un générateur de faits à partir d'autres faits. Un état courant du système est constitué de l'ensemble des objets qui se trouvent dans la mémoire de travail. Le changement d'un état à un autre se fait par le retrait, l'assertion et/ou la modification des objets. La transformation d'une base de règles hybride en un réseau de Petri doit pouvoir représenter les classes, les règles, les conditions, les actions et les objets. Cette transformation doit faire correspondre à ces éléments les places, les transitions, les expressions sur les arcs en entrée, les expressions sur les arcs en sortie et les jetons, respectivement, dans un modèle de réseaux de Petri comme nous allons le voir par la suite. Pour permettre cela, nous avons exploité les propriétés des réseaux de Petri colorés pour modéliser une base de règles hybride. Ainsi, nous proposons les transformations suivantes :

- Les places : les classes utilisées par les règles sont représentées par les places. Ainsi, l'ensemble coloré est constitué des classes du système. Dans la suite de ce chapitre les termes place et classe sont interchangeables
- Les transitions : les règles de la base sont représentées par les transitions. Nous utiliserons dans ce qui suit indifféremment les termes transition et règle.
- Les arcs en entrée : un arc qui lie une place à une transition signifie qu'il y a une condition dans la règle correspondante à la transition qui porte sur les objets de la place. D'après l'étude de certains langages hybrides objets-règles (Rules, JRules, OPS5), et comme expliqué précédemment, deux types de conditions existent. Une condition positive qui filtre un ensemble d'objets vérifiant ses tests, et une condition négative qui est vraie s'il n'existe aucun objet qui vérifie ses tests. Les conditions positives donnent lieu directement à des expressions sur les arcs en entrée, mais le problème se pose dans la transformation d'une règle contenant des conditions négatives. Pour résoudre ce problème, chaque condition négative va être représentée par une expression qui permet de filtrer les objets qui vérifient ses tests et une fonction garde est définie sur la transition qui représente cette règle tel que le nombre de ces objets doit être égal à zéro.
- Les arcs en sortie : les jetons filtrés par les conditions ou ceux nouvellement crées vont être réinjectés selon les cas suivants :
  - Une action *retract* signifie que les jetons référencés par cette action doivent être retirés, et donc on ne doit pas les réinjecter, ce qui donne lieu à aucun arc en sortie.
  - Une action *modify* modifie les jetons de la place sur laquelle porte l'action. Ces jetons doivent être réinjectés avec leurs nouvelles valeurs, et donc un arc doit lier la transition avec leur place.
  - Une action assert crée de nouveaux jetons, donc un arc doit lier la transition de l'action à la place sur laquelle porte l'action.
  - Les jetons d'une place en entrée, sur la quelle ne porte aucune action doivent être réinjectés tels qu'ils sont en créant des arcs de la transition vers leurs places d'origine.

  Ainsi, l'action qui donne lieu à un arc en sortie définit aussi son expression. On note aussi que la réinjection dans ce cas est due aux actions et non à des événements externes.

- Les jetons : chaque place peut être munie de plusieurs jetons. Un jeton est donc un des objets de la classe représentée par cette place. Les termes jeton et objet sont aussi interchangeables par la suite
- Le marquage : pour définir le marquage de notre réseau, il suffit de munir l'ensemble des places de jetons qui représentent les objets instances des différentes classes du système.

Dans notre modélisation, les expressions des arcs en entrée du réseau sont formées à partir des conditions, et les expressions des arcs en sortie à partir des actions. Donc le langage d'inscription est défini à partir des conditions et des actions des règles. Ce langage respecte bien les définitions d'un langage d'inscription pour les réseaux de Petri puisqu'il est constitué d'expressions dont l'évaluation filtre les jetons au niveau des arcs en entrée ayant comme couleur celle de la place en entrée, et ajoute ou modifie des jetons au niveau des arcs en sortie ayant comme couleur celle de la place en sortie.

Contrairement à un réseau de Petri de bas niveau où les arcs contiennent seulement des nombres qui indiquent les multiplicités des places en entrée ou en sortie d'une transition, les arcs définies dans notre modélisation comportent des expressions obtenues à partir des conditions et des actions des règles. Pour déterminer qu'une transition est déclenchable à partir d'un marquage donné, il ne s'agit pas de vérifier qu'un certain nombre de jetons est disponible dans certaines places. Mais il faut évaluer les expressions des arcs en entrée de cette transition pour filtrer les jetons. Une évaluation statique ne permet pas de détecter les jetons qui peuvent traverser une transition car ces expressions contiennent des tests plus complexes et des appels de méthodes. Nous allons voir dans le chapitre de l'implantation qu'un simulateur basé sur le moteur d'inférence fourni avec la grammaire étudiée est le moyen adéquat pour implanter un simulateur qui permet d'exécuter le réseau et chercher les transitions déclenchables.

Graphiquement la modélisation d'une base de règles utilisant un ensemble de classes peut être schématisée dans la figure 5-8.

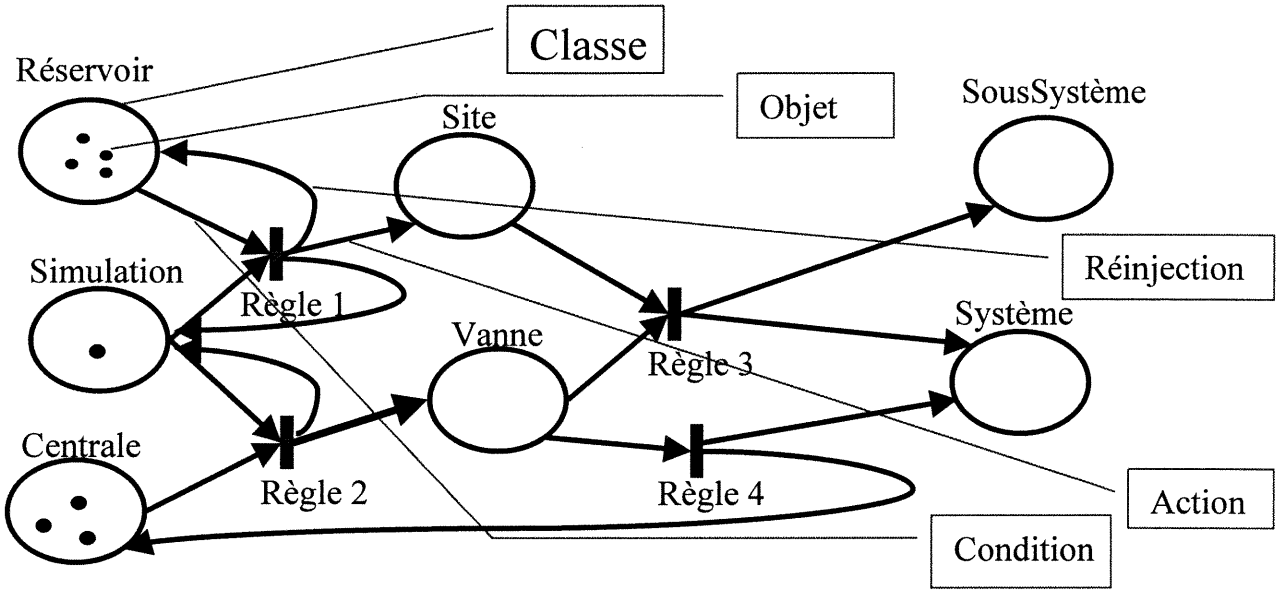


Figure 5-8 : Un réseau de Petri coloré modélisant une BR hybride

5.5.3 Construction du graphe d'occurrence pour effectuer la V&V

5.5.3.1 Construction du graphe d'occurrence

La technique que nous utilisons ici est basée sur la construction d'un graphe d'occurrence qui capte les résultats de simulation du réseau de Petri. Chaque nœud de ce graphe représente un marquage atteignable à partir du marquage initial. Ainsi, le GO permet d'obtenir un arbre d'exécution de la base de règles. Un marquage dans notre cas est l'état de simulation à un moment donné. Il est constitué des objets de chaque classe du système. En démarrant la simulation à partir d'un marquage initial, c'est à dire l'ensemble d'objets en entrée, toutes les règles déclenchables sont prises en considération et le résultat de leurs déclenchements constitue les marquages du GO.

Pour illustrer la notion de graphe d'occurrence obtenu à partir de la simulation d'une base de règles, nous donnons l'exemple de quatre règles qui calculent la fonction Fibonacci (Fib) pour un entier donné. La première calcule le Fibonacci de 1, la deuxième celui de 2, la troisième pour le cas récursif pour tout n > 2, et la quatrième qui retrouve le Fib pour un nombre n > 2 tel que Fib(n-1) et Fib(n-2) sont déjà calculés. Le détail des règles dans le langage Jrules est donné dans l'annexe A. Ces règles font référence à une classe Fib, elle aussi décrite dans l'annexe B, ayant deux attributs : n le nombre pour lequel on veut calculer le Fibonacci et v la valeur calculée. Si l'on effectue la simulation de ce système pour trois objets : Fib(n = 5, v = 0), Fib(n = 3, v = 0), et Fib(n = 1, v = 0), le résultat est un GO dont une partie est décrite dans la figure 5-9.

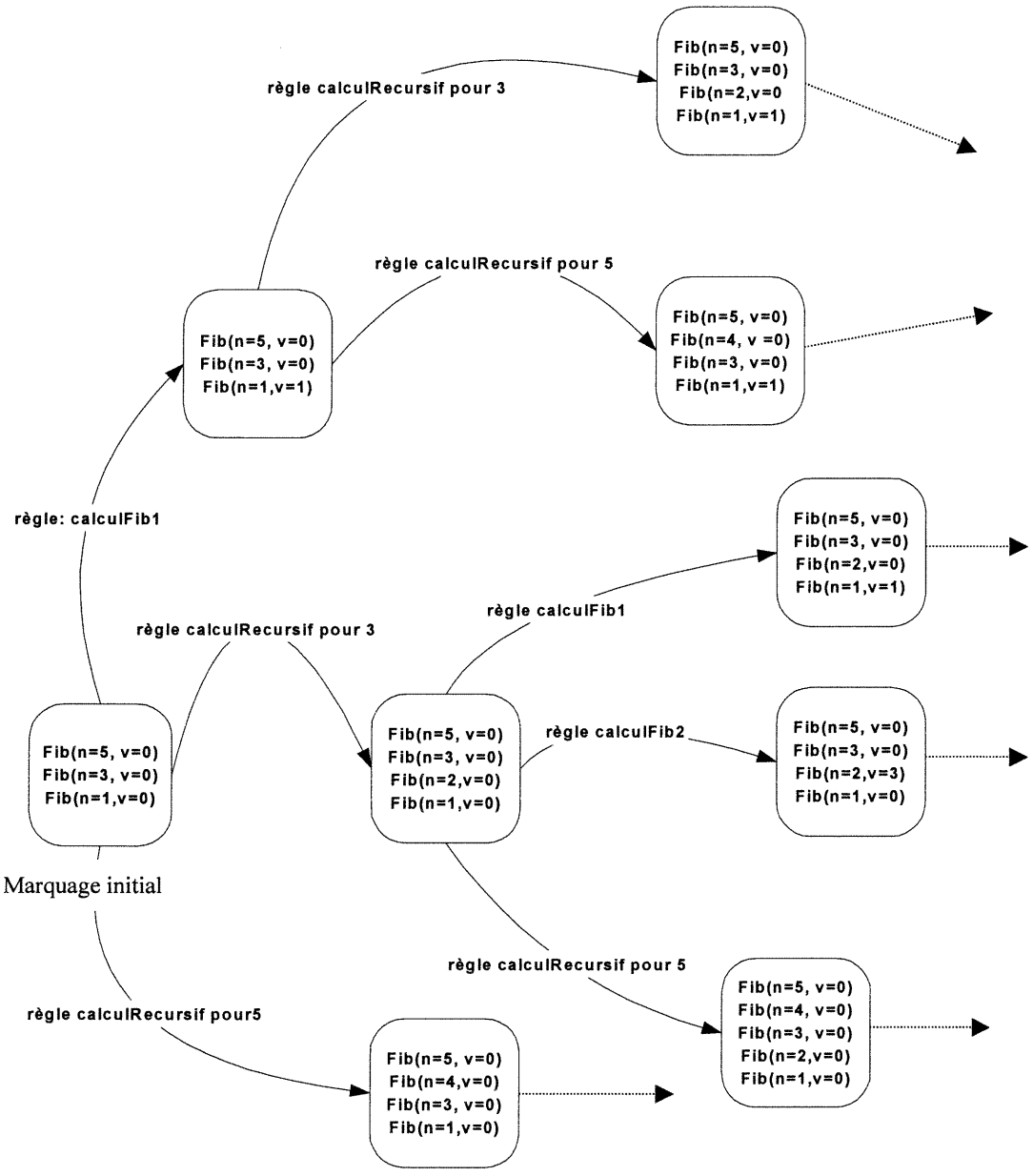


Figure 5-9 : Une partie du GO du Fibonacci de 3 nombres en utilisant des règles

Un arc du graphe est constitué d'un marquage initial, une transition représentant la règle déclenchée, et un marquage final. L'algorithme de simulation et de construction du GO est le suivant :

S : ensemble d'occurrence = $\{M_0\}$, M_0 est le marquage initial

G : graphe d'occurrence = {}

L : liste de marquage non déclenchée = $[M_0]$

répéter

* sélectionner un marquage M à partir de L
* pour chaque transition t qui est déclenchable à partir de M faire
** générer M' le marquage après déclenchement de t
** si M' n'est pas un élément de S alors
*** ajouter M' à S
*** ajouter M' à L
** ajouter l'arc (M,T,M') a G

jusqu'à ce que L est vide

Au cours d'une utilisation normale de la base de règles, le moteur d'inférence part d'un état initial et essaie de suivre un seul chemin en choisissant à tout moment une seule règle à déclencher. L'algorithme de construction du GO prend en considération toutes les règles déclenchables à partir de la même situation. Il procède par saturation pour couvrir toutes les situations possibles d'exécution de la base de règles.

Pour exécuter le réseau, nous avons besoin d'un marquage initial, c'est à dire l'ensemble des jetons pour les différentes places (initialisation de la simulation). En général, les réseaux de Petri colorés sont munis d'expressions d'initialisation qui permettent de générer ce marquage initial. Dans notre cas, nous exploitons la spécification des états définis pour chaque classe, à l'aide du design pattern State et des contraintes OCL, pour générer les objets couvrant les états initiaux des différentes classes. Ces objets constituent donc le marquage initial. Chaque état initial est défini par des contraintes sur certains attributs de la classe. Pour générer des objets caractérisant cet état, nous affectons des valeurs à ces attributs de telle façon à ce que les contraintes soient respectées. Les autres attributs non contraignants peuvent prendre des valeurs quelconques de leurs types. Une façon simple de procéder est de récupérer des objets à partir de la base d'objets et de changer les valeurs d'attributs pour respecter les contraintes des états initiaux.

5.5.3.2 Vérification et validation

Une fois le GO construit, il s'agit ensuite de parcourir ce dernier pour chercher les marquages qui induisent des situations d'erreurs et d'anomalies obtenues à partir de l'exécution de la base de règles. Le but est d'aviser l'expert pour qu'il effectue les changements nécessaires. Notre recherche dans le GO est basée sur la comparaison des marquages. Un marquage est défini par les jetons des places qui correspondent aux objets des

différentes classes de la base modélisée. Le problème se réduit à une comparaison des objets. Nous donnons dans ce qui suit les définitions suivantes:

- Égalité des objets : pour qu'ils soient égaux, deux objets doivent d'abord avoir la même classe. L'égalité des objets est définie par rapport aux attributs de leur classe. Chaque classe doit fournir les attributs sur les quels on sebase pour obtenir l'égalité de ses objets. En OO, ceci traduit par l'implantation d'une méthode qui compare les objets de la classe (equals en Java). Par exemple, deux objets de la classe Fib sont identiques s'ils ont la même valeur pour l'attribut n pourlequel on veut calculer le Fibonacci.
- Égalité des places : deux places sont égales si leur nombre de jetons (objets) est le même, et les jetons sont égaux un à un.
- Égalité des marquages : deux marquages sont égaux si leurs places sont égales une à une.

Enfin, L'étude des marquages du graphe d'occurrence nous permet d'analyser la base de règles. Nous donnons la définition des deux classes suivantes (syntaxe Java) que nous allons utiliser pour les exemples :

```
public class C1 {
        boolean a;
        float b;
}
public class C2 {
        char c;
        int d;
}
```

avec les contraintes suivantes sur les attributs:

valeurs possibles de b : 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0

valeurs possibles de c : 'p' (petit), 'm' (moyen), 'g' (grand)

valeurs possibles de d : 1,2,3,4,5,6,7,8,9,10

Dans ce qui suit, nous présentons la détection des principales erreurs et anomalies. Les exemples que nous allons montrer sont simples à détecter manuellement. En réalité, les règles hybrides sont plus complexes surtout lorsqu'elles contiennent des appels de méthodes.

a) Vérification de la cohérence

Elle porte sur les éléments suivants :

- Les règles en conflit

Comme montré dans la figure 5-10, toutes les règles déclenchables à partir de la même situation (même marquage) sont potentiellement en conflit.

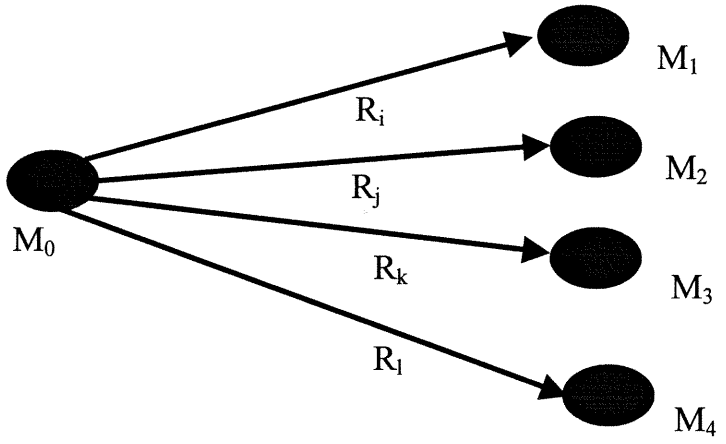


Figure 5-10 : Situation potentielle de conflit

Cet ensemble de règles déclenchables dans la même situation doit être étudié par l'expert pour vérifier si ces règles peuvent effectivement se déclencher dans la même situation. Même si une stratégie de résolution de conflits peut être prédéfinie par l'expert dans une situation d'un ensemble de règles candidates au déclenchement (affectation de priorités, regroupement par paquet, etc.), la possibilité de conflits dûs à la présence d'autres règles non désirées ou à l'absence d'une règle désirée est réelle. Ces possibilités sont dûes aux incohérences des assertions ou aux connaissances incomplètes. Contrairement aux systèmes monotones pour lesquels on peut détecter automatiquement les situations de conflit ($a \rightarrow b$ et $a \rightarrow \neg b$ sont automatiquement en conflit), les systèmes non monotones dépendent du domaine d'application des règles, et c'est à l'expert de trancher après détection de situations potentielles de conflit.

Pour chaque arc du graphe d'occurrence, on cherche les arcs qui peuvent avoir le même marquage initial, ces arcs constituent l'ensemble de règles qui peuvent se déclencher dans la même situation. Nous donnons l'exemple des deux règles suivantes (syntaxe Jrules, voir annexe C) :

```
rule  règle1 {
        when {
                obj1 : C1(a == true; b < 0.5);
                obj2 : C2 (c != 'p');
        }
        then {
                modify obj1 {
                        b = 0.9;
                }
        }
}
rule règle2 {
        when {
                obj1 : C1(a == true);
                obj2 : C2(c == 'g'; d < 3);
        }
        then {
                modify obj1 {
                        a = false;
                }
        }
}
```

Si à un moment donné, nous avons dans la mémoire de travail les objets obj1a(a = true, b = 0.1), obj1b(a = true, b = 0.2) de la classe C1 et obj2a(c = 'g', d = 1), obj2b(c = 'g', d = 2) de la classe C2, et donc ils représentent un marquage du GO à partir duquel nous pouvons déclencher les deux règles *règle1* et *règle2*. Nous constatons donc que ces deux règles sont potentiellement en conflit et il faut les étudier pour décider que cette situation est voulue ou inappropriée dans la BC. La règle *règle2* change l'attribut a à false alors que la règle *règle1* ne le change pas et reste à true, ce qui induit deux situations différentes pour cet attribut. Selon le domaine d'application et la signification de ces attributs, les experts peuvent prendre une décision.

- Règles redondantes

Si des règles différentes aboutissent au même marquage (figure 5-11), nous pouvons parler de règles redondantes.

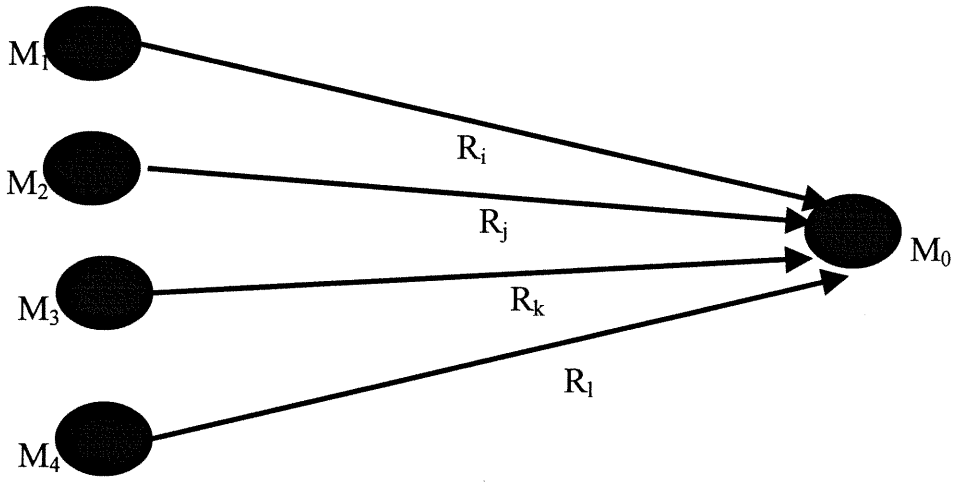


Figure 5-11 : Situation de redondance

Pour détecter ce type de règles, on examine les marquages finaux des arcs du graphe d'occurrence. Si on trouve des règles aboutissant aux mêmes marquages finaux, alors elles sont potentiellement redondantes. Même si les conditions et les actions de ces règles ne sont pas identiques et/ou subsumées, le fait d'aboutir au même marquage induit une situation possible de redondance. La décision finale revient également à l'expert qui les a défini et qui doit décider des changements à effectuer. Il est important de souligner que même dans les systèmes classiques (monotones), les situations de redondance ne sont pas rejetées automatiquement. Elles peuvent être crées volontairement pour des raisons de performance par exemple. Dans un système hybride, la sémantique de règles ne peut pas être obtenue directement à partir de ces dernières. L'implication des experts est toujours nécessaire pour la prise de décisions.

Nous donnons l'exemple des deux règles suivantes :

```
rule règle1 {
    when {
        obj1 : C1 (a == false; b > 0.3);
        obj2 : C2(c == 'g');
    }
    then {
        modify obj1 {
            a = true;
```

```
        }
    }
}
```

À partir du marquage : obj1a(a = false, b = 0.5) de la classe C1 et obj2a(c = 'g', d = 3) de la classe C2, la règle *règle1* est déclenchable et nous aboutissons au marquage : obj1a(a = true, b = 0.5) et obj2a(c = 'g', d = 3).

```
rule règle2 {
    When {
        obj1 : C1(b < 0.8);
        obj2 : C2(c != 'p'; d < 0.5);
    }
    then {
        modify obj2 {
            c = 'g';
            d = d+1;
        }
    }
}
```

À partir du marquage : obj1a(a = true, b = 0.5) de la classe C1 et obj2a(c = 'm', d = 2) de la classe C2, la règle *règle2* est déclenchable et nous aboutissons au marquage : obj1a(a = true, b = 0.5) et obj2a(c = 'g', d = 3). Même si les deux règles ont des conditions et des actions différentes, nous constatons que leur déclenchement peut aboutir au même marquage. Ceci traduit une situation de redondance qui doit être examinée par l'expert.

- Règles inutiles

Une règle peut être utilisée dans un ou plusieurs arcs du GO suivant le nombre de fois où elle est exécutée. Elle peut être inutile s'il n'y a aucune règle dans le GO qui utilise les marquages produits par elle et si aucun de ces marquages n'est considéré comme but final de l'inférence. Si chaque marquage final de l'arc de la règle, non déclaré comme but final de l'inférence, n'est pas utilisé par aucun autre arc, la règle est totalement inutile sinon elle l'est partiellement. Pour savoir si un arc est un but final de l'inférence, on examine ses objets. Si chaque objet fait référence à un état final de sa classe alors ce marquage est un but final de l'inférence.

Nous considérons la règle suivante :

```
rule règle1 {
```

```
when {
        obj1 : C1(a == false; b == 0.2);
        obj2 : C2(c == 'm'; d < 5);
}
then {
        modify obj1 {
                a = true;
        }
        modify obj2 {
                d = d-1;
        }
}
}
```

À partir du marquage : obj1a(a = false, b == 0.2) de la classe C1 et obj2a(c = 'm', d = 4) de la classe C2, cette règle est déclenchable et son déclenchement donne lieu au marquage: obj1a(a = true, b = 0.2) et obj2a(c = 'm', d = 3). Si ce marquage n'est utilisé par aucune autre règle de la base alors nous pouvons constater que cette règle est inutile. Si tous les marquages obtenus par déclenchement de cette règle ne sont pas utilisés par les autres règles alors cette règle est totalement inutile, et si certains marquages sont utilisés et d'autres ne le sont pas, cette règle est partiellement inutile.

- Règles inatteignables

Pour qu'une règle soit atteignable, il faut qu'un de ses états déclencheurs soit obtenu directement à partir du marquage initial ou indirectement à partir des marquages intermédiaires du GO. Si dans un GO, une règle n'utilise aucun marquage initial ou intermédiaire, elle est alors inatteignable.

Nous donnons l'exemple de la règle suivante :

```
rule règle1 {
        when {
                obj1 : C1(a == true; b < 0.1);
                obj2 : C2(c == 'm');
        }
        modify obj1 {
                b = 0.9;
        }
```

```
}
```

S'il n'existe aucun marquage (initial ou intermédiaire) dans le GO qui permet de déclencher cette règle alors elle est inatteignable. Dans la première condition, le deuxième test porte sur des valeurs impossibles pour l'attribut b c'est pourquoi aucun marquage du GO ne peut rendre cette règle atteignable.

b) Vérification de la cohérence interne

Comme défini dans le premier chapitre, la vérification de la cohérence interne consiste à s'assurer que la base de règles ne viole aucune des contraintes définies dans le modèle de cohérence. Afin de maintenir la satisfaction de ces contraintes, on doit s'assurer que toutes les exécutions du système aboutissent à des données cohérentes. Le graphe d'occurrence contient l'ensemble des marquages obtenus par simulation de la base de règles. En examinant ces marquages, on peut détecter les objets qui ne respectent pas les contraintes OCL définies sur leurs classes. Ainsi, les règles qui produisent des marquages contenant ces objets sont responsables de la violation de ces contraintes. Ces règles doivent être examinées par les experts pour corriger les situations d'incohérence interne.

Nous donnons les exemples des règles suivantes dont leur déclenchement permet de violer les contraintes définies sur les attributs :

```
rule règle1 {
        when {
                obj1 : C1(a == false; b > 0.3);
                obj2 : C2(c == 'g'; d == 1);
        }
        then {
                modify obj2 {
                        d = d-2;
                }
        }
}
```

Après déclenchement de cette règle, l'attribut d aura une valeur −1 qui est une valeur non permise.

```
rule règle2 {
        When {
                obj1 : C1( b == 0.8);
```

```
        obj2 : C2(c == 'm'; d > 5);
    }
    then {
        modify obj1 {
            b = b*2;
        }
    }
}
```

si cette règle se déclenche, elle va donner une valeur non permise pour l'attribut b, c'est à dire 1.6.

c) Vérification de la complétude

L'utilisation du design pattern ***State*** nous a permis de définir les états caractéristiques des différentes classes de la base de connaissances. Il est donc possible de connaître tous les états d'une classe. Chaque état est défini par des expressions OCL sur les attributs de la classe. Une base de règles est complète si elle peut produire au moins un objet par état pour chaque classe de la BC. Pour chaque classe, on examine donc ses objets qui se trouvent dans les marquages du GO. Si certains états ne sont pas référencés dans aucun des marquages, on peut constater que certaines règles sont probablement manquantes et donc leur absence fait que certains états ne sont pas atteints. nous nous basons sur les états des classes pour étudier la complétude de la base de règles parce que ces états reflètent la réalité des données du domaine d'application. Ils doivent être définis par les experts du domaine pour modéliser la dynamique du SBC.

d) Validation

Comme nous l'avons vu dans le premier et le troisième chapitre, la validation s'intéresse à l'étude du fonctionnement du SBC. Il s'agit de prouver que ce dernier donne les résultats attendus par les experts. Le test de fonctionnement d'un SBC est indépendant du formalisme de représentation des connaissances. En général, il s'agit de disposer d'un ensemble de cas de test comme entrées au système, faire des simulations de fonctionnement, et analyser les résultats obtenus.

Les applications développées à l'aide des systèmes de production ne possèdent pas un graphe de contrôle de flux. Cependant, un graphe de dépendance peut être généré en cherchant les dépendances entre les règles. Nous exploitons les résultats de simulation de la base par le réseau de Petri pour chercher dynamiquement ces dépendances. À partir du graphe

d'occurrence, et partant du marquage initial en utilisant une recherche en profondeur, nous pouvons construire les différents chemins d'exécution jusqu'aux marquages finaux qui constituent les buts d'inférence. Il s'agit de mettre à plat les différents chemins de déductions qui se trouvent dans le graphe d'occurrence. Chaque marquage final d'un arc peut constituer un marquage initial d'un ou plusieurs autres arcs. À partir de ces relations d'activation entre les arcs, nous construisons le graphe de dépendance. L'algorithme de construction du graphe d'occurrence prend en considération toutes les règles déclenchables à partir de la règle courante, ainsi toutes les dépendances possibles peuvent être retrouvées dans le graphe. L'analyse du graphe de dépendance permet de détecter les situations du mal fonctionnement du SBC.

5.6 Conclusion

Dans ce chapitre nous avons proposé un modèle de V&V. Ce modèle est basé sur les éléments suivants :

- Exploitation des réseaux de Petri de haut niveau pour représenter un environnement hybride objet-règle
- Adaptation et redéfinition des erreurs et des anomalies en fonction de cette représentation
- Exploitation de la modélisation objet (UML, OCL, Design pattern State, etc.) pour supporter la V&V. Elle offre des mécanismes puissants pour la spécification d'états et de contraintes sur un SBC.

Un des résultats importants de notre travail est le rapprochement entre les méthodes de V&V du génie logiciel et celles des SBC. Ce mariage entre les deux formalismes utilisés (objets et règles) permet d'exploiter les techniques puissantes d'un domaine dans un autre. Ainsi la spécification des états, technique souvent utilisée en OO pour spécifier, tester et valider des systèmes OO, nous a permis de vérifier et valider une base de règles.

Après avoir étudié la V&V d'une base de règles hybride, nous allons dans le chapitre suivant faire les choix relatifs aux différents outils d'implémentation, à la représentation physique des connaissances utilisées, et à l'implantation du modèle de V&V de la base de connaissances. Nous allons détailler les implémentations réalisées au niveau du gestionnaire de connaissances suivant l'architecture proposée dans le quatrième chapitre.

Chapitre 6

Implantation et Expérimentation

L'utilisateur principal de l'environnement PReCI est le groupe de *Gestion des ressources hydriques* (GRH) de la division Énergie Électrique, Québec (ÉÉQ) d'ALCAN. ÉÉQ gère les installations de production, de transport et de distribution hydroélectrique d'ALCAN au Saguenay Lac-Saint-Jean. Le réseau hydrique d'ALCAN a une capacité énergétique annuelle d'environ 2000 mégawatts en moyenne; il compte 6 centrales hydroélectriques, 28 ouvrages de retenue, 43 groupes turbines-alternateurs, 850 kilomètres de lignes de transport d'énergie, un réseau d'une trentaine de stations hydrométéorologiques, etc. L'objectif ultime de la planification de l'opération d'un tel système est de produire l'énergie hydroélectrique demandée par les usines d'ALCAN au coût le plus bas tout en respectant les points suivants :

- Utilisation efficace de l'eau
- Prise en compte de l'incertitude hydrologique future
- Respect des contraintes sécuritaires

Pour ce faire, un processus décisionnel de gestion des ressources hydriques s'appuyant sur quatre étapes est utilisé. Il est décrit dans la figure 6-1.

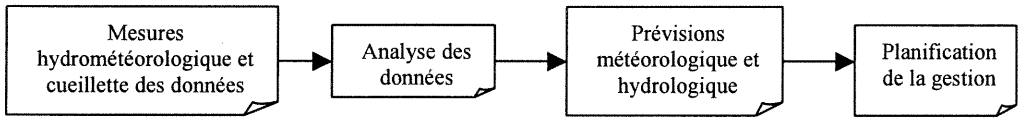


Figure 6-1 : processus décisionnel effectué par le groupe GRH

Dans ces tâches de planification, des systèmes informatiques basés sur des modèles mathématiques éprouvés pour ce genre d'applications, sont utilisés à des fins d'optimisation ou de simulation. Cependant, les connaissances utilisées dans le processus décisionnel au sein du groupe GRH, sont nombreuses et diverses. Le problème est que celles-ci sont souvent enfouies dans le code des programmes qui les utilisent. Cette situation devient plus

problématique lorsque l'ingénieur veut explorer de nouveaux scénarios, en modifiant l'une de ces connaissances. Il n'a d'autre recours que de parcourir le code des routines en question, afin d'y apporter les modifications escomptées. Le principal inconvénient d'une telle solution est l'absence de séparation entre le niveau connaissance et le niveau inférence. Il a pour conséquence immédiate la restriction des champs d'investigation et d'exploration envisagés par les analystes du groupe GRH. De plus, l'architecture actuelle de type « boîte noire» engendre un manque de flexibilité des outils utilisés. Une conséquence de cela est la difficulté à maintenir et à faire évoluer un tel système. Nous travaillons sur une nouvelle solution respectant les critères suivants :

- Être une solution type Système à base de connaissances, délimitant donc nettement les niveaux connaissance et raisonnement. Le niveau connaissance va inclure plusieurs formalismes de représentation des connaissances, les mieux adaptés à la résolution du problème.

- Exploiter dans le courant du processus décisionnel, les modèles conventionnels (actuellement à l'usage) retenus, et les intégrer dans la solution. Les routines implantant ces modèles, passent à travers un processus de réingénierie, et cohabitent avec des routines d'inférences sur les connaissances formalisées.

- Assurer une gestion intelligente et automatique de la connaissance, par le biais d'un gestionnaire de connaissances offrant toutes les fonctionnalités souhaitées.

Dans ce projet nous avons opté pour le formalisme objet-règle aussi bien au niveau de la conception que de l'implantation des différents types de connaissances. Le choix des outils d'implantation s'inscrit donc dans ce sens. L'utilisation d'une conception objet, exprimée sous formes de diagrammes UML, pour décrire la partie statique de notre système, ainsi qu'une description selon un style déclaratif de la partie raisonnement du système, sous formes de règles, nous oblige à faire le choix des langages qui offrent ces deux possibilités. Au cours de ce chapitre, nous allons présenter les outils utilisés pour implanter la base de connaissances et décrire une façon de coder les contraintes et les états du système. Les différentes réalisations et problèmes rencontrés au niveau du gestionnaire de connaissances vont être détaillés aussi.

6.1 Base de connaissances

6.1.1 Base d'objets

Ces dernières années, le langage Java est devenu de plus en plus utilisé pour la réalisation de plusieurs applications dans différents domaines, aussi bien dans l'industrie que dans le domaine de la recherche. En plus d'être indépendant de la plate forme et de bien respecter les concepts de l'orienté objet, Java est un langage riche, car il offre plusieurs API (Application Programming Interface) utiles pour différents besoins (interfaces, réseau, base de données, sécurité, systèmes distribués, etc.).

Au niveau de la représentation de connaissances, Java nous permet de bien exprimer la partie statique du système (connaissances intentionnelles) sous formes de classes; les diagrammes de modélisation UML sont traduits sous forme de classes écrites en Java qui décrivent le contenu du système en terme de données (attributs) et de comportement (méthodes). Les connaissances extensionnelles sont gérées par l'ensemble des objets instances des différentes classes. Un mécanisme de persistance doit être mis en place pour stocker ces objets d'une façon permanente dans une base de données. Pour cela, on a utilisé le système de gestion de bases de données objet, ObjectStore. C'est un ensemble d'API qui permettent de stocker des objets Java et offrir des mécanismes de recherche et de navigation dans ces objets.

Le diagramme de classes du système hydrique d'ALCAN se trouve en annexe D. Ce diagramme nous a permis de générer les classes Java comme point de départ pour gérer la base d'objets et construire la base de règles. Il faut noter qu'après le processus d'ingénierie du système actuel d'ALCAN, nous avons décidé de réutiliser les modèles de planification faisant appel à des calculs mathématiques. Ces modèles, disponibles sous forme de routines FORTRAN, vont être appelés à partir des méthodes des différentes classes nécessitant ces informations. Pour ce faire, nous avons utilisé le JNI (Java Native Interface) pour définir des méthodes natives qui font appel à ces routines. Il faut noter que ces appels ne sont pas directes; une couche Java - C++ a été réalisée pour interfacer Java avec Fortran (JNI fonctionne seulement avec C et C++).

6.1.2 Base de Règles

La partie déclarative de notre système (base de règles) doit être exprimée sous forme de règles de production. Dans le formalisme objet-règle, les règles sont exprimées par des conditions et des actions sur les classes. Sous sa forme standard, Java n'offre pas la possibilité

de représenter des règles de production. Cependant plusieurs outils (Jess, Jrules, etc.) sont disponibles et peuvent être considérés comme des extensions aux librairies de base de Java pour permettre la définition de règles sur des classes Java. Nous avons choisi l'outil Jrules [30] pour implanter notre base de règles. C'est un ensemble d'API Java qui nous permet, après avoir défini les règles en utilisant sa propre grammaire BNF (Backus Naur Form), de manipuler et d'utiliser ces règles pour faire de l'inférence.

Avant de parler de la grammaire BNF de Jrules, on décrit comment son interaction se fait avec Java. La figure 6-2 montre l'interface Jrules avec Java.

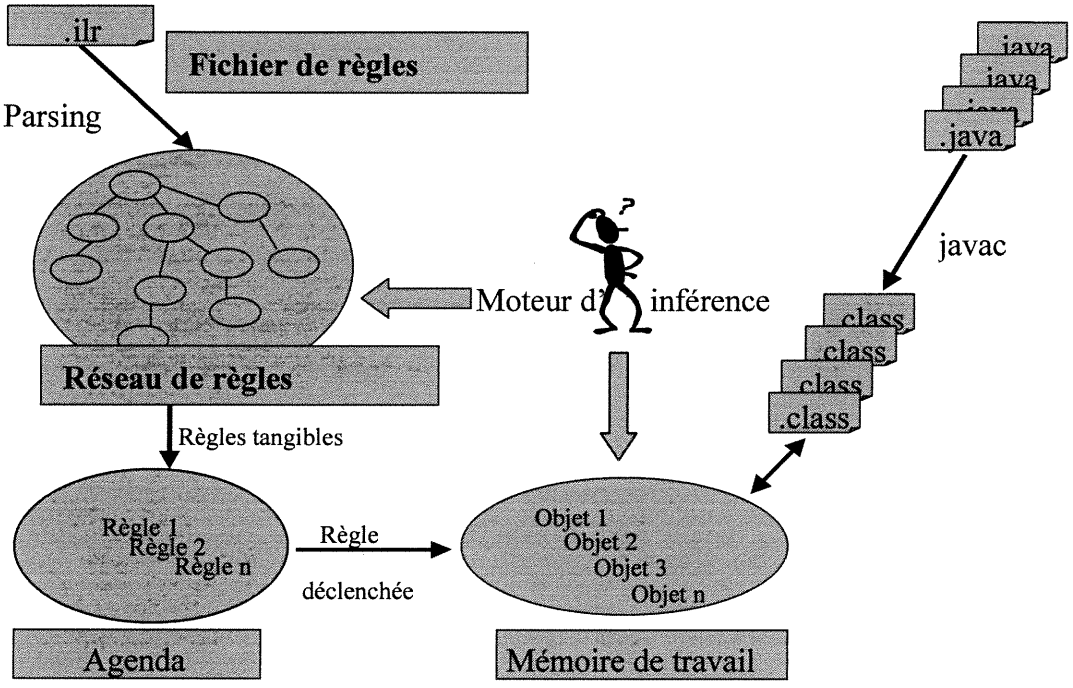


Figure 6-2 : Interface Jrules - Java

À partir des classes Java compilées et un fichier de règles d'entrée chargé par le moteur d'inférence, ce dernier initialise la mémoire de travail par des objets Java et construit un réseau de règles. À ce moment là, la session de raisonnement commence par la recherche des règles tangibles et leur instantiation dans l'agenda. Après, une seule règle est choisie pour être déclenchée. Ceci va modifier le contenu de la mémoire de travail. Ce cycle continue jusqu'à ce qu'il n'y ait plus de règle tangible ou que la session d'inférence est arrêtée explicitement par l'utilisateur. Le but final de l'inférence est constitué par l'ensemble d'objets dans la mémoire de travail qui ont subi tous les changements effectués par les règles déclenchées (ajout, modification et retrait).

La grammaire BNF de Jrules est une grammaire hybride non monotone. Jrules utilise Java comme langage hôte et exploite tous ses mécanismes objets. Il utilise les classes

compilées de java (fichiers .class) et n'essaie à aucun moment d'analyser le code source. Les classes référencées dans Jrules doivent être déclarées *public*. Les attributs, les méthodes et les constructeurs utilisés dans les règles doivent aussi être déclarés *public*. Ces classes, attributs, méthodes et constructeurs sont définis par l'utilisateur dans un autre paquet. S'ils ne sont pas déclarés public, ils ne peuvent pas être utilisés par aucun autre paquet externe, et leur accès par Jrules déclenche des exceptions. La grammaire de Jrules est définie dans l'annexe C.

Ainsi l'intégration Java-Jrules nous offre une plate forme complète en Java pour implanter une base de connaissances hybride objet-règle. Dans le cadre du gestionnaire de connaissances, une couche va être ajoutée au dessus de ces outils. Cela va permettre de considérer les aspects suivants de notre système :

- Les utilisateurs de notre système sont des experts de leur domaine. Ils ne sont pas obligés de travailler directement sur la grammaire BNF de Jrules, ou de maîtriser le langage Java. Le rôle du module de maintenance est de leur permettre la manipulation de la base de connaissances (base d'objets et base de règles) en utilisant des interfaces conviviales, et donc avec plus de souplesse et de facilité.
- Après avoir construit la base de connaissances ou suite à des modifications apportées à celle-ci, des incohérences ou des incomplétudes peuvent facilement s'infiltrer. Dans le module de vérification, nous allons pouvoir vérifier la base de connaissances en cherchant ces incohérences et ces incomplétudes.
- Une base de connaissances exempte d'incohérences et d'incomplétudes n'assure pas automatiquement un bon fonctionnement du système. La logique définie par l'expert peut engendrer des résultats inattendus par celui-ci. La validation du système est donc requise pour assurer son bon fonctionnement.

6.2 Implantation des états et des contraintes du système

Nous avons proposé une façon de modéliser les contraintes définies dans le modèle de cohérence et spécifier les états des différentes classes de la BC. En exploitant le langage OCL, nous avons pu enrichir les diagrammes de classes de notre système pour définir le modèle de cohérence et les états du système. OCL est un langage formel qui vient d'être adopté avec le standard UML au sein de la communauté du génie logiciel. Jusqu'à date, Java ne dispose pas de techniques explicites et directes pour implanter des contraintes, il n'y a pas de mécanismes en Java pour décrire des invariants ou des pré et post conditions dans une classe Java. Or, en

exploitant certains aspects de Java (réflexion et techniques de composants de Java, JavaBeans), on peut implanter certains types de contraintes [25]. Il s'agit de :

- Valeurs minimums et maximums
- Valeurs valides et invalides
- Types admissibles (types possibles des valeurs valides pour un attribut)
- Valeurs par défaut

Java propose un standard de codification qui permet de reconnaître les attributs, les méthodes, ainsi que les événements définis dans une classe en suivant son architecture de composants JavaBean. Ceci permet de réutiliser facilement du code Java. Cette technique est exploitée aussi par les environnements de développement intégrées pour éditer visuellement le contenu des composants Java (attributs, méthodes, et événements) qui suivent ce standard de codification. Ainsi, une méthode portant la signature *int getAge()* définie dans une classe *Personne* signifie que cette classe a un attribut *age* de type *int*. En suivant ce standard de nomination, on peut adopter une façon pour définir des méthodes qui implémentent des contraintes des types cités précédemment. On définit les conventions suivantes pour coder ces méthodes :

- Valeurs minimums et maximums: pour les attributs de type numérique, les méthodes suivantes peuvent être utilisées pour définir des valeurs minimums et maximums :

  <type> getXXXMinValue()

  <type> getXXXMaxValue()

  XXX désigne un nom d'attribut d'une classe sur lequel est définie la contrainte. Par exemple une contrainte qui définit la valeur minimum d'un attribut *debit* de la classe *Reservoir* est implémentée par la méthode suivante :

  public float getDebitMinValue() { retrun 0; }

- Valeurs valides et invalides : on spécifie explicitement la liste des valeurs possibles pour un attribut par la méthode suivante :

  <type>[] getXXX ValidValues()

  De même pour la liste des valeurs impossibles, on la définit ainsi:

  <type>[] getXXXInvalidValues()

- Types admissibles : les valeurs possibles d'un attribut non primitif doivent être de certains types, ils peuvent être définis par :

  Class[] getXXXValidClasses()

- Les valeurs par défaut : pour définir une valeur par défaut d'un attribut, on utilise la méthode suivante :

<type> getXXXDefaultValue()

À partir du paquet général dans lequel sont définies les classes du système ALCAN, nous dérivons des sous paquets, chacun définit les contraintes et les états d'une classe donnée. Ainsi, chaque paquet contient la classe principale (Reservoir par exemple) et un ensemble de classes définissant ses états. Au niveau de la classe principale, nous définissons les méthodes qui implémentent les contraintes du modèle de cohérence, et au niveau des classes qui décrivent des états les méthodes qui implémentent des invariants caractérisant chaque état.

D'après l'étude du système ALCAN, nous avons constaté que la plupart des attributs manipulés sont des valeurs réelles exprimant des grandeurs (élévations, volumes, apports, débits, etc.). Les contraintes qui peuvent être définies sur ces grandeurs sont:

- Liste des valeurs possibles
- Valeurs minimums et maximums
- Valeurs par défaut

Certaines contraintes définies sur des attributs font référence à d'autres attributs de la même classe ou des autres classes du système. Ces liens peuvent être obtenus par navigation entre les différents types de relations entre les classes (association, agrégation, composition , et héritage).

On donne l'exemple des contraintes définies sur la classe Reservoir:

- Élévations et volumes dans les limites possibles:

Réservoir	*Élévation Min (mètres)*	*Élévation max (mètres)*	*Volume min (hmc)*	*Volume max (hmc)*
RLM	487.18	495.41	0	3140.1
RPD	408.59	440.59	17.9	5227.5
RCD	163.71	174.07	0	538.7
RCS	133.34	140.05	0	120.4
RLSJ	96.51	102.91	0	6628.8
RIM	92.55	103.37	0	180.1
RCC	57.04	67.86	0	184.1

Tableau 2 : Valeurs minimums et maximums pour les élévations et les volumes

- Débit >= 0
- Débit <= Débit maximal (fonction du niveau)
- pourcentage plein (réservoir) >= 0

Le tableau contient des contraintes de types minimum et maximum. On donne l'exemple d'implémentation de la méthode définissant des contraintes de minimum sur l'élévation. les autres méthodes sont codées de le même façon :

```
public float getElevationMin() {
  if this.getCode().equals("RLM") return 487.18;
   else if this.getCode().equals("RPD") return 408.59;
    else if this.getCode().equals("RCD") return 163.71;
     else if this.getCode().equals("RCS") return 133.34;
      else if this.getCode().equals("RLSJ") return 96.51;
       else if this.getCode().equals("RIM") return 92.55;
        else if this.getCode().equals("RCC") return 57.04;
         else return 0;
}
```

Java offre le mécanisme de réflexion que l'on peut utiliser en mode exécution pour connaître les informations d'une classe (attributs, méthodes, événements). En disposant de la classe d'un objet, on peut dynamiquement l'interroger en utilisant l'API de réflexion de Java [26] pour chercher les méthodes qui implémentent des contraintes sur les classes. L'appel de ces méthodes sur l'objet nous retourne les valeurs qu'on doit utiliser pour vérifier la violation des contraintes ou la détection de l'état d'un objet.

6.3 Gestionnaire de connaissances

6.3.1 Module de Maintenance

Ce module inclut deux parties :

6.3.1.1 Éditeur de règles

L'écran principal de l'éditeur de règles est constituée de deux parties :

- Un arbre qui affiche la base organisée par paquets
- Un panneau dans lequel sont affichés le détail de la règle sélectionnée.

A ce niveau, on peut soit ajouter un nouveau paquet ou supprimer un paquet. En parcourant l'arbre, on peut consulter les règles de toute la base. Les règles affichées ne peuvent pas être modifiées directement à partir de cet écran (figure 6-3), il s'agit d'un mode consultation dans lequel les règles sont représentées sous un format plus ou moins naturel traduit à partir de la grammaire de Jrules.

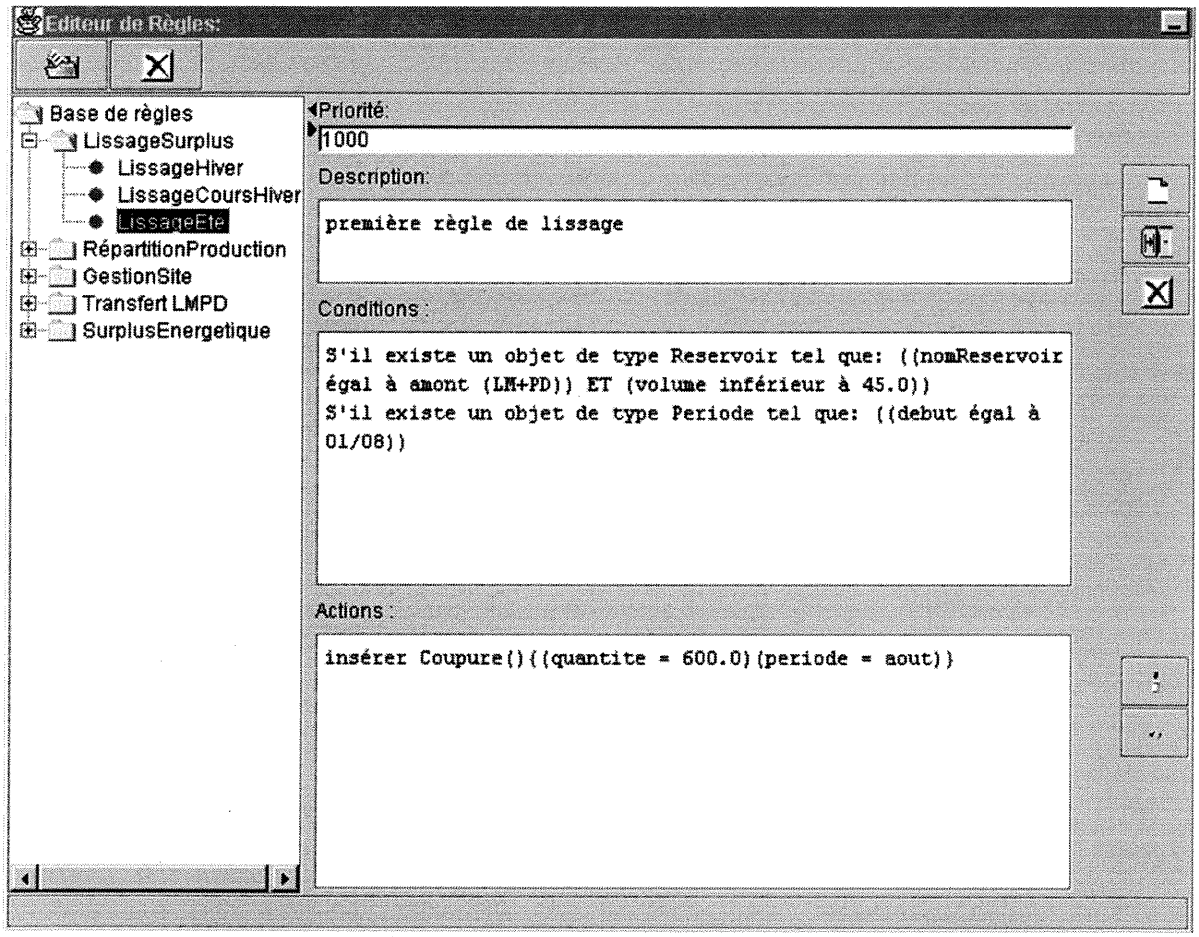


Figure 6-3 : Interface de l'éditeur de règles

Une règle est constituée des éléments suivants :

- Nom de la règle : chaîne de caractère
- Priorité de la règle : entier
- Description de la règle : chaîne de caractère
- Conditions de la règle : liste de conditions
- Actions de la règle : liste d'actions.

Pour entrer une nouvelle règle, on clique sur le bouton « Ajouter une règle». L'écran de la figure 6-4 s'affiche.

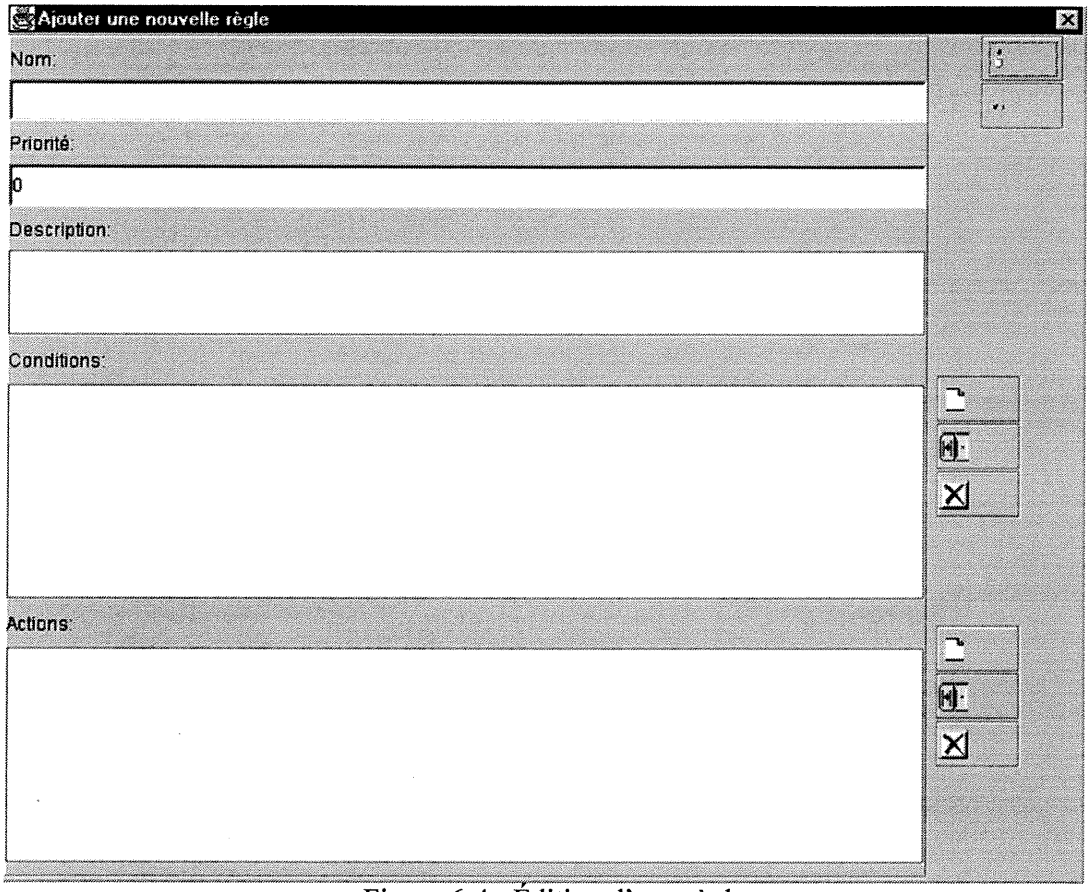


Figure 6-4 : Édition d'une règle

Après avoir spécifié le nom, la priorité, et la description de la règle, on utilise les boutons disponibles au niveau conditions (sous forme de liste) pour manipuler celles-ci (ajouter, modifier ou supprimer une condition). On donne l'exemple de l'ajout de la règle de lissage :

Si volume des réservoirs amont (LM+PD) en début de journée au 1er août < 45%

Alors coupure de 600 MW.

On a deux objets à manipuler dans les conditions de cette règle, l'objet Reservoir pour spécifier son nom et son volume, et l'objet Periode désignant la période définie dans la condition. Pour entrer une condition on clique sur le bouton nouvelle condition. On choisit le type de la condition à entrer (simple, exist, not) et un nouvel écran (figure 6-5) s'affiche pour éditer cette condition.

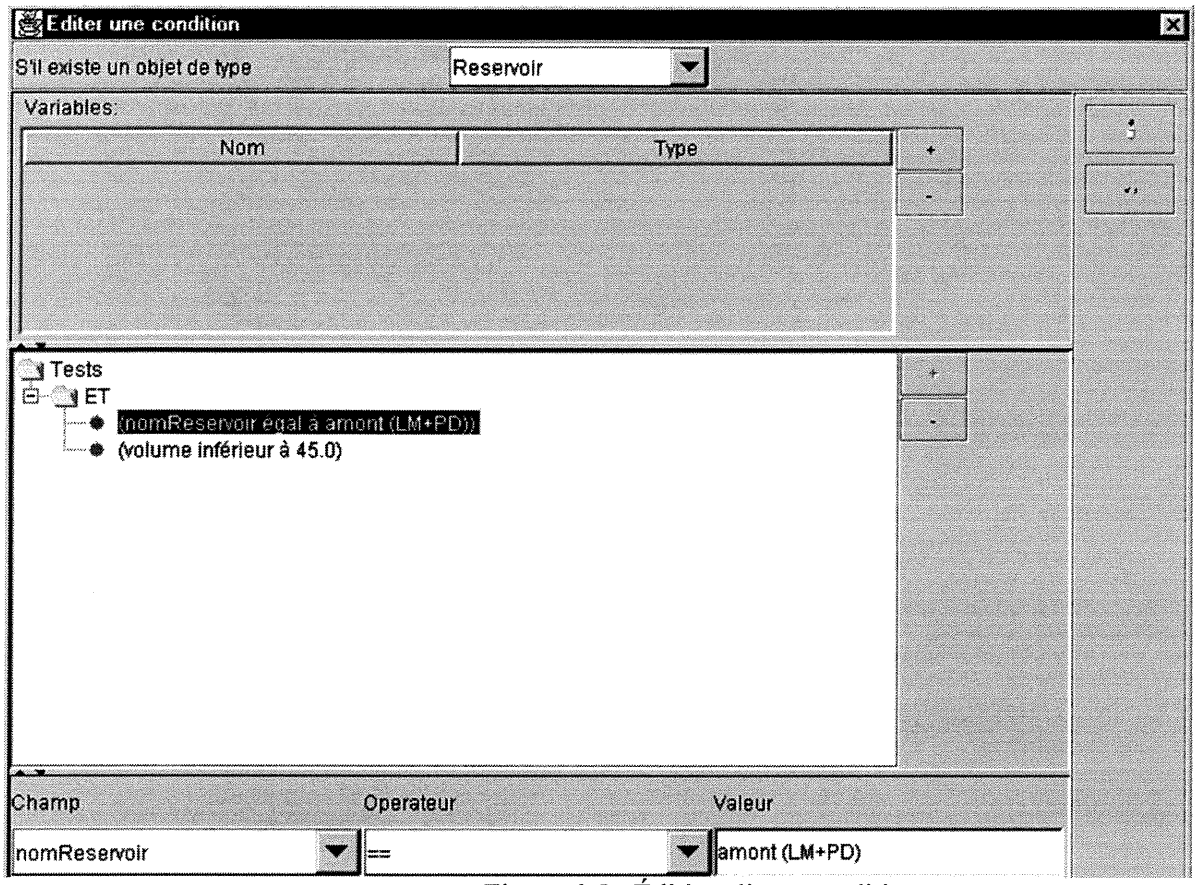


Figure 6-5 : Édition d'une condition

Au niveau de la partie variables, on peut déclarer des variables à utiliser dans la condition. Ensuite, on définit les tests de la condition en utilisant le bouton «+» au niveau Tests. Quatre types de tests sont disponibles :

- ET : deux tests binaires connectés par le ET
- OU : deux tests binaires connectés par le OU
- NOT : la négation d'un test binaire
- Test Simple : un seul test binaire.

Ensuite on définit les actions d'une règle. On peut choisir entre les types d'actions suivants :

- Action Assert : pour ajouter un nouvel objet dans la mémoire de travail
- Action Modify : pour modifier un objet existant
- Action Retract : pour retirer un objet de la mémoire de travail.

Pour notre règle de l'exemple, on doit ajouter une action Assert pour insérer un objet Coupure dans lequel on définit la quantité et la période. À l'aide de l'écran de la figure 6-6 on ajoute cette action.

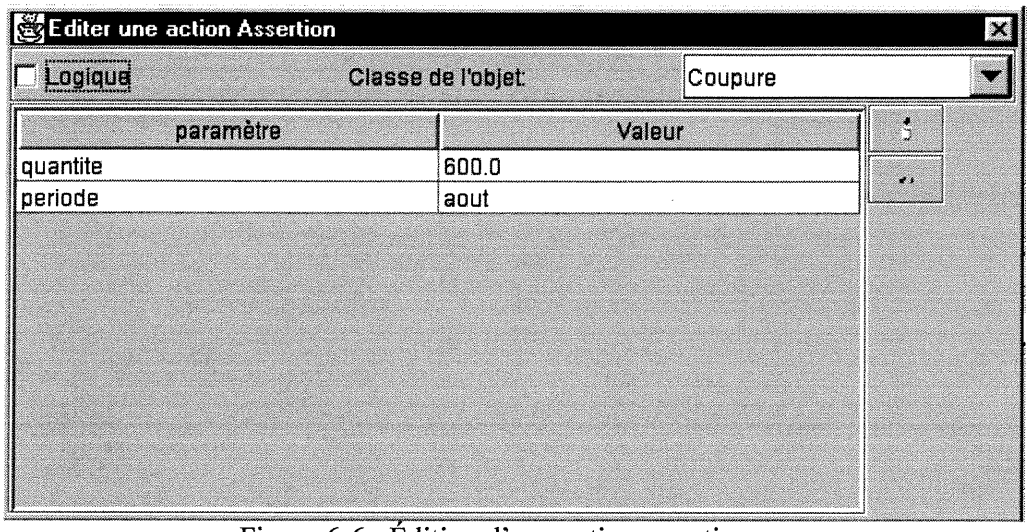


Figure 6-6 : Édition d'une action assertion

De la même façon, l'action Modify nous permet de modifier un objet en affichant la liste de ses champs pour changer leurs valeurs.

Pour l'action Retract, une liste d'objets disponible au niveau de la règle est affichée. On choisit l'objet à retirer au niveau de cette règle. L'écran de la figure 6-7 est utilisé pour ajouter ou modifier une règle Retract.

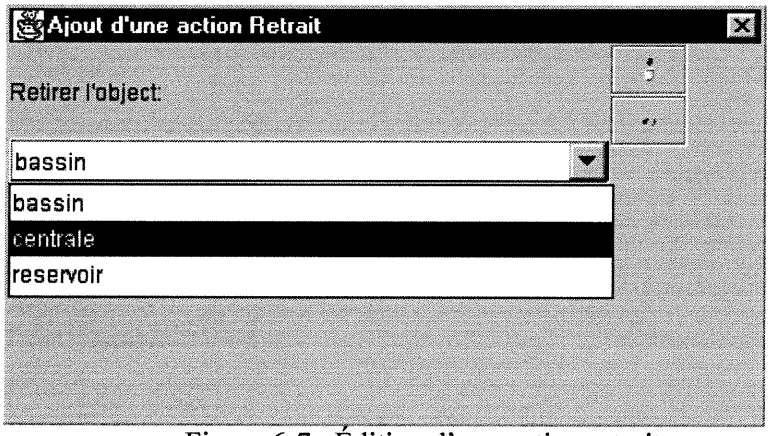


Figure 6-7 : Édition d'une action retrait

Une fois la règle éditée, on peut la valider en cliquant sur le bouton enregistrer. Les contrôles suivants sont prévus pour chaque règle :

- La priorité d'une règle est un entier
- Chaque règle doit contenir au moins une condition et 0 ou plusieurs actions.

Nous travaillons actuellement sur le développement d'une deuxième version de l'éditeur de règles apportant les améliorations suivantes :

- Permettre les appels de méthodes dans les conditions et les actions
- Gérer des expressions plus complexes dans la définition des règles
- Développer une couche intermédiaire comme abstraction de la grammaire de Jrules en utilisant un langage simplifié. Ce langage réduit la complexité de la grammaire Jrules afin de permettre à l'utilisateur de se concentrer sur la sémantique de ses règles
- Édition tabulaire de certaines règles ; ces règles se différencient uniquement par les valeurs des différents éléments sur les-quels portent les conditions et donc une édition tabulaire (règles en lignes, conditions et actions en colonnes) de ce genre de règles est plus efficace.

6.3.1.2 Éditeur d'objets

Il s'agit d'offrir aux experts la possibilité de gérer les objets réels du système pour refléter l'état courant du domaine. Il agit comme un configurateur du système. Le prototype de la figure 6-8 a été réalisé pour offrir la possibilité de gérer le système hydrique d'ALCAN :

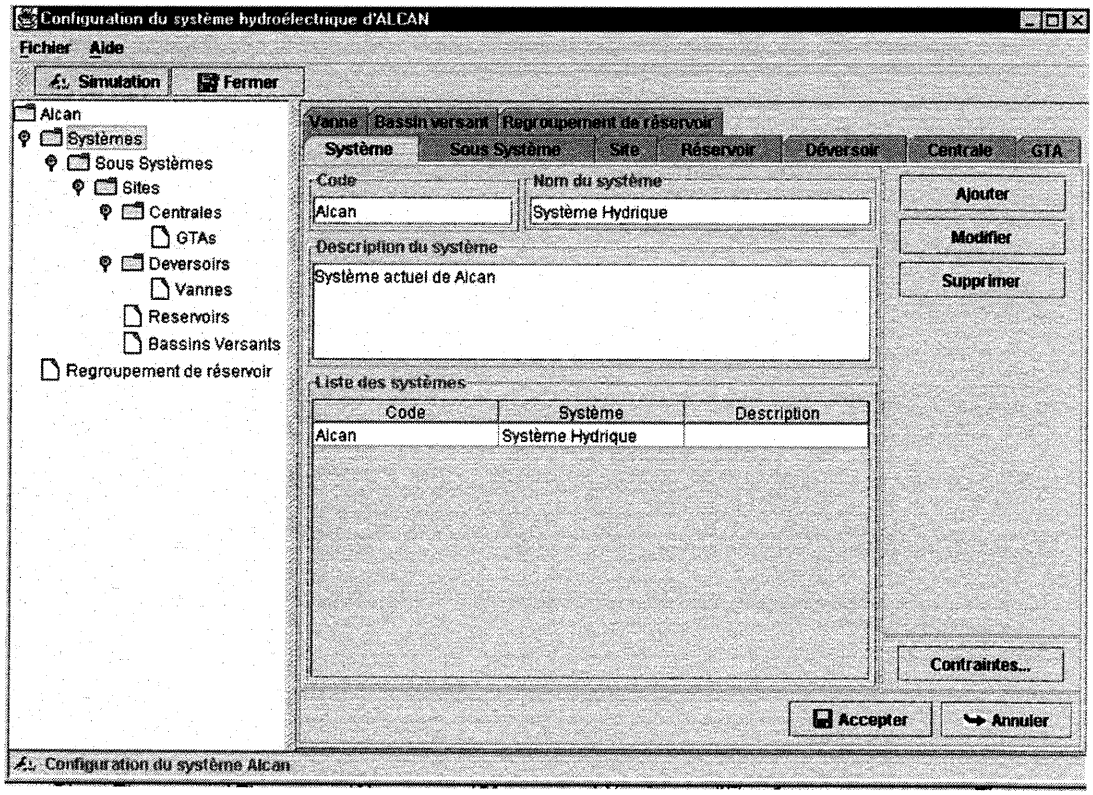


Figure 6-8 : Éditeur d'objets de l'environnement PReCI

6.3.2 Module de Vérification et Validation

Après avoir modélisé la base de connaissances par un réseau de Petri coloré, nous avons essayé d'exploiter des outils disponibles pour implanter notre modèle de vérification et validation. Les tests ont porté sur l'outil suivant :

- Renew [29] : c'est un outil développé au sein du groupe sur les systèmes distribués du département de l'informatique à l'Université de Hambourg en Allemagne. Il s'agit d'un éditeur graphique qui permet de construire un réseau de Petri coloré modélisant un système donné. Il utilise Java comme langage d'inscription; les expressions définies sur les arcs du réseau sont des expressions Java. Renew dispose d'un simulateur qui permet de simuler le réseau à partir d'un marquage initial mais il ne permet pas d'obtenir le graphe d'occurrence.

D'autres outils de simulation par réseau de Petri sont disponibles [32]. Leur utilisation dans notre projet suppose que l'on veuille introduire manuellement notre modèle en utilisant leurs éditeurs graphiques. Or, dans notre cas on dispose d'une base de connaissances déjà existante que l'on doit transformer automatiquement en un réseau de Petri pour l'étudier. Un autre obstacle majeur est que la plupart des outils disponibles proposent leur propre langage d'inscription (par exemple Java pour Renew) ou implémentent des expressions selon le langage d'inscription standard des réseaux de Petri [24]. Dans notre modélisation, les expressions sur les arcs sont dérivées à partir des conditions et des actions des règles de Jrules. Leur syntaxe est particulière bien que leur définition respecte un langage d'inscription; leur évaluation produit des objets (jetons) en entrée et en sortie. Ceci rend l'utilisation des outils disponibles inadéquate. Dans notre modélisation, les couleurs sont des classes et les jetons sont des objets. Jrules dispose d'un moteur d'inférence qu'on peut utiliser pour faire de la simulation. Tous les éléments sont disponibles pour implanter notre propre modèle de réseau de Petri à base de Java et Jrules. Nous proposons le diagramme de classes de la figure 6-9 dans lequel nous modélisons et nous simulons une base de règles avec un réseau de Petri coloré.

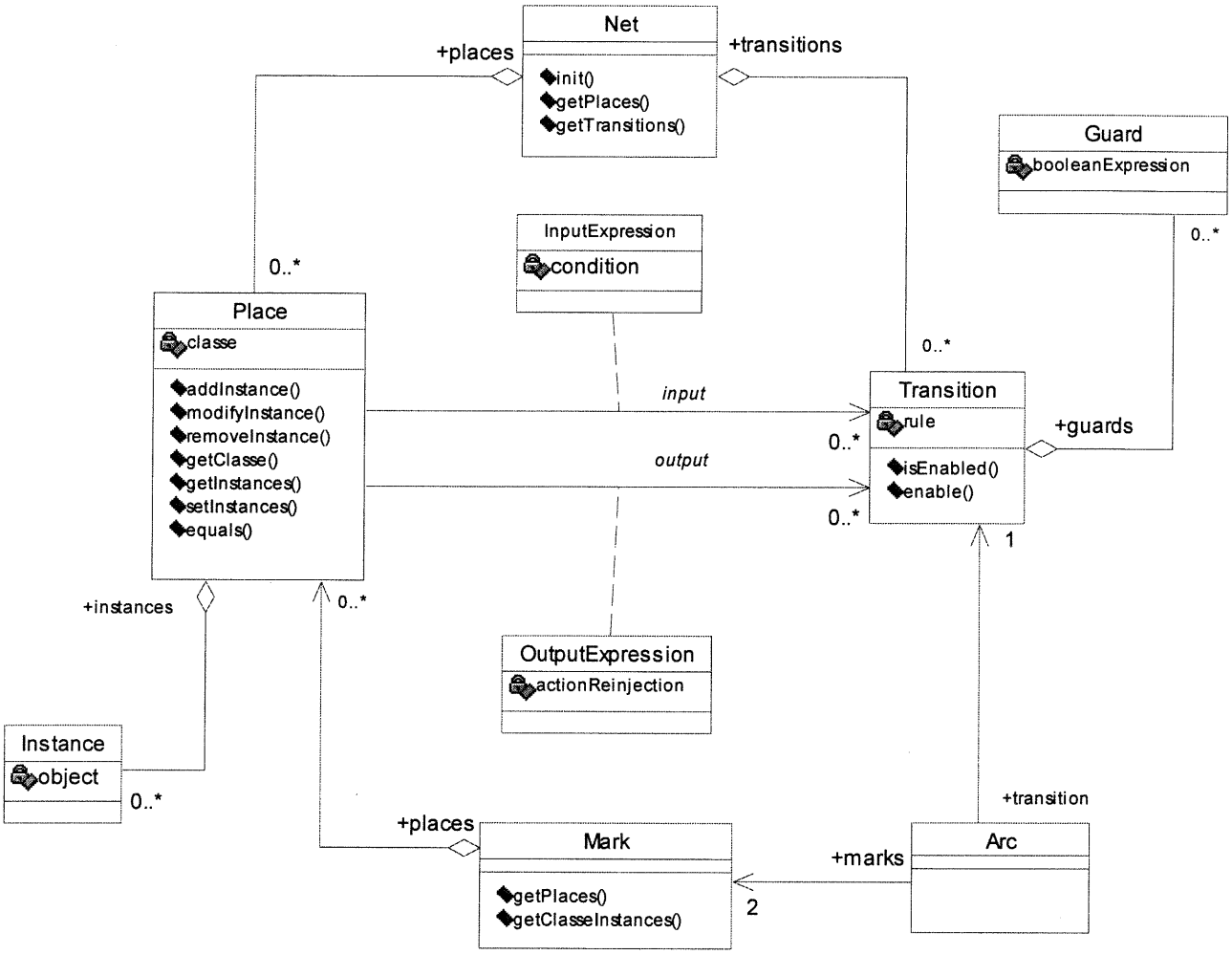


Figure 6-9 : Modèle de représentation d'une BR hybride par un réseau de Petri coloré

Les étapes suivantes permettent de construire notre modèle :

1. Chercher le fichier de la base de règles
2. Chaque règle de la base correspond à une instance de la classe Transition
3. Identifier les classes utilisées dans la base de règles, chaque classe correspond à une instance de la classe Place
4. En examinant les conditions et les actions des règles, on construit les arcs d'input et d'output de chaque transition ainsi que leurs fonctions de gardes. Ces éléments sont représentés par des instances des classes InputExpression, OutputExpression et Guard
5. À toute classe (Place) on lui associe ses instances, chacune des ces instances est représentée par la classe Instance.

Cette transformation de la base de règles en un réseau de Petri est dynamique. Après avoir parcouru cette base, les composantes de chaque règle sont représentées par des instances des classes de notre modèle.

Après avoir construit le réseau, on doit l'exécuter pour chercher le graphe d'occurrence. On utilise la classe Mark pour capter l'état du système avant (marquage initial) et après (marquage final) chaque activation d'une transition. La classe Arc est utilisée pour construire le graphe; un arc est constitué d'un marquage de départ, d'une transition et d'un marquage final.

Tout le modèle est codé en Java. Nous avons généré les classes nécessaires à partir du diagramme de classes de la figure 6-9, et nous avons implanté l'algorithme de construction du GO en utilisant les structures de données Java (Hashset, Vector et ArrayList). La recherche dans les marquages du GO est basée sur la comparaison de leurs objets (jetons). Chaque classe de la base de connaissances redéfinit les deux méthodes *equals()* et *compareTo()* héritées de la classe *Object* pour comparer ses objets.

Une base de règles Jrules est organisée sous formes de paquets. Un paquet contient plusieurs règles ayant un lien sémantique entre elles. Jrules offre une API qui permet d'analyser la base de règles et l'organiser en paquets. La vérification et la validation peuvent porter sur toute la base de règles ou une partie (un ou plusieurs paquets). Nous utilisons l'écran de la figure 6-10 pour démarrer une session de V&V :

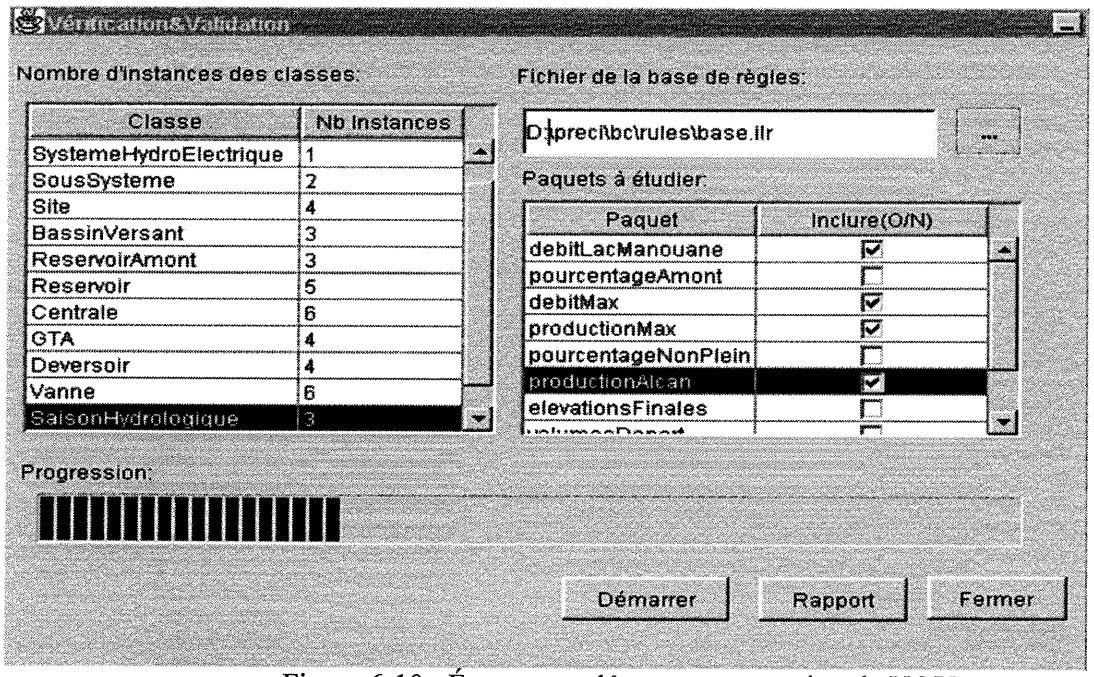


Figure 6-10 : Écran pour démarrer une session de V&V

6.4 Conclusion

Mis à part l'éditeur d'objets, qui est spécifique au projet PReCI en offrant les interfaces nécessaires pour éditer les objets propres au système hydrique d'ALCAN, les autres modules du gestionnaire ne sont pas spécifiques à un domaine d'application donné. Ils peuvent être réutilisables dans d'autres systèmes qui utilisent le même formalisme objet-règle. Pour l'éditeur de règles, il suffit de spécifier le chemin dans lequel se trouvent les classes Java compilées qui servent comme connaissances statiques du domaine, afin de définir les connaissances dynamiques de celui-ci (règles). Le modèle de représentation de la base de règles par un réseau de Petri est aussi facilement réutilisable pour tout langage objet-règle non monotone. Il suffit de développer un transformateur pour chaque langage que l'on veut étudier vers ce modèle. Le reste des traitements (construction du GO, vérification et validation) restent les mêmes en utilisant le moteur d'inférence du langage à étudier comme simulateur.

Conclusion Générale

Notre implication dans le projet PReCI avait pour but de proposer une architecture pour le gestionnaire de connaissances et d'implanter les modules de vérification et validation en s'inspirant des travaux déjà réalisés. Notre choix du formalisme de représentation de connaissances (objet-règle) a rendu cette tâche très difficile. Après avoir fait un parcours de plusieurs travaux sur la V&V, nous n'avons trouvé aucun travail qui porte sur la V&V des systèmes hybrides objets-règles. L'une des dernières publications sur la V&V considère le sujet comme un nouveau champ pour des futures investigations [27].

L'utilisation du formalisme hybride pour représenter les connaissances a permis une bonne structuration, mais elle a rendu les tâches de V&V plus difficiles. Les règles sont devenues plus complexes. Une analyse statique de ces règles est pratiquement impossible. Nous avons proposé un modèle de V&V utilisant une approche dynamique basée sur les réseaux de Petri colorés. Ainsi, nos contributions majeures sont:

- Nous avons proposé une architecture pour un gestionnaire de connaissances. En plus de la V&V, il permet aux experts l'accès à la BC pour effectuer tout changement nécessaire sur la base d'objets et la base de règles. Ce module est aussi utilisé suite à des sessions de V&V pour corriger les erreurs et les anomalies. Nous avons opté pour une approche d'intervention manuelle de l'expert, elle a l'avantage de le laisser seul responsable des actions à effectuer. Puisque il connaît bien le domaine, il est capable de fournir les bons changements à effectuer sur la base.
- L'utilisation d'UML dans la conception et la modélisation des connaissances nous a permis d'exploiter les mécanismes de ce langage pour dériver toutes les informations nécessaires pour la V&V. Nous avons proposé le design pattern *state* avec des extensions à l'aide d'OCL comme un moyen uniforme pour représenter les états et le modèle de cohérence. Tout système hybride objet-règle peut exploiter le même mécanisme pour définir ses états et son modèle de cohérence. Cette technique est indépendante des langages d'implantation utilisés (Jrules et Java dans notre exemple).

- En s'inspirant du modèle des réseaux de Petri et en utilisant une modélisation objet, nous avons proposé un modèle de transformation de la base de connaissances en un réseau de Petri coloré pour des besoins de V&V. Dans ce modèle, tous les composants de la BC (classes, règles, conditions, actions et objets) sont représentés par des éléments de modélisation des réseaux de Petri (places, transitions, expressions des arcs d'entrée, expressions des arcs de sortie et jetons).
- Nous avons exploité une des techniques d'analyse des réseaux de Petri, à savoir la construction des graphes d'occurrence, pour effectuer la V&V en utilisant une approche dynamique basée sur la simulation de la BC. L'analyse des marquages du graphe d'occurrence nous a permis de détecter les situations d'erreurs et d'anomalies de la BC.

Des extensions futures sont envisageables pour notre travail, il s'agit de :

- Faire des expérimentations sur des bases de connaissances de grande taille. Avec la croissance d'utilisation du formalisme objet-règle, plusieurs bases de règles de différentes tailles deviennent disponibles. Elles peuvent faire l'objet de différentes expérimentations.
- Porter notre travail sur d'autres langages objets-règles. Nous avons utilisé Jrules pour effectuer les différentes expérimentations dans le cadre du projet PReCI. Nous pouvons réutiliser les mêmes techniques pour la V&V avec d'autres langages similaires à Jrules (rules, OPS5, etc.)
- Utiliser des techniques plus avancées pour la génération des marquages initiaux à partir des états du système. En génie logiciel, plusieurs travaux ont été réalisés dans ce sens (State based testing) [33].

Bibliographie

[1] A. D. Preece et R. Shinghal. (Page consultée le 15 janvier 2000). *Foundation and Application of Knowledge Base Verification,* [En ligne].
http://www.csd.abdn.ac.uk/~apreece/Pubs/IJIS94.html.

[2] A. D. Preece, A. Batarekh et R Shinghal. *Verifying Rule-Based Systems.* Rapport pour Bell Canada, Centre for Pattern Recognition and Machine Intelligence, Université de Concordia, Montréal, Canada, Août 1990.

[3] A. D. Preece. (Page consultée le 15 janvier 2000). *Methods for Verifying Expert System Knowledge Bases* , [En ligne]. http://www.csd.abdn.ac.uk/~apreece/Pubs/Preece91.html.

[4] T. A. Nguyen, W. A. Perkins, T. J. Laffey et D. Pecora. *Checking an Expert Systems Knowledge Bases for Consistency and Completeness.* Proceeding of the Ninth International Joint Conference on Artificial Intelligence (IJCAI 85), 1985, Los Angeles, Californie, États-Unis, Vol. 1, pages 375-378.

[5] T. A. Nguyen, W. A. Perkins, T. J. Laffey et D. Pecora.*Knowledge Base Verification.* AI Magazine, 1987, Vol. 8, No. 2 , pages 69-75.

[6] W. A. Perkins, T. J. Laffey, D. Pecora et T. A. Nguyen. *Knowledge Base Verification.* chapitre dans Topics in Expert System Design, Elsevier Science Publisher B.V., North Holland, 1989, pages 353-376.

[7] M. Suwa, A. C. Scott et E. H. Shortliffe *An Approach to Verifynig Completeness and Consistency in Rule-Based Expert System.* AI Magazine, 1982, Vol 3, No. 4, pages 16-21.

[8] C. L. Chang, J. B. Combs et R. A. Stachowitz.*A Report on the Expert Systems Validation Associate(EVA).* Expert Systems with Applications, 1990, Vol. 1, pages 217-230.

[9] R. A. Stachowitz, J. B. Combs et C. L. Chang. *Validation of Knowledge-Based Systems.* Second AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program, Mars 9-11, 1987, Arlington, Virginia, États-Unis, pages 1-10.

[10] R. A. Stachowitz, C. L. Chang, T. S. Stock et J. B. Combs.*Building Validation Tools for Knowledge-Based Systems.* SOAR87, 1st Annual Workshop on Space Operations, Automation, and Robotics, NASA/JSC, Aout 1987, Houston, États-Unis, pages 209-216.

[11] R. A. Stachowitz et J. B. Combs. *Validation of Expert Systems.* Proceeding of the Twentieth Annual Hawaii International Conference on System Sciences, 6-9 janvier 1987, Kailua-Kona the Big Island, Hawaii, États-Unis, Vol 1, pages 686-695.

[12] A. D. Preece. *Verification of Rule-Based Expert Systems in Wide Domains.* Research and Development in Expert Systems VI : Proc. Expert Systems 89, British Computer Society Specialist Group on Expert Systems, september 20-22, 1989, London, Angleterre, pages 66-77.

[13] A. D. Preece. *Towards a methodology for evaluating expert systems.* Expert Systems, november 1990, Vol. 7, No. 4, pages 215-223.

[14] A. Ginsberg. *Knowledge-Base Reduction: A new Approach to Checking Knowledge Bases for Inconsistency & Redondancy.* AAAI 98, the Seventh National Conference on Artificial Intelligence, Aout 21-26, 1988, Saint Paul, Minnesota, États-Unis, pages 585-589.

[15] B. J. Gragun et H. J. Steudel. *A decision-table-based processor for checking completeness and consistency in rule-based expert systems.* International Journal Man-Machine Studies, 1987, Vol. 26, pages 633-648.

[16] A. D. Preece, R. Shinghal et A. Batarekh. *Verifying Expert Systems: A Logical Framework and Practical Tool.* Expert Systems with Applications, 1992, Vol. 5, pages 421-436.

[17] M. Ayel et M. C. Rousset, *La cohérence dans les bases de connaissances.* Toulouse, France, Éditions Cepadues, 1990. 106 pages.

[18] N. Zlatareva et A. D. Preece. *State of the Art in Automated Validation of Knowledge-Based Systems.* Expert Systems with Applications, 1994, Vol.7, No. 2, pages 151-167.

[19] M. Ayel et L. Vignollet. *SYCOJET and SACCO, Two Tools for Verifying Expert Systems.* International Journal of Expert Systems, 1993, Vol. 6, No. 3, pages 357-382.

[20] D. L. Nazareth. *Investigating the Applicability of Petri Nets for Rule-Based System Verification.* IEEE Transactions on Knowledge and Data Engineering, Juin 1993, Vol. 4, No. 3, pages 402-415.

[21] E. Gamma, R., R. Johnson et J. Vlissides *Design Patterns : Elements of Reusable Object-Oriented Software*. Boston, États-Unis, Addison-Wesley, 1995.395 pages.

[22] J. Warmer et A. Kleepe. *The Object Constraint Language: Precise Modelling with UML.* Reading, Massachusetts, États-Unis, Addison-Wesley, 1998. 112 pages.

[23] J. L. Peterson. *Petri Nets Theory and the Modelling of Systems*. Englewood Cliffs, New Jersey, États-Unis, Prentice-Hall, 1981. 290 pages.

[24] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use*, Volume 1. Berlin, Allemagne, Springer-Verlag, 1992. 234 pages.

[25] H. Knublauch, M. Sedlmayr et T. Rose. (Page consultée le 5 octobre 2000) *Design Pattern for the implementation of constraints on JavaBeans*, [En ligne]. http://www.netobjectdays.org/node00/de/Conf/publish/talks.html#Design_Patterns_for_.t

[26] D. Green. (Page consultée le 21 septembre 2000). *The Reflection API*, [En ligne].http://www.javasoft.com/docs/books/tutorial/reflect/index.html.

[27] A. Vermesan et F. Coenen. *Validation and Verification of Knowledge Based Systems : Theory, Tools and Practice*. Dordrecht, Pays-Bas, Kluwer Academic Publishers, 1999. 364 pages.

[28] H. Lounis. *Vérifier, évaluer et réviser des systèmes à base de connaissances: contribution de l'apprentissage automatique*. 1994. 187 pages. Thèse de doctorat en sciences, Université Paris XI, Orsay, France.

[29] F. Lackinger et S. Schlee. (Page consultée le 13 février 2000). *The ViVa Framework for V&V of KBS,* [En ligne]. http://spd-web.terma.com/Projects/ViVa/intro.html#References.

[30]. Compagnie Ilog. (Page consultée le 5 décembre 1999). *JRules,* [En ligne]. http://www.ilog.com/products/rules/.

[31]. Renew. (Page consultée le 11 juin 2000). *Renew tool,* [En ligne]. http://www.renew.de/.

[32]. International Petri Nets community. (Page consultée le 25 mai 2000). *Welcome to the Petri Nets World,* [En ligne]. http://www.daimi.au.dk/PetriNets/.

[33] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima and C. Chen. *On Object State Testing.* Eighteenth Annual International Computer Software & Applications Conference, IEEE Computer Society Press, 1993, Los Alamitos, Californie, États-Unis, pages 222-227.

ANNEXE A

Classe Fib en Java

```
package fib;
import java.io.*;

public class Fib
{
    public int number = 0;
    public long value = 0;

    public Fib(int number)
      {
        this.number = number;
      }
};
```

ANNEXE B
Règles de calcul de la fonction Fibonacci

```
// on doit importer la classe Fib défini dans le package fib
import fib.*;

rule calculateFib1
{
    when
      {
        ?f: fib.Fib(number==1; value==0);
      }
    then
      {
        modify ?f
          {
            value=1;
          }
      }
};


rule calculateFib2
{
    when
      {
        ?f: fib.Fib(number==2; value==0);
      }
    then
      {
        modify ?f
          {
            value=3;
          }
      }
};


rule makeRecursiveGoal
{
    when
      {
        fib.Fib(?n:number; ?n>1; value==0);
        not fib.Fib(number==?n-1);
      }
    then
      {
        assert Fib(?n-1);
      }
};

rule computeValue
{
    when
      {
        ?f: fib.Fib(?n:number; ?n>2; value==0);
        ?f1: fib.Fib(number==?n-1; value!=0);
        ?f2: fib.Fib(number==?n-2; value!=0);
      }
```

```
    then
      {
        modify ?f
          {
            value=?f1.value + ?f2.value;
          }
      }
};
```

ANNEXE C
Grammaire de Jrules

Jrules permet de définir un groupe de règles dans une unité d'entrée qui peut être soit un fichier sur disque, une chaîne de caractères, ou un flux de données. La définition BNF d'une unité d'entrée est :

*RulesetDefinition::= (ImportDefinition)**
*(RuleDefinition)**

La clause *ImportDefinition* permet d'établir le lien avec les classes Java qui vont être utilisées dans les différentes règles de l'unité. On doit faire l'import de tout paquet contenant la définition de ces classes.

La clause RuleDefinition est la partie essentielle d'une unité d'entrée, elle permet de définir l'ensemble de règles de l'unité. Sa grammaire est la suivante :

RuleDefinition::= "rule" Identifier "{"
RuleParameters
RuleConditions
RuleActions "};"

Identifier est l'identificateur de la règle c'est à dire son nom. Au niveau de chaque règle on peut définir deux paramètres, le premier permet de spécifier le paquet de la règle. Un paquet de règles est une organisation logique de la base de règles qui contient des règles ayant des liens entre elles. On peut regrouper des règles qui, ensemble dans une session d'inférence, peuvent être déclenchées pour résoudre une partie ou tout un problème. Le deuxième paramètre définit la priorité d'une règle, c'est à dire un entier signé (type *int* de Java). Ce paramètre est exploité par le moteur d'inférence Jrules dans le cas où il y a plusieurs règles tangibles au déclenchement, et dans ce cas il choisira la règle de plus haute priorité.

Au niveau de *RuleConditions* on définit les conditions de la règle, sa syntaxe est la suivante :

RuleConditions ::= "when" "{" (RuleCondition)+ "}"

RuleCondition ::= Condition Simple | Condition NOT | Condition Exists

Il existe trois type de conditions :

- Condition simple : elle filtre les objets de la mémoire de travail correspondant à la classe sur laquelle porte la condition et qui vérifient les tests de celle-ci. Sa syntaxe est la suivante :

Condition Simple ::= (<VARIABLE>":")? ClassCondition ";"

Exemple : ?x: Number(intValue() >20);

?i : java.lang.String(?this. equals("Jrules"));

La variable est instantiée autant de fois qu'il y a d'objets qui vérifient la condition.

- Condition *Not* : une condition *Not* est vraie s'il n'y a aucun objet d'un classe donnée dans la mémoire de travail qui vérifie la condition, sa syntaxe est la suivante:

  NotCondition ::= "not" ClassCondition ";"

  Exemple : not Number(?n : intValue(); ?n > 20 && ?n < 30);

  not java.lang.String(?this.equals("Jrules"));

- Condition *Exists* : une condition *Exists* est vraie s'il existe au moins un objet d'une classe donnée dans la mémoire de travail qui vérifie la condition, sa syntaxe est :

  ExistsCondition ::= "exists" ClassCondition ";"

  Exemple : exists Number(?n : intValue(); ?n > 20 && ?n < 30);

  exists java.lang.String(?this.equals("Jrules"));

La *ClassCondition* définit des tests qui portent sur une clase Java, sa syntaxe est la suivante :

ClassCondition ::= ClassName "(" (TestAssignmentSuite)? ")"

TestAssignmentSuite permet de définir les différents types de tests pour chaque condition. Comme Jrules utilise Java comme langage hôte, les tests peuvent contenir toute expression Java correcte incluant des expressions booléennes, des appels de méthodes, des surcharges, etc. Ces expressions sont séparées par des points virgules comme opérateur ET. Une expression peut être aussi composée de deux ou plusieurs autres expressions séparées par && pour le ET ou || pour le OU.

Au niveau de *RuleActions*, on définit les actions de chaque règle. Cette partie est définie par:

RuleActions ::= "then" "{" (RhsStatement) "}"*

RhsStatement ::= AssertAction | RetractAction | UpdateAction

| ModifyAction | ApplyAction

On trouve les types d'actions suivantes :

- Assertion : une action assertion construit un nouvel objet d'une classe donnée et l'insère dans la mémoire de travail. Sa syntaxe est la suivante :

AssertAction ::= *"assert" ("logical")?*
ClassName (Arguments)?
(("{" (ExecutableStatement) "}" (";")?) | ";")*

Un des constructeurs de la classe va être choisi pour créer l'objet selon les arguments qui suivent le nom de la classe. On peut aussi appeler du code (clause *ExecutableStatement*), ceci s'exécute juste avant l'insertion de l'objet dans la mémoire de travail. La clause *logical* permet de faire une assertion logique de l'objet, l'objet existe déjà il ne va pas être réinséré une deuxième fois, il a une existence unique dans la mémoire de travail et la règle qui le référencie est responsable de sa validité.

Exemple :
```
assert Integer(12);
assert Form()
{
color = Red;
shape = Circle;
}
```

- Retrait : une action retrait enlève un objet donné de la mémoire de travail. Sa syntaxe est la suivante :

  RetractAction ::= "retract" PrimaryExpression ";"

  La clause *PrimaryExpression* définit un objet quelconque qui peut se trouver dans la mémoire de travail.

Exemple :
```
rule RemoveDisplay {
When {
 ?c: Computer();
}
then {
 retract ?c;
}};
```

- Modification : elle inclut les variantes suivantes :
  - L'action *Update* : lorsque un objet est modifié par une autre application Java qui utilise les mêmes objets que ceux dans la mémoire de travail, Jrules doit être notifié de cette modification en utilisant l'action *update*:

    UpdateAction ::= "update" ("refresh")? PrimaryExpression ";"

    La clause *refresh* est utilisée pour permettre la réinsertion dans l'agenda des règles qui restent vraies et qui utilisent l'objet en question.

- L'action *Modify* : l'action *modify* diffère légèrement de l'action *update* par le fait que les modifications à apporter sur un objet sont définies explicitement dans l'action elle même :

  ModifyAction ::= "modify" ("refresh")? PrimaryExpression "{" (ExecutableStatement)+ "}" (";")?

  Exemple : rule UpdateCounter {

  ```
  When {
   ?c: Counter();
  }
  then {
   modify ?c {
    value += 1;
   }}};
  ```

- L'action *apply* : au contraire des actions *update* et *modify*, l'action *apply* effectue seulement les changements définis dans celle-ci et ne prend pas en considération les règles utilisant l'objet (pas de *refresh*) :

  ApplyAction ::= "apply" PrimaryExpression "{" (ExecutableStatement)+ "}" (";")?

On donne l'exemple complet d'une règle Jrules qui calcule la fonction Fibonacci d'un nombre n supérieur à deux :

```
import fib.Fib;
rule computeValue {
    packet = fibonaci;
    priority = high;
    when  {
        ?f: Fib(?n:number; ?n>2; value==0);
        ?f1: Fib(number==?n-1; value!=0);
        ?f2: Fib(number==?n-2; value!=0);
    }
    then  {
        modify ?f  {
            value=?f1.value + ?f2.value;
        }
    }};
```

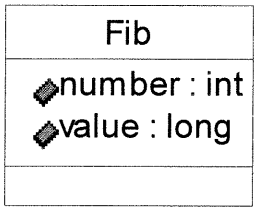
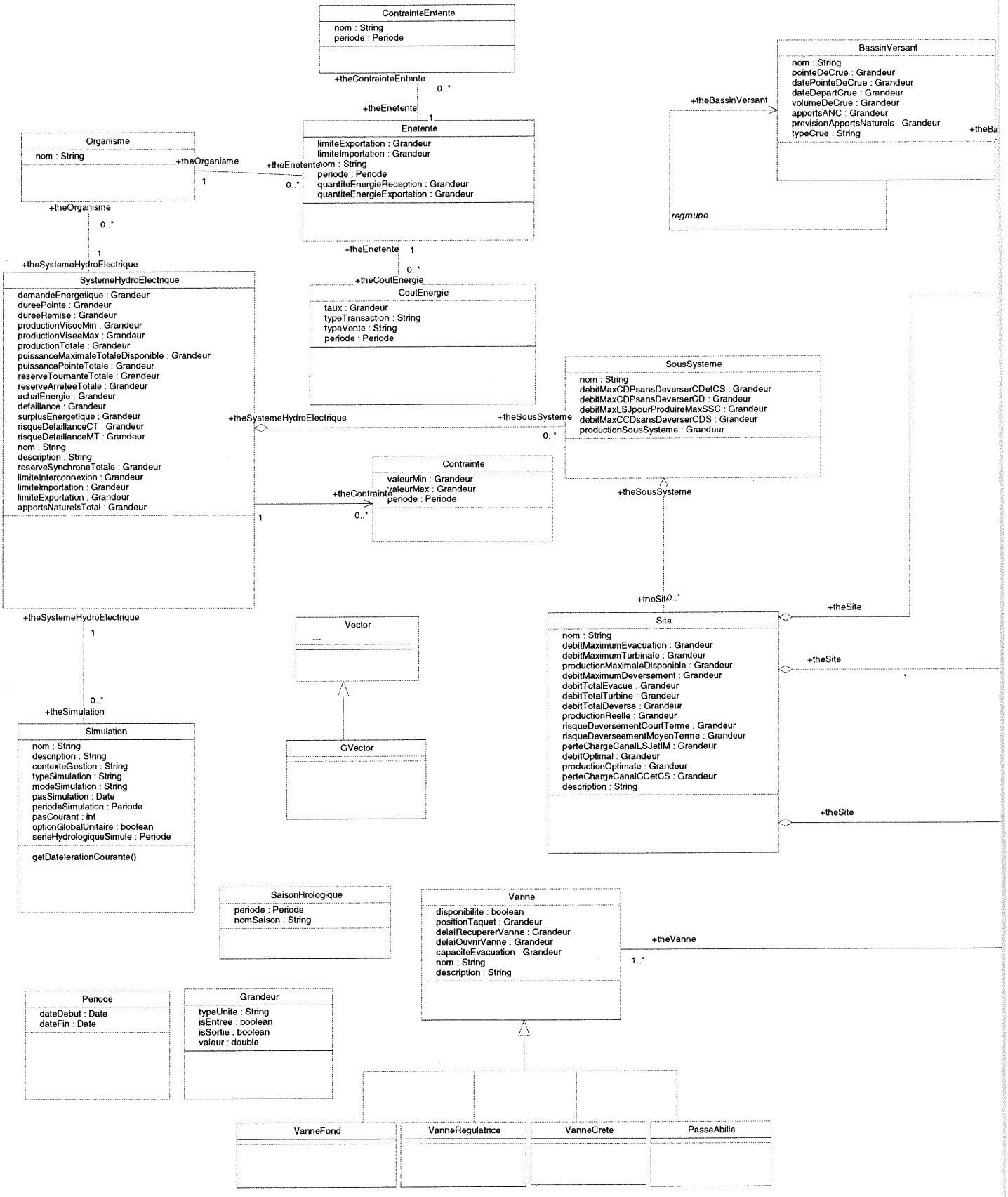


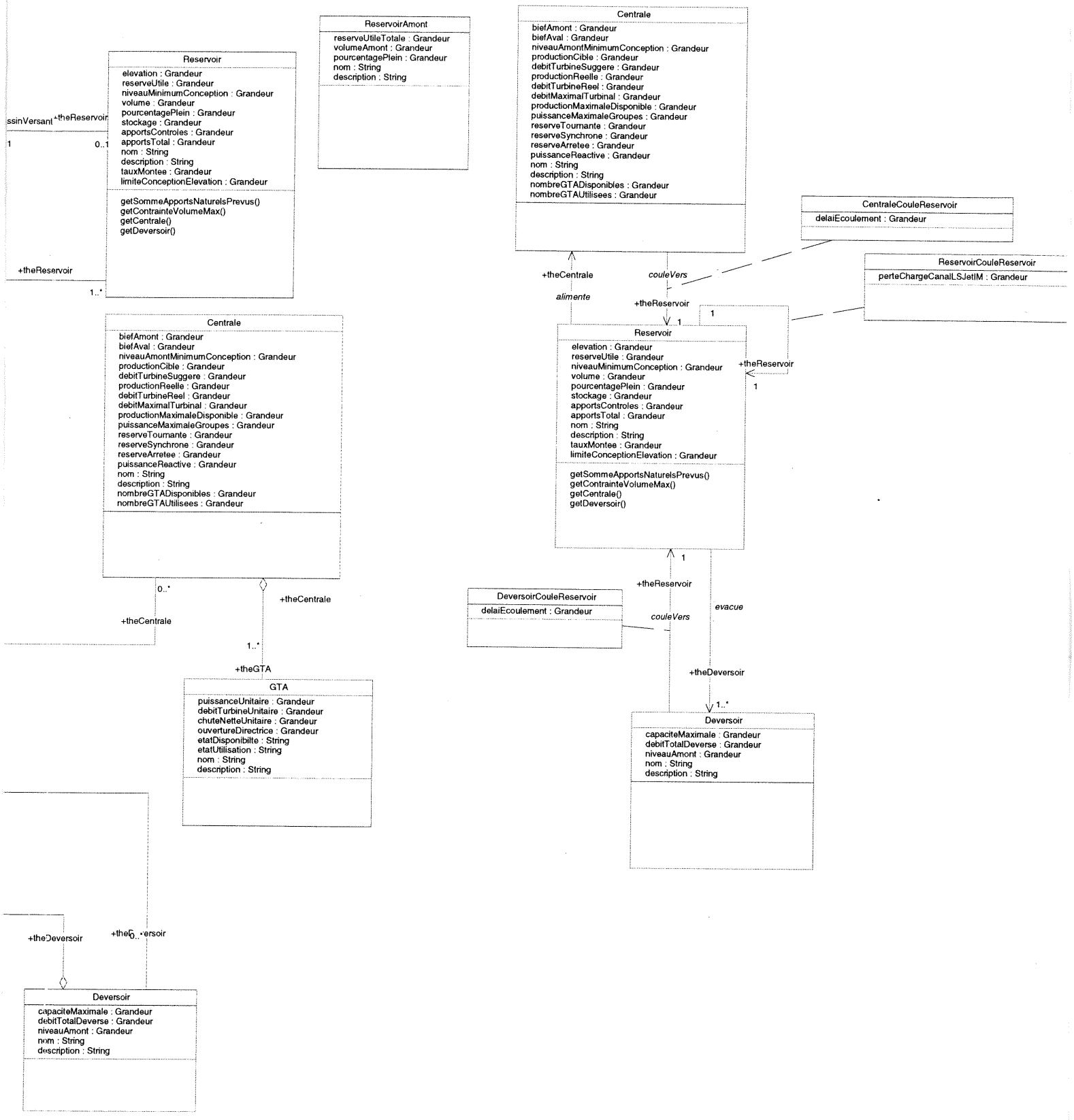
Schéma de la classe Fib

Pour le paramètre priorité, Jrules dispose aussi de quatre constantes prédéfinies (**maximum** = 1 000 000 000, **high** = 1 000 000, **low** = -1 000 000, **minimum** = -1 000 000 000).

ANNEXE D
Diagramme de classes du système hydrique d'ALCAN


ContrainteEntente
nom : String
periode : Periode
+theContrainteEntente
0..*
+theEnetente
1
BassinVersant
nom : String
pointeDeCrue : Grandeur
datePointeDeCrue : Grandeur
dateDepartCrue : Grandeur
volumeDeCrue : Grandeur
apportsANC : Grandeur
previsionApportsNaturels : Grandeur
typeCrue : String
+theBassinVersant
+theBa
regroupe
Organisme
nom : String
+theOrganisme
1
+theEnetente
0..*
Enetente
limiteExportation : Grandeur
limiteImportation : Grandeur
nom : String
periode : Periode
quantiteEnergieReception : Grandeur
quantiteEnergieExportation : Grandeur
+theOrganisme
0..*
1
+theSystemeHydroElectrique
+theEnetente
1
0..*
+theCoutEnergie
SystemeHydroElectrique
demandeEnergetique : Grandeur
dureePointe : Grandeur
dureeRemise : Grandeur
productionViseeMin : Grandeur
productionViseeMax : Grandeur
productionTotale : Grandeur
puissanceMaximaleTotaleDisponible : Grandeur
puissancePointeTotale : Grandeur
reserveTournanteTotale : Grandeur
reserveArreteeTotale : Grandeur
achatEnergie : Grandeur
defaillance : Grandeur
surplusEnergetique : Grandeur
risqueDefaillanceCT : Grandeur
risqueDefaillanceMT : Grandeur
nom : String
description : String
reserveSynchroneTotale : Grandeur
limiteInterconnexion : Grandeur
limiteImportation : Grandeur
limiteExportation : Grandeur
apportsNaturelsTotal : Grandeur
CoutEnergie
taux : Grandeur
typeTransaction : String
typeVente : String
periode : Periode
SousSysteme
nom : String
debitMaxCDPsansDeverserCDetCS : Grandeur
debitMaxCDPsansDeverserCD : Grandeur
debitMaxLSJpourProduireMaxSSC : Grandeur
debitMaxCCDsansDeverserCDS : Grandeur
productionSousSysteme : Grandeur
+theSystemeHydroElectrique
+theSousSysteme
0..*
Contrainte
valeurMin : Grandeur
valeurMax : Grandeur
periode : Periode
+theContrainte
1
0..*
+theSousSysteme
+theSystemeHydroElectrique
1
0..*
+theSimulation
+theSit
0..*
+theSite
Site
nom : String
debitMaximumEvacuation : Grandeur
debitMaximumTurbinale : Grandeur
productionMaximaleDisponible : Grandeur
debitMaximumDeversement : Grandeur
debitTotalEvacue : Grandeur
debitTotalTurbine : Grandeur
debitTotalDeverse : Grandeur
productionReelle : Grandeur
risqueDeversementCourtTerme : Grandeur
risqueDeverseementMoyenTerme : Grandeur
perteChargeCanalLSJetIM : Grandeur
debitOptimal : Grandeur
productionOptimale : Grandeur
perteChargeCanalCCetCS : Grandeur
description : String
+theSite
+theSite
Vector

GVector
Simulation
nom : String
description : String
contexteGestion : String
typeSimulation : String
modeSimulation : String
pasSimulation : Date
periodeSimulation : Periode
pasCourant : int
optionGlobalUnitaire : boolean
serieHydrologiqueSimule : Periode
getDateIterationCourante()
SaisonHrologique
periode : Periode
nomSaison : String
Vanne
disponibilite : boolean
positionTaquet : Grandeur
delaiRecupererVanne : Grandeur
delaiOuvrirVanne : Grandeur
capaciteEvacuation : Grandeur
nom : String
description : String
+theVanne
1..*
Periode
dateDebut : Date
dateFin : Date
Grandeur
typeUnite : String
isEntree : boolean
isSortie : boolean
valeur : double
VanneFond
VanneRegulatrice
VanneCrete
PasseAbille


ReservoirAmont
reserveUtileTotale : Grandeur
volumeAmont : Grandeur
pourcentagePlein : Grandeur
nom : String
description : String
Reservoir
elevation : Grandeur
reserveUtile : Grandeur
niveauMinimumConception : Grandeur
volume : Grandeur
pourcentagePlein : Grandeur
stockage : Grandeur
apportsControles : Grandeur
apportsTotal : Grandeur
nom : String
description : String
tauxMontee : Grandeur
limiteConceptionElevation : Grandeur
getSommeApportsNaturelsPrevus()
getContrainteVolumeMax()
getCentrale()
getDeversoir()
ssinVersant
+theReservoir
1
0..1
+theReservoir
1..*
Centrale
biefAmont : Grandeur
biefAval : Grandeur
niveauAmontMinimumConception : Grandeur
productionCible : Grandeur
debitTurbineSuggere : Grandeur
productionReelle : Grandeur
debitTurbineReel : Grandeur
debitMaximalTurbinal : Grandeur
productionMaximaleDisponible : Grandeur
puissanceMaximaleGroupes : Grandeur
reserveTournante : Grandeur
reserveSynchrone : Grandeur
reserveArretee : Grandeur
puissanceReactive : Grandeur
nom : String
description : String
nombreGTADisponibles : Grandeur
nombreGTAUtilisees : Grandeur
0..*
+theCentrale
+theCentrale
1..*
+theGTA
GTA
puissanceUnitaire : Grandeur
debitTurbineUnitaire : Grandeur
chuteNetteUnitaire : Grandeur
ouvertureDirectrice : Grandeur
etatDisponibilte : String
etatUtilisation : String
nom : String
description : String
+theDeversoir
+theD..ersoir
Deversoir
capaciteMaximale : Grandeur
debitTotalDeverse : Grandeur
niveauAmont : Grandeur
nom : String
description : String
Centrale
biefAmont : Grandeur
biefAval : Grandeur
niveauAmontMinimumConception : Grandeur
productionCible : Grandeur
debitTurbineSuggere : Grandeur
productionReelle : Grandeur
debitTurbineReel : Grandeur
debitMaximalTurbinal : Grandeur
productionMaximaleDisponible : Grandeur
puissanceMaximaleGroupes : Grandeur
reserveTournante : Grandeur
reserveSynchrone : Grandeur
reserveArretee : Grandeur
puissanceReactive : Grandeur
nom : String
description : String
nombreGTADisponibles : Grandeur
nombreGTAUtilisees : Grandeur
CentraleCouleReservoir
delaiEcoulement : Grandeur
ReservoirCouleReservoir
perteChargeCanalLSJetIM : Grandeur
+theCentrale
couleVers
alimente
+theReservoir
1..1
1
Reservoir
elevation : Grandeur
reserveUtile : Grandeur
niveauMinimumConception : Grandeur
volume : Grandeur
pourcentagePlein : Grandeur
stockage : Grandeur
apportsControles : Grandeur
apportsTotal : Grandeur
nom : String
description : String
tauxMontee : Grandeur
limiteConceptionElevation : Grandeur
getSommeApportsNaturelsPrevus()
getContrainteVolumeMax()
getCentrale()
getDeversoir()
+theReservoir
1
1
+theReservoir
DeversoirCouleReservoir
delaiEcoulement : Grandeur
couleVers
evacue
+theDeversoir
1..*
Deversoir
capaciteMaximale : Grandeur
debitTotalDeverse : Grandeur
niveauAmont : Grandeur
nom : String
description : String