DRISS DIRI

ÉLABORATION D'UN SYSTÈME DE MAINTIEN DE VÉRITÉ : une approche orientée objet

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en informatique
pour l'obtention du grade de maître ès science (M. Sc.)

DÉPARTEMENT D'INFORMATIQUE ET DE GÉNIE LOGICIEL
FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

2009

Résumé

Le but de ce mémoire est de présenter une approche orientée objet pour l'élaboration d'un Système de Maintien de Vérité à base de Justifications à Négation et Non Monotone (SMVJNNM). Un SMV est un module utilisé dans les systèmes à base de connaissances pour réviser des croyances. On distingue trois principaux types de SMV: à base de justifications, à base logique et à base d'assomptions. Ils utilisent des structures en réseau pour enregistrer les instances d'un ensemble de règles et tous s'inscrivent dans un paradigme orienté listes. Nous proposons un paradigme objet pour l'élaboration d'un SMV. Les étapes de la démarche suivie sont: étude des SMV existants, modélisation d'un SMV au niveau des connaissances, conception par patrons, implémentation et tests. Deux exemples tirés de la documentation scientifique montrent que notre système offre des fonctionnalités équivalentes à celles des SMV étudiés. Notre système a aussi été utilisé comme mini-moteur de recherche.

Abstract

The objective of this master's degree dissertation is to propose an object oriented approach for the design of negated non-monotonic justifications-based truth maintenance systems (NNMJTMS). A truth maintenance system (TMS) is a module assisting knowledge-based systems to conduct belief revision. There are three main types of TMS: justification-based, logical-based and assumption-based. All of these systems use network structures to register instances of a set of production rules according to a list-oriented paradigm. We propose in our work to adopt an object-oriented approach for the design of a TMS. We went through the following steps: review of existing TMS, modeling a TMS at the knowledge level, design and implementation using patterns and testing. To test the TMS in conjunction with a client system, two examples borrowed from scientific literature indicate that our system offers functionalities equivalent to those of the TMS found in the literature. In the first example, we validate some textbook cases. And in the second one, we test the load capacity of the TMS system while assisting a tiny search engine.

Avant-Propos

Comprendre le fonctionnement des SMV est une tâche complexe, l'expliquer l'est encore un peu plus. Modéliser et implémenter un tel système est un vrai défi. Je présuppose que le lecteur possède non seulement les notions de base de l'informatique mais aussi de solides connaissances de l'approche orientée objet et des patrons de conception *(design patterns)*. Le lecteur qui possède des connaissances dans une méthodologie de développement à objet peut facilement se familiariser ave la méthodologie CommonKADS qui fait appel au formalisme UML.

Je tiens à exprimer mes remerciements à madame Nicole Tourigny d'avoir accepté de diriger ce mémoire ainsi que madame Laurence Capus codirectrice. Je les remercie pour leurs contributions à ce projet.

Je tiens à remercier énormément aussi monsieur André Gamache et monsieur Luc Lamontagne qui ont accepté d'examiner ce mémoire.

Je remercie particulièrement monsieur Mario Marchand pour ses interventions et son aide et Monsieur Lamontagne pour ses corrections et ses commentaires constructifs.

Enfin, je remercie madame Sylvie Tremblay pour avoir corrigé l'orthographe des nombreuses ébauches que je lui ai soumises.

Table des matières

Liste des tableaux

Liste des figures

Chapitre 1 : Introduction

Ce mémoire porte sur le domaine de l'ingénierie des connaissances et plus particulièrement sur les systèmes de maintien de vérité (SMV). D'après Forbus et De Kleer (1993), un SMV est un module qui sert à maintenir des raisonnements effectués par un autre agent. Les SMV sont utilisés en révision des croyances en général comme nous allons le voir dans le paragraphe 2.3. En ingénierie des connaissances plus particulièrement, ils sont utilisés pour le maintien de la cohérence des systèmes à base de connaissances (Ayel et Rousset, 1990) et en tant que modules pour accélérer le fonctionnement des moteurs d'inférence (Forbus et De Kleer, 1993). Pour Doyle (p. 235, 1979), une croyance ou assomption[1] peut être tout ce qu'on accepte sans forcément avoir besoin d'une justification ou d'une raison. Schreiber *et al.* (p. 286-288, 2000) soulignent l'importance de systèmes pour maintenir la vérité ou pour garder des traces des raisonnements pour le raffinement des systèmes à base de connaissances.

L'ingénierie des connaissances a pour but le développement et la mise au point des SBC, appelés aussi systèmes experts (Ayel et Rousset, p. 5, 1990). Contrairement aux systèmes d'informations, les SBC conviennent surtout pour les domaines qui requièrent des connaissances complexes ou des connaissances difficiles à représenter de façon algorithmique (Preece, p. 1-2, 2001).

Le développement des SMV a connu son apogée au cours des années 80 et 90 comme en témoigne Martins (1990). Au cours de cette période, plusieurs algorithmes aussi complexes les uns que les autres ont été avancés dans la littérature. Par la suite, le champ d'application a été étendu à d'autres domaines, particulièrement en résolution des problèmes

[1] Assomption : Proposition admise à titre d'hypothèse en vue d'en démontrer une autre (Office de la langue française). http://www.granddictionnaire.com.

de satisfaction des contraintes (Jussien, 1997). Actuellement, les SMV sont utilisés dans des milieux universitaires et malgré nos recherches, nous n'avons pu trouver aucune application commerciale. Dans les milieux académiques, les SMV sont utilisés dans des champs spécialisés tels que le traitement de la langue naturelle comme on peut le voir avec les travaux du groupe de recherche *SNeRG*[2] (Johnson *et al.*, 1999-2000; Shapiro *et al*., 2007).

En génie logiciel, le développement de tout système requiert en général une méthodologie à suivre et il en existe plusieurs. En ingénierie des connaissances, diverses approches destinées pour développer des SBC existent. On peut citer entre autres : Generic Tasks, Role-Limiting Methods, MIKE, CommonKADS, etc. (Studer et *al.*, 1998; Fensel et Benjamins, 1998).

Dans le reste de ce chapitre, nous allons décrire brièvement ce qui a motivé notre intérêt pour ce domaine. Nous allons présenter la problématique, les objectifs poursuivis ainsi que les résultats obtenus.

1.1 Motivation

Notre premier contact avec les SMV s'est effectué dans le cadre du cours "Représentation des connaissances et modélisation" à l'Université Laval. Lors d'un travail sur la cohérence des SBC, travail effectué à partir des travaux d'Ayel et Rousset (1990) et d'Ayel et Laurent (1991), nous avons réalisé à quel point ce sujet peut être complexe et la cohérence difficile à définir. Ayel et Rousset (1990) la présentent indirectement en étudiant son contraire, l'incohérence. Par la suite ils étudient celle-ci, comme plusieurs autres d'ailleurs

[2] http://www.cse.buffalo.edu/sneps/

(Harmelen, 1998; Preece, 2001; Bellefeuille, 2001), d'un point de vue statique puis d'un point de vue dynamique. Au niveau statique, il existe parmi ces auteurs un large consensus quant à ce qui peut être qualifié d'incohérent. En général, on s'intéresse à un ensemble de règles qui doivent être exemptes d'un certain nombre d'anomalies[3] déterminées pour être déclarées cohérentes. Au niveau dynamique, la détection des anomalies s'effectue en cours d'exécution. Des outils spécialisés tels les SMV sont nécessaires pour accomplir cette tâche (Ayel et Rousset, 1989).

Une recherche exploratoire concernant ces systèmes nous a permis de réaliser que souvent, ce sont des algorithmes partiels qui sont proposés dans la littérature et que dans la plupart des cas, ces algorithmes portent sur la propagation d'états dans un réseau de nœuds.

En génie logiciel, nous savons qu'il existe des patrons de conception *(design patterns)* qui s'appliquent parfaitement à cette catégorie de problèmes, celle de la propagation d'événements dans un réseau. Les patrons de conception sont des solutions qui ont fait leurs preuves pour résoudre des problèmes récurrents (Gamma et *al.*, 1995). D'après Gamma et *al.* (1995), recourir à des patrons de conception lors de la résolution d'un problème présente l'avantage de mettre l'accent sur l'aspect conception plutôt que de le mettre sur le côté algorithmique. Ceci rejoint partiellement le choix qui a été effectué de considérer un SMV en tant qu'un SBC devant être modélisé avec la méthodologie CommonKADS. Cette stratégie a une implication très importante par la suite. En effectuant une revue de littérature, nous avons donc écarté tout ce qui a trait à l'aspect algorithmique et remonté aux sources pour étudier les SMV de base tels qu'ils étaient proposés initialement. Ces systèmes sont : le SMV à base de justifications de Doyle (1979), les SMV à base d'assomptions de De Kleer (1986) et le SMV à base logique de McAllester (1978). En effet, c'était le seul moyen qui permettait de comprendre le fonctionnement complet des SMV. L'étude d'algorithmes à partir d'articles ne nous a pas été très utile, notamment parce qu'ils sont trop partiels.

[3] Bellefeuille (2001) décrit en détail ce sujet qui dépasse le cadre de ce mémoire.

Les SMV sont en développement continu dans les milieux universitaires. En plus des travaux du groupe de recherche *SNePS* de l'Université d'état de New York à Buffalo, ils sont aussi enseignés et en développement à l'Université North Western et à l'Université d'Urbana Champagne en Illinois sur la base des travaux de Forbus et De Kleer. Du point de vue théorique, le sujet semble avoir été clos. À notre connaissance, aucune nouvelle théorie n'est venue enrichir celles qui existaient déjà.

1.2 Problématique

Avant d'expliquer notre problématique et aller plus loin, nous allons devoir définir un SMV et pour ce faire, une brève présentation historique s'impose.

D'après Forbus et De Kleer (1993), un SMV est un module qui sert à maintenir des raisonnements effectués par un autre agent. Ce système doit aussi détecter des anomalies et permettre de réviser des croyances. L'agent en question peut être un moteur d'inférence, un humain ou n'importe quel programme. Plus précisément, un SMV doit enregistrer (ou mettre en cache) un raisonnement effectué par n'importe quel autre agent. Il doit en même temps détecter des contradictions si elles surviennent et les signaler. Il doit aussi fournir des explications sur demande et restituer tout raisonnement mis en cache. Indirectement, il peut servir à augmenter les performances de l'agent qui raisonne en lui évitant d'effectuer les mêmes raisonnements plusieurs fois (Forbus et De Kleer, 1993). Pour mettre en cache les raisonnements qui lui sont soumis, tout SMV construit un réseau de nœuds qu'il doit gérer par la suite.

Doyle (1979) a été le premier à populariser l'expression "Système de Maintien de Vérité". Son système s'inscrit dans la suite de plusieurs autres travaux en la matière notamment, ceux de Stallman et Sussman (1977), de London (1978), etc. Le SMV de Doyle fonctionne à base de justifications en enregistrant les faits et leurs justifications. Pour enregistrer un fait, ce système recourt à un nœud et pour représenter la négation de ce même fait, il recourt à un second nœud. Pour "réviser des croyances", ce système associe des états à chaque nœud et selon les besoins, il active ou désactive ceux qui sont considérés comme des assomptions ainsi que tout ce qui en découle.

Le cache que le SMV doit maintenir est construit sous forme d'un réseau qu'il met à jour au fur et à mesure que des raisonnements lui sont transmis. Pour représenter un fait et sa négation, le système avancé par Doyle (1979, 1983) crée deux nœuds. Si un nœud positif et sa négation existent et sont actifs simultanément alors un troisième nœud est créé pour représenter leur contradiction.

Pour éviter de représenter un fait et sa négation par deux nœuds, McAllester (1978) proposa un système où tout raisonnement est transformé sous une forme normale, c'est-à-dire une disjonction de clauses. Ceci permet, d'après lui, de réaliser une économie de mémoire et éviter d'avoir à effectuer des retours en arrière *(backtracks)* pour expliquer les sources d'une contradiction. Le problème de cette approche c'est qu'elle élimine complètement le sens des relations et par conséquent elle handicape le SMV de l'une de ses principales fonctionnalités, celle de fournir des explications.

De Kleer (1986) a développé un système qui calcule tous les contextes dans lesquels un nœud serait actif ou en contradiction. Pour économiser l'utilisation de la mémoire, il proposa de représenter les états par un ensemble de bits. Cependant pour la négation, il a eu recourt à des nœuds séparés dans certains cas et à la normalisation dans d'autres (De Kleer, 1986; Reiter et De Kleer, 1987).

Par souci de simplification, Doyle (1979) et De Kleer (1986) limitent ce qui peut être assumé et ce qui peut être considéré comme une négation. Par conséquent, plusieurs variantes de leurs systèmes ont été proposées. Cependant, la représentation de la négation à elle seule explique grandement l'existence des différents types de SMV.

Le problème que nous voulons résoudre consiste donc à développer un SMV, qui traite la négation efficacement, qui est capable de fournir plusieurs sortes d'explications et qui ne pose aucun type de restrictions.

Du point de vue des structures, plusieurs améliorations peuvent être apportées. Nous allons d'abord remodeler l'entité nœud qui sert de structure fondamentale dans l'approche de Doyle (1979) puis ensuite, nous allons remodeler la structure réseau du système, c'est-à-dire la manière dont les nœuds sont reliés. Cette restructuration devra faciliter la mise au point d'un SMV à base de justification non monotone à négation (SMVJNMN). D'après Forbus et De Kleer (1993), un tel système est difficile à construire et à utiliser. Après de maintes recherches, nous n'avons trouvé aucun SMVJNMN fonctionnel et nous souhaitons contribuer à corriger cette situation. D'après nous, un SMVJNMN n'est pas obligatoirement plus complexe qu'un SMVJNM. Nous pensons plutôt que cette complexité n'est vraie que si le système est modélisé selon les approches traditionnelles avec des structures mal définies.

Pour développer un SMVJNMN, nous allons nous inspirer des meilleures techniques du génie logiciel en adoptant une approche objet et en appliquant un ensemble de patrons de conception. Ceci nous permettra de lever un certain nombre de restrictions introduites dans les SMV de base afin de les simplifier. Dans les prochains chapitres, nous reviendrons sur ces points.

1.3 Objectifs et méthodologie

Le principal objectif de notre mémoire de maîtrise est de proposer une approche orientée objet pour élaborer un SMV. Pour atteindre cet objectif, nous avons franchi les étapes suivantes : étude des SMV décrits dans la documentation scientifique afin de mieux en comprendre la problématique, modélisation du système au niveau des connaissances d'un SMV, conception par patrons, implémentation et tests.

Le niveau des connaissances *(Knowledge level)* a été introduit par Newell (1982) pour décrire comment la modélisation peut être effectuée à un certain niveau d'abstraction et comment elle doit l'être indépendamment du type d'implémentation (niveau des symboles). Au niveau des connaissances, seul ce qu'un agent connaît et comment ses buts sont atteints doivent être spécifiés alors que les détails doivent être traités au niveau de l'implémentation.

La méthodologie CommonKADS propose une suite de modèles que Schreiber et *al.* (pp. 18-19, 2000) classent en trois catégories. La première porte sur la modélisation au niveau contextuel (organisation, tâches, agents) c'est-à-dire d'un point de vue managérial ou "administration des affaires" (Schreiber et *al.*, p. 46, 2000). Cette partie ne fait pas partie de la proposition de recherche. La seconde concerne l'aspect conceptuel et se situe au niveau des connaissances et celui de la communication. La troisième porte sur les artéfacts.

Entre le fameux "niveau des connaissances" de Newell (1982) et le modèle des connaissances de CommonKADS il va falloir établir une correspondance. CommonKADS

suit le formalisme UML[4] (Booch *et al.,* 1999) et un autre qui lui est propre. Comme toute autre méthodologie utilisant ce formalisme, CommonKADS décrit comment un modèle peut être spécifié à un certain niveau d'abstraction et comment il peut être raffiné progressivement jusqu'à la phase d'implémentation. Nous reviendrons sur ce sujet dans le paragraphe 4.1.

1.4 Résultats obtenus

Mettre au point un SMVJNMN fonctionnel est le principal résultat attendu et c'est ce que nous avons réalisé. Un SMV fonctionnel est celui qui remplit les fonctionnalités que nous avons déjà énumérées, à savoir :

- mettre en cache un raisonnement;
- détecter des contradictions;
- fournir des explications sur demande;

Dans notre cas, montrer que le système réalisé remplit ces fonctionnalités nécessite absolument le développement d'un système client qui doit utiliser le SMV. Un système minimaliste a donc été développé avec deux types exemples d'application qui couvrent les points que nous venons d'énumérer.

À travers le premier type exemple, le système client soumet au SMV une suite de cas triviaux, que nous allons décrire dans le chapitre 2 et que nous allons tester puis décrire dans le chapitre 6. Des traces d'exécution pour chacun des cas seront fournies. Chacun de ces exemples teste un aspect particulier du système et l'ensemble a été repris de l'ouvrage de Forbus et De Kleer (1993) où le tout est documenté.

[4] *Unified Modeling Langage.*

Le second exemple sert à montrer comment le SMV peut servir en tant que cache de plusieurs dizaines de milliers de nœuds et du même coup, comment il peut servir dans la résolution d'un problème concret. Dans cet exemple, le système client consiste en un mini moteur de recherche qui cherche une suite d'expressions parmi une multitude d'autres et affiche les résultats selon un ordre donné. Au cours de son exécution, ce système va requérir la création d'un très grand nombre de nœuds de la part du SMV. Le but poursuivi dans cet exemple est de montrer que le SMV peut être mis à rude épreuve en termes de charge au niveau de la mémoire.

Chaque exemple que nous présentons cible une des fonctionnalités du système mais sans forcément exclure les autres.

1.5 Contributions

La principale contribution de ce mémoire est de mettre au point un système qui supporte tous les opérateurs logiques de base tout en accomplissant toutes les fonctionnalités qu'un SMV est sensé effectuer. Le système en question reprend les points forts des systèmes étudiés et évite ceux qui ont été jugés problématiques. Plus exactement, le SMV qui sera développé le sera à base de justifications, non monotone et à négation (SMVJNMN). Dans le chapitre 3, nous verrons exactement en quoi consiste ce type de systèmes.

La seconde contribution a été de modéliser le système au niveau des connaissances en mettant complètement en arrière plan tout aspect algorithmique. Ceci a permis de le concevoir à partir d'un agencement assez dense de patrons de conception. Nous faisons ici référence à la densité dans le sens de Riehle (1997) mais en tenant compte des contraintes

de "refactorisation" comme le proposent Gamma et *al.* (1995) et Fowler (1999). Il faut entendre par "refactorisation" un système qui est facilement extensible et dont les structures sont découplées des fonctions. À notre connaissance, aucun SMV n'a été abordé selon ces dimensions et en particulier, en termes de patron de conception. Cette dernière approche permet de présenter le modèle du SMV comme une structure en réseau avant tout. Par conséquent, elle permet éventuellement de l'appliquer dans des domaines similaires tels les graphes de Pétri, les réseaux de neurones, etc. Dans ces cas, seules les fonctions restent à adapter selon le champ d'intérêt. Le lecteur averti pourra établir un lien entre ce type de structures et celle des systèmes des bases de données réseau[5]. Concernant l'approche par patrons de conception (incluant les idiomes), nous nous sommes inspirés grandement des pratiques utilisées dans la plateforme Mac Os X[6].

Développer un système en adoptant une approche objet est un choix qui favorise la réutilisation et la maintenance au détriment d'une approche purement algorithmique. En effet, ce qui nous intéresse ce n'est pas de trouver un algorithme quelconque qui améliore une fonction du système ou sa rapidité (cette dernière peut dépendre de plusieurs facteurs) mais plutôt de décrire le système d'une façon générique avec un ensemble de patrons de conception déjà assimilés. À notre avis, le lecteur ayant des connaissances de base en génie logiciel peut facilement appréhender le fonctionnement d'un SMV sans forcément rentrer dans des détails proches de l'implémentation (pseudo code, algorithmes, etc.).

1.6 Plan du mémoire

[5] Un SMV construit un réseau qui sert de cache et ce réseau peut être comparé à celui des systèmes de bases de données réseau de type entités/relations des normes Codasyl de 1981. Dans ces systèmes, les entités sont unies par des relations *(sets)* de types *"Owners/members"*. Notre système construit sa cache réseau selon ce principe.

[6] http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/chapter_5_section_3.html

Ce mémoire est organisé en sept chapitres incluant celui-ci. Le chapitre 2 porte sur l'état de l'art. On y décrit les SMV de base, les systèmes qui ont été développés ainsi qu'une mise au point concernant la révision des croyances en général.

Le troisième chapitre porte sur la problématique et sur la méthodologie CommonKADS. Dans la problématique, nous faisons ressortir les points forts et les points faibles de chacun des trois types des SMV décrits ainsi que les éléments de solution qui en découlent. Nous situons d'abord l'approche de McAllester (1978) par rapport à celle de Doyle (1979-1983) et puis nous évaluons dans quelle mesure cette approche constitue vraiment une alternative utiles. Par la suite, nous procédons de la même façon avec l'approche de De Kleer. Après une brève présentation de la modélisation au niveau des connaissances, nous décrivons la méthodologie CommonKADS. Nous effectuons un survol de la suite des modèles qui y sont proposés en mettant l'accent sur ceux qui nous concernent le plus.

Le chapitre 4 est consacré à la modélisation du système en fonction de nos objectifs. Une attention particulière est accordée aux modèles des connaissances et de la communication.

Dans le chapitre 5, nous expliquons le modèle de conception en passant en revue les spécifications imposées par CommonKADS. Plus particulièrement, nous montrons comment le système envisagé s'intègre dans le paradigme Modèle-Vue-Contrôleur[7]. Auparavant, nous décrivons rapidement un ensemble des patrons de conception et la façon dont ils ont été appliqués pour modéliser notre système.

Le chapitre 6 contient les éléments suivants : implémentation réalisée, tests effectués, présentation et discussion des résultats obtenus. Nous exposons la façon dont le SMV a été

implémenté, comment il doit être exploité ainsi que le système client qui sera utilisé pour le manipuler. Ce dernier système sera modélisé et implémenté pour :permettre de soumettre un ensemble de requêtes au SMV pour des fins de test et pour résoudre un problème concret, actuel et très exigeant en ressources. Le chapitre 7 porte sur la conclusion.

[7] http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/chapter_5_section_4.html

Chapitre 2 : État de l'art

Dans ce chapitre, nous allons présenter les trois principaux types de SMV à savoir : le SMV à base de justifications de Doyle (1979), le SMV à base logique de McAllester (1978) et le SMV à base d'assomptions de De Kleer (1986-1993). Nous présenterons la logique et les stratégies adoptées dans chacun de ces systèmes. À partir de ces systèmes de base, plusieurs variantes et plusieurs algorithmes ont été proposés dans la littérature (Martins, 1990). On peut citer entre autres, les systèmes de gestion de clauses et les systèmes non monotones à négation. Ces derniers sont très complexes et difficiles à comprendre et à utiliser (Forbus et De Kleer, p. 169, 1993). Pour ces raisons, nous mettrons donc l'accent sur la compréhension des systèmes de base, nous présenterons les outils développés qui sont disponibles puis nous verrons quelques domaines où ils ont été appliqués.

2.1 Les systèmes de maintien de vérité

Un SMV est un programme qui fonctionne en tandem avec un moteur d'inférence ou avec n'importe quel autre agent à qui il offre certains services en enregistrant les raisonnements qu'ils ont effectués (Forbus et De Kleer p. 157, 1993).

D'après Doyle (p. 2, 1983), l'origine de ces systèmes remonte au milieu des années soixante-dix avec les travaux de Stallman et Sussman qui s'intéressaient aux problèmes des retours en arrière guidés par dépendance *(Dependency directed backtracking).* Au cours de cette période, plusieurs systèmes ont été développés entre autres par London (1978), McAllester (1978), Doyle (1979), Reiter et De Kleer (1987), De Kleer et William (1986), etc. (Doyle, p. 2, 1983). L'expression "Système de maintien de vérité" a été cependant

popularisée par Doyle (1979) qui la remplaça par celle de "Système de Maintien des Raisons"[8] afin d'éviter certaines ambiguïtés[9] (Doyle, p. 2, 1983).

Le fait de recourir aux SMV présente l'avantage de laisser le moteur d'inférence s'occuper de la résolution du problème alors que la tâche de maintien de vérité est déléguée à un système indépendant. Pour fonctionner, les deux systèmes doivent utiliser un même protocole de communication et chacun doit gérer ses propres bases de données (figure 2.1). Ces bases doivent être synchronisées. Par conséquent, à chaque assertion de la base de faits du moteur d'inférence, un élément correspondant doit exister dans celle du SMV (Forbus et De Kleer, p. 158, 1993).

D'après Forbus et De Kleer (pp. 152-156, 1993), tout SMV doit être capable d'accomplir les fonctions suivantes :

- Mettre en cache un raisonnement et le fournir sur demande;
- Permettre d'effectuer des révisions;
- Détecter des contradictions quand elles surviennent;
- Fournir des explications;
- Servir de guide lors des retours en arrière guidés par dépendances;
- Permettre d'effectuer des raisonnements par défaut.

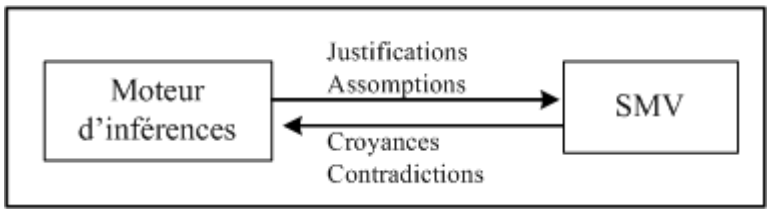


Figure 2.1 Architecture d'un solveur de problème
Source : De Kleer et Forbus (p. 158, 1993)

[8] *Reasons Maintenance System (RMS).*
[9] D'après Doyle, l'expression SMV est conservée simplement pour des raisons historiques (Doyle, p. 2, 1983).

Chaque type de SMV s'acquitte de ces fonctions à sa manière et diverses stratégies peuvent être adoptées. Cependant, le recours à un SMV peut s'avérer contre productif s'il est couplé avec un moteur d'inférence qui est déjà capable de réaliser efficacement ces tâches (Forbus et De Kleer, p. 169, 1993).

Pour mettre en cache les raisonnements qui leur parviennent, les SMV construisent un réseau où les assertions du moteur d'inférence sont représentées par des nœuds reliés entre eux suivant les raisonnements effectués (Forbus et De Kleer, p. 159, 1993). Au lieu de rétracter un fait dans la base du moteur d'inférence, une simulation peut être effectuée par le SMV qui désactive le nœud correspondant et tout autre nœud qui en dépend. Pour maintenir l'état d'activité, chaque nœud est muni d'un ensemble d'étiquettes que le système met à vrai ou à faux. Cet ensemble varie en fonction des stratégies suivies.

Dans les points qui vont suivre, nous allons présenter la logique de chacun des trois SMV de base. Plus particulièrement, nous allons voir comment chacun étiquette ses nœuds, traite la négation et la non monotonie, détecte les contradictions et fournit des explications. La détection des contradictions porte sur les anomalies reliées à l'existence simultanée d'un fait et de sa négation dans la base du SMV et c'est le seul type d'anomalies qu'un SMV peut détecter automatiquement. Pour les autres, seules des explications peuvent être fournies sous forme de traces d'exécution. L'interprétation de ces explications relève du système client qui recourt au SMV (Forbus et De Kleer, 1993).

2.1.1 Les SMV à base de justifications

D'après Doyle (p. 232, 1979), un SMV est un programme qui sert à déterminer un ensemble de croyances à partir d'un ensemble de raisons ou justifications. Ce système permet de mettre à jour cet ensemble de croyances en tenant compte de toute nouvelle "raison" qui y

est ajoutée ou rétractée au cours d'un processus de raisonnement. Une croyance qui est mise hors de l'ensemble des croyances est dite *OUT*. Elle est dite *IN* quand elle est introduite (Doyle, p. 234, 1979). D'après Forbus et De Kleer (p. 174, 1993), *IN* et *OUT* peuvent être interprétées respectivement par "il est vrai" et "il est faux" qu'un fait existe dans la base des croyances.

Les croyances initiales qui peuvent être rétractées sont appelées "assomptions"[10]. Donc, un SMV doit déterminer les conséquences de tout changement dans l'état de ces assomptions comme il doit pouvoir retracer les raisons qui ont conduit à ces changements. Le même processus est entrepris si une croyance doit être reconsidérée (Doyle, p. 235, 1979). L'ensemble des changements dans la base du SMV simule donc l'état dans lequel la base des faits du moteur d'inférence devrait être si celle-ci a été modifiée réellement.

Pour montrer le rapport qui existe entre un SMV et un simple moteur d'inférence, nous allons prendre un exemple fictif de trois règles et voir comment elles sont instanciées. Nous verrons ensuite comment elles sont représentées au niveau du SMV.

Dans le tableau 2.1, les étapes d'instanciation des règles : $D \rightarrow E$, $A \wedge B \rightarrow C$ et $E \wedge C \wedge F \rightarrow G$ sont décrites. Les symboles $\wedge$, $\vee$, $\rightarrow$, et $\neg$ représentent respectivement la conjonction la disjonction, l'implication et la négation. Le contenu de la base de faits à chacune des étapes figure dans la dernière colonne et c'est ce contenu qui est transmis au SMV. Celui-ci construit alors le réseau représenté par la figure 2.2.

[10] Dans la littérature en anglais, "*Assumption*" est utilisée au lieu de "*Hypothesis*". D'après l'office de la langue française du Québec, "une assomption est une proposition admise à titre d'hypothèse en vue d'en démontrer une autre".

Dans le tableau 2.1, l'assertion de *D* lors de la première étape se traduit par son ajout dans la base des faits du moteur d'inférence. Ceci déclenche la règle *R1* qui à son tour engendre automatiquement *E* (étape deux). À l'étape trois, l'assertion de *B* ne déclenche rien tant que celle de *A* n'a pas été effectuée. Quand cela se produit à l'étape quatre, *C* est ajouté automatiquement. L'introduction de *F* à l'étape six se traduit par celle de *G*. À la fin de la septième étape, la base des faits contiendra une liste ordonnée qui correspond à la cinquième colonne du tableau 2.1.

Étapes	**Assertions**	**Règles déclenchées**	**Assertions automatiques**	**Base des faits**
1	*D*			*D*
2		*R1*	*E*	*E*
3	*B*			*B*
4	*A*			*A*
5		*R2*	*C*	*C*
6	*F*			*F*
7		*R3*	*G*	*G*

Tableau 2.1 Exemple d'exécution d'un système de règles

La figure 2.2 est une représentation du réseau construit en parallèle par le SMV selon les conventions de De Kleer et Forbus (p. 162, 1993). Selon ces conventions, les assomptions sont représentées par des rectangles, les justifications par des triangles et tout autre nœud par des ovales. Ainsi par exemple, le nœud *B,* qui est considéré comme une assomption, est représenté par un rectangle et il est le seul qui peut être rétracté. Les nœuds *A* et *D* seront toujours actifs.

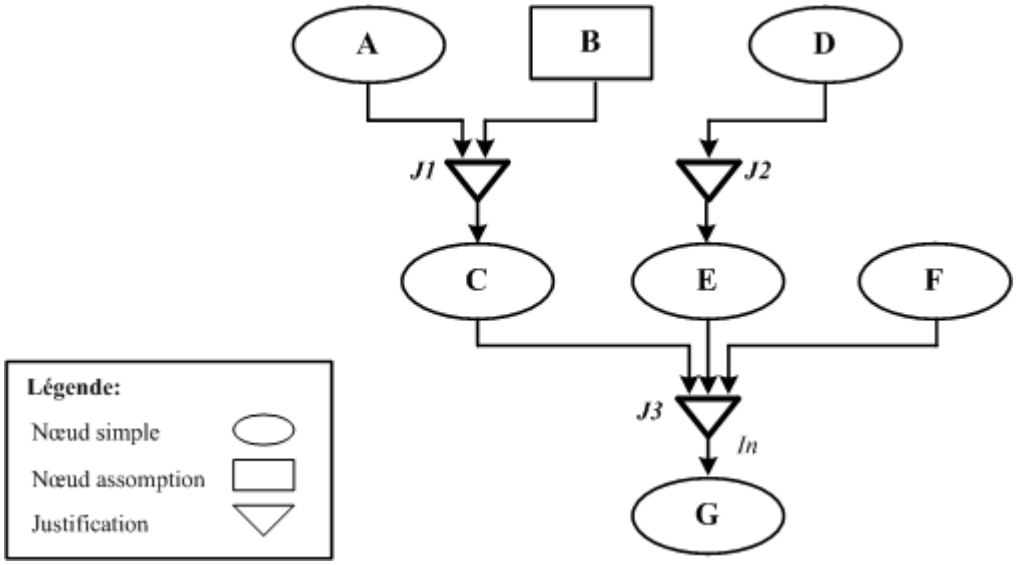


Figure 2.2 Exemple d'un simple réseau

D'après Shapiro (Shapiro, 1998), le fait qu'un nœud soit considéré comme étant *OUT* implique que tout le raisonnement qui en dérive doit être reconsidéré et défait. Par conséquent, tous les nœuds qui dérivent de ce nœud doivent être mis à *OUT*. Si par la suite ce nœud est remis à *IN*, alors le processus inverse doit être effectué. Ce processus de propagation monotone doit donc être déclenché à chaque fois que l'état d'un nœud est modifié. Donc, dans l'exemple de la figure 2.2, l'état de *C* et de *G* dépendra directement de celui de *B*.

Dans certaines situations, pour être mis à *IN* certains nœuds peuvent nécessiter que d'autres soient dans l'état contraire (Shapiro, 1998; Doyle, 1979). Le processus de propagation doit donc tenir compte de l'état à propager comme il doit tenir compte de certaines contraintes. Il doit, entre autres, éviter de rentrer dans des boucles sans fin et détecter toute contradiction qui survient. Le SMV de Doyle (1979) est non monotone et remplit toutes ces conditions.

2.1.1.1 Traitement de la négation et des contradictions

D'après Forbus et De Kleer (1993), la représentation explicite de la négation dans un SMVJ nécessite la création d'un nœud distinct. Une contradiction est détectée automatiquement lorsqu'un nœud ainsi que le nœud représentant sa négation existent et sont à *IN* simultanément. Une contradiction peut résulter aussi de la combinaison de n'importe quels autres nœuds qui sont mutuellement exclusifs quand cela est spécifié expressément. Dans les deux cas, un nœud est automatiquement créé par le SMV pour la représenter.

La figure 2.3 est un exemple fourni par Forbus et De Kleer (p. 177, 1993) qui illustre ces deux situations ainsi que les choix possibles qui doivent être pris pour résoudre les contradictions qui en résultent.

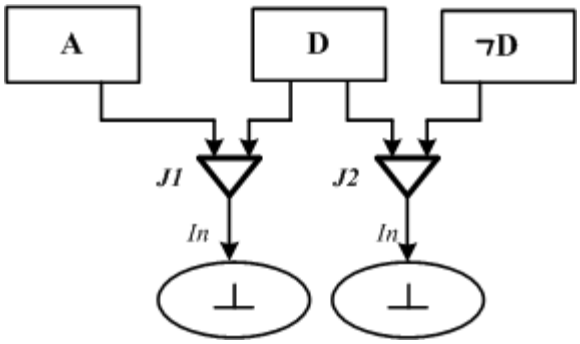


Figure 2.3 Gestion des contradictions
Source : adapté de Forbus et De Kleer (p. 177, 1993)

Lorsque le SMV construira ce réseau, il réagira de différentes façons dépendamment de la manière dont les nœuds sont créés. S'il reçoit en premier la relation $A \wedge D \xrightarrow{In} \perp$, il va enregistrer la contradiction et la signaler et l'une des deux assomptions, *A* ou *D*, va devoir être désactivée. La relation $\xrightarrow{Étiquette}$ indique le sens de la propagation ainsi que l'étiquette à propager lorsque le précédent est à *IN*.

Si *A* est désactivé, alors quand $\neg D$ sera introduit, le SMV va automatiquement générer une nouvelle contradiction et un des deux nœuds, *D* ou $\neg D$, va devoir être désactivé. Si *D* est désactivé à son tour, alors seul $\neg D$ restera actif dans tout le réseau.

Si au lieu de *A*, *D* a été désactivé en premier, l'introduction de $\neg D$ ne pourrait pas générer de contradiction. Par conséquent, *D* n'aurait pas été désactivé et deux nœuds sur trois seraient restés actifs. Pour cette raison, Shapiro suggère de développer des SMV qui seraient capables de minimiser automatiquement l'ensemble des nœuds devant être désactivés (Shapiro, 1998). Normalement, comme le soulignent Forbus et De Kleer (p. 173, 1993), c'est à l'agent qui effectue le raisonnement que revient le choix de décider quel noeud désactiver. En effet, en plus des critères quantitatifs, d'autres aspects qualitatifs doivent être considérés comme nous allons le voir dans le paragraphe 2.3.

2.1.1.2 Propagation et non monotonie

Un simple SMVJ ne fait que propager les mêmes états alors qu'un SMVJ non monotone permet de déduire l'état d'activité d'un nœud en l'absence de celle d'un autre (Shapiro, 1998). Dans ce cas, le changement de l'état d'un nœud peut se traduire aussi bien par la propagation de cet état que par celle de l'état contraire. Le support de la non monotonie est essentiel car il permet de représenter l'exclusion mutuelle et la logique des défauts (Forbus et De Kleer, p. 156, 1993).

Pour supporter la non monotonie dans le système de Doyle, chaque nœud est muni d'une paire de listes, une liste des nœuds à mettre à *IN* et une liste des nœuds à mettre à *OUT*. Quand l'état d'un nœud change, les nœuds des deux listes sont mis à jour en conséquence.

2.1.1.3 Propagation et circularité

Lorsqu'un nœud doit être activé ou désactivé, le SMVJ doit propager tout changement dans le reste du réseau, détecter en même temps toute contradiction et composer avec les chemins circulaires. Un changement peut survenir si l'état d'activité d'un nœud change, si un nœud est ajouté au réseau ou si une quelconque relation entre nœuds est modifiée (Doyle, 1979; Forbus et De Kleer, 1993).

D'après Forbus et De Kleer (1993), le processus de propagation doit tenir compte d'un certain ensemble de contraintes. Premièrement, un et un seul processus de propagation est effectué à la fois. Deuxièmement, si une contradiction est détectée, le processus doit être arrêté. Troisièmement, si un état est propagé vers un nœud qui est déjà dans cet état, alors aucun changement ne doit être effectué et la propagation doit être arrêtée. Elle peut se poursuivre éventuellement sur d'autres chemins tant que ceux-ci contiennent des nœuds dont l'état peut être modifié. Finalement, le SMVJ doit être en mesure d'éviter de rentrer dans des boucles infinies. Pour éviter les circuits, le SMV marque simplement tout nœud visité et le maintient dans cet état tant que le processus de propagation n'a pas été complété.

2.1.1.4 Circularité et supports bien fondés

Un nœud peut posséder plus qu'un précédent et il ne peut être désactivé que si tous ses précédents ne sont pas dans un état qui lui permet de rester actif. On appelle "support bien fondé" toute justification qui, sans être parmi les conséquents d'un nœud, permet à ce dernier de rester actif (Doyle, 1979). En effet, quand un nœud a été désactivé, l'état de l'ensemble de ses précédents doit être vérifié pour trouver un support lui permettant de rester actif. Cependant, pour éviter qu'un support ne soit trouvé parmi les conséquents, sa recherche ne doit être entreprise qu'à la fin d'un même processus de propagation (Forbus et

De Kleer, p.174, 1993). Par exemple dans la figure 2.4, si le nœud *C* est désactivé, *B* le sera aussi. Si le SMV cherche immédiatement un support pour *B*, il va trouver *A* par J_1 alors que celui-ci sera mis à *OUT* à travers J_2. Dans ce cas, *A* ne peut pas servir de support pour *B* et le SMV doit continuer la propagation jusqu'à la fin du circuit pour éviter de le considérer.

Pour détecter la présence de circuits, le SMV marque tout nœud visité. À la fin de chaque processus, tout nœud marqué est remis dans son état initial et un nouveau chemin peut être exploré alors (Forbus et De Kleer, p.174, 1993).

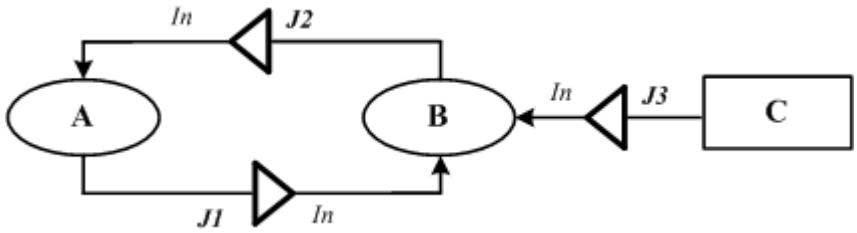


Figure 2.4 Supports bien fondés

La figure 2.5 illustre un cas non monotone qui peut aboutir à deux résultats différents mais tous valables. Dans cet exemple, si le nœud *A* est actif alors *B* et *C* vont l'être automatiquement. S'il est désactivé alors deux cas sont possibles dépendamment du chemin que la propagation empruntera en premier, par J_3 ou par J_4.

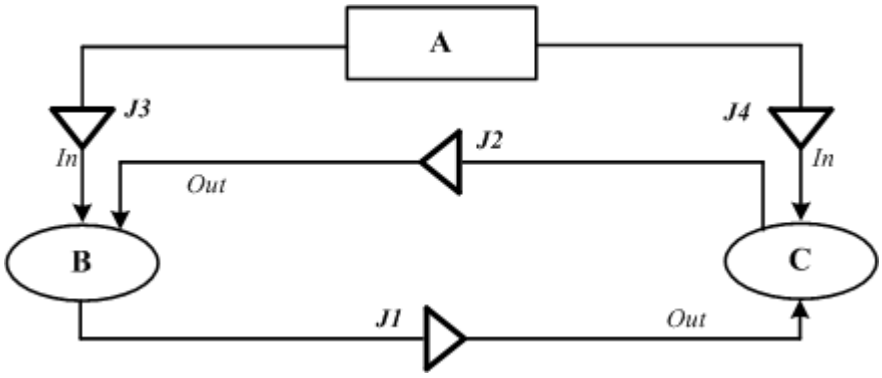


Figure 2.5 Circularité et propagation
Source : adapté à partir de Forbus et De Kleer (p. 183 ,1993)

Si la propagation débute par J_3 alors *B* sera mis à *OUT* puis marqué. Puisque *B* est *OUT*, le SMV essaiera de mettre *C* à *IN* mais celui-ci l'est déjà. *B* sera ensuite démarqué et un support sera cherché mais aucun ne sera trouvé. En effet, *A* étant à *OUT*, il ne reste que *C* mais celui-ci est à *IN*. Si *C* est à *IN*, *B* doit être à *OUT* et il l'est déjà. Le processus de propagation par J_3 prend fin et continue par J_4. *C* sera mis à *OUT* puis marqué, *B* sera ensuite mis à *IN* puis marqué. La propagation prend alors fin puisqu'elle ne peut pas continuer par J_1 car *C* est déjà marqué. Un support sera ensuite cherché pour *C* mais ni *A* ni *B* ne peut servir à cette fin. *C* sera alors démarqué ainsi que *A* et la propagation s'arrête définitivement. *A* et *C* finiront donc à *OUT* et *B* à *IN*.

Si la propagation avait débuté par J_4, alors *A* et *B* auraient fini à *OUT* et *IN* respectivement puisque le réseau est symétrique.

L'ordre dans lequel le SMV enregistre ses nœuds ainsi que celui qu'il choisit pour la propagation déterminent donc l'état final du réseau. Cet ordre doit être considéré dans la stratégie de la résolution d'un problème de l'agent qui recourt au SMV. D'après Forbus et De Kleer (p. 174, 1993), la compréhension de la propagation dans un contexte non monotone et circulaire est essentielle et les multiples résultats possibles ne doivent pas forcément être pris pour des dysfonctions du système.

2.1.1.5 Explications bien fondées

Pour pouvoir propager des changements d'états, le SMV maintient des structures de données pour stocker les conséquents, les justifications, les états ainsi que toute information appropriée. Ces structures servent aussi pour suivre à la trace n'importe quel raisonnement effectué et l'expliquer dans les deux sens par chaînage arrière ou par chaînage avant. Comme lors des propagations, tout SMV doit être en mesure de fournir des explications

bien fondées tout en composant avec les chemins circulaires (Forbus et De Kleer, p. 174, 1993).

2.1.2 Les SMV à base logique

McAllester (1978) proposa un SMV dit à base logique (SMVL) à trois valeurs où chaque nœud est étiqueté par *VRAI*, *FAUX* ou *INCONNU*. Dans ce système, les raisonnements sont encodés sous forme d'un ensemble de clauses normalisées (McAllester, p. 5, 1978). D'après McAllester (1978), cette approche permet de représenter n'importe quelle formule y compris la négation et ce, sans recourir à un second nœud pour représenter la négation. D'après lui, cette représentation est plus économe en termes d'utilisation de la mémoire.

D'après Forbus et De Kleer (p. 270, 1993), dans un SMVL, un nœud qui dérive d'un ensemble de clauses est étiqueté *VRAI*. Si c'est sa négation qui l'est alors il est étiqueté *FAUX*. Dans toute autre situation, il est étiqueté *INCONNU*. *VRAI* et *FAUX* peuvent être respectivement associés à l'étiquette *IN* et *OUT* du SMVJ.

Dans le système de McAllester, quand une expression est normalisée, ses opérandes sont simplement post fixés avec leurs étiquettes. Par exemple, $A \wedge C \rightarrow D$ est représentée par $A.FAUX \vee C.FAUX \vee D.VRAI$ bien que cela puisse donner lieu à de multiples interprétations. En effet d'après McAllester (p. 5, 1978), cette expression peut résulter aussi bien de $A \wedge C \rightarrow D$, de $\neg D \wedge C \rightarrow \neg A$ que de n'importe quelle autre formule équivalente.

2.1.2.1 Gestion des contradictions

D'après McAllester (p. 9, 1978), une contradiction est détectée dès que tous les éléments d'une même formule sont mis à *FAUX*. Par exemple, dans la suite des formules suivantes : $\neg A \vee B$, $\neg C \vee D$, $\neg C \vee E$ et $\neg B \vee \neg D \vee \neg E$, la dernière expression sera étiquetée par $B.FAUX \vee D.FAUX \vee E.FAUX$ et sera en contradiction. En effet, comme le notent Forbus et De Kleer (p. 284, 1993), si *A, B, C, D* et *E* sont des assomptions initialement étiquetées *INCONNU* et si *A* et *C* sont mis à *VRAI*, le SMVL déduira *B* puis *D* et la quatrième expression sera contradictoire.

Pour résoudre cette contradiction, le SMVL va automatiquement étiqueter *C* par *FAUX* et puis générer et enregistrer $\neg A \vee \neg C$ comme *nogood*. *"Nogood"* est une expression utilisée dans le langage des SMV pour désigner un ensemble d'assomptions qui engendrent une contradiction. Comme le montre McAllester (p. 10, 1978), les formules précédentes peuvent être interprétées par les formules équivalentes $A \rightarrow B$; $C \rightarrow D$; $C \rightarrow E$ et $B \wedge D \rightarrow \neg E$.

D'après ces formules, il devient évident que l'existence de *A* et celle de *C* impliquent nécessairement celle de *E* et de $\neg E$, d'où la contradiction. D'après Forbus et De Kleer (1993), si *C* doit être réintroduit plus tard, le SMVL sera en mesure d'étiqueter automatiquement *A* et *C* de telle manière qu'ils ne soient pas à *VRAI* simultanément. Ceci évite par le fait même la nécessité de tout retour en arrière guidé par dépendances.

2.1.2.2 Explications bien fondées

D'après McAllester (p. 5, 1978), si la normalisation permet de résoudre la représentation explicite de la négation, le sens de l'implication se perd malheureusement. Étant donné qu'un SMV doit pouvoir fournir des explications bien fondées, des mécanismes supplémentaires doivent être prévus pour les générer. Forbus et De Kleer (p. 276, 1993) recourent à une structure où chaque nœud est associé avec trois autres éléments : le numéro de l'étape du raisonnement, le numéro des étapes précédentes ainsi que la justification associée.

2.1.3 Les SMV à base d'assomptions

En adoptant une approche complètement différente, De Kleer (1986) a proposé un SMV dit à base d'assomptions (SMVA) où chaque nœud est étiqueté par l'ensemble des environnements dans lesquels il est actif. De Kleer (p. 143, 1986) définit un environnement comme étant un ensemble conjonctif d'assomptions qui sert de support pour un nœud donné.

Le recours à des environnements en tant qu'étiquettes avait pour but de résoudre certaines catégories de problèmes qui nécessitent de considérer plusieurs contextes simultanément. D'après De Kleer (p. 187, 1986), les SMVJ ne permettent de considérer qu'un contexte à la fois. Les SMVA ont été donc avancés pour dépasser cette limite en calculant pour chaque nœud toutes les possibilités qui lui permettent d'être actif. Cependant, cette stratégie peut nécessiter un nombre exponentiel de calculs et d'après De Kleer (p. 164, 1986), il peut exister jusqu'à 2^n combinaisons possibles pour un nœud qui dérive de n assomptions.

Pour résoudre à ce problème, différentes techniques ont été mises au point. Premièrement, le SMVA procède de façon incrémentale (comme les autres systèmes d'ailleurs) c'est-à-dire, qu'il effectue ses calculs au fur et à mesure qu'il enregistre ses nœuds. Deuxièmement, les environnements sont calculés pour chaque noeud automatiquement auxquels les environnements des précédents sont ajoutés. Troisièmement, l'ensemble des environnements est minimisé pour chaque nœud en tenant compte des subsomptions (Forbus et De Kleer, 1993).

Toutes ces opérations rendent le système plus complexe et des contraintes supplémentaires ont dû être introduites. Ainsi, un simple SMVA ne traite pas la négation et se limite aux seules relations monotones (Forbus et De Kleer, 1993).

Pour traiter la négation, d'autres variantes ont été développées plus tard notamment, les SMVA à négation (De Kleer, 1988), les systèmes de gestion de clauses (Forbus et De Kleer, 1993), etc. Dans ces systèmes, les mêmes techniques de normalisation utilisées dans les SMVL sont reprises avec toute la complexité que cela engendre.

2.1.3.1 Exemple

Pour montrer comment un SMVA étiquette ses nœuds, nous allons reprendre un exemple classique de De Kleer (1986) et Forbus et De Kleer (1993), soit celui d'un simple SMVA défini par les relations monotones suivantes :

$A \longrightarrow E$,	$D \longrightarrow F$,
$B \longrightarrow E$,	$E \longrightarrow H$,
$C \longrightarrow F$,	$F \wedge G \longrightarrow I$,
$C \longrightarrow G$,	$H \wedge I \longrightarrow J$.

Dans cet exemple, *A, B, C* et *D* sont des assomptions et comme on peut le voir dans la figure 2.6, chaque nœud est étiqueté par son ensemble d'environnements. Ainsi, *E* possède les environnements {*A}, {B}* ainsi que *{A, B}* mais ce dernier est inutile. En effet, seul *{A}* ou *{B}* peut suffire. Il en est de même pour *H* qui hérite les environnements de *B* mais n'en possède pas d'autres.

F quant à lui, possède les environnements *{{C}, {D}, {C, D}},* mais *{C}* et *{D}* suffisent. *G* ne possède que *{D}* mais *I* hérite des environnements de *F* et de *G*. Comme *C* ne détermine en rien l'état de *I* alors seul *{D}* est retenu. Finalement, *J* hérite des environnements de *H* et de *I* et aucune minimisation n'est plus possible. L'état de *J* se retrouve donc déterminé par les environnements *{A, D}* et *{B, D}* au lieu de *{A, B, C}; {A, C, D}; {B,C, D}; {A, C}* et *{B, C}*.

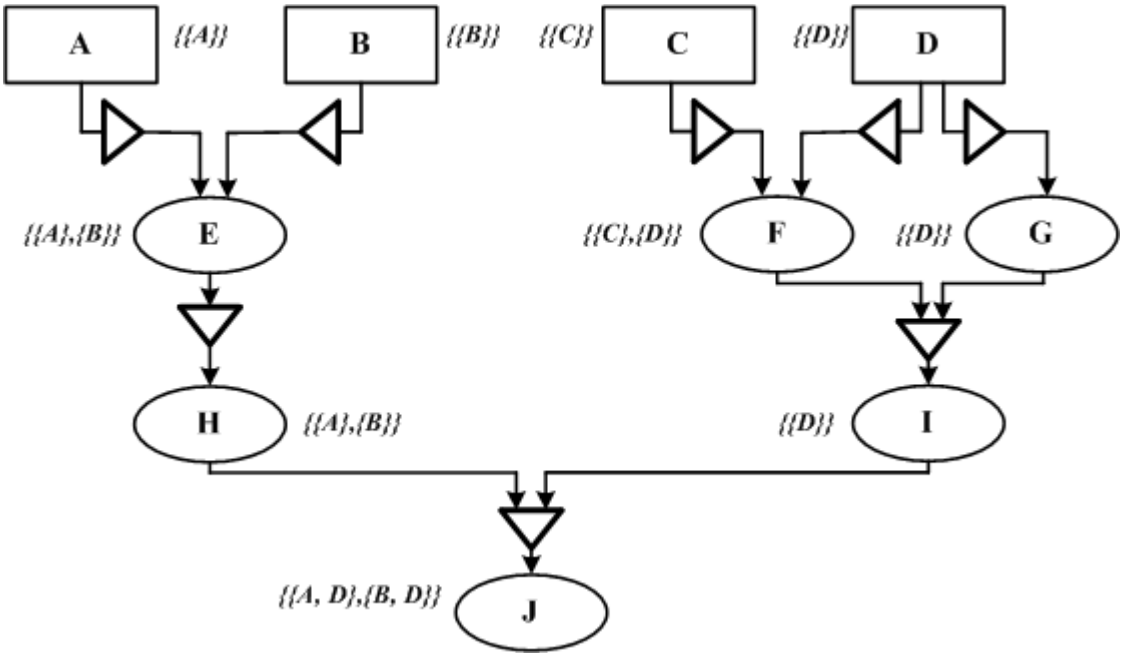


Figure 2.6 Exemple d'étiquettes dans un réseau SMVA
Source : Forbus et De Kleer (p. 449, 1993)

2.1.3.2 Gestion des contradictions

Dans un SMVA comme dans un SMVJ, la contradiction est représentée par un nœud à part. Pour un nœud donné, tout environnement qui débouche sur une contradiction est enlevé de tout ensemble d'environnements sains puis stocké en tant que *nogoods* (De Kleer, 1986).

Dans l'exemple précédent, si une relation $K \wedge A \longrightarrow \perp$ est introduite et si *K* est une prémisse mais sans être une assomption alors *A* sera exclu respectivement des environnements de *E*, de *H* puis de *J* (figure 2.7). L'étiquette de ce dernier nœud serait alors réduite à *{D, B}*.

Pour associer chaque nœud avec ses environnements (sains et *nogoods*) respectifs, le SMVA doit maintenir des structures supplémentaires (De Kleer, 1986). Donc, en plus de devoir enregistrer tout le réseau de la figure 2.7, toutes les informations contenues dans le tableau 2.2 doivent l'être aussi.

Nœuds	**environnements**	***Nogoods***
H	*{B}*	*{A}*
E	*{B}*	*{A}*
F	*{C}, {D}*	
G	*{D}*	
I	*{D}*	
J	*{D, B}*	*{A}*

Tableau 2.2 Ensemble d'étiquettes d'un SMVA

L'exemple que nous venons de voir est assez simple dans le sens qu'il est non monotone et ne traite pas de la négation. Il donne cependant une idée sur la quantité de calcul qui sera générée ainsi que sur les structures de données qui seraient nécessaires pour stocker tous les types d'environnements.

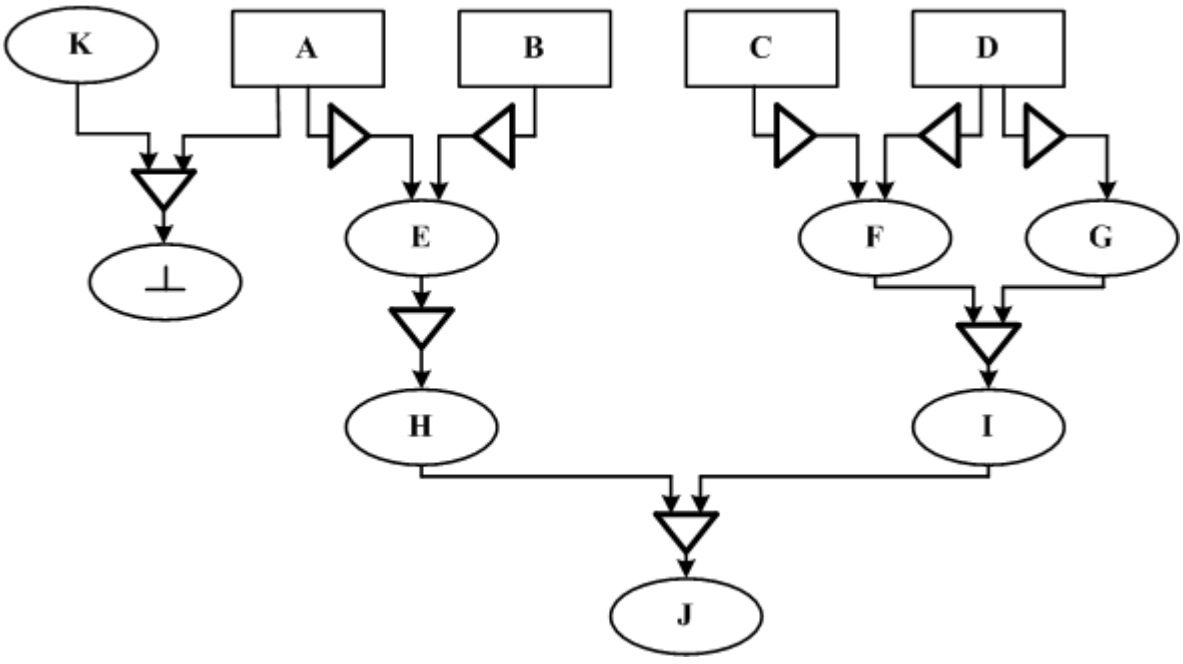


Figure 2.7 Exemple d'un réseau et contradiction
Source : Forbus et De Kleer (1993)

2.1.3.3 Explications bien fondées

Comme n'importe quel autre SMV, un SMVA doit fournir des explications bien fondées. Donc pour chaque nœud, des structures sont nécessaires pour pouvoir retracer l'ensemble des justifications et éviter les chemins circulaires (Forbus et De Kleer, p. 449, 1993). Comme le mentionnent ces auteurs, le système ne peut pas se servir des environnements pour atteindre cette fin mais doit nécessairement suivre les relations entre nœuds.

2.2 Quelques outils existants

McAllester (1978) avait proposé un SMVL en LISP. Forbus et De Kleer (1993) ont mis au point un SMVJ, un SMVL et SMVA dans le même langage ainsi que plusieurs versions d'un moteur d'inférence appelé "*Tiny Rule Engine"* pour les faire fonctionner. Ces SMV sont en développement continu et leur code est disponible sur la toile[11].

Plusieurs systèmes de vérification de la cohérence dynamique ont été mis au point depuis des années. Certains utilisent des techniques qui leurs sont propres et ne sont pas des SMV à proprement parler. D'après Ayel et Rousset (1990), *INDE* par exemple est un outil axé sur les règles mutuellement exclusives et *SUPER* en est un autre qui peut détecter des contradictions en construisant un réseau de type et/ou. Ces systèmes ont été conçus avec l'idée de détecter différentes anomalies concernant des règles de production. *COVADIS* est un autre système décrit par Ayel et Rousset (1990) qui d'après lui, se rapproche des SMVA de De Kleer mais sans recourir à une base de faits particulière. Ces outils ne sont pas disponibles et il est donc difficile de les classer en tant que SMV. Il est en effet difficile de savoir dans quelle mesure ils permettent d'effectuer des révisions de croyances, de quel type de croyances il s'agit et s'ils opèrent dynamiquement.

Plus récemment, des systèmes incorporant des SMV ont été développés et le sont encore. On peut citer notamment *STALKER* (Carbonara et Sleeman, 1999) ainsi que *SNePS* (Shapiro, 1998).

*STALKER (**S**peeding up The **AL**gorithm for **K**nowledg**E** base **R**efinement)* est un outil de raffinement de SBC développé à la suite d'un autre outil similaire appelé *KRUST* (Craw et Boswell, 1999; Wirantunga et Craw, 2000). D'après Carbonara et Boswell (1999), dans *STALKER* les opérations de test sont gérées par un SMV dont le but est d'améliorer la rapidité d'exécution et de pouvoir effectuer plusieurs tests à la fois.

SNeBR (Semantic Network Belief Revision) est un SMV fonctionnant avec le système *SNePS (Semantic Network Processing System)* développé par le groupe de recherche *SNeRG*[12] sous la direction de Shapiro. *SNePS* est un système de représentation des connaissances qui est destiné pour le traitement du langage naturel. Ce système utilise *SNeBR* pour gérer les contradictions automatiquement. Il peut désactiver les "croyances" qui ont un impact minimal comme il peut tenir compte de certaines préférences spécifiées en tant que métaconnaissances. *SNePS* et *SNeBR* sont disponibles sur la toile mais sont spécialisés dans un domaine spécifique de recherche. Ils sont écrits dans un langage qui s'inspire de LISP et *SNePS* a été porté vers le langage Java.

[11] http ://www.qrg.northwestern.edu/BPS/BPS1024.zipil
[12] http ://www.cse.buffalo.edu/sneps/index.html

2.3 Récapitulation

Dans ce chapitre, nous avons présenté les principaux SMV de base sans nous attarder aux nombreux algorithmes proposés dans la littérature. Pour pouvoir comprendre la logique utilisée dans chacun d'eux, nous avons décidé d'analyser les systèmes d'origine tels que proposés par leurs concepteurs. Le but de cette analyse a été de faire ressortir les stratégies adoptées dans chacun des systèmes afin de pouvoir en adopter une plus tard.

Lors de notre revue de littérature, nous avons constaté qu'en plus des algorithmes ponctuels, celle-ci couvre aussi les domaines où les SMV sont appliqués. Les SMV sont cités partout où la révision des croyances est abordée. Concernant ce domaine, une mise au point a été nécessaire pour souligner ce que ces systèmes permettent de réviser exactement. Dans tous les cas, peu d'outils développés sont disponibles pour fins d'essais. Nous avons cité ceux que nous avons pu trouver et pour lesquels des références ont été fournies.

Dans le prochain chapitre, nous allons comparer les points forts et les points faibles de chaque type de SMV étudié. À partir de cette comparaison, nous déciderons quelle stratégie adopter pour modéliser un nouveau système. Nous décrirons aussi la méthodologie qui sera utilisée lors de cette modélisation.

Chapitre 3 : Problématique et méthodologie

Ce chapitre est divisé en deux parties. La première porte sur la problématique et la seconde sur la méthodologie CommonKADS. Dans la problématique, nous allons identifier les points faibles et les points forts de chacun des trois types de systèmes décrits dans le chapitre précédent. Pour ce faire, nous allons prendre le SMVJ de Doyle (1979) comme point de référence et lui opposer les deux autres. Par la suite, nous évaluerons les stratégies adoptées dans chacun de ces systèmes puis nous proposerons notre approche.

Dans la seconde partie, nous allons exposer les grands points de la méthodologie CommonKADS que nous allons appliquer pour développer un SMV.

3.1 Problématique des SMV et leur typologie

Doyle (1979) avait pour but de montrer comment la révision des croyances peut être gérée au moyen de justifications. Il a donc proposé un système où l'entité nœud, sa négation ainsi que la contradiction sont représentées séparément. On peut comprendre cependant que le mode de représentation n'était pas alors sa principale préoccupation.

McAllester (1978) s'est donné un autre but et s'est intéressé à optimiser cette représentation en recourant à la normalisation. Cependant comme Doyle (p. 238, 1979) l'a fait remarquer, cette approche est loin d'être satisfaisante puisqu'elle handicape le système de l'une de ses principales fonctions à savoir celle de fournir des explications.

Dans le paragraphe 2.1.2, nous avons montré comment une expression normalisée peut donner lieu à plusieurs interprétations possibles et c'est ce qui fait que le sens des explications se perd. Le SMVL doit donc être doté de structures de données supplémentaires pour qu'il puisse retracer les raisonnements à la manière d'un SMVJ. Malheureusement, cette option au lieu de réduire la consommation de mémoire ne fait que l'augmenter. Par ailleurs, la normalisation alourdit le système qui doit jouer le rôle d'un interpréteur de formules.

Forbus et De Kleer (p. 281, 1993) ont noté d'autres failles dans l'approche de McAllester (1978) au sujet de la propagation booléenne des contraintes qu'il utilise. D'après eux, cette propagation ne serait pas complète et ne permettrait pas un étiquetage correct dans certaines situations. Par exemple, si les deux expressions $(x \vee \neg y)$ et $(x \vee y)$ sont étiquetées *INCONNU* alors le SMVL doit être en mesure de mettre automatiquement *x* à *VRAI* mais il en est incapable. Il n'est pas capable non plus, par exemple, de détecter qu'aucun état *VRAI/FAUX* ne peut satisfaire les expressions $(x \vee \neg y)$, $(x \vee y)$, $(\neg x \vee \neg y)$, $(x \vee \neg y)$ si celles-ci ont été étiquetées *INCONNU* au départ. Tout ceci explique pourquoi les SMVL avaient connu peu d'écho dans la littérature[13].

Les systèmes de base que nous avons décrits sont des versions simples. Des variantes plus complexes ont été proposées en fonction de leur capacité de gérer la négation et la non monotonie.

Un SMVJ simple est monotone et il est connu dans la littérature par l'abréviation anglaise de JTMS *(Justifications-Based Truth Maintenance System).* S'il supporte la non monotonie, on parle alors d'un *Non Monotonic Justifications Based Truth Maintenance System* ou NMJTMS.

Quand un NMJTMS est capable de propager des changements d'états vers des nœuds négatifs, on le nomme *Negated Non Monotonic Justifications-Based Truth Maintenance System* ou NNMJTMS. D'après Forbus et De Kleer (1993), ce type de système est complet mais il est extrêmement complexe à expliquer et à utiliser. D'après nous, cette complexité découle en grande partie du fait que l'on dérive le NNMJTMS à partir d'un simple JTMS tout en continuant de représenter la négation et la contradiction par des entités à part. Dans le chapitre 4, nous verrons comment cette complexité peut être résolue en adoptant des structures plus appropriées.

Il existe aussi des SMVA simples et des SMVA à négation. Ces systèmes sont connus respectivement par les abréviations ATMS *(Assumptions-Based Truth Maintenance System)* et NATMS *(Negated Assumptions-Based Truth Maintenance System)*. Quand il s'agit de traiter la négation d'une façon plus approfondie, De Kleer (1986, 1987) a choisi d'opter pour la normalisation avec toutes les limites qu'elle implique. Sur cet aspect, un NATMS est fondamentalement différent d'un ATMS.

Après ces considérations, les questions principales que l'on peut se poser sont : quel type de système doit-on élaborer et comment le faire ?

D'après ce qui précède, il est primordial de comprendre que la représentation de la négation explique grandement l'existence de ces différents types de SMV. Pour les raisons déjà citées, la normalisation est une stratégie qui est rejetée d'emblée. Il en résulte que l'on ne peut pas développer ni un SMVL ni un SMVA à négation sur cette base. Le choix d'un SMVJ s'impose donc de lui-même. Pour que celui-ci soit complet, il faut qu'il soit non

[13] On peut facilement le vérifier en jetant un coup d'œil sur la monographie de Martins (1990).

monotone et à négation (SMVJNMN). Il reste à déterminer comment le faire et c'est ce que nous allons expliquer dans les deux prochains chapitres.

Dans le chapitre 4, nous verrons comment un seul nœud pourra être modélisé pour englober un fait et sa négation. Si les deux existent, ce même nœud pourrait être étiqueté comme étant une contradiction. Nous verrons aussi à la suite de De Kleer (p. 158, 1986) comment les états de chaque nœud peuvent être représentés par un ensemble minimal de bits. Ceci résout la question de McAllester (1978) et rend la normalisation inutile. Dans le chapitre 5, nous verrons comment la consommation en mémoire peut globalement être allégée encore en recourant au patron de conception poids plume.

Si l'on réussit à représenter un fait et sa négation par une même superstructure alors représenter la contradiction par un nœud séparé devient inutile. *Ceteris paribus* (toute chose étant égale par ailleurs), ce système ne pourra dans le pire des cas nécessiter que le tiers de nœuds requis par le SMVJ de Doyle (1979). Par exemple, si on crée 100 000 nœuds et si pour chacun on génère respectivement son nœud opposé, le système de Doyle générera 200 000 nœuds et un autre 100 000 pour représenter chaque contradiction pour un total de 300 000. Avec notre approche, 100 000 nœuds suffiront peu importe la situation.

Pour la non monotonie, Doyle (1979) utilise deux types de listes, une liste pour les *OUTs* et une liste pour les *INs*. Dans notre cas, une seule liste sera utilisée mais avec quatre types d'états possibles à propager. On peut propager un état *IN* ou *OUT* vers la partie négative ou positive d'un nœud ce qui donne au total quatre combinaisons possibles. Ceci évitera d'utiliser pour chaque justification deux listes au lieu d'une. Au niveau de l'ensemble du système, ceci réduira donc le nombre de ce type de listes par deux.

La consommation en mémoire n'est pas le plus grand handicap des systèmes de base que nous avons présentés. Tout système modélisé avec des structures mal définies devient difficile à faire évoluer, à décrire, à utiliser et surtout à maintenir. D'après nous, recourir en plus un paradigme procédural orienté liste n'est pas une stratégie appropriée. Tout ceci explique pourquoi Forbus et De Kleer (1993) trouvaient le NNMJTMS complexe à expliquer et à utiliser. D'après nous, il l'est à cause des stratégies de conception qui ont été adoptées alors.

Pour ces raisons, nous allons développer un SMVJNMN (NNMJTMS) en recourant plutôt à une approche orientée objet et une grande attention sera accordée à la modélisation et à la conception. Plusieurs patrons de conception seront alors utilisés dans le but de mettre l'aspect algorithmique en arrière plan. À notre connaissance, aucun SMV n'a été abordé selon cette approche et malgré l'abondance de la littérature concernant le sujet, nous n'avons trouvé aucun SMVJNMN fonctionnel ou opérationnel.

Une approche orientée objet nous permettra de modéliser le système au niveau des connaissances, de le concevoir avec plus de rigueur et de réduire le nombre de types de structures utilisées. Dans la littérature, nous avons remarqué que l'on confond souvent les entités nœuds avec leurs attributs. Forbus et De Kleer (1993), par exemple, parlent de nœuds contradiction, de nœuds prémisses et de nœuds assomptions mais aussi d'états assomptions et d'états prémisses. Pour nous, un nœud sera considéré comme une entité abstraite qui peut posséder autant d'attributs que nécessaire et nous allons éviter de confondre une entité avec ses propriétés. Une fois que cette mise au point est effectuée, nous n'aurons pas besoin de trouver des parades pour limiter l'espace du problème du SMV. Nous n'aurons pas besoin, par exemple, d'identifier expressément ce qu'est un nœud prémisse ni présupposer que les contradictions ne découlent que de faits incertains. Par conséquent, nous n'aurons pas besoin d'identifier expressément les assomptions mais nous allons garder cette fonctionnalité pour des raisons de compatibilité.

On peut se demander pourquoi cette stratégie n'a pas été adoptée auparavant. D'après nous, elle ne l'a pas été pour des raisons historiques. En effet, le développement des SMV a connu son apogée à la fin des années quatre vingt alors que le paradigme orienté objet ne s'est imposé que quelques années plus tard avec l'adoption d'UML (Booch *et al.* 1999). Au cours de cette période, de nouveaux formalismes, méthodologies et méthodes de conception ont fait leur apparition. On peut citer par exemple, le formalisme UML (Booch et *al.*, 1999), la méthodologie *RUP (Rational Unified Process*) et les patrons de conception (Gamma *et al*., 1995; Buschman *et al*., 1996). Toutes ces techniques ouvraient donc des opportunités qui permettent de concevoir des systèmes selon de nouvelles approches.

3.2 Méthodologie CommonKADS

D'après Schreiber *et al.* (p. 3, 2000), la méthodologie CommonKADS a été mise au point spécialement pour le développement des SBC étant donné que le concept de connaissance joue un rôle central dans ces systèmes. Pour ces auteurs, une connaissance n'est rien d'autre qu'une information à laquelle une compétence a été attachée alors qu'une information serait une donnée ou un simple signal qui aurait subi un traitement quelconque. Cependant, pour eux, ce qui caractérise le plus une connaissance c'est surtout sa capacité de générer une action et de nouvelles connaissances d'où la notion de compétence. Elle peut, par conséquent, être présentée sous la forme : *"Si conditions alors actions*". Pour ne pas se limiter à un paradigme déclaratif, Iglesias et *al.* (1997) avaient proposé une extension à CommonKADS qu'ils ont renommée alors MAS-CommonKADS. Selon eux, cette extension qui s'inscrit dans le paradigme objet conviendrait mieux au développement des systèmes multiagents car dans ces systèmes la communication et la coordination sont primordiales.

La méthodologie CommonKADS couvre l'ensemble du cycle de développement selon un processus itératif et les grandes phases du cycle de développement y ont été divisées en trois grandes catégories : le contexte, les concepts et les artéfacts. Ces catégories qui comportent un ou plusieurs "modèles" ont pour but d'expliquer le pourquoi, le quoi et le comment du système à développer (Schreiber *et al.*, 2000). La première partie porte sur la modélisation au niveau contextuel et comporte les trois modèles suivants : le "modèle de l'organisation", le "modèle de la tâche" et le " modèle des agents". Rappelons que notre système n'est pas développé pour une organisation donnée et le but de ce mémoire ne porte que sur la modélisation au niveau des connaissances ainsi que les phases subséquentes.

Les concepts sont spécifiés dans deux modèles : "le modèle des connaissances" et "le modèle de la communication". Les artéfacts sont décrits au moyen d'un seul modèle, "le modèle de conception" (figure 3.1).

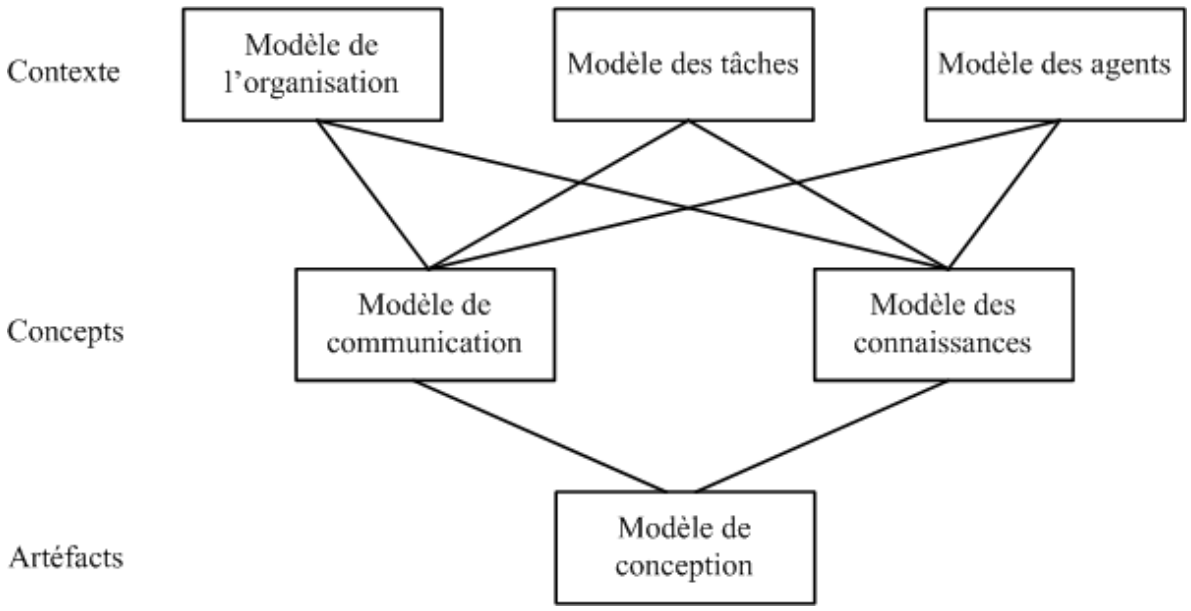


Figure 3.1 Les modèles de CommonKADS
Source : Schreiber *et al.* (p. 18, 2000)

Chaque modèle CommonKADS comprend plusieurs "feuilles de travail" qui doivent être complétées et qui passent en revue tous les aspects devant être couverts. Pour les points qui doivent être spécifiés, CommonKADS propose d'adopter au choix soit des spécifications textuelles soit des diagrammes en suivant le formalisme UML[14] (Schreiber *et al.*, p. 116, 2000).

3.2.1 Le modèle des connaissances

Appelé aussi modèle de l'expertise, le modèle des connaissances a pour but de décrire, à un certain niveau d'abstraction, l'ensemble des raisonnements du futur système. C'est au moyen de ce modèle que les connaissances requises pour que le système accomplisse ses tâches sont décrites. Schreiber *et al.* (2000) décomposent ce modèle en trois couches : la couche du domaine, la couche des inférences et la couche des tâches (figures 3.2).

La couche du domaine sert à spécifier les "concepts" et les connaissances du domaine de l'application et consiste en un schéma du domaine et une ou plusieurs bases de connaissances.

La couche des tâches spécifie les buts qui doivent être atteints pour résoudre un problème donné ainsi que l'usage qui est fait des connaissances décrites dans la couche du domaine.

La couche des inférences comporte l'ensemble des connaissances statiques qui peuvent être utilisées lors d'un processus de raisonnement et chaque processus peut être représenté par

[14] UML : *Unified Modeling Langage.*

des structures d'inférences. Ces structures consistent en inférences proprement dites, en rôles d'inférences et en fonctions de transfert (Schreiber *et al.*, 2000).

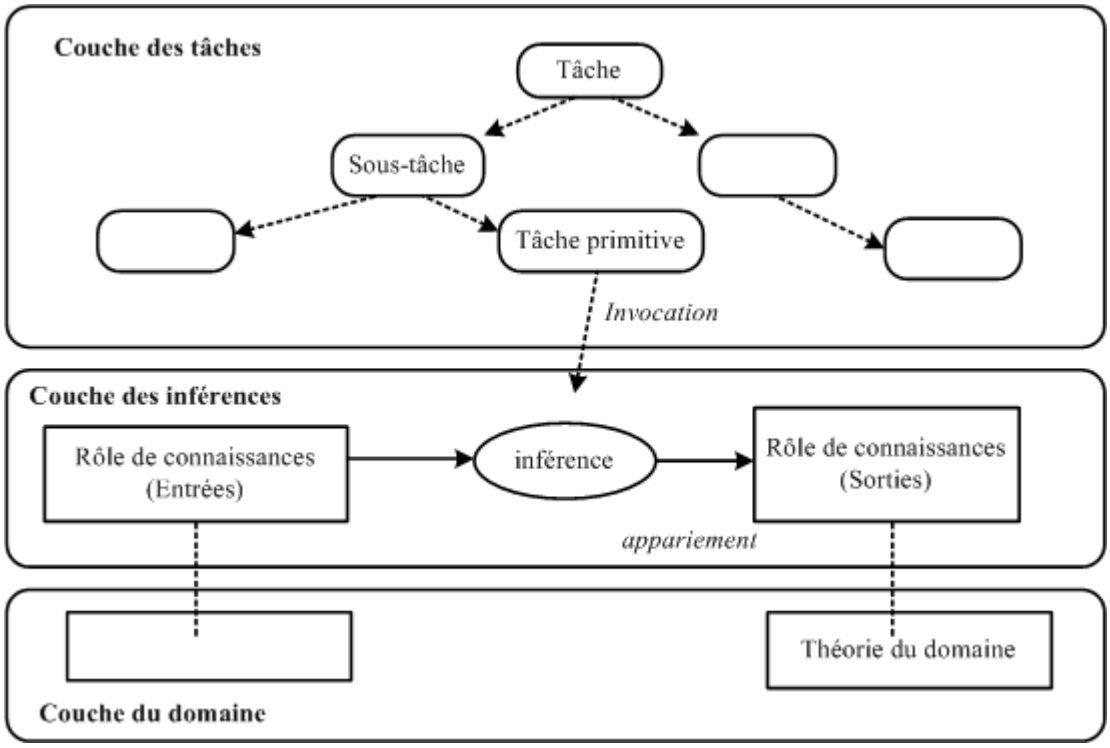


Figure 3.2 Modèle des connaissances
Source : Fensel *et al*. (p. 5, 1998)

3.2.1.1 La couche des tâches

Dans cette couche, les tâches sont spécifiées du point de vue du système à développer et elles correspondent à celles qui ont été identifiées dans le modèle organisationnel. Une tâche implique un ou plusieurs buts à atteindre et par conséquent nécessite une ou plusieurs méthodes de résolution de problèmes. Des "méthodes de tâche" doivent donc être définies pour décrire comment chaque tâche sera réalisée par décomposition en sous-fonctions. Ces sous-fonctions peuvent être décomposées à leur tour en d'autres tâches, en inférences et en fonctions de transfert (Schreiber *et al., 2000*).

D'après Schreiber *et al.* (p. 124, *2000*), puisque les tâches que les SBC peuvent accomplir sont peu nombreuses, elles sont par conséquent facilement classifiables et identifiables. Ils proposent un catalogue de tâches génériques ainsi que les gabarits qui leur correspondent. Ceux-ci peuvent être appliqués comme tels ou être adaptés selon les circonstances. Ils peuvent notamment être combinés puisque les tâches le peuvent aussi.

Le catalogue de Schreiber *et al.* (*2000*) contient dix types de tâches groupées en deux catégories : les tâches d'analyse et les tâches de synthèse. Une tâche d'analyse peut être une de classification, d'évaluation, de diagnostic, de surveillance *(monitoring)* ou de prédiction. Les tâches de synthèse concernent la conception (conception et configuration), la modélisation, la planification, la programmation *(scheduling)* ou l'affectation.

3.2.1.2 La couche du domaine

La couche du domaine contient toutes les informations et connaissances se rapportant au système à développer. Elle consiste essentiellement en un ou plusieurs schémas du domaine et une ou plusieurs bases de connaissances.

Le schéma du domaine porte sur la définition des types et décrit les structures du domaine. On y définit notamment les "concepts", les "relations" et les "types de valeur". Un schéma peut être spécifié textuellement ou à l'aide d'un diagramme de classes.

3.2.1.3 La couche des inférences

Selon Schreiber *et al.* (p. 104, *2000*), une inférence représente les pas de raisonnement primitifs. Elle utilise des connaissances contenues dans une quelconque base de

connaissances pour générer de nouvelles informations à partir de ses entrées dynamiques. Pour Schreiber *et al.* (*2000*), une inférence peut être décrite en termes d'entrées/sorties et des rôles qu'elle accomplit. Ces rôles, appelés "rôle de connaissances", peuvent être dynamiques (types) ou statiques (instances).

Le rôle de la couche des inférences consiste à décrire comment les structures du schéma du domaine sont utilisées pour exécuter un processus de raisonnement donné. Ce processus peut être décrit par des structures d'inférences qui font interagir les inférences, les rôles de connaissances ainsi que les fonctions de transfert. Ces dernières sont nécessaires pour faire communiquer le système avec le monde extérieur (Schreiber *et al.,* 2000).

Comme pour les tâches, CommonKADS offre par défaut un ensemble d'inférences, des gabarits de structures d'inférences ainsi que des fonctions de transferts. Ces fonctions ont été catégorisées sur la base de deux critères : qui possède l'initiative et qui détient les informations. À partir de ces deux dimensions, Schreiber *et al.* (p. 108, 2000) ont identifié les types suivants : "`Obtenir`", "`Recevoir`", "`Présenter`" et "`Fournir`".

3.2.2 Le modèle de communication

Le rôle du modèle de la communication est de spécifier le processus d'échange d'informations pour faciliter l'identification des transferts des connaissances entre agents lors de la réalisation de leurs tâches (Schreiber *et al.,* 2000). La notion de transaction joue donc un rôle central dans ce modèle. Schreiber *et al.* (2000) proposent de définir ce modèle en le décomposant en trois couches. La première a pour but de spécifier le plan de communication global. La seconde couche concerne la description de chaque transaction individuelle et la troisième porte sur les spécifications de l'échange d'informations pour chacune des transactions identifiées.

La spécification du plan de communication permet de décrire au moyen de diagrammes de dialogue les échanges entre différents agents pour chaque tâche. Cependant, comme ces diagrammes ne contiennent pas de mécanismes de contrôle, ces derniers doivent être spécifiés séparément. Il en est de même pour chaque transaction identifiée, le plan de communication dont elle fait partie, les agents impliqués, l'objet d'information et finalement les contraintes possibles (Schreiber *et al.*, 2000).

Les spécifications des échanges d'informations pour une transaction donnée doivent identifier les agents expéditeurs et receveurs, les items d'information, les mécanismes de contrôle ainsi que les spécifications du message. Celles-ci doivent indiquer le contenu, les références et le type de communication. Plusieurs types de messages possibles ont été identifiés par Schreiber *et al.* (p. 232, 2000) qui les classent selon l'intention des agents concernés et selon la source des initiatives. D'après ces auteurs ces types sont : "requête/proposer; requérir/offrir[15]; ordonner/accepter; refuser une délégation de tâche /adopter une tâche; demander/répondre; reporter et informer. "

D'après Schreiber *et al.* (2000), pour de simples échanges d'informations, les fonctions : "`demander`", "`répondre`, "`rapporter`" et "`informer`" peuvent être utilisées. L'adoption d'une tâche peut s'opérer par "`proposition`", "`offre`", "`agrément`" ou "`rejet`" alors qu'une délégation peut être effectuée par "`requête`" ou par "`ordre`".

3.2.3 Le modèle de conception

[15] Requête et requérir sont pris tels quels textuellement (Schreiber *et al.* p. 233, 2000) !

Selon Schreiber *et al.* (2000), le modèle de conception décrit l'ensemble de l'architecture logicielle du système en termes de sous-systèmes ainsi que les mécanismes de contrôle qui régissent leurs interactions. D'après ces auteurs, le processus de conception doit s'effectuer en suivant certaines étapes, en préservant les structures et en se conformant à des critères de qualité déterminés. Par préservation des structures, Schreiber *et al.* (2000) entendent que tout ce qui a été modélisé doit se retrouver nécessairement dans le modèle de conception puis dans l'implémentation. Comme critères de qualité, ils identifient la réutilisation du code, la maintenabilité, les explications et le support de CommonKADS.

D'après Schreiber *et al.* (2000), quatre étapes sont nécessaires pour mener à bien le processus de conception. Ces étapes portent sur la conception de l'architecture du système, l'identification de la plateforme cible, la spécification des composantes du système et la spécification de l'application avec l'architecture. La conception de l'architecture consiste à décomposer le système global selon la métaphore du Modèle-Vue-Contrôleur (MVC) en spécifiant le mode de représentation des connaissances, la disponibilité de librairies d'interfaces usager ("Vue") et le support des librairies CommonKADS.

Le Modèle-Vue-Contrôleur est un patron de conception de haut niveau qui porte sur l'architecture globale d'une application qu'il classifie en trois types objets selon leurs rôles et selon les liens de communication qui existent entre eux. Les objets modèles du MVC représentent l'expertise et les connaissances du domaine alors que les objets de vue représentent les éléments de l'interface graphique (fenêtres, boutons, listes déroulantes, champs textes, images, menus etc.). Les objets de type contrôleur servent à établir un lien entre les objets de type modèles et les vues et ce de façon plus au moins découplée.

Dans la version traditionnelle de ce patron, le modèle est relié avec la vue alors dans la version plus récente, le contrôleur joue un rôle de médiateur entre les deux (figure 3.3) . Cette dernière version est appelée aussi patron Modèle-Vue-Présentateur *(Model-View-*

Presenter). Le MVC est généralement décomposé en patrons plus élémentaires (Gamma et *al*. 1995)

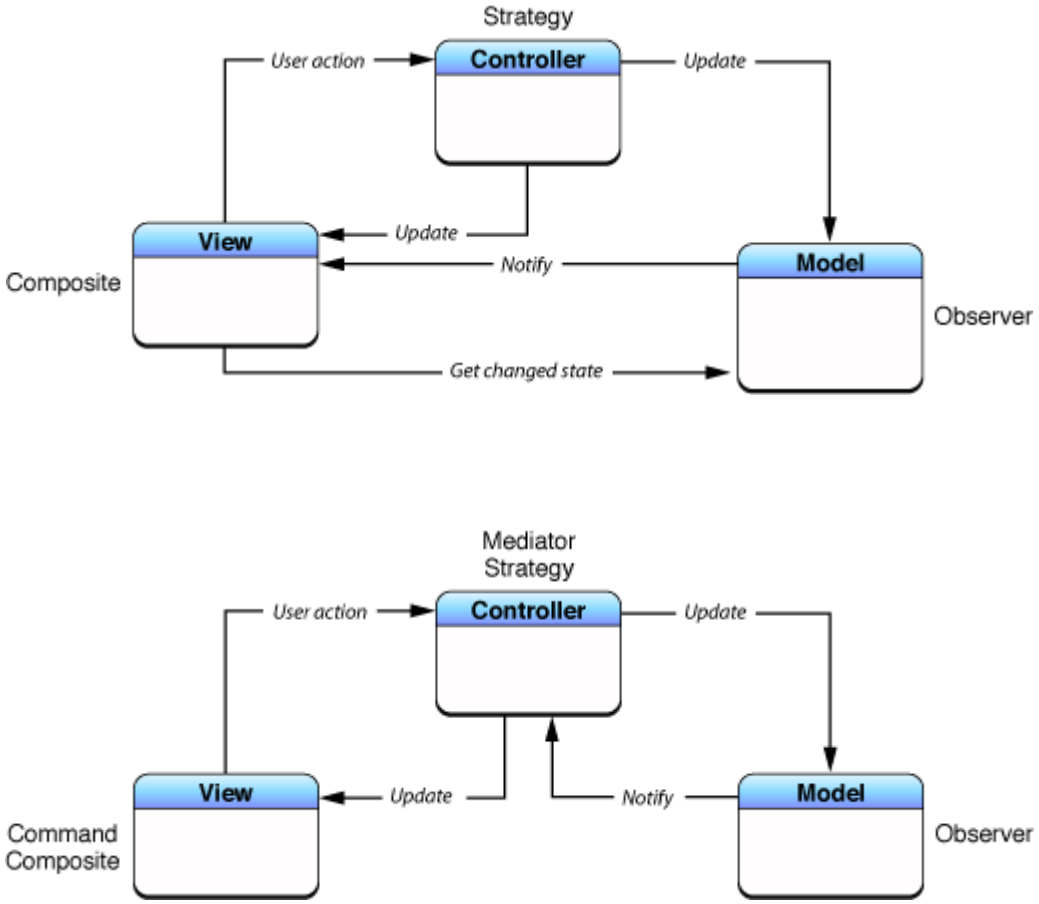


Figure 3.3 Version traditionnelle et version récente du MVC
Source : Apple (2008)[16]

Selon l'approche CommonKADS, la partie "Modèle" doit contenir les structures de contrôle pour les méthodes des tâches et associer des méthodes aux inférences. C'est à ce niveau aussi que le format de représentation des bases de connaissances, la façon de manipuler ces bases, les rôles statiques ainsi que les rôles dynamiques nécessaires sont définis (Schreiber *et al*. p. 294, 2000) . Ces derniers peuvent être des listes, des collections ou n'importe quel

16 http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/chapter_5_section_4.html#//apple_ref/doc/uid/TP40002974-CH6-SW1

type de conteneur (et il en existe une multitude[17]). Ils peuvent être ordonnés, accepter des doublons, avoir des méthodes, etc.

La partie "Contrôleur" peut être associée au modèle de la communication. Ce dernier doit servir à déterminer entre autres les divers types d'interactions du système, par exemple si être guidé par évènements, s'il doit être concurrentiel ou non et ainsi de suite (Schreiber *et al.*, p. 294, 2000).

D'après Schreiber *et al.*, (p. 294, 2000), la manière dont les différentes parties du MVC doivent être articulées et la plateforme visée déterminent la conception du système ainsi que sa complexité. Ils suggèrent donc de recourir à des ouvrages spécialisés ainsi qu'aux techniques de conception utilisées en génie logiciel puisque en ingénierie des connaissances les moyens sont limités.

3.3 Récapitulation

Dans ce chapitre, nous avons présenté les points problématiques des principaux types de SMV. Ceux-ci ont été abordés selon un même paradigme orienté liste (ceux décrits par Forbus et De Kleer, 1993). À notre avis, un tel paradigme qui était à la mode à une certaine époque n'est pas forcément le moyen idéal pour concevoir et développer un système aussi complexe qu'un SMV. En effet, ce paradigme impose des structures de données qui ne permettent pas de modéliser un système à un haut niveau d'abstraction. Ces limites peuvent constituer un frein à la façon dont un problème peut être posé et résolu. Par conséquent, elles sont responsables de la complexité des systèmes obtenus.

[17] Voir par exemple la liste des conteneurs STl à l'adresse : http://www.sgi.com/tech/stl/table_of_contents.html

En adoptant un paradigme orienté objet, nous proposons de développer un SMV selon une approche qui, à notre connaissance, n'a pas été explorée. Ce choix va nous permettre d'appliquer les dernières techniques du génie logiciel et de l'ingénierie des connaissances et développer un SMV qui est conforme aux pratiques actuelles.

Nous avons exploré l'utilisation de CommonKADS pour développer ce système en tant qu'un SBC. Nous avons donc exposé les différents modèles de cette méthodologie mais sommairement. Cependant, dans les prochains chapitres nous aurons l'occasion de les détailler puisque nous allons compléter toutes les feuilles de travail prévues par cette méthodologie.

Chapitre 4 : Modélisation

Dans les chapitres précédents, nous avons décrit les principaux types de SMV existants, leurs limites et les éléments de solutions envisagés. Le SMV que nous allons modéliser le sera surtout à partir des trois types de SMV qui ont été décrits dans le chapitre 2.

Dans le modèle des connaissances, nous passerons en revue les différentes couches de ce modèle et un schéma du domaine sera proposé. Ce schéma sera progressivement raffiné au cours du prochain chapitre portant sur la conception. Comme le recommandent (Schreiber et *al.*, p. 273, 2000), le schéma du domaine complet préservant toutes les structures sera alors proposé et c'est à partir de ce schéma que le système final sera implémenté.

La communication est une activité centrale pour un SMV. Elle l'est aussi bien pour la réalisation des tâches du système lui-même qu'elle l'est pour la synchronisation et l'échange de services avec le système client. Le modèle de la communication sera présenté immédiatement à la suite du modèle des connaissances.

Les modèles de CommonKADS sont accompagnés par des feuilles de travail. Les feuilles relatives à chacun des modèles présentés ici figurent en annexes.

4.1 Modélisation et "niveau des connaissances"

Newell (1982) a introduit le concept "niveau des connaissances *(The knowledge level)* pour décrire comment la modélisation doit être effectuée à un certain niveau d'abstraction et

comment elle doit l'être indépendamment de tout type d'implémentation. Newell (1982) précise qu'au niveau des connaissances, seul ce qu'un agent connaît et comment ses buts sont atteints doivent être spécifiés selon (principe de rationalité). Les détails quant à eux, doivent être laissés pour la phase d'implémentation, celui des symboles.

La méthodologie CommonKADS propose une suite de modèles que Schreiber et *al.* (pp. 18-19, 2000) classent en trois catégories. La première porte sur la modélisation au niveau organisationnel (organisation, tâches, agents) c'est-à-dire d'un point de vue managérial ou "administration des affaires" (Schreiber et *al.*, p. 46, 2000). La seconde concerne l'aspect conceptuel et se situe au niveau des connaissances et celui de la communication. Finalement, la troisième porte sur les artéfacts. Pour les spécifications Schreiber et *al.* (2000) proposent de modéliser un système textuellement ou en adoptant le formalisme UML auquel ils ont ajouté quelques diagrammes qui leur sont propres. On peut citer entre autres les diagrammes des structures d'inférences et ceux des tâches.

En 1995, Booch, Rumbaugh et Jacobson proposèrent le formalisme UML avec sa suite de diagrammes pour modéliser un système à divers niveaux d'abstraction (Booch et *al.* 1999). Chaque type de diagrammes convient à la modélisation d'un besoin donné. Le niveau le plus haut pour les diagrammes de classes permet de décrire des concepts par des classes (vides) où seul le nom est généralement requis. Si les noms de ces classes correspondent aux concepts du domaine, il s'agit cependant de noms propres qui doivent être préservés lors des étapes ultérieures. Schreiber et *al.* (2000) parlent alors de "préservation de structures" et insistent sur ce principe qui est primordial en vérification et validation. Dans ce domaine, Preece (2001) précise que ce qui a été spécifié doit nécessairement correspondre à ce qui a été implémenté sinon le système est incomplet ou faux. Par conséquent, nous allons nous conformer rigoureusement à ces principes.

4.2 Le modèle des connaissances

Le modèle des connaissances de CommonKADS se divise en trois couches appelées catégories des connaissances : la couche du domaine, la couche des tâches et la couche des inférences. Dans les points qui suivent, nous allons commencer par la couche du domaine. Nous allons identifier les concepts et les types de valeur nécessaires puis proposer un schéma du domaine. Dans la couche des tâches, nous allons voir comment la tâche *"Maintien de vérité"* sera décomposée. Nous verrons ensuite quelles sont les sous-tâches et les méthodes de tâches associées ainsi que les inférences et les fonctions de transfert qui composent le tout.

Comme le suggèrent Iglesias et *al.* (1997) nous allons adopter une approche orientée objet dès le début de la conception afin de concevoir un système multiagent où chaque entité sera munie de compétences spécifiques lui permettant de contribuer à l'accomplissement de chacune des sous-tâches identifiées. Schreiber et *al.* (pp. *119-120*, 2000), nous semblent ambigus sur ce point. Ils affirment que leur approche se situe entre l'approche orientée objet et l'approche orientée donnée :

> *"O-O people state that the information structure (data) are usually much more stable (and thus likely to change) than the function of system. Therefore, in O-O analysis, the data is the entry point for modelling an application domain. In the CommonKADS we take a position between the two approaches."*

Cependant, ils finissent par adopter une approche objet comme ils l'affirment et je cite:

> *"However, realizing this architecture might indeed be easier in an O-O programming language…Integration and/or coordination with other systems becomes more important in the design stage. As O-O is the prevailing paradigm in the contemporary software engineering, an O-O based design of a knowledge-intensive system will make integration easier"* (Schreiber et al., pp. 277-279, 2000).

Dans le chapitre 12 sur l'implémentation, c'est cette approche qu'ils appliquent comment ils le soutiennent :

> *"Using the O-O primitive discussed above, we implemented the CommonKADS architecture described in the previous chapter"* (Schreiber et *al.,* p. 297, 2000)".

4.2.1 La couche du domaine

Dans cette partie, nous allons identifier les différents concepts relatifs au SMV et les représenter d'un point de vue objet. Nous verrons ensuite les types de valeur qui leur seront associés et nous proposerons ensuite un schéma du domaine. Ce schéma sera raffiné progressivement dans le modèle de conception et sera appliqué tel quel en phase d'implémentation et ce, conformément au principe de "préservation des structures" suggéré par Schreiber et *al.* (p. 273, 2000).

4.2.1.1 Les concepts et leurs relations

La construction d'un réseau est effectuée à partir de l'ensemble des raisonnements qui parviennent au SMV sous forme d'un ensemble d'opérateurs logiques et d'opérandes. Chaque opérande peut être considéré comme un littéral et un ensemble d'opérandes constitue une justification comme le décrivent Forbus et De Kleer (1993). Doyle (1979) parle de justification dans la mesure où tout ensemble de faits peut servir de "raison" qui sert de support à un autre fait.

Un réseau est constitué par un ensemble de nœuds, chacun représentant un littéral ou une justification. L'entité nœud constitue le premier concept identifiable qui peut être soit un

littéral soit une justification. Les Littéraux et les justifications sont des nœuds et constituent les concepts fondamentaux et élémentaires du système. Une relation de type "est un" relie donc les deux derniers concepts au premier.

Le SMV est responsable de construire le réseau de nœuds et de le gérer. Il constitue donc un autre concept clé qui peut être représenté par une entité à part. Nœud, littéral et justification seront nommés[18] arbitrairement : `"DNode"`, `"DJustification"`, `"DLiteral"`. Le concept associé au SMV lui-même sera appelé *`"DTMSproxy"`*. Dans le chapitre 6 portant sur la conception, nous expliquerons le choix des noms ainsi que celui de toute autre valeur définie dans ce chapitre et qui fera partie du schéma du domaine.

Justifications et littéraux partagent au moins un certain nombre d'attributs notamment le fait d'avoir un nom (ou contenu), un ensemble de machines d'états et une paire d'agrégations. Une justification doit posséder une liste pour retracer ses éléments et une autre pour retracer ses conséquents. Inversement, un littéral doit posséder une liste pour retracer ses précédents et une autre pour retracer les justifications qui le contiennent. En généralisant, chaque nœud devra donc être muni d'une paire de collections. Nous avons ici deux relations d'agrégation.

4.2.1.2 Les valeurs types et autres relations

Schreiber et *al.* (p. 93, 2000) indiquent que les attributs (valeurs et valeurs typées[19]) de chaque concept doivent être spécifiés. Par conséquent, le contenu d'un nœud sera représenté par une variable appelée "*`content`*" et qui consiste en une simple chaîne de

[18] Il s'agit de noms propres.
[19] Valeur typées pour "*value types*". Une valeur typée est une instance qui n'est pas référencée. http://msdn2.microsoft.com/en-us/library/s1ax56ch(VS.80).aspx

caractères de type *"string"*. Pour stocker un ensemble d'états, une variable à huit bits appelée "*states*" sera nécessaire. Quatre bits de cette variable seront réservés pour la partie positive et quatre autres pour la partie négative si elle existe. Pour chacune des parties, le premier bit indiquera si l'on est en présence d'une assomption on non. Le second indiquera si cette partie est active ou non (*IN* ou *OUT*). Le troisième sert pour marquer un nœud et éviter la circularité. Le dernier bit est pour indiquer simplement si la partie en question existe ou non (figure 4.1).

À la création d'un littéral *P* par exemple, son contenu sera mis à "P" et certains de ses états mis à un. Plus exactement, en commençant le compte à partir du bit le moins significatif, le deuxième bit ainsi que le quatrième seront mis à un. Si par la suite la négation de *P* doit être ajoutée alors les bits six et huit seront mis à un aussi. Dans ce cas, puisque les deux parties existent et puisque les deux sont actives *(IN)* en même temps alors le nœud en question est en contradiction. Pour détecter une telle contradiction, à chaque fois que l'un de ces bits est modifié cette vérification doit être effectuée.

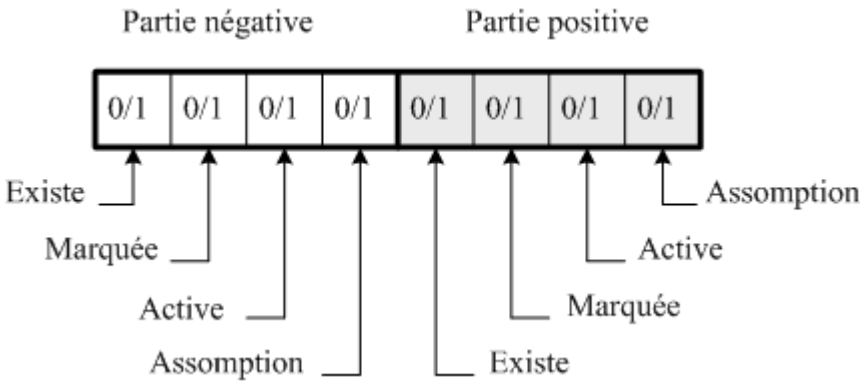


Figure 4.1 : États d'un littéral

Un nœud justification est muni des mêmes structures mais n'a besoin que de deux bits pour gérer ses états (figure 4.2). Une justification peut être active ou non et, elle peut être marquée ou non. Une justification est marquée lors du processus d'explications pour éviter les circularités.

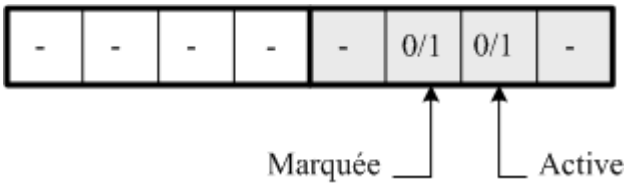


Figure 4.2 : États d'une justification

Pour manipuler tous ces attributs, des valeurs types ont été définies et sont détaillées dans la page 3 de l'annexe 2. Certaines de ces valeurs seront décrites textuellement dans le paragraphe 4.3.

4.2.1.3 Le schéma du domaine

Nous venons d'identifier trois types de relations structurelles reliant les nœuds entre eux. La classe *DTMSproxy* doit à son tour gérer aussi bien des littéraux que des justifications. Une paire d'agrégations sera donc nécessaire pour chaque catégorie de nœuds. Une seule agrégation aurait pu suffire puisqu'un objet *DTMSproxy* gère les nœuds indépendamment de leur sous-type. Cependant, comme le nombre de justifications sera en général moindre que celui des littéraux et pour accélérer les mécanismes de recherche, nous avons décidé de les séparer.

À partir des concepts et des relations identifiés, il est d'ores et déjà possible de présenter le schéma de domaine (figure 4.3) mais auparavant, une mise au point est nécessaire. Les agrégations dont il est question ne peuvent pas être de simples listes. En effet, dans certains cas un nœud doit maintenir une liste et des éléments et de l'état à propager. Dans d'autres cas, la liste doit contenir et les éléments et leur signe. Les listes doivent donc être

des conteneurs associatifs[20]. Conformément aux spécifications de CommonKADS (Schreiber et *al.*, p. 286, 2000), ces conteneurs seront présentés dans la section 5.10.4 du modèle de conception.

D'après ce schéma, une instance de la classe `DTMSproxy` pourra contenir deux ensembles de *n* nœuds qu'elle pourra manipuler indépendamment de leur sous-type. Il en va de même pour n'importe quel objet de type `DNode` qui pourra contenir zéro ou plusieurs objets du même type.

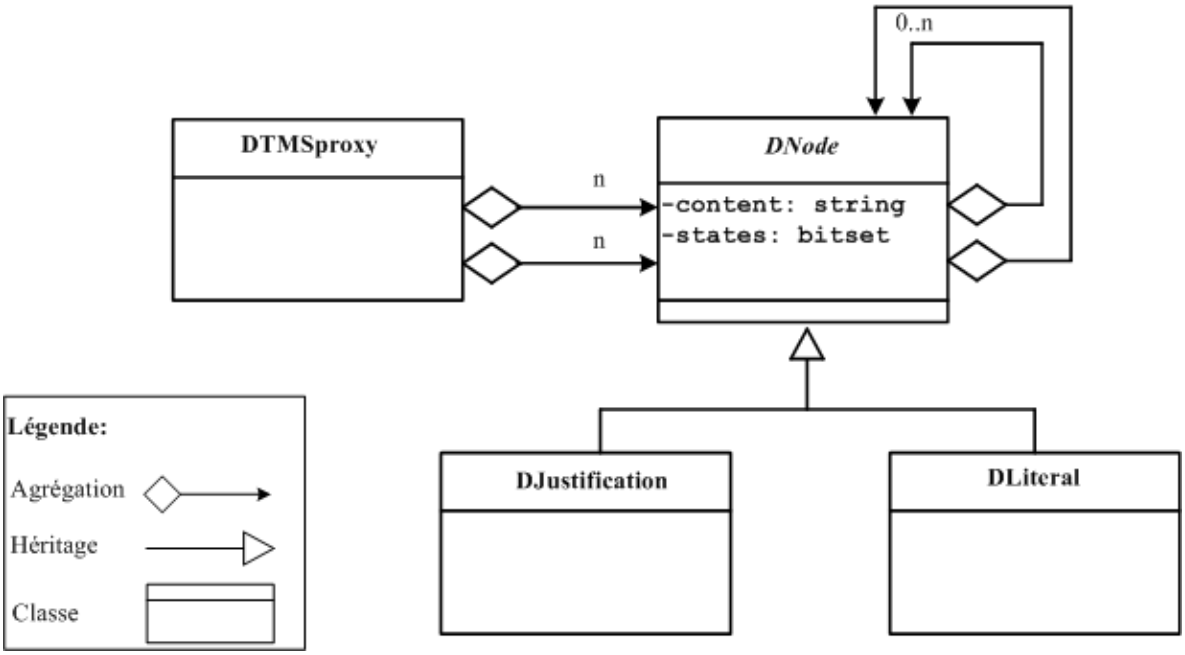


Figure 4.3 : Schéma du domaine

[20] Un objet destiné à stocker d'autres objets et d'effectuer des opérations sur eux. Les conteneurs associatifs sont appelés aussi dictionnaires ou *hash tables*. Pour plus d'informations visiter le site : http://www.sgi.com/tech/stl/table_of_contents.html

4.2.1.4 La base de connaissances

Les connaissances qu'un SMV doit représenter sont très limitées et déterminées et le type de règles qui en découle l'est par conséquent. Un SMV consiste en un ensemble de nœuds et nous allons considérer l'entité nœud comme un agent qui va encapsuler toutes les connaissances du système. Autrement dit, toutes les connaissances que le SMV doit reproduire seront déléguées au niveau de cette entité. Chaque objet nœud sera en mesure de gérer un ensemble de machines d'états et propager ces changements vers ses accointances. Il sera aussi en mesure de fournir des explications et demander à d'autres nœuds de se comporter de façon similaire. Ces connaissances sont spécifiées en UML au moyen de cinq machines d'états[21] alors que les interactions le sont par des diagrammes d'activité. Les machines d'états décrivent comment le passage d'un état à un autre peut être effectué selon des relations de type *antécédents/conséquents* et correspondent à la définition que donnent Schreiber et *al.*, (p. 4, 2000) à la notion de connaissance.

La figure 4.11 est une représentation générique du type de machines d'états que nous utilisons. Les machines d'états détaillées figurent en Annexe 3 puisqu'elles ne peuvent pas être complétées à ce stade. En effet, la méthodologie CommonKADS ne permet de décomposer les inférences en méthodes qu'en phase de conception et les spécifications de ces machines requièrent celles de ces méthodes (Schreiber et *al.*, p. 286, 2000). Les interactions inter nodales quant à elles, sont spécifiées par les diagrammes d'activités des figures 4.5 à 4.7.

Les rôles dynamiques que le système prend en entrées et en sorties se limitent à de simples chaînes de caractères identifiant un nœud et constituant son contenu. Par la suite, tout objet nœud sera manipulé par référence alors que les états sont définis par les valeurs types et sont spécifiés en annexe 2. Les types des rôles statiques sont décrits en détail lors de la

[21] Pour plus d'informations, on peut consuler l'excellent article publié sur le *web* par Martin R-C. (1998). http://www.objectmentor.com/resources/articles/umlfsm.pdf

phase de conception comme le recommande Schreiber et *al.*, (p. 286, 2000) et les spécifications complètes figurent dans le schéma du domaine complet en annexe 2.

Notons qu'il est difficile de suivre la méthodologie CommonKADS d'une façon linéaire surtout si le système à développer est réactif. Concernant ce point, plusieurs auteurs (Iglesias et *al.*, 1997; Chaib-draa et *al.*, 2001) ont d'ailleurs souligné les limites de cette méthodologie et ont opté MAS-CommonKADS. D'après eux, cette extension convient mieux pour la modélisation de systèmes où les interactions et la communication sont centrales. Dans notre cas, nous allons simplement adopter un paradigme objet. Nous allons conformer le plus possible avec le formalisme UML et modéliser le système à un haut niveau d'abstraction.

4.2.2 La couche des tâches

Les fonctionnalités que le système doit accomplir sont les mêmes que celles que nous avions identifiées pour les autres SMV et la même tâche *"Maintien de vérité"* sera reprise ici. Dans les points qui suivent, nous allons voir de quel type de tâche il s'agit et ensuite comment cette tâche sera décomposée en sous-tâches, en inférences, etc.

D'après le catalogue de CommonKADS, la tâche *"Maintien de vérité"* est une tâche de diagnostic dans la mesure où elle permet de détecter certaines anomalies et de remonter à leurs causes. Elle peut être considérée aussi comme une tâche de surveillance puisqu'elle permet d'effectuer cette détection en cours d'exécution du système client.

La figure 4.4 montre comment cette tâche est décomposée[22] en trois sous-tâches : *"Construction du réseau"*, *"Changement d'états"* et *"Génération d'explications"*. Ces sous-tâches sont elles-mêmes décomposées en fonctions de transfert et en inférences.

À chacune des trois sous-tâches identifiées, une méthode de tâche sera associée. Pour la construction du réseau, la méthode "*Générer-Insérer*" décrit la façon dont un nœud sera généré et inséré dans le réseau. La détection des contradictions est effectuée par la méthode "Modifier-Propager-Détecter". Elle est appelée à chaque fois que l'état d'activité d'un nœud ou que la structure du réseau a été modifiée. Une méthode "*Expliquer*" a été associée avec la sous-tâche "*"Génération d'explications"*.

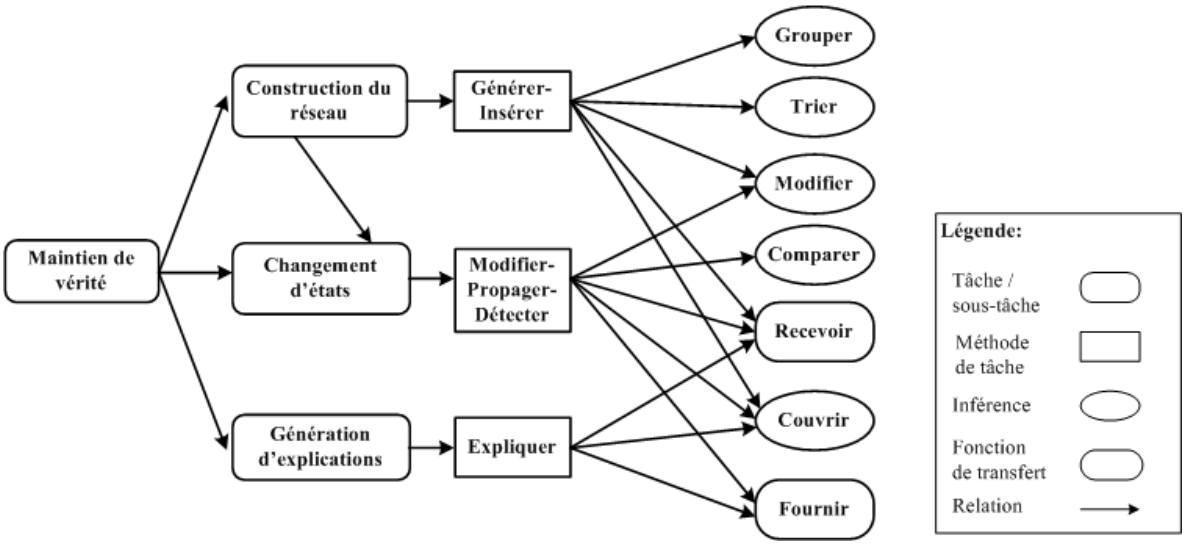


Figure 4.4 : Décomposition des Tâches du système

Comme nous avons adopté une approche objet où chaque entité joue le rôle d'un agent, la tâche *"Maintien de vérité"* sera réalisée par la classe *DTMSproxy* en collaboration

[22] Elle ne montre pas les relations de type "est un"

avec chacun des types de nœud identifiés. Les fonctions de transfert seront du ressort de la classe *DTMSproxy* qui aura l'exclusivité de communiquer avec tout système client.

La méthode de tâche *Générer-Insérer* a pour rôle de créer et de configurer un nœud puis l'insérer dans le réseau (figure 4.5). Étant donné qu'un nœud littéral est une entité duale qui stocke aussi bien la partie positive que la partie négative, une demande d'ajout d'un nœud ne se traduit pas forcément par la création d'un nouvel objet. Si le nœud existe déjà, il est simplement reconfiguré en conséquence. Le système s'assure d'abord que la partie n'existe pas et si tel est le cas, il ne fait que mettre à un (la valeur 1) les états appropriés. Lorsque le nœud en entier n'existe pas, l'objet *DTMSproxy* le crée, le configure et l'ajoute dans l'une de ses deux listes.

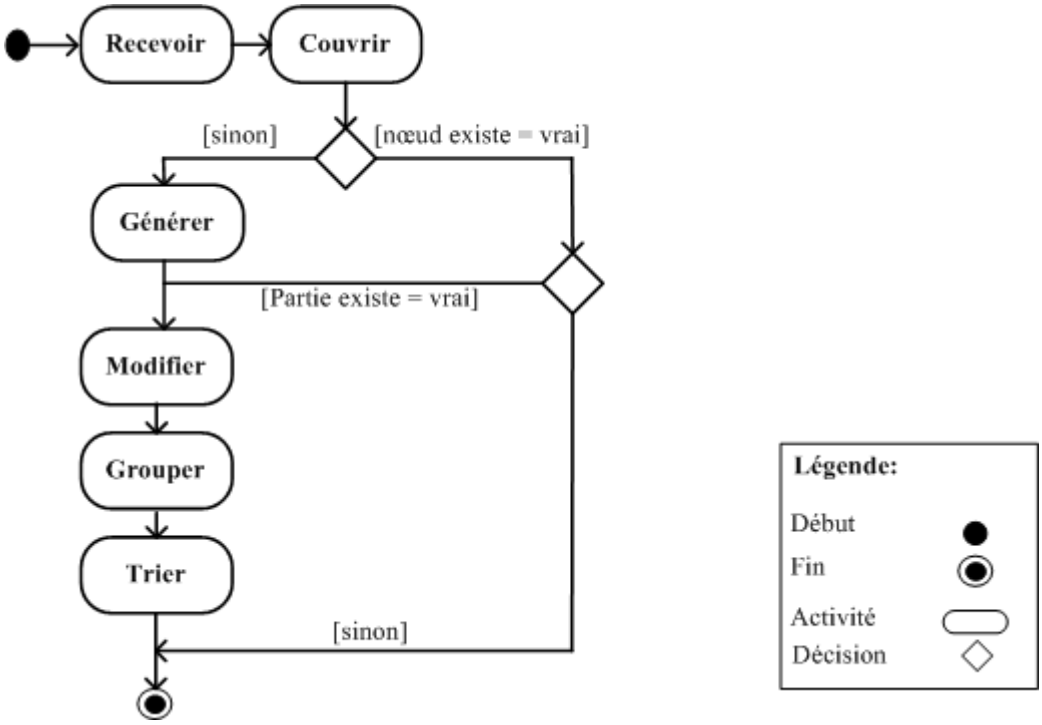


Figure 4.5 : La méthode de tâche "Générer-Insérer"

Si un nœud doit être ajouté dans l'une des listes d'un autre nœud, cette tâche est déléguée aux nœuds eux-mêmes. Ainsi, une justification doit être capable d'insérer un littéral en tant qu'élément ou en tant que conséquent et lui déléguer le reste de la tâche. Le littéral en

question ajoute alors cette justification en tant que possesseur ou en tant qu'antécédent selon le cas. Ces doubles opérations sont faites au niveau de la classe nœud indépendamment des sous types.

La construction du réseau s'effectue de la façon suivante : la classe *DTMSproxy* reçoit une requête pour créer un nœud. Elle parcourt alors l'une de ses listes pour localiser le nœud en question et vérifier qu'il n'existe pas déjà. S'il n'existe pas, alors le nœud est créé, ses états modifiés puis il est inséré dans l'une des deux listes selon un ordre établi. Si le nœud existe, alors ses états peuvent être modifiés en fonction de la requête. Toute modification d'états d'un nœud requiert la *"Détection des contradictions"*. La construction du réseau consiste aussi à relier des justifications et des littéraux. Comme nous l'avons décrit, cette activité est déléguée à ces entités.

La sous-tâche *"Changement d'états"* est effectuée selon les mêmes mécanismes de délégation (figure 4.6). Un objet *DTMSproxy* reçoit une requête et localise le nœud dont l'état doit être modifié et lui délègue le reste de la tâche. Le nœud en question modifie ses états et éventuellement relaie le message aux nœuds qui dépendent de lui. Plus précisément, un littéral modifie ses états et avertit les justifications auxquelles il appartient. Celles-ci peuvent modifier leurs états et à leur tour, avertir leurs conséquents. Le processus de propagation continue ainsi tant que chaque entité ne parcourt pas l'ensemble des éléments de l'une de ses listes.

Chaque littéral qui modifie son état d'activité, vérifie s'il n'est pas en contradiction, s'auto marque, propage la modification et annule le bit qui a été marqué. Si une contradiction est détectée, le processus est interrompu et la contradiction retournée récursivement dans le sens inverse de la propagation. Quand l'objet *DTMSproxy* reçoit une contradiction, il la signale au système client.

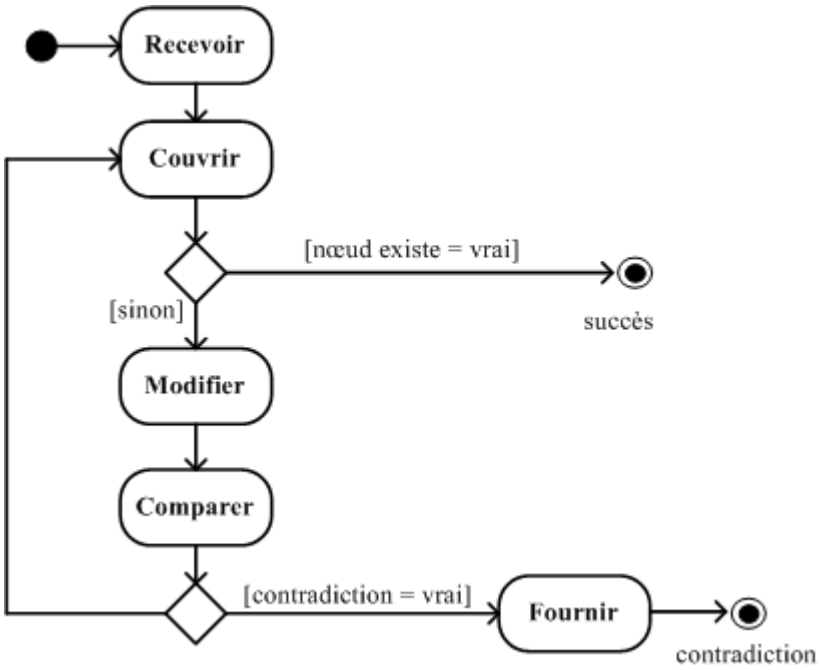


Figure 4.6 : La méthode de tâche "Modifier-Propager-Détecter"

La sous-tâche "*Expliquer*" est effectuée selon la même logique (figure 4.7). Le système reçoit une requête et localise le nœud à être expliquer puis lui envoie un message en conséquence. Le nœud en question s'auto explique et peut éventuellement demander aux nœuds concernés de s'expliquer. Plusieurs modes d'explication sont possibles : il est possible de suivre un raisonnement à la trace, de nœud en nœud, de façon assez détaillée, etc. Une explication peut être ponctuelle et ne porter que sur une justification ou sur un littéral donné. Parfois, seules les conclusions d'un raisonnement (sauter des prémisses aux conclusions) ou inversement, les prémisses d'une conclusion peuvent suffire. La sous-tâche "*Expliquer*" est initiée par *DTMSproxy*.

Une explication ponctuelle doit pouvoir fournir le contenu d'un nœud, son signe, son état d'activité et éventuellement mentionner si la partie du nœud en question est une assomption ou non. Une suite d'explications ponctuelles constitue une explication globale et cette explication peut être affichée en détail ou non.

Lors d'un processus d'explication, chaque partie d'un nœud qui a été expliquée est marquée pour ne pas être expliquée une seconde fois. À la fin du processus, tout état marqué d'un nœud est remis à l'état initial.

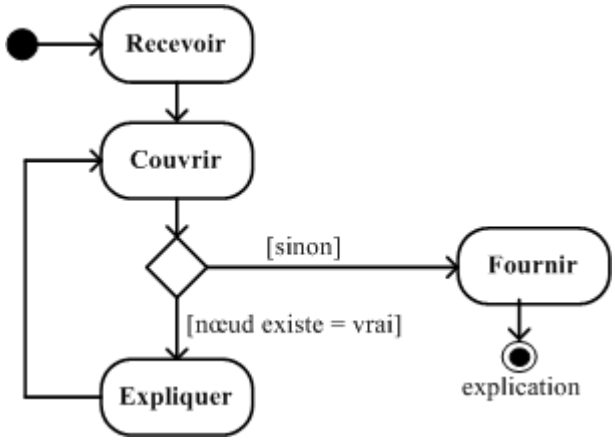


Figure 4.7 : La méthode de tâche "Expliquer"

Dans un contexte multiagent, l'exécution d'une tâche prend une autre dimension que nous devons préciser immédiatement. Chacune des inférences décrites dans les figures[23] 4.5 à 4.7 va être exécutée au niveau de chaque agent identifié (concept) et elle va l'être virtuellement. Par virtuel, nous entendons ici surcharge de fonctions par liens dynamiques comme cela se pratique dans les approches à objets. Dans le modèle conception, les méthodes associées aux inférences doivent être spécifiées (Schreiber et al, 2000) et ce sont ces méthodes qui seront surchargées. La communication entre agents sera par conséquent essentielle dans la réalisation de chaque tâche.

[23] D'après Schreiber *et al.* (p. 116, 2000), les spécifications peuvent être effectuées en pseudo-code ou par des diagrammes UML.

4.2.3 La couche des inférences

Pour construire un réseau, le système doit créer et grouper des nœuds, les trier selon leur ordre d'arrivée puis modifier des états. Chacun de ces nœuds correspond à un fait transmis par le système client et que le système "reçoit" par une fonction de transfert. Le reste du processus a été décrit dans la méthode de tâche ***Générer-Insérer***. À partir du catalogue de CommonKADS, les inférences "***Grouper***", "***Trier***", "***Modifier***" et "***Couvrir***" ainsi que la fonction de transfert "***Recevoir***" ont été donc sélectionnées pour la réalisation de la sous-tâche ***"Construction du réseau"*** (figure 4.8).

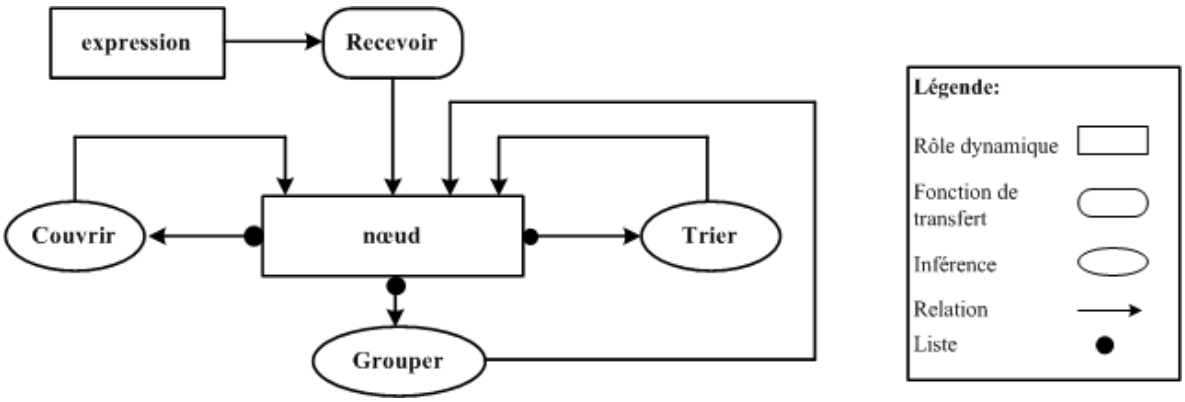


Figure 4.8 : Structure d'inférences pour construire un réseau

Pour construire un réseau, le système a besoin d'une expression qui décrit le type de nœud (contenu, signe, etc.). Celui-ci est alors ajouté dans une liste de nœuds qui doit être "couverte" et préalablement et triée. La figure 4.8 montre comment toutes les inférences sont centrées autour d'un nœud mais sans pouvoir rendre compte de l'aspect dynamique de l'ensemble. En effet, comme le mentionnent Schreiber *et al.* (p. 180, 2000), les structures d'inférences sont une représentation essentiellement statique d'un raisonnement et ne peuvent convenir pour la représentation d'un raisonnement dynamique. Ils conseillent par conséquent d'être plus flexible avec les rôles joués par ces structures et de les interpréter de manières élargies (Schreiber et *al.*, p. 180, directive 7-25, 2000).

La sous-tâche *"Changement d'états"* s'effectue à chaque fois que le système reçoit une demande pour modifier l'état d'activité d'un nœud. À chaque modification, le système doit comparer certaines valeurs pour vérifier si une contradiction est survenue. Si tel est le cas, il doit le signaler en précisant dans quel contexte cela s'est produit. Une fois l'état d'activité d'un nœud modifié, le système doit la propager et couvrir les branches appropriées du réseau. Les fonctions de transfert "*Recevoir*" et "*Fournir*" ainsi que les inférences "*Modifier*", "*Comparer*" et "*Couvrir*" ont été identifiées pour la réalisation de cette sous-tâche (figure 4.9).

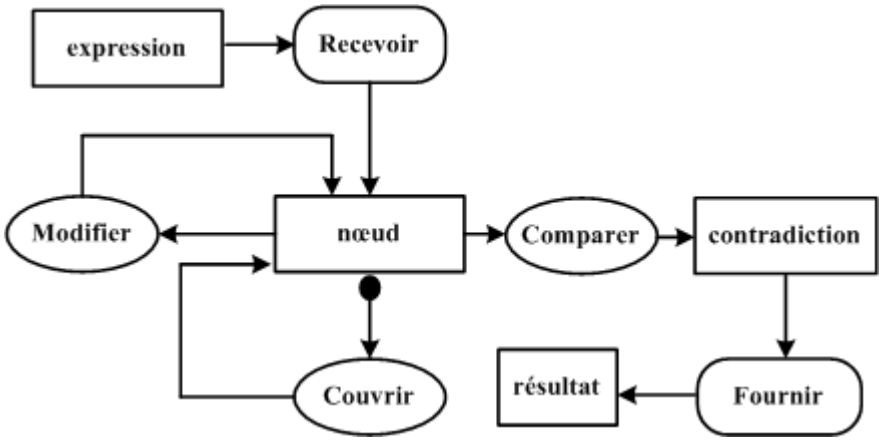


Figure 4.9 : Structure d'inférences pour modifier des états

Pour fournir des explications, le système doit obtenir des informations sur le nœud à expliquer et fournir des résultats en retour. Pour accomplir cette dernière opération, le système doit couvrir une partie du réseau de la même manière qu'il le fait pour propager des états. Les fonctions de transfert "*Recevoir*" et "*Fournir*" ainsi que l'inférence "*Couvrir*" ont été donc identifiées pour remplir cette troisième sous-tâche (figure 4.10). En couvrant les parties du réseau, le système stocke les explications partielles dans un différentiel dont le contenu est fourni à l'agent client sous forme d'une explication finale.

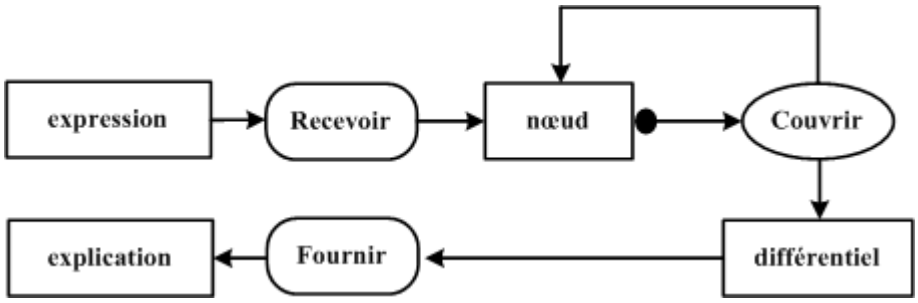


Figure 4.10 : Structure d'inférences pour générer des explications

L'ensemble de ces inférences et de ces fonctions de transfert provient du catalogue de CommonKADS. Nous verrons dans le chapitre 7 conformément aux spécifications de cette méthodologie (Schreiber et *al.*, p. 285, 2000) quels seront les appels de méthodes *(Method calls)* qui leur seront associés.

4.3 Le modèle de la communication

Dans ce modèle, nous allons présenter le plan de communication, les transactions entre agents et les détails des échanges d'informations pour chacune des transactions définies.

4.3.1 Le plan de communication

Comme nous l'avons décrit, une tâche est distribuée par décomposition entre différents types d'agents. La communication joue donc un rôle central dans la réalisation de chaque tâche et elle en fait quasiment partie. Comme le nombre des tâches à réaliser par le système est limité, celui des transactions l'est aussi.

Schreiber et *al.* (p. 218, 2000) présentent les transactions comme des relations entre tâches impliquant différents agents. En général dans notre cas, une transaction concernera différents types d'agents dans la réalisation d'une même tâche mais elle peut se limiter à un seul et même type. Une transaction sera répétée autant de fois qu'il y a d'agents d'impliqués et tant qu'une sous-tâche à laquelle elle est associée n'a pas été complétée.

Nous avons trois transactions et un seul plan de communication que nous avons nommé *"plan principal"*. Les transactions sont :*"Ajouter un nœud"*; *"Changer d'état"* et *"Expliquer"*.

4.3.2 Les transactions entre agents

Du point de vue du SMV, trois types d'agents sont impliqués dans le processus de communication à savoir :

- le système client qui va communiquer avec un objet de type *DTMSproxy;*
- l'objet de type *DTMSproxy* qui interagit avec un objet de type DNode;
- l'objet de type DNode qui va communiquer avec d'autres objets du même type et ce, de façon récursive.

Initialement, une transaction parvient au SMV en tant qu'une requête effectuée par le système client. La classe *DTMSproxy* intercepte la requête, localise le nœud concerné et lui relaie le message. Le nœud en question traite la demande et relaie le message à tout autre nœud concerné. Chaque nœud retourne toujours une réponse à celui qui l'a invoqué et à la fin du processus, cette réponse atteint l'objet *DTMSproxy* qui la transmet au système client. Au cours de ce processus, un nœud littéral va toujours faire suivre une transaction vers des nœuds justification et inversement un nœud justification va toujours faire suivre une transaction vers des nœuds littéraux.

4.3.2.1 CM-1 : Transaction *"Ajouter un nœud"*

La transaction *"Ajouter un nœud"* concerne la sous-tâche *"Construction du réseau"* qui à son tour peut impliquer la sous-tâche *"Changement d'états".* Cette transaction peut donc être suivie par la transaction *"Changer d'état"*.

Quand la transaction émane du système client et dépendamment des cas, un nœud et le signe de l'une de ses parties constituent l'objet d'information. Quand cette transaction implique des agents de type *DTMSproxy* et DNode, l'objet d'information inclut en plus la liste dans laquelle le nœud doit être inséré.

4.3.2.2 CM-1 : Transaction *"Changer d'état"*

Cette transaction est associée à la réalisation de la sous-tâche *"Changement d'états".* Comme pour la transaction précédente, un nœud ainsi que le signe d'une partie de ce nœud constituent l'objet d'information. Quand cette transaction concerne les agents internes du SMV, elle inclut en retour (réponse) une référence sur tout nœud qui peut être en contradiction. Cette référence est nulle si aucune contradiction n'est détectée.

4.3.2.3 CM-1 : Transaction *"Expliquer"*

La transaction *"Expliquer"* concerne la réalisation de la sous-tâche *"Fournir des explications"* et elle est de la même nature que les deux précédentes. La seule différence dans ce cas c'est que l'objet d'information inclut en plus le type d'explications

souhaitées, le sens dans lequel les explications doivent être générées et éventuellement un différentiel pour stocker les résultats. Les feuilles de travail CM-1 concernant ces trois transactions sont fournies en annexe.

4.3.3 Détails et spécifications des échanges d'informations

Pour chacun des trois types de transactions identifiés, toutes les communications sont de type *"demander/répondre"* (ask/reply). Chaque requête est initiée par un objet de façon non impérative sous forme d'un message et chaque objet fournit une réponse appropriée selon le contexte.

Dans les trois cas, les transactions suivent le même chemin selon l'ordre suivant :

1.1 **Expéditeur** : Système client;
1.2 **Destinataire** : *DTMSproxy;*

2.1 **Expéditeur** : *DTMSproxy;*
2.2 **Destinataire** : DNode;

3.1 **Expéditeur** : DNode;
3.2 **Destinataire** : DNode.

Le système client initie toujours une transaction avec un objet *DTMSproxy*. Celui-ci initie une seconde transaction du même type avec un objet de type DNode. Par la suite, ce dernier objet initie à son tour et récursivement ce même type de transaction avec d'autres objets du même type. Les réponses, quant à elles, sont acheminées dans le sens contraire et peuvent être soit une chaîne de caractères soit un différentiel.

4.3.3.1 CM-2 : *"Ajouter un nœud"*

Pour la création d'un nœud, le système client envoie un message à un objet *DTMSproxy*. Le message doit contenir le nom du nœud à créer et éventuellement le signe (négatif ou positif). Le nom consiste en une simple chaîne de caractères alors qu'une simple constante représente le signe. Cette constante est égale à 0 si le signe est négatif et elle est égale à 1 s'il est positif. Elle est définie par l'énumération suivante :

```
nodePart {kNegPart = 0, kPosPart = 1}
```[24].

Puisque il faut distinguer dans quelle liste un nœud est ajouté, l'énumération suivante a été définie :

```
theSets {kTheObserversSet = 1, kTheObservablesSet = 2}
```

Quand un nœud est inséré dans l'une des deux listes d'un objet *DTMSproxy*, cette liste peut concerner ou les justifications ou des littéraux. L'énumération précédente a été donc redéfinie comme suit :

```
theSets {kTheLiteralsSet = 1, kTheJustificationsSet};
```

Le choix de ces noms sera expliqué dans le paragraphe 5.7.

4.3.3.2 CM-2 : *"Changer d'état"*

La modification de l'état d'un nœud nécessite que le système client envoie un message à un objet *DTMSproxy* précisant le nom du nœud, éventuellement la partie du nœud dont l'état

[24] Schreiber *et al.* (p. 93, 2000) indiquent comment les valeurs types doivent être spécifiées. Schreiber *et al.* (p. 116, 2000), les spécifications peuvent être effectuées en pseudocode ou par des diagrammes UML.

doit être modifié ainsi que l'état devant être modifié. Deux types d'états seulement peuvent être concernés par cette demande : les états d'activité *(IN/OUT)* ou les états d'assomption.

Cette transaction est reprise par l'objet *DTMSproxy* qui la relaie vers le nœud concerné puisqu'ils sont respectivement observable/observateur. S'il s'agit d'une modification pour assumer ou non un nœud, la transaction prend fin une fois que le nœud exécute la tâche à son niveau. Si par contre il s'agit d'une modification d'état d'activité, cette transaction va prendre d'autres dimensions.

Une fois qu'un nœud modifie son propre état d'activité, il doit décider seul quel message envoyer aux autres nœuds qui dépendent de lui. Il doit avertir certains de se mettre à *IN* et il doit avertir d'autres du contraire. En plus, il doit préciser pour chaque nœud quelle partie doit être modifiée. Il existe donc quatre types d'étiquettes possibles pour représenter l'état qu'un nœud doit propager et que nous avons nommé `kIn/kOut` quand la partie positive est ciblée et `kNegIn/kNegOut` quand c'est la partie négative qui l'est. Ces étiquettes et les valeurs qui leurs sont associées sont décrites par l'énumération suivante :

```
label {kIn = 1, kOut = 2, kNegIn = 3, kNegOut = 4}.
```

Rappelons que dans le système de Doyle, deux listes sont utilisées pour la propagation des états. La première est réservée aux nœuds devant être mis à *IN* et l'autre à ceux devant être mis à *OUT*. Donc dans notre cas, nous recourrons plutôt à une seule liste mais avec quatre valeurs possibles à propager.

Une fois qu'un nœud modifie l'un de ses états d'activité, il met à un le bit approprié pour s'auto marquer et éviter les circuits comme nous l'avons déjà expliqué. Après chaque opération de propagation, ce bit est ensuite remis à zéro, son état initial.

Conformément aux recommandations de Schreiber et *al.* (p. 230, 2000), le contrôle des messages peut être spécifié par du pseudo code ou par des machines d'états. La figure 4.11 est une machine d'états globale qui peut être appliquée aux différents états d'un nœud. Lorsque les inférences seront décomposées en "appels de méthodes" dans le chapitre 5 comme le spécifient Schreiber et *al.* (p. 285, 2000), une machine d'états pour chacun des états d'un nœud a été fournie.

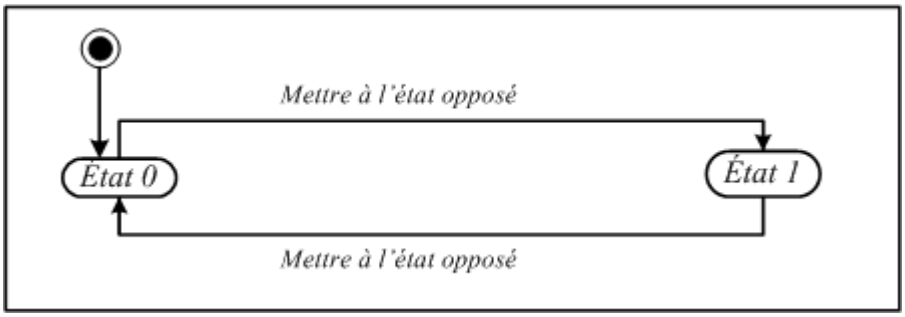


Figure 4.11 : Machine d'état générique associée à chaque état d'un nœud

4.3.3.3 CM-2 : *"Expliquer"*

Une transaction *"Expliquer"* suit le même cheminement que celui des modifications d'états d'activités. Elle est initiée par le système client qui envoie un message à la classe *DTMSproxy* spécifiant le nom du nœud et le signe en option comme nous l'avons décrit dans le paragraphe 4.3.3.1. Le message spécifie aussi le type d'explications souhaitées (détaillé ou non) ainsi que le sens (chaînage avant ou chaînage arrière).

Toute demande d'explications est relayée par la classe *DTMSproxy* au nœud concerné. Éventuellement, un différentiel est ajouté en paramètre pour stocker les résultats. Si l'explication doit être détaillée et concerne plus qu'un nœud alors chaque nœud qui reçoit le message le relaie récursivement aux autres nœuds concernés. Une explication qui n'est pas détaillée peut être limitée au nœud lui-même uniquement. Il s'agit alors d'une explication

ponctuelle. Elle peut se limiter aussi soit à ses prémisses soit à ses conclusions. Dans ce cas, les nœuds intermédiaires sont ignorés et seuls ceux qui sont à l'extrémité d'une branche (aux limites) sont expliqués. Pour décrire les types d'explications possibles, l'énumération suivante a été définie :

```
explanationMode {kForward = 0, kBackward = 1, kWithoutDetails = 0,
                    kWithDetails = 1, kBounds = 3};
```

Une transaction *"Expliquer"* impliquant deux nœuds est soit ponctuelle soit détaillée. Une explication ponctuelle consiste pour un nœud de retourner simplement son contenu et ses états (actif ou non assumé ou non). Une explication détaillée est ponctuelle en plus elle est récursive *(kWithDetails)*. Chaque nœud s'auto-explique et relaie cette transaction vers tout autre nœud dépendant de lui. L'objet *DTMSproxy* qui reçoit les explications cumulées dans un différentiel peut décider de retourner au système client l'ensemble de ces explications ou uniquement les explications aux limites (kBounds) c'est-à-dire, les prémisses ou les conclusions.

L'objet d'information d'une transaction *"Expliquer"* peut varier dynamiquement d'un agent à un autre dépendamment du contexte. Dans le modèle conception, nous verrons les différentes combinaisons possibles des méthodes qui seront associées à ce type particulier de transactions. Le processus d'explication constitue l'activité la plus complexe de notre système. Comme on va le voir plus tard, elle nécessite la surcharge de plusieurs méthodes polymorphes, récursives, imbriquées et croisées à la fois que seule une approche objet peut permettre de réaliser.

4.4 Récapitulation

Dans ce chapitre nous avons présenté les modèles CommonKADS qui se situent au niveau du contexte organisationnel et celui des concepts. Une attention particulière a été accordée au modèle de connaissances qui constitue notre centre d'intérêt. Dans ce modèle, les concepts et les types de valeurs ont été définis et un schéma du domaine a été construit. C'est ce schéma qui sera progressivement repris et raffiné au cours des deux prochains chapitres.

À la suite du modèle des connaissances, nous avons décrit le modèle de la communication. À chacune des sous-tâches identifiées dans la couche des tâches, un type de transaction a été associé. Le prochain chapitre, le modèle conception, apportera un éclairage supplémentaire sur la dynamique de l'ensemble. Un schéma complet du domaine sera détaillé comme le spécifient Schreiber et *al.* (p. 185, 2000).

Chapitre 5 : Le modèle conception

"Design is a funny word. Some people think design means how it looks, but of course, if you dig deeper, it's really how it works."
Steve Job.

Dans le modèle des connaissances, des concepts, des relations ainsi que certaines propriétés et leurs types ont été identifiés. Le tout a été ensuite représenté par un schéma du domaine. Dans ce chapitre, nous allons passer en revue les différents points du modèle de conception de CommonKADS, mais auparavant, nous allons expliquer pourquoi ce schéma du domaine a été conçu ainsi.

En matière de conception, Schreiber et *al.* (p. 294, 2000) soulignent, qu'à part ce qu'ils proposent, la littérature dans le domaine de l'ingénierie des connaissances est limitée à ce sujet. Par conséquent, ils suggèrent d'étudier des ouvrages spécialisés en conception de logiciels et de les adapter. Donc, avant de décrire le modèle de conception, nous allons voir en premier comment le système a été décomposé en un ensemble de patrons de conception. Nous détaillerons un peu plus ce qu'est un patron de conception. Par la suite, nous montrerons comment nous avons appliqué les patrons qui conviennent à notre situation.

En génie logiciel, les idiomes utilisés en conception sont généralement associés aux patrons de conception (Eckel, 2003) et doivent être justifiés. Un paragraphe portera sur cet aspect en particulier afin d'expliquer le choix des noms des types (les classes et les autres structures).

5.1 Les patrons de conception

D'après Buschmann et *al.* (1996), "les patrons de conception sont des solutions éprouvées pour des problèmes récurrents qui surviennent dans des situations particulières de conception". À la différence d'un algorithme qui s'attache à décrire d'une manière formelle comment résoudre un problème particulier, les patrons décrivent des procédés de conception généraux où la communication entre objets est centrale. C'est pour cette raison d'ailleurs que Cooper (1998) les qualifie même de "patron de communication".

Dans la littérature, il existe plusieurs patrons de conception comme il existe plusieurs façons de les classifier. Gamma et *al.* (1995), dont l'ouvrage constitue la référence en la matière, proposent un catalogue[25] de vingt-trois patrons classés en trois catégories : les patrons de création, les patrons structurels et les patrons de comportement. Ces patrons sont largement couverts dans la littérature et nous n'exposons ici que la partie applicative essentielle à notre modèle.

Dans les chapitres précédents, nous avons décidé d'aborder les SMV en termes de patrons de conception au lieu de le faire en termes d'algorithmes complexes. Un algorithme nécessite toujours d'être expliqué et d'être formulé selon une spécification quelconque. Les patrons de conception quant à eux sont plutôt génériques et offrent des solutions qui le sont aussi. Une bonne partie de la logique du SMV que nous avons modélisée pourrait être appréhendée intuitivement pour le lecteur familier avec les patrons les plus courants et avec les bonnes pratiques du génie logiciel.

[25] http://www.dofactory.com/Patterns/Patterns.aspx
http://fr.wikipedia.org/wiki/Motif_de_conception#Liste_des_patrons_de_conception_du_GoF

Le schéma du domaine présenté dans le chapitre précédent a été déjà modélisé par un agencement de patrons comme nous allons le montrer ici. Ceci présuppose qu'il fallait savoir lesquels utiliser. Pour sélectionner un patron il faut (Gamma et *al.*, 1995) :

- étudier comment les patrons collaborent en général;
- déterminer les causes qui peuvent amener une révision de conception;
- considérer les facteurs de changement ou ce que Eckel (2003, p. 572) appelle "vecteur de changement". Ces facteurs sont essentiels lors de tout processus de "refactorisation" dans le futur (Fowler (1999);
- savoir comment les utiliser;
- définir les structures et les collaborations;
- définir les classes;
- implémenter en tenant compte des responsabilités et des compétences.

À notre avis, la sélection d'un patron relève de l'expérience. Normalement, une fois qu'un problème est formulé, le choix des patrons à utiliser s'impose de lui-même ou inversement, un problème peut être facilement circonscrit quand on le modélise avec des patrons en arrière plan. Pour éviter tout problème de sur-ingénierie[26], le recours à des patrons de conception ne doit pas constituer une fin en soi et ceux-ci doivent vraiment faire partie de la solution. Comme le souligne Kerievsky (2002) à ce sujet, une utilisation à outrance de patrons peut avoir des effets pervers et peut compliquer la conception au lieu de la simplifier.

En considérant tous ces points, nous allons recourir aux six patrons suivants : `"observateur"` *(observer)*, `"itérateur"` *(iterator)*, `"composite"` *(composite)*, `"poids plume"` *(flyweight)*, `"fabrique"` *(factory method)* et `"mandataire"` *(proxy)*. Bien que notre système comporte huit machines d'états, le patron `"État"` *(state)* qui peut sembler convenir pour les modéliser a été exclu à cause du très grand nombre d'objets qui sont susceptibles d'être générés.

[26] *Over engineering*

La plupart de ces cinq patrons, comme d'autres d'ailleurs, exploitent trois mécanismes fondamentaux des langages à objets : l'abstraction, l'agrégation (incluant la composition) et l'héritage. Ils l'exploitent selon une configuration bien particulière qui peut être présentée comme une sorte de méta-patron à son tour (figure 5.1). Cette configuration consiste à relier deux classes abstraites, dont l'une est conteneur alors que l'autre représente les éléments à contenir. Tout objet contenu l'est selon son Supertype mais, à chaque fois qu'il doit être invoqué, il l'est selon le type dérivé. Le polymorphisme et la surcharge permettent par la suite une flexibilité qui ne pourrait être obtenue autrement que par des artifices très complexes.

Regardons maintenant comment le schéma du domaine du chapitre précédent a été articulé pour concevoir un SMVJ complet, compact et extensible

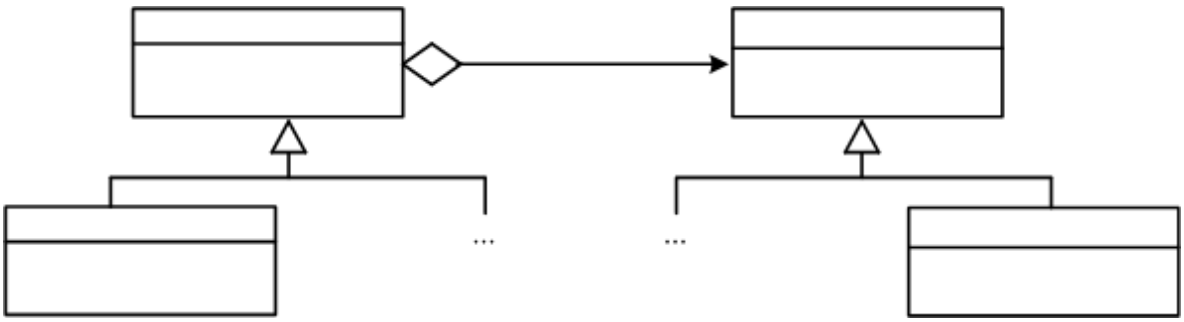


Figure 5.1 Un méta-patron

5.2 Gestion de l'itération : les patrons observateur et itérateur

Pour parcourir une liste d'objets qui sont à l'écoute d'un message quelconque, une itération doit être effectuée. Un objet, appelé itérateur est généralement créé. Les objets qui sont à l'écoute *(listeners)*, c'est-à-dire qui sont en attente de tout message qui peut leur être envoyé, quant à eux sont appelés observateurs.

La façon d'organiser des objets dans une liste pour les contacter correspond au patron de conception observateur et celle de l'itération, au patron itérateur. Ces deux patrons vont généralement de pair.

5.2.1 Le patron observateur *(observer)*

Le patron observateur est un patron qui suit la structure de la figure 5.1 et il met en œuvre deux types de classe, une classe "observable" (appelée aussi émettrice ou diffuseur) et une classe "observatrice"(figure 5.2). C'est l'objet "observable" que revient la responsabilité de maintenir une liste pour stocker des objets "observateurs" (appelés aussi écouteurs ou accointances). En général, les objets notifiés n'ont pas à avoir connaissance de l'objet qui les contacte et les deux types peuvent par conséquent, être faiblement couplés. La liste elle, peut être n'importe quel conteneur.

Pour construire et gérer cette liste, l'observable doit définir des méthodes pour les ajouter (`attacher`), les ôter (`détacher`) et leur envoyer des messages (`notifier`). La classe dite observateur quant à elle, doit définir au moins une méthode pour se mettre à jour *(`update`)* et que la méthode "`notifier`" peut invoquer. Des mécanismes comparables peuvent être mis en œuvre et plusieurs types de messages peuvent être émis. Il suffit que les deux types de classe puissent se conforment à un protocole établi.

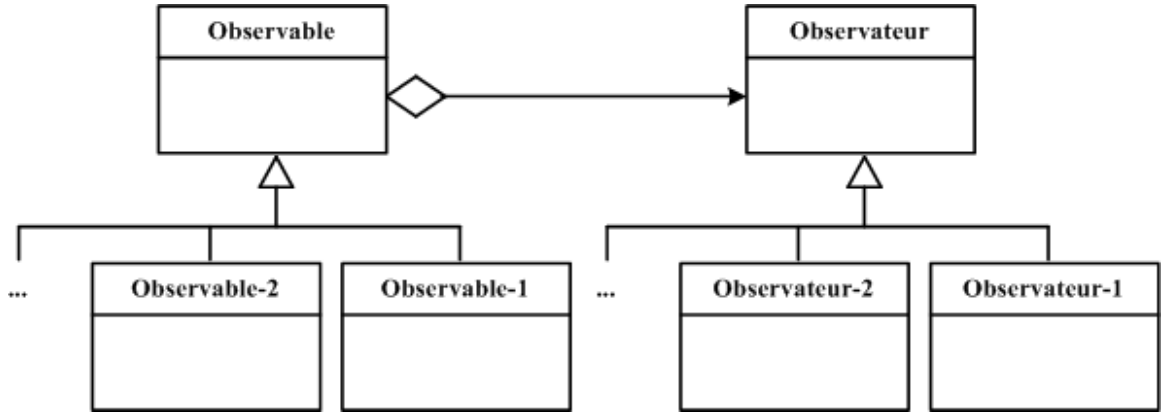


Figure 5.2 Patron observateur
Source : adapté de Gamma et *al.* (p. 164, 1995)

Bien que ce patron soit connu sous l'appellation "observateur", c'est au niveau de la classe observable que l'essentiel de la logique se déroule excepté la création des éléments qui peut être effectuée ailleurs. L'objet observable doit cependant référencer les objets observateurs qui peuvent être créés et détruits n'importe où. Normalement, les opérations pour ajouter, retrancher et parcourir la liste des éléments, sont déléguées à la classe conteneur[27] elle-même et c'est cette dernière qui recourt à son tour au patron itérateur.

Dans notre schéma du domaine, la classe nœud est un observable pour un ensemble d'autres nœuds qui sont des observateurs. Elle est donc une classe de type observateur. Elle l'est aussi vis-à-vis de la classe `DTMSProxy` qui joue le rôle d'un observable à son tour. En effet, c'est cet objet qui doit en premier lieu notifier chaque nœud d'un événement qui lui parvient du système client.

Le schéma du domaine de la figure 4.3 peut donc être affiné en lui ajoutant deux nouvelles classes que nous nommerons `"DObservable"` et `"DObserver"`. Nous obtenons alors le schéma suivant :

[27] Chaque agrégation dans la figure 5.3 va se traduire par une instance d'une classe. Cette classe n'est pas encore définie à ce niveau d'abstraction mais on peut le voir dans le schéma final détaillé en annexe.

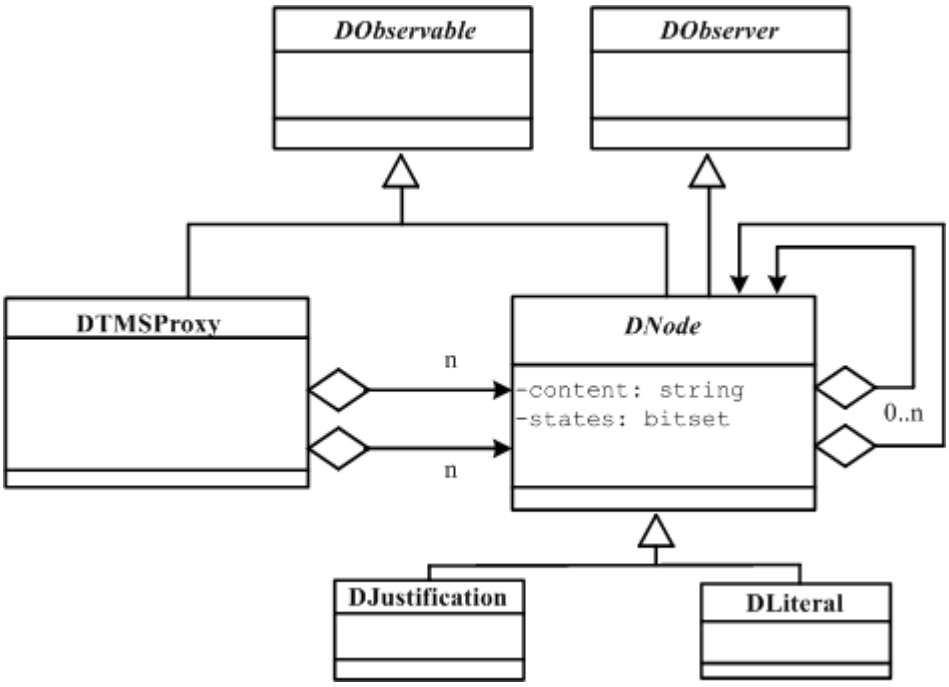


Figure 5.3 Schéma du domaine et patron observateur

Le problème avec ce nouveau schéma, c'est qu'il a recourt à l'héritage multiple. Malheureusement ce type d'héritage que plusieurs langages à objets ne supportent pas, doit être évité autant que possible. D'ailleurs Eckel (p. 569, 2002) le compare à l'instruction "*goto*" qui était en vogue dans certains langages procéduraux.

Donc, comme la classe `DNode` est la seule qui est de type `DObserver` alors ce type peut être supprimé. Nous allons considérer *de facto* que `DNode` est de type observateur sans avoir à le représenter explicitement. Le schéma précédent peut finalement être simplifié comme suit :

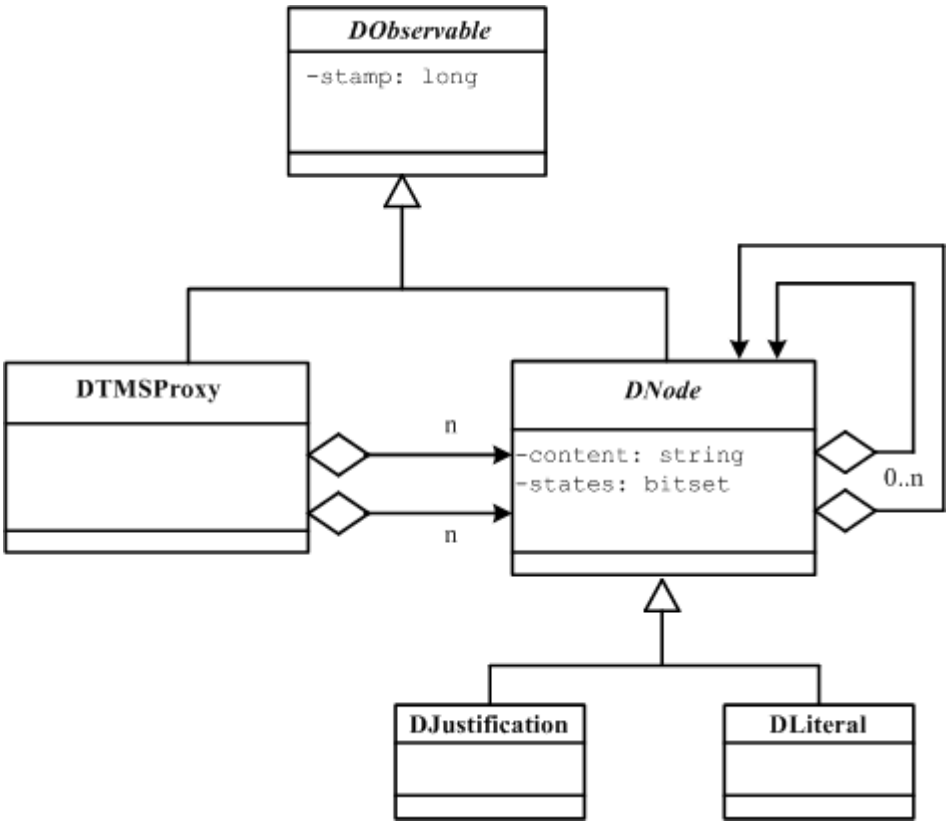


Figure 5.4 Patron observateur et schéma du domaine simplifié

5.2.2 Le patron itérateur *(iterator)*

Parcourir une liste consiste à effectuer une itération à travers tous ses éléments. Le patron de conception itérateur a été conçu spécialement pour parcourir divers conteneurs sans que ceux-ci n'aient à exposer leurs représentations (Gamma et *al.*, 1995). Pour que cela soit possible, la classe conteneur doit être en mesure de produire des objets de type itérateur sur demande. Un itérateur doit être capable de localiser, entre autres, le premier et le dernier élément de la liste à laquelle il est associé, l'élément suivant, l'élément précédent, etc. Il existe plusieurs types d'itérateurs qui permettent de traverser divers conteneurs, de différentes manières.

Dans notre cas, nous n'avons pas besoin de concevoir d'itérateur puisque nous allons réutiliser ceux des librairies standards *(Standard template Librairies : STL*[28]*)* ainsi que les conteneurs qui leurs sont associés. Cependant, pour comprendre comment nous allons les exploiter, nous allons les présenter rapidement.

Normalement, une classe abstraite itérateur est associée avec une classe abstraite conteneur qui définit l'interface pour générer des objets itérateurs. Cette classe conteneur permet de définir des objets dont le rôle est de stocker d'autres objets et de fournir des méthodes pour manipuler objets (contenus). Des méthodes courantes servant à ajouter, ôter, insérer et chercher sont aussi définies ainsi que des itérateurs pour parcourir l'ensemble des objets. Les sous-classes respectives de la classe abstraite sont responsables de définir les objets concrets appropriés. Ainsi chaque conteneur dérivé peut décider quel type d'itérateur générer.

La classe conteneur qui a pour responsabilité de générer des objets itérateurs définit une fonction pour cette fin. Cette fonction constitue en elle-même un autre patron de conception appelé `"fabrique"` que nous allons expliquer un peu plus loin puisque nous l'appliquons. La figure 5.5 représente le diagramme de classes du patron itérateur avec une "fabrique" `"CreerIterateur"` tel que décrit par Gamma et *al.* (1995).

La présentation de ce patron de conception est primordiale pour la compréhension de la logique du SMV que nous avons modélisé. Des itérateurs vont être créés et détruits à chaque fois qu'une liste doit être parcourue. Une multitude d'itérateurs vont donc intervenir dans tout processus de propagation d'états ou d'explication.

[28] http ://www.sgi.com/tech/stl/

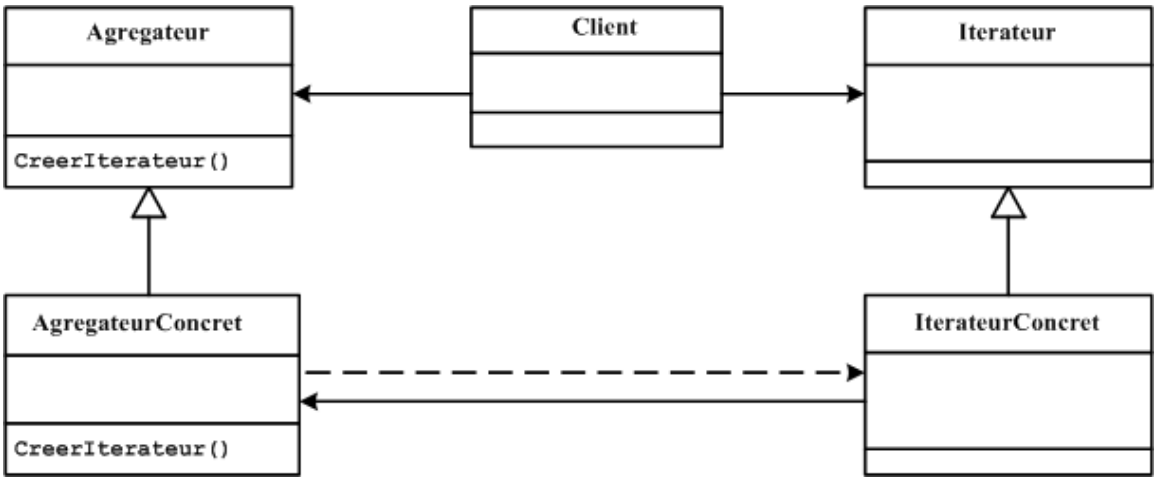


Figure 5.5 Patron itérateur
Source : adapté de Gamma et *al.* (p. 257, 1995)

5.3 Gestion de la récursivité : le patron composite

D'après Gamma et *al.* (p. 163, 1995), le patron composite sert à structurer des objets sous la forme d'arbre. Les objets composés et les objets composants sont structurés selon la même configuration décrite dans la figure 5.1. Les objets composés sont stockés dans la classe composite qui est en même temps un sous-type du composant. Ceci permet aux composants et aux composites d'être traités uniformément d'une manière récursive (figure 5.6).

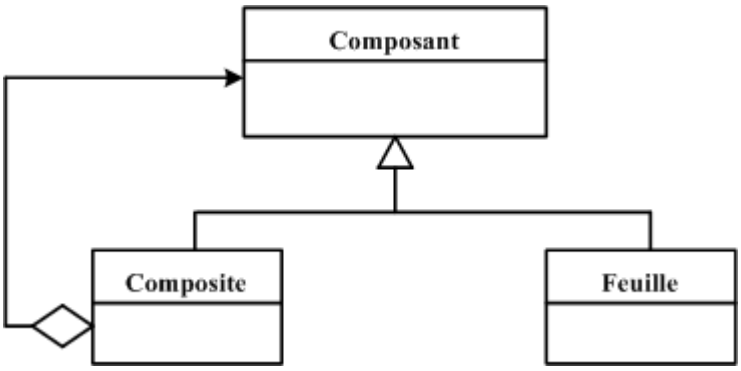


Figure 5.6 Illustration du patron composite
Source : adapté de Gamma et *al.* (p. 164, 1995)

La figure 5.6 illustre le cas général du patron composite. Celui-ci peut représenter n'importe quel graphe et tout autre cas particulier peut en être dérivé. Dans notre cas, nous n'avions pas à nous demander comment modéliser le réseau du SMV mais uniquement comment agencer les nœuds à partir de ce patron et des patrons précédents. Il fallait uniquement considérer qu'un nœud est composite et composant (figure 5.7) puis observateur et observable à la fois.

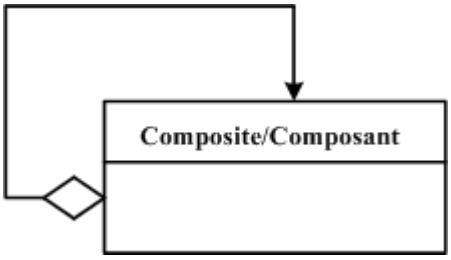


Figure 5.7 Un cas particulier du patron composite

Ce type de patron est courant dans la conception des "cadriciels" *(framework)* d'interfaces graphiques pour usagers. Dans ces environnements les objets "vues" sont généralement composés par d'autre "vues" et chacune peut recevoir et propager des événements.

La classe `DNode` a été conçue en suivant ce principe. En plus, comme chaque nœud possède une paire de listes alors il est doublement composite/observable comme on peut le voir dans la figure 5.8.

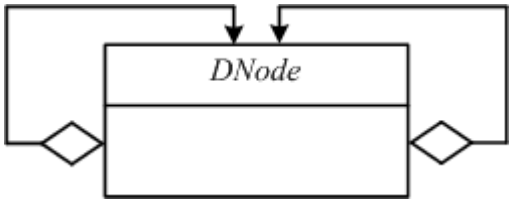


Figure 5.8 Un nœud en tant qu'un double composite

La structure de la figure 5.8 est un amalgame de composite, d'observateur, et d'observable et contient en filigrane les patrons itérateur et fabrique. En termes de conception, elle est de "haute densité " comme le qualifient Riehle et *al.* (2000)[29].

5.4 Le composite en tant que poids plume *(flyweight)*

Un système comme celui que nous mettons en place doit être en mesure de gérer un nombre "illimité" de nœuds. La gestion efficace de la mémoire dans ce cas constitue une préoccupation principale. Il est assez évident qu'un nœud qui appartient à plusieurs autres nœuds ne peut pas y figurer sous plusieurs copies et que de simples références sont appropriées. Cependant dépendamment des contextes, un nœud doit maintenir des étiquettes qui varient de l'un à l'autre. Par conséquent, ces étiquettes ne peuvent pas être référencées et doivent être stockées par valeur.

D'après Gamma et *al.* (p. 195, 1995), "poids plume" est un patron de conception qui a justement pour rôle de résoudre ce type de problèmes et il possède une structure similaire à celle de la figure 5.1. Une classe conteneur y joue le rôle d'une fabrique d'objets appelés "poids plume". Un "poids plume" est un objet qui figure dans plusieurs contextes et qui possède des attributs propres (intrinsèques) qui le décrivent, et des attributs variables (extrinsèques) qui dépendent du contexte dans lequel il se trouve. Les propriétés qui sont communes et qui doivent figurer dans tous les contextes sont stockées par références ce qui permet de réduire les besoins en termes de mémoire vive. Par opposition, les propriétés qui dépendent du contexte, les propriétés extrinsèques, sont stockées par valeurs.

[29] *Design density*

5.5 Création de nœuds et le patron fabrique *(factory method)*

Lors de la demande de la création d'un nouvel objet, la classe conteneur doit s'assurer qu'il n'existe pas déjà. Si tel est le cas, alors elle en crée un, sinon elle le localise. Dans les deux cas, elle retourne par la suite une référence sur l'objet en question. La classe `DTMSProxy` a été munie de fonctions dont le seul but est de générer des nœuds. Selon les cas, ceux-ci sont instanciés par la suite soit en tant que justification soit en tant que littéral.

À chaque fois qu'un nœud est créé, la classe `DTMSProxy` doit lui attribuer un ordre. En tant que `DObservable`, chaque nœud est muni d'une variable `"stamp"` qui sert comme critère de tri. En tant que `DObservable`, la classe `DTMSProxy` est munie de cette variable aussi, mais elle s'en sert pour maintenir le compte du nombre de nœuds créés. À chaque fois qu'un nœud est généré, ce compte est incrémenté d'une unité puis il est assigné à la variable appropriée du nouvel objet.

Quand un nœud ajoute un autre nœud dans l'une de ses listes, celle-ci l'insère en se servant de la variable `"stamp"` en tant que critère de tri. Un foncteur, appelé "`DSortFunctor`" a été défini pour cette fin comme cela est spécifié dans le schéma final fourni en annexe.

Lorsqu'un Objet `DTMSProxy` ajoute un nœud nouvellement créé dans l'une de ses listes, il utilise le contenu (le nom) de ce nœud en tant critère de tri. Ceci permet de localiser facilement chaque nœud par son nom. Un itérateur de `"map"` paramétré en conséquence a été redéfini. Cet itérateur a été nommé `"NodesIterator"` dans le schéma final de l'annexe 2.

5.6 Le système en tant que mandataire *(proxy)*

La classe `DTMSProxy` joue un autre rôle au sein du SMV, celui d'un mandataire. Un mandataire est un objet mandataire qui contrôle l'accès à un autre objet (Gamma et *al.*, p. 195, 1995) et il peut avoir d'autres fins. Il peut servir de cache, de mécanisme de sécurité, d'accélérateur de traitement, etc.

Un mandataire de protection permet l'accès à un objet par un autre en offrant à l'objet client les mêmes interfaces que celles de l'objet accédé. Ceci donne à l'accesseur l'impression qu'il y accède réellement (figure 5.10). Un mandataire distant est un objet qui maintient en cache d'un ensemble d'objets et qui les fournit à un objet client sur demande économisant ainsi les opérations de recharger les objets à partir d'une source lointaine ou coûteuse d'accès. L'accès à travers un mandataire et toujours indirect et permet ainsi un meilleur niveau de sécurité.

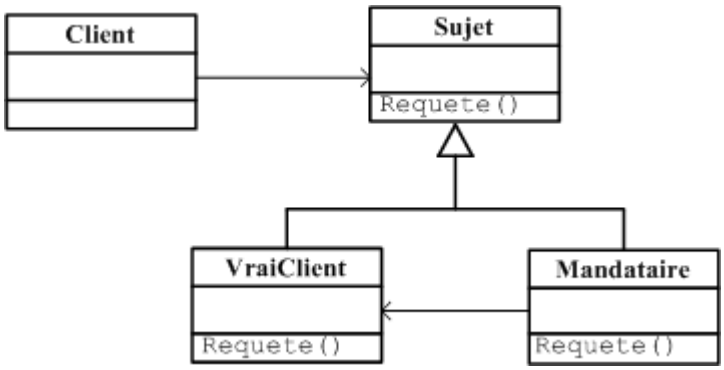


Figure 5.9 Patron mandataire
Source : adapté de Gamma et *al.* (1995)

La classe `DTMSProxy` a été conçue en tant que mandataire (d'où son nom) avec toutes ces caractéristiques. Elle est la seule classe qui peut être invoquée par le système client et elle sert de mécanisme de protection. Elle met en cache un ensemble de nœuds et de raisonnements et donne l'impression au système client qu'il y accède directement en

fournissant les mêmes interfaces que les nœuds. Finalement, elle met en cache l'ensemble d'un raisonnement et le fournit sur demande.

Le patron mandataire est le dernier qui a été mis en œuvre dans notre modèle. La figure qui suit représente le schéma du domaine annoté en fonction de tous les patrons que nous venons de décrire. Le schéma complet est détaillé en annexe deux et il contient l'ensemble des classes avec leurs propriétés typées, les méthodes qui leur sont associées ainsi que l'ensemble des valeurs énumérées.

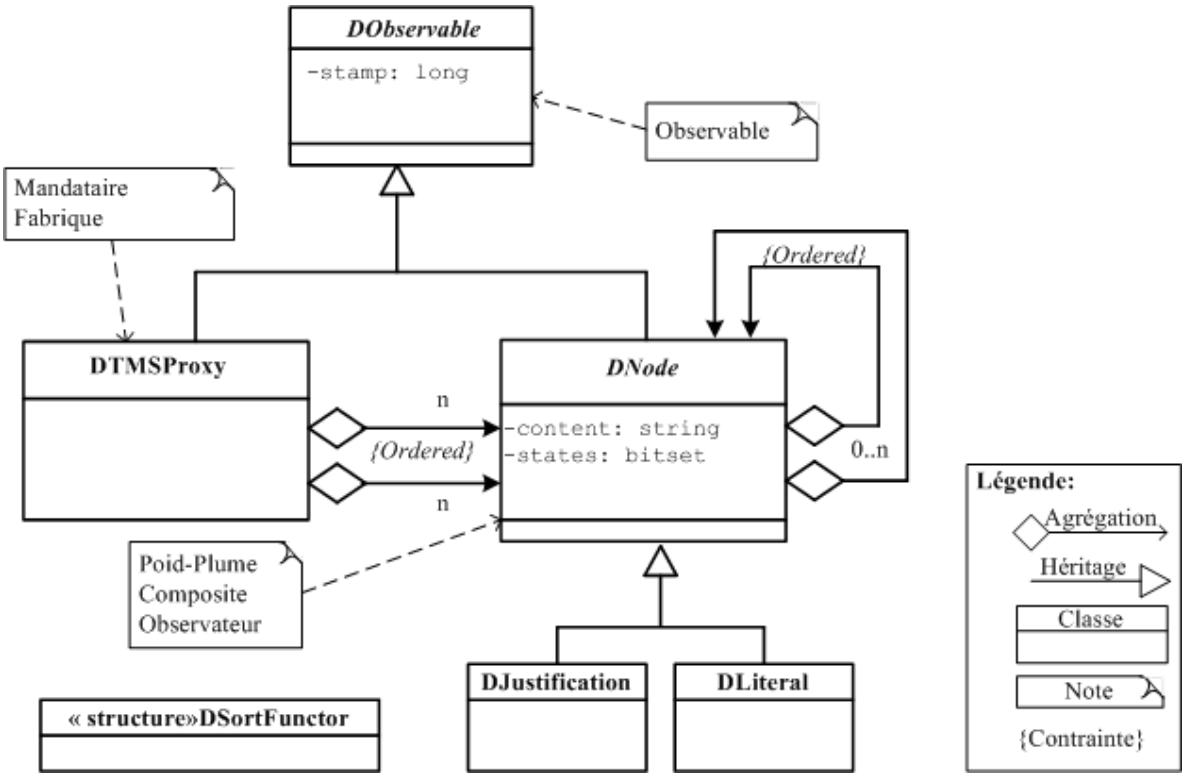


Figure 5.10 Schéma du domaine et patrons de conception

5.7 Idiomes[30]

Si les patrons de conception peuvent permettre de saisir intuitivement la logique derrière un modèle, les idiomes et autres conventions peuvent y contribuer aussi. Ils facilitent la maintenabilité et la lisibilité des spécifications et du code généré (Krzysztof et Brad, 2005). Dans sa taxonomie des patrons, Eckel (2003) définit un idiome comme étant la manière d'écrire un code en fonction d'un langage particulier. D'après lui, un idiome est adopté par une communauté de développeurs s'intéressant à une plateforme et à un langage de programmation donné. Les idiomes, les conventions et les pratiques communes facilitent la lecture et la compréhension par des pairs et allègent la documentation.

Le choix des noms des concepts, des méthodes, des variables et des énumérations du schéma final a été déterminé par celui du langage de programmation choisi, la plateforme d'intérêt et la syntaxe des langages cibles possibles (Objective-C). Le SMV que nous développons sera réutilisé dans le futur dans d'autres applications écrites en C++ et Objective-C et son code sera éventuellement partagé sur internet. Il doit donc être accessible et lisible par une large audience. L'idiome que nous allons suivre est le même que celui utilisé en Objective-C[31] sur la plateforme du Mac Os X.

Puisque nous avons décrit comment le schéma du domaine du chapitre précédent a été conçu et après l'avoir raffiné, nous allons dans le reste de ce chapitre appliquer le modèle conception de la méthodologie CommonKADS.

---

[30] http://en.wikipedia.org/wiki/Idiom#Computer_science
James O. Coplien, Advanced C++:Programming Styles And Idioms; Addison-Wesley Professional, 1992.
[31] http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf

5.8 DM-1 : Architecture du système

Le patron MVC est un patron de conception d'architecture globale et de haut niveau. D'après Gamma et *al.* (p. 6, 1995) chacune des parties de ce patron peut être décomposée à son tour en patrons plus élémentaires[32] comme ceux que nous avons exposés. Le système que nous avons modélisé peut être considéré comme un module serveur qui a pour rôle de fournir des services à un système client. Celui-ci est responsable d'effectuer un raisonnement et d'inter agir avec les usagers. C'est donc à ce dernier que revient la responsabilité de fournir la partie "Vue" et la partie "Contrôleur". Comme le soulignent Gamma *et al.* (1995), le paradigme MVC ne s'applique pas forcément tel quel dans toutes les applications. Dans le cas d'une application de traitement de texte par exemple, il est difficile de séparer la partie vue de la partie modèle. Dans les applications client/serveur, la présentation relève aussi des applications clientes. Dans notre cas, un SMV n'est qu'un simple module qui complète la partie "Modèle" d'un système client.

5.9 DM-2 : Plateforme d'implémentation

Le SMV a été modélisé en fonction d'un langage objet en évitant l'héritage multiple puisque ce ne sont pas tous les langages qui le supportent. Aucune plateforme ni système d'exploitation en particulier n'ont été ciblés mais le système a été conçu pour être réutilisé par la suite sur les principales plateformes connues. C'est pour cette raison que seules les librairies standards[33] du langage C++ on été utilisée. Ces librairies sont fournies avec la plupart des compilateurs et leur équivalent vient avec la plupart des environnements de développement.

---

[32] http ://www.dofactory.com/Patterns/Patterns.aspx

[33] http ://www.sgi.com/tech/stl/table_of_contents.html

5.10 DM-3 : Spécification de l'architecture

Dans les points qui suivent, nous allons passer en revue les différents aspects des spécifications de l'architecture (voir figure 2.1).

5.10.1 Modèle d'application : Méthode de tâche

Pour débuter l'exécution de la tâche `"Maintien de vérité"`, nous n'avons pas défini une opération `"execute"` comme Schreiber et *al.* (p. 284, 2000) suggèrent de la nommer. Pour les raisons idiomatiques que nous avons soulignées dans le paragraphe 5.7, nous l'appelons `"Start".` Ceci permettra d'obtenir des expressions plus explicites lors de l'implémentation comme par exemple :

- `client.Start(withTheDtmsHelp); en C++;`
- `[client Start: withTheDtmsHelp];` en Objective-C.

L'objet `client` serait une instance du système client, c'est-à-dire l'agent qui raisonne. `Start` serait l'opération qui débute le processus de raisonnement avec l'aide d'une instance du `DTMSProxy`. Cette instance serait appelée `withTheDtmsHelp` dans les circonstances. Cet exemple illustre parfaitement ce que nous avons expliqué dans la partie sur les idiomes et comment ceux-ci constituent un complément aux patrons de conception.

D'après Schreiber et *al.* (p. 284, 2000), les structures de contrôle doivent être considérées à ce niveau. Comme nous avons adopté une approche objet et comme chaque objet est responsable de ses propres activités, alors le patron itérateur y joue ce rôle. Nous assumons que le lecteur est familier avec ce type de structures de données. Dans le cas contraire, les spécifications détaillées peuvent être consultées sur les sites fournis dans références.

5.10.2 Modèle d'application : Inférence

Schreiber et *al.* (p. 286, 2000) mentionnent que pour chaque inférence une opération d'exécution doit être prévue et si des solutions sont trouvées, des mécanismes de stockage en mémoire doivent être déterminés. Nous verrons dans le paragraphe 5.10.4 comment ces activités sont réalisées par les rôles qui sont identifiés.

5.10.3 Modèle d'application : Méthodes d'inférences

Lors de l'identification des inférences au niveau des connaissances, la façon dont ces inférences sont réalisées n'a pas décrite et c'est à ce niveau que Schreiber et *al.* (p. 286, 2000) recommande de le faire.

Dans le schéma du domaine complet fourni en annexe figure l'ensemble des prototypes des méthodes associées à chacune des classes. Le tableau 5.1 décrit en plus dans quelle inférence chacune de ces méthodes joue un rôle et combien de fois elle est surchargée. Comme le décrivent Schreiber et *al.* (p. 285-286, 2000), les relations inférences/méthodes ne sont pas forcément de type un à un. Une même méthode peut correspondre à plusieurs messages et peut venir sous plusieurs formes. Par exemple, la méthode "`Explain`", qui est relative à l'inférence `"Couvrir",` existe dans chacune des classes et elle existe sous quatre versions surchargées (avec différentes signatures) dans la classe `DJustification`.

Inférences	**Méthodes**	**Classes**				
		DObservable	DTMS	DNode	DJustification	DLiteral
Grouper; Trier et Modifier	AddNod	1	1	1	1	1
	AddConsequent		1			
	AddElement		1			
	NewLiteral		1			
	NewJustification		1			
Modifier	SetStamp	1				
	SetBit			1		
	ClearBits			1		
	SetContent	1		1	1	1
	SetAssumption	1	1	1		1
	SetToIn	1	1	1	1	1
	SetToOut	1	1	1	1	1
	SetMarquee			1	1	1
	UnsetAssumption		1	1		1
	UnsetMarquee		1			1
Comparer	TestBit			1		
	Exists			2		
	IsIn		1	1	1	1
	IsAssumption		1	1		1
	IsContradictory			1		1
	IsMarqued			1	1	1
Couvrir	Explain	1	1	4	4	3
	SetToIn		1	1	1	1
	SetToOut		1	1	1	1
	FindNodeByName		1			
	NewLiteral		1			
	NewJustification		1			

Tableau 5.1 : Correspondance entre les inférences et leurs méthodes

Toutes les méthodes qui sont des accesseurs sont publiques comme cela est spécifié dans le schéma final de l'annexe 2 et toutes les méthodes du tableau 5.1 le sont sauf les deux dernières. Après ce processus de décomposition des inférences en appels de méthodes, on peut consulter les spécifications détaillées de l'annexe 3.

5.10.4 Modèle d'application : Rôles

Puisque nous utilisons une approche orientée objet, les méthodes de tâches ont été directement associées avec les données sur lesquelles elles opèrent. Comme le recommande Schreiber et *al.* (p. 286, 2000), les rôles dynamiques doivent être spécifiés ainsi que leur caractéristiques. Nous avons réutilisé :

- "`map`" en tant que conteneur associatif (clé/valeur) et ordonné;
- "`vector`" en tant que conteneur simple et ordonné;
- "`string`" en tant que conteneur de chaînes de caractères;
- "`bitset`" en tant que conteneur de bits pour stocker les différents états;
- "`iterator`" en tant qu'itérateur pour traverser différents conteneurs.

La plupart de ces types de données viennent avec leurs propres opérations que nous allons réutiliser. Ils sont définis dans les librairies du langage C++ et sont largement documentés sur le Web[34].

Le foncteur de tri décrit dans le point 5.5 a été associé avec le type "`map`" lorsque utilisé au niveau de la classe `DNode` puisque les nœuds du réseau doivent être ordonnés selon leur ordre d'arrivée. À titre d'information, le conteneur standard "`map`" trie par défaut, son contenu selon l'ordre croissant des clés. Un foncteur doit lui être passé en paramètre si un autre ordre de tri doit être utilisé et c'est ce que nous avons donc fait.

Puisque nous allons stocker des paires de nœud/valeur dans chaque conteneur "`map`" et que ces nœuds sont soit des observateurs soit des observables alors ce type de conteneur a été nommé "`Observers`" et "`Observables`" selon les cas (voir les spécifications en

[34] `http://www.sgi.com/tech/stl/table_of_contents.html`
[34] "map" est une classe gabarit *(template)*.

annexe 2). Dans le même ordre d'idées, le type d'itérateur qui leur est associé a été nommé "`ObserversIterator`" et "`ObservablesItrator`" respectivement (annexe 2). Ces redéfinitions s'inscrivent dans l'idiomatique que nous avons décrite dans la section 5.7 et sont spécifiées dans le schéma du domaine final de l'annexe 2.

En ce qui concerne les rôles statiques, les accesseurs ont été décrits indirectement dans le paragraphe précédent et l'ensemble est repris dans le schéma du domaine complet en annexe 2.

5.10.5 Modèle d'application : Base de connaissances

Le mode de représentation des connaissances suit le paradigme orienté objet. Toutes les méthodes sont donc associées avec leurs propres données. Elles sont donc soit des accesseurs soit des mutateurs comme nous l'avons exposé dans le paragraphe 5.10.3.

5.10.6 Spécification de l’application avec l'architecture

L'avantage d'avoir choisi une approche objet dès la phase de modélisation a permis de construire un schéma du domaine lors de la modélisation, de le raffiner progressivement lors de la conception et de l'implémentation. Ceci garantit que les structures de départ sont et seront rigoureusement préservées. Rappelons que la plupart des environnements de développement intégrés (EDI) modernes permettent de manipuler directement le code à partir d'un schéma du modèle et inversement. Dans le chapitre 6, nous présenterons quelques prises d'écran en exemple pour montrer comment nous nous sommes conformé au principe de conservation des structures préconisé par CommonKADS.

5.11 Récapitulation

Dans ce chapitre, nous avons présenté la façon dont le système a été conçu ainsi que les points relatifs à la conception que la méthodologie CommonKADS prescrit.

Le système a été conçu par une combinaison de patrons de conception judicieusement choisis. Ces patrons ont été indirectement utilisés en filigrane lors de la modélisation au niveau des connaissances. C'est en les prenant comme référence aussi que les trois SMV de base ont été analysés puis leurs points faibles identifiés. Comme le soulignent William et *al.* (1998), le recours à des patrons de conception permet d'éviter de tomber dans des "patrons d'erreurs" (appelés aussi anti-patrons)[35].

Après avoir décrit les patrons utilisés et comment ils ont été articulés, nous avons présenté l'architecture du système en complétant les différentes feuilles de travail de la méthodologie CommonKADS. Nous avons montré aussi comment le système s'intègre dans la partie modèle d'un système client.

Dans le chapitre suivant, nous allons voir comment le schéma du domaine va être implémenté ainsi qu'un système pour le manipuler.

[35] http ://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

Chapitre 6 : Implémentation, tests et discussion des résultats

Un SMV fonctionne avec un système client auquel il offre des services et même s'il doit être utilisé directement par un agent humain, une interface minimale est nécessaire pour le manipuler. Dans ce chapitre, nous allons décrire comment le système que nous avons conçu a été implémenté puis nous décrirons le système client qui a été développé pour l'exécuter. Ce dernier système sera minimaliste puisque son développement ne fait pas partie de notre étude.

Pour montrer que le SMV remplit toutes les fonctionnalités déjà décrites, nous allons lui soumettre un ensemble de requêtes triviales au moyen du système client. Ces requêtes vont porter sur la présence des circuits, la détection des contradictions ainsi que sur la gestion de la non monotonie. Les résultats seront affichés par génération d'explications ce qui montrera que le système est capable d'accomplir cette fonction. Ces cas triviaux sont repris à partir des exemples fournis par Forbus et De Kleer (1993) et ont été expliqués dans le chapitre 2. Nous les reprenons ici pour montrer que le système produit des résultats conformes à ceux avancés par ces auteurs.

Des tests avec des cas triviaux peuvent montrer que le système se comporte comme prévu mais ils ne permettent pas d’évaluer la taille du réseau qu'il peut gérer. Un exemple complémentaire a été donc conçu spécialement pour cette fin. À travers cet exemple, nous verrons surtout comment le SMV peut servir de cache. Rappelons que servir de cache pour un système client est l'une des tâches qu'un SMV doit accomplir. Ce qui nous intéresse en particulier dans cet exemple, c'est de tester la taille de ce cache et la robustesse du système. Avant de conclure, nous discuterons des résultats obtenus puis des points forts et faibles du système.

6.1 La relation système client - SMV

Le rôle du système client développé pour invoquer le SMV est minimaliste consiste à fournir une simple interface d'entrées et sorties par des commandes en ligne. Toutes les entrées/sorties seront effectuées par entrée standard (qui peut être une console, un fichier, etc.). Il a été modélisé par une simple classe nommée `DClient` (*alias* `DReasonner`) dont les fonctionnalités seront étendues selon les besoins. Dans une première phase, ce système servira à effectuer des opérations génériques. Dans une seconde, ses fonctionnalités seront étendues pour résoudre un problème concret. Le diagramme de classes de la figure 6.1 qui est une représentation plus formelle de la figure 2.1, montre comment le SMV et le système client seront associés au moyen des deux classes `DClient` et `DTMSProxy`.

Figure 6.1 Relation système client /SMV

Comme nous l'avions spécifié lors de la conception, la classe `DTMSProxy` constitue le point d'entrée pour le SMV et elle est la seule à être accessible pour tout agent client.

6.2 Implémentation

Le système client a été implémenté dans le même langage que le SMV c'est-à-dire en C++ et seules les librairies standard de ce langage ont été utilisées. Comme nous l'avons

spécifié dans le chapitre 5, le SMV a été développé indépendamment de toute plateforme. Il a été testé sur *Unix/Mac Os X* ainsi que sur *Windows* en utilisant respectivement les environnements de développement *XCode/gcc* (voir Annexe 4) et *Visual Studio* 2005. Dans ces deux cas, nous nous sommes limité aux librairies standard que nous avons exploitées intensément en réutilisant les classes "`map`[36]", `iterator`[37] et "`bitset`[38]" que nous avons déjà identifiés comme rôles dynamiques mais que nous allons décrire brièvement.

La classe "`map`" est un conteneur associatif qui enregistre une occurrence, et une seule, d'une paire d'éléments qui lui sont soumis. Le premier élément de la paire qui est de type "`pair`" est appelé "`first`" et sert de clé unique. Le second élément, appelé "`second`" contient la valeur. Par défaut, cette classe trie ses éléments par ordre croissant mais elle prend un troisième paramètre optionnel. Ce paramètre qui est une fonction-objet[39] sert à déterminer tout autre ordre de tri. En tant que conteneur standard, cette classe a été conçue pour fonctionner avec des itérateurs qu'elle est en mesure de générer automatiquement.

La classe "`bitset`" est un conteneur variable de bits qui vient avec plusieurs fonctionnalités permettant d'effectuer des opérations détaillées aussi bien au niveau de chacun des éléments qu'au niveau de l'ensemble. Cette classe a été réutilisée au niveau d'un nœud pour représenter l'ensemble des états décrits dans les figures 4.1 et 4.2. Plus particulièrement, ce sont surtout les méthodes pour modifier et tester un bit qui ont été utilisées.

Les fonctionnalités du SMV se basent en grande partie sur celles des trois classes standard bien que d'autres ont été réutilisées accessoirement notamment les classes "`vector`" et

[36] http://www.sgi.com/tech/stl/Map.html
[37] http://www.sgi.com/tech/stl/Iterators.html
[38] http://www.sgi.com/tech/stl/bitset.html

"`multimap`" (annexe 2). Tel que décrit dans les structures d'inférences, la classe "`vector`" a été utilisée comme différentiel pour stocker les explications que le système génère (page 1 de l'annexe 4). La classe "`multimap` " fonctionne comme "`map`" mais accepte des doublons. Celle-ci a été utilisée au niveau du système client pour effectuer des opérations de tri.

Toutes ces classes apparaissent dans le schéma du domaine final tel que spécifié en annexe 2. Ce schéma a été intégré rigoureusement dans le processus de développement et figure parmi les fichiers sources dans chacun des deux environnements utilisés (page 1 de l'annexe 4).

Le SMV a été implémenté avec une centaine de fonctions et deux mille lignes de code approximativement. La majorité des fonctions sont très élémentaires et ont été testées séparément puis en groupe. Dans certains cas, elles sont récursives, croisées, polymorphes et surchargées à la fois. Tous ces mécanismes sont exploités avec les patrons de conception déjà expliqués.

6.3 Le système client

Le système client développé a été muni de plusieurs opérations essentielles dont une fonction principale nommée "`Start`" (Schreiber et *al.*, p. 284, 2000). Cette fonction qui est la seule à être publique a pour rôle de générer un menu avec des items permettant d'effectuer des opérations de base comme créer des nœuds, les relier, désactiver ou d'activer un littéral, générer des explications, etc. (figure 6.2). Chacun de ces items est associé avec une fonction au niveau de la classe du système. Chacune de ces fonctions est associée à

[39] http://www.sgi.com/tech/stl/functors.html

son tour avec une autre au niveau du SMV. D'autres fonctions purement utilitaires pour gérer les opérations d'entrées/sorties et le traitement des erreurs ont été également définies.

La figure 6.2 est une prise d'écran du menu principal du système client (À ne pas confondre avec le SMV) alors que la figure 6.3 présente la classe associée à ce menu.

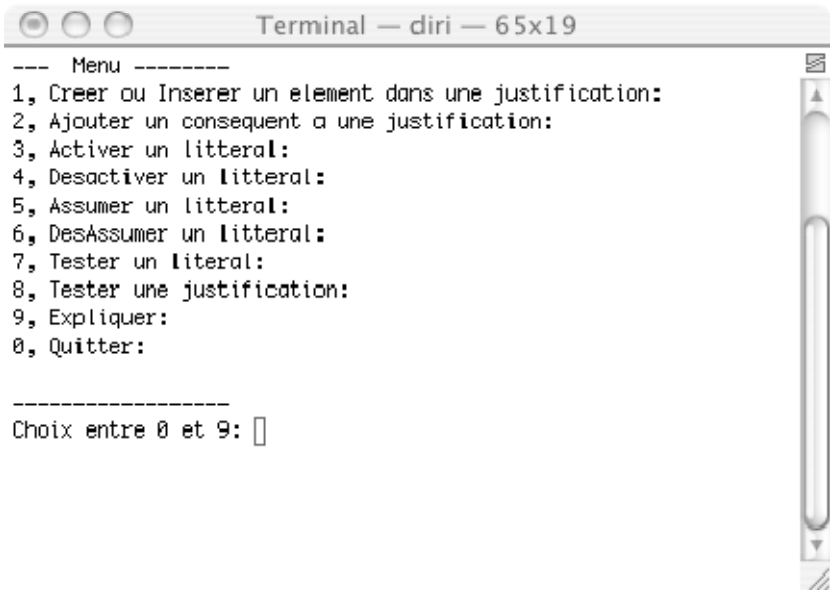


Figure 6.2 Menu du système client

Figure 6.3 Le système client de base

6.4 Tests génériques

Pour tester le système, nous allons lui soumettre quelques cas triviaux pour montrer qu'il les gère correctement. Nous profitons de l'occasion pour reprendre les exemples des paragraphes 2.1.1.1 et 2.1.1.4 (figures 2.3 et 2.5) du chapitre 2 et les vérifier en les exécutant. Nous verrons alors comment le système gère la contradiction et la circularité. Dans chacun de ces cas, deux scénarios seront testés l'un à la suite de l'autre puis les résultats seront affichés. Nous rappelons que ces exemples triviaux proviennent de l'ouvrage de Forbus et De Kleer (1993) où ils sont amplement documentés. Ici, nous allons les tester pour montrer que le système produit les résultats auxquels ces auteurs arrivent et pour corroborer le texte du chapitre 2.

6.4.1 Contradictions

Dans le paragraphe 2.1.1.1, nous avons présenté l'exemple d'un réseau qui contient deux nœuds qui représentent la contradiction. L'un de ces noeuds résulte de l'existence d'un nœud et de sa négation alors que le second résulte de l'existence de deux nœuds qui sont mutuellement exclusifs. Nous rappelons que l'ensemble de ces nœuds était défini par les relations $D \wedge \neg D \xrightarrow{kIn} \perp$ et $A \wedge D \xrightarrow{kIn} \perp$.

Dans notre système, nous n'avons pas besoin de créer un second nœud pour représenter la négation puisque nous utilisons des nœuds duaux. Quand notre système recevra une requête pour enregistrer *D* et une requête pour enregistrer $\neg D$, un seul nœud sera créé lors de la première requête. Lors de la seconde requête, les états de la partie négative du nœud *D* seront simplement configurés et il deviendra une contradiction.

Dans le SMVJ de Doyle, la relation $A \wedge D \xrightarrow{kIn} \perp$ est spécifiée par le système client. Quand *A* et *D* sont à *IN*, le SMV doit signaler cette contradiction et un retour en arrière est entrepris pour désactiver l'un des ces deux nœuds. Dans notre système, *A* et *D* sont encodés directement comme étant mutuellement exclusifs. Par conséquent, aucun nœud contradiction ni aucun retour en arrière ne seront nécessaires. Si l'un des deux nœuds est à *IN*, alors il propagera automatiquement l'état contraire *(OUT)* vers l'autre. La relation $A \wedge D \xrightarrow{kIn} \perp$ devra donc être transmise au SMV sous la forme des deux relations non monotones suivantes : $A \xrightarrow{kOut} D$ et $D \xrightarrow{kOut} A$. En plus des nœuds *A* et *D*, deux nœuds justification que nous nommerons J_1 et J_3 sont nécessaires puisque *A* justifie *D* et *D* justifie *A*. La justification J_1 va inclure la partie positive de *D* et la justification J_3 va inclure celle de *A*. Une justification J_2 est créée pour contenir la partie négative de *D*. Ainsi plus tard, il serait toujours possible de relier les deux parties de *D* pour qu'elles soient mutuellement exclusives.

Puisque *A* et *D* sont mutuellement exclusifs, alors l'état final du nœud *D* dépendra de l'ordre de création des relations qui unit ces deux nœuds. En effet, il possible de transmettre au SMV $A \xrightarrow{kOut} D$ puis $D \xrightarrow{kOut} A$ comme il est possible de lui transmettre $D \xrightarrow{kOut} A$ puis $A \xrightarrow{kOut} D$. Dans le premier cas, *A* restera à *IN* alors que *D* sera mis à *OUT* et le noeud *D* ne sera pas contradictoire. Dans le second cas, *D* sera mis à *IN* et *A* sera mis à *OUT* et une requête supplémentaire sera nécessaire pour résoudre cette contradiction.

Les figures 6.4 et 6.5 montrent un exemple d'exécution des ces deux possibilités. Dans les deux cas, le système client effectue des requêtes auprès du SMV pour créer *D*, ¬*D* et *A* et ensuite les relations qui les unissent. Des messages ne sont affichés que pour signaler la contradiction qui survient ainsi que les requêtes qui sont exécutées. Pour afficher l'état final des noeuds *A* et *D*, nous avons dans chacun des cas suivi les directives du menu affiché pour expliquer *A* en détail, par chaînage avant.

Dans la figure 6.5, la relation $A \xrightarrow{kOut} D$ a été déclarée avant la relation $D \xrightarrow{kOut} A$. Par conséquent, le système a généré le résultat suivant :

```
1- A.IN;
2- involved in {J3.IN};
3- J3.IN  J3.IN A = {A.IN};
4- Implies {(D).OUT};
5- (D).OUT;
6- involved in {J1.OUT};
7- {J1.OUT} = {(D).OUT};
8- Implies {A.IN};
9- A.IN;
10-involved in {J3.IN};
11-J3.IN
```

Ce résultat peut être interprété respectivement par :

1- *A* est à *IN*;
2- *A* est inclus dans J_3 qui est à *IN*;
3- Explication de J_3 qui est à *IN* : J_3 qui est à *IN* et contient un élément *A* qui est à *IN;*
4- J_3 implique *D* qui est à *OUT;*
5- Explication de *D* qui est à *OUT*;
6- *D* qui est à *OUT* est inclus dans J_1 qui est à *OUT*;
7- J_1 qui est à *OUT* contient un élément *D* qui est *OUT* ;
8- J_1 qui est à *OUT* implique *A* qui est à *IN;*
9- Explication de *A* qui est à *IN*;
10- *A* est inclus dans J_3 qui à *IN*;
11- Explication de J_3. J_3 a été expliqué. Le du processus prend fin.

L'explication détaillée[40] de *A* est obtenue par une suite d'explications ponctuelles et ce de façon récursive. Les deux dernières explications sont réaffichées à la fin pour montrer la présence du circuit.

[40] Voir paragraphe 4.3.3.3.

Dans la figure 6.6, la relation $D \xrightarrow{kOut} A$ est déclarée avant la relation $A \xrightarrow{kOut} D$. Par conséquent, *A* finit dans un état *OUT* alors que *D* finit dans un état *IN* et en contradiction comme nous l'avions déjà expliqué.

Dans la figure 6.5, la relation $D \xrightarrow{kOut} A$ est déclarée avant la relation $A \xrightarrow{kOut} D$. Par conséquent, *A* finit dans un état *OUT* alors que *D* finit dans un état *IN* et en contradiction comme nous l'avions déjà expliqué.

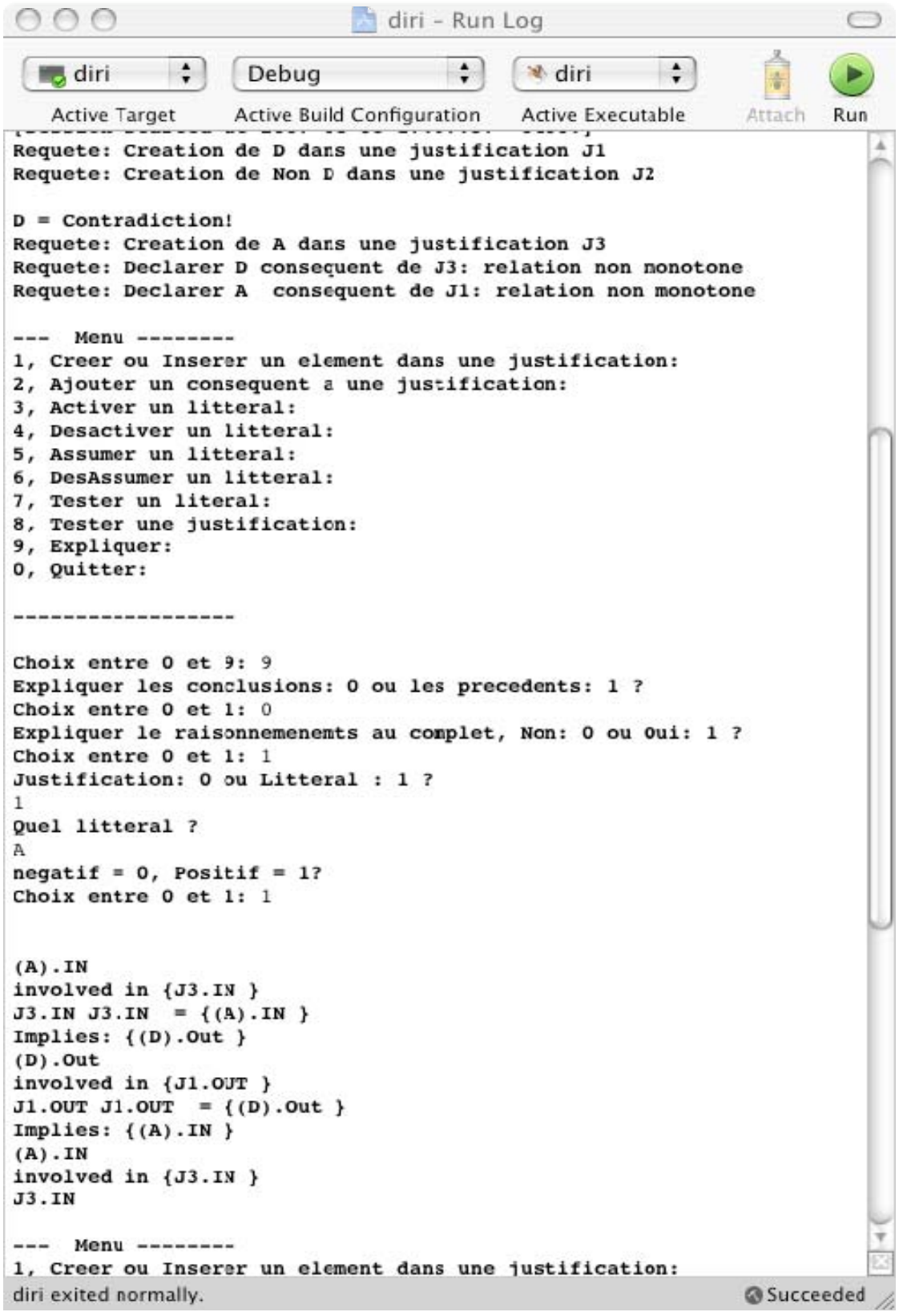


Figure 6.4 Premier exemple de contradictions

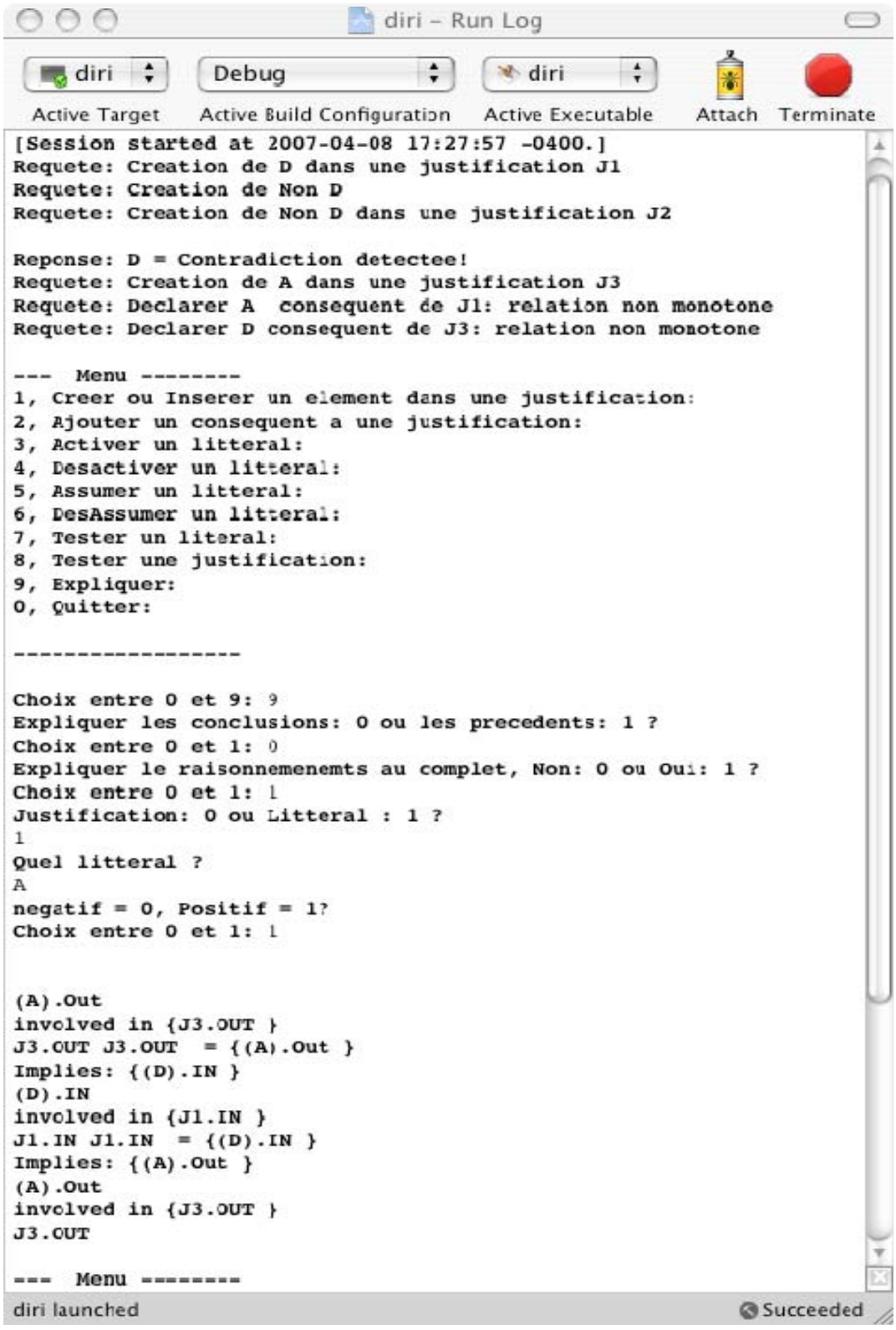


Figure 6.5 Second exemple de contradictions

6.4.2 Circularité

Dans le paragraphe 2.1.1.4, nous avons montré comment un SMV doit composer avec la propagation en présence de circuits. Les relations $A \xrightarrow{kIN} B$, $A \xrightarrow{kIN} C$, $B \xrightarrow{kOut} C$ et $C \xrightarrow{kOut} B$ ont alors été utilisées comme exemple. Nous avions expliqué que, dépendamment de l'ordre de création des nœuds *A* et *B*, la désactivation de *A* peut avoir deux effets sur leur état d'activité final. Deux exemples d'exécution sont fournis en annexe 5 pour montrer les résultats obtenus avec le système. Dans cet exemple, on peut voir en effet que le système finit avec *A* à *IN* et *B* à *OUT* dans un cas, puis avec *A* à *OUT* et *B* à *IN* dans l'autre cas. Pour éviter de saisir les données manuellement, pour alléger les illustrations et pour simplifier notre travail, la création des nœuds et l'inversion de l'ordre d'exécution ont été insérés directement dans le code.

6.4.3 Explications

Pour afficher les résultats des exemples 6.4.1 et 6.4.2, des explications ont été demandées au SMV qui les a générées. Ces explications, qui sont constituées d'une suite d'explications ponctuelles concaténées tracent en détail un raisonnement en partant vers les conclusions (annexes 4 et 5). Pour montrer que le système est en mesure de produire des explications dans le sens contraire, l'exemple décrit dans le paragraphe 6.4.2 a été reprise en demandant au système de générer des explications pour le nœud *B* (figure 6.6). Dans le paragraphe 6.5, deux exemples d'explications ponctuelles seront fournis.

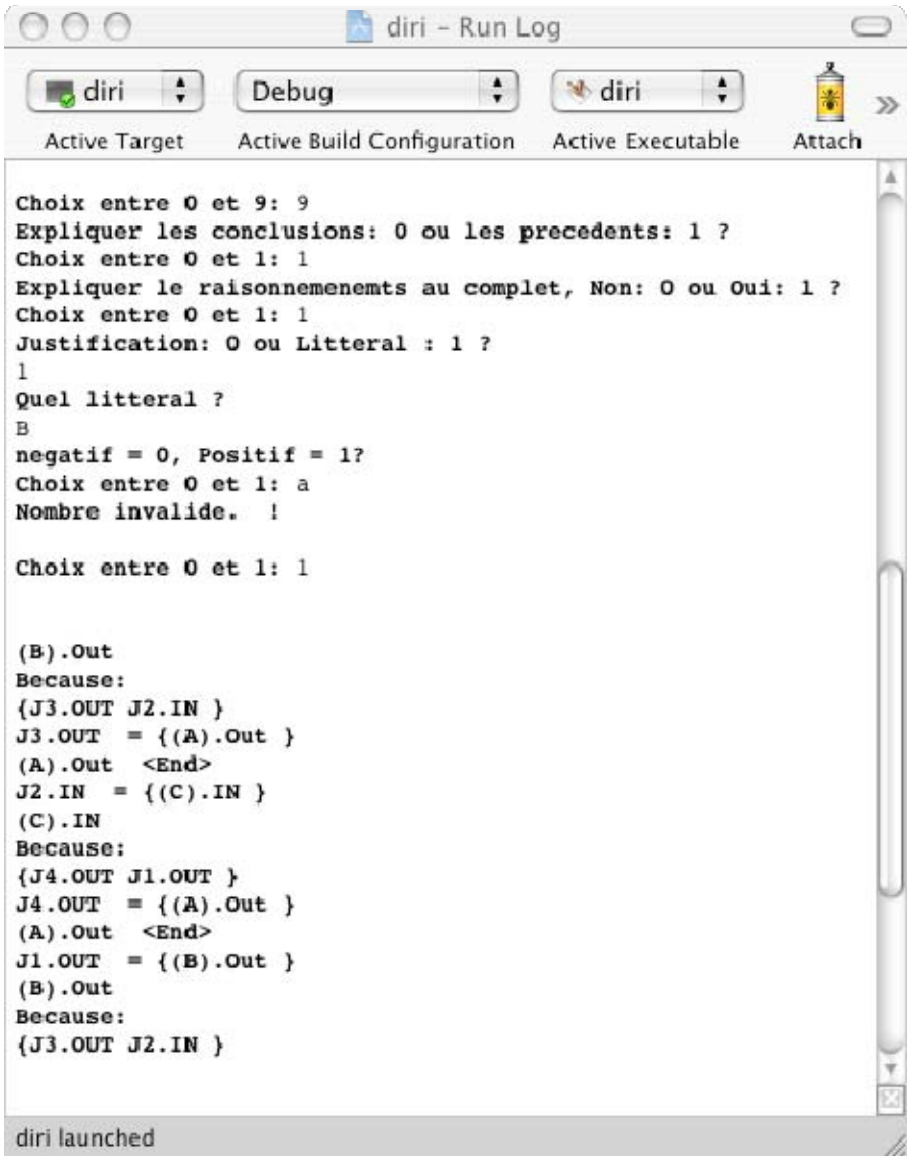


Figure 6.6 Exemple d'explications

6.5 Test de capacité par un exemple concret

Dans cet exemple, nous allons tester la capacité de charge du SMV et sa robustesse. Indirectement, nous allons montrer comment un SMV peut être exploité pour la résolution d'un problème concret et d'actualité.

Le système client qui utilisera le SMV est un mini moteur de recherche qui doit chercher des expressions parmi une multitude d'autres stockées dans des fichiers et afficher les résultats selon un certain ordre. Il s'agit du même système client déjà présenté mais auquel cinq autres fonctions ont été ajoutées (figure 6.7). Afin de réduire le transfert des données entre le client et le SMV, une fonction spéciale a été ajoutée à ce dernier de rediriger directement l'entrée standard vers lui[41]. Pour les autres données telles que les expressions à chercher, toutes les requêtes ont été incorporées dans le code pour éviter les saisies manuellement (page 3 de l'annexe 4).

Le raisonnement que le système client effectue pour accomplir sa tâche nous importe peu et on peut en faire abstraction. Ce qui compte pour nous ainsi que du point de vue SMV, ce sont les services que ce dernier doit fournir.

Le système client aura besoin de certains types de services. Il aura besoin d'enregistrer ses données dans le cache du SMV sous forme de nœuds et ceux-ci doivent être reliés sous forme ensembles/éléments et/ou antécédents/conséquents (en réseau). Il aura aussi besoin de certaines explications en cours d'exécution. Certaines de ces explications serviront à le guider par dépendance. La troisième figure de l'annexe 4 est un fragment de code qui montre l'implémentation d'un objet nommé `"tms"` qui invoque entre autres une méthode `NewLiteral` (ligne 374), `AddConsequent` (ligne 385) et `Explain` (ligne 412). Ceci illustre comment le système client (*alias* `DReasonner` dans l'illustration) effectue un raisonnement et invoque l'objet SMV quand cela est nécessaire.

[41] Autrement, il aurait fallu saisir les entrées (258 060 nœuds) manuellement par commandes lignes (tableau 6.1).

Le système client a été implémenté et testé avec quelques fichiers de petite taille. Par la suite, quatre fichiers en format texte contenant respectivement 49 837, 60 554, 84 090, 63 577 expressions (chaînes de caractères) pour un total de 258 054 lui ont été soumises (tableau 6.1). Lors de la résolution de "son" problème, le client invoque le SMV lui demande de mettre en cache ces expressions, les expressions recherchées ainsi que le nom des fichiers utilisés. Le SMV s'exécute et crée autant de nœuds que nécessaire mais sans doublons. Quand des explications sont demandées le SMV s'exécute et le système client continue son raisonnement puis affiche les résultats par ordre d'importance.

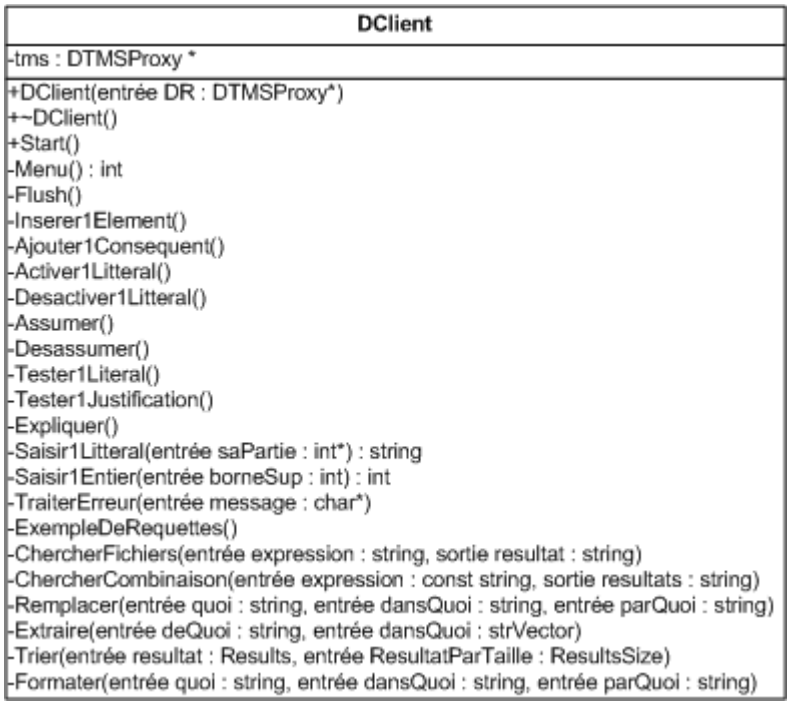


Figure 6.7 Extension du système client

Pour éviter tout résultat biaisé, des expressions à rechercher spécifiques ont été insérées dans les fichiers en s'assurant qu'elles n'y existaient pas déjà. Les expressions que le système client cherche sont : *"Expr104", "Expr102", "Expr103","Expr104""Expr105"* et *"Expr106"*. Il s'agit là de chaîne de caractères qui sont stockés dans l'attribut "`contenu`" de chaque nœud. Rappelons que le SMV manipule ses nœuds par référence.

"Expr104" et *"Expr105"* ont été insérées dans un fichier nommé *"Fre1.txt"*. Les expressions *"Expr101 "* et *"Expr102"* ont été insérées dans *"Fre2.txt"*. Les expressions *"Expr101", "Expr103"*et *"Expr104"* l'ont été dans *"Fre3.txt"* alors que les expressions *"Expr101"*et *"Expr103"* ont été insérées dans *"Fre4.txt"*. Finalement, une expression *"Expr106"* n'a été insérée dans aucun des fichiers cités (tableau 6.1). Au niveau du SMV, chaque fichier a été associé avec une justification qui lui est propre.

Fichiers	**Justifications associées**	**Nombre d'expressions**	**Expressions recherchées**
Fre1.txt	*FichJ1*	49 837	*Expr104, Expr105*
Fre2.txt	*FichJ2*	60 554	*Expr101, Expr102*
Fre3.txt	*FichJ3*	84 090	*Expr101, Expr103, Expr104*
Fre4.txt	*FichJ4*	63 577	*Expr101, Expr103*
			Expr106
Total : 4		**258 054**	**6**

Tableau 6.1 : Données utilisées lors de la requête

Les résultats obtenus dans nos expérimentations indiquent que notre système a réussi à retrouver toutes les expressions qu'il devait repérer dans les textes (voir Fig. 6.8). Toutes les expressions insérées ont été localisées puis affichées par fichier et par nombre d'occurrences. L'expression *"Expr106"* qui n'a été insérée nulle part a été ignorée. Comme on peut voir dans la même figure, les statistiques générées par le SMV indiquent que 21 649 nœuds ont été créés avec succès.

Pour montrer comment le système client a généré ses résultats, nous avons exécuté manuellement les mêmes requêtes pour l'expression "*Expr101"* (figure 6.9) et pour la justification *"FichJ3"* dans laquelle cette expression est contenue (figure 6.10). Dans cette dernière figure, la justification affiche tous ses éléments et ensuite ses conséquents (3 expressions) entre accolades.

En exploitant davantage les fonctionnalités du SMV, les capacités de ce moteur de recherche peuvent être étendues pour qu'il puisse filtrer des fichiers. En effet, si des expressions sont mises à *OUT*, les justifications où elles figurent le seront automatiquement aussi. Lors de la recherche, il suffira de rejeter toutes les justifications qui sont désactivées ainsi que les fichiers qui leurs sont associés. Le recours à des filtres peut servir comme anti-pourriels, outils de contrôle parental, etc. Nous n'avons pas besoin ici de montrer que le système peut activer ou désactiver quoi que ce soit puisque nous ne testons que la capacité du cache. Cependant, le système client pourra être complété et raffiné dans ce sens.

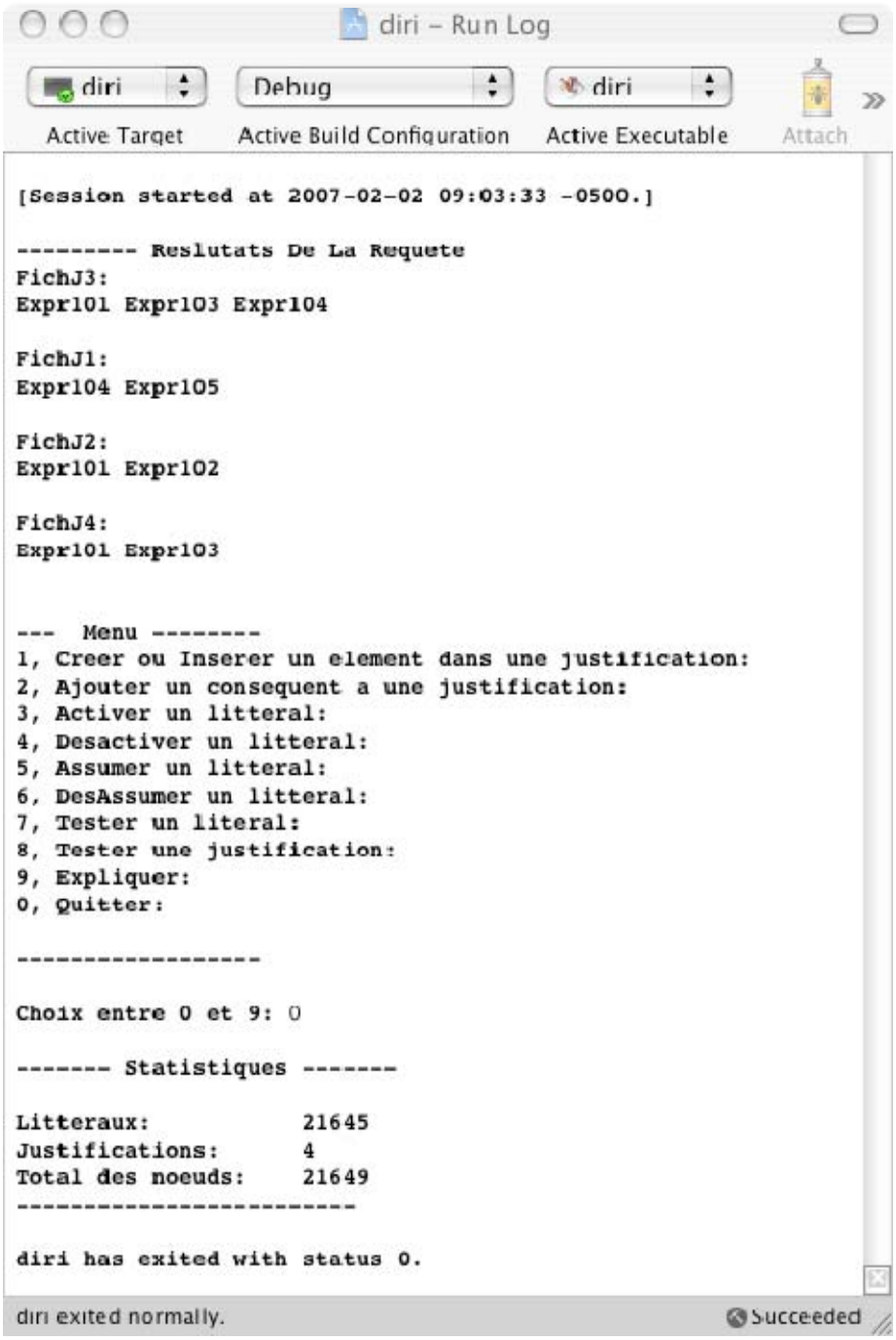


Figure 6.8 Test concret du système en tant que cache

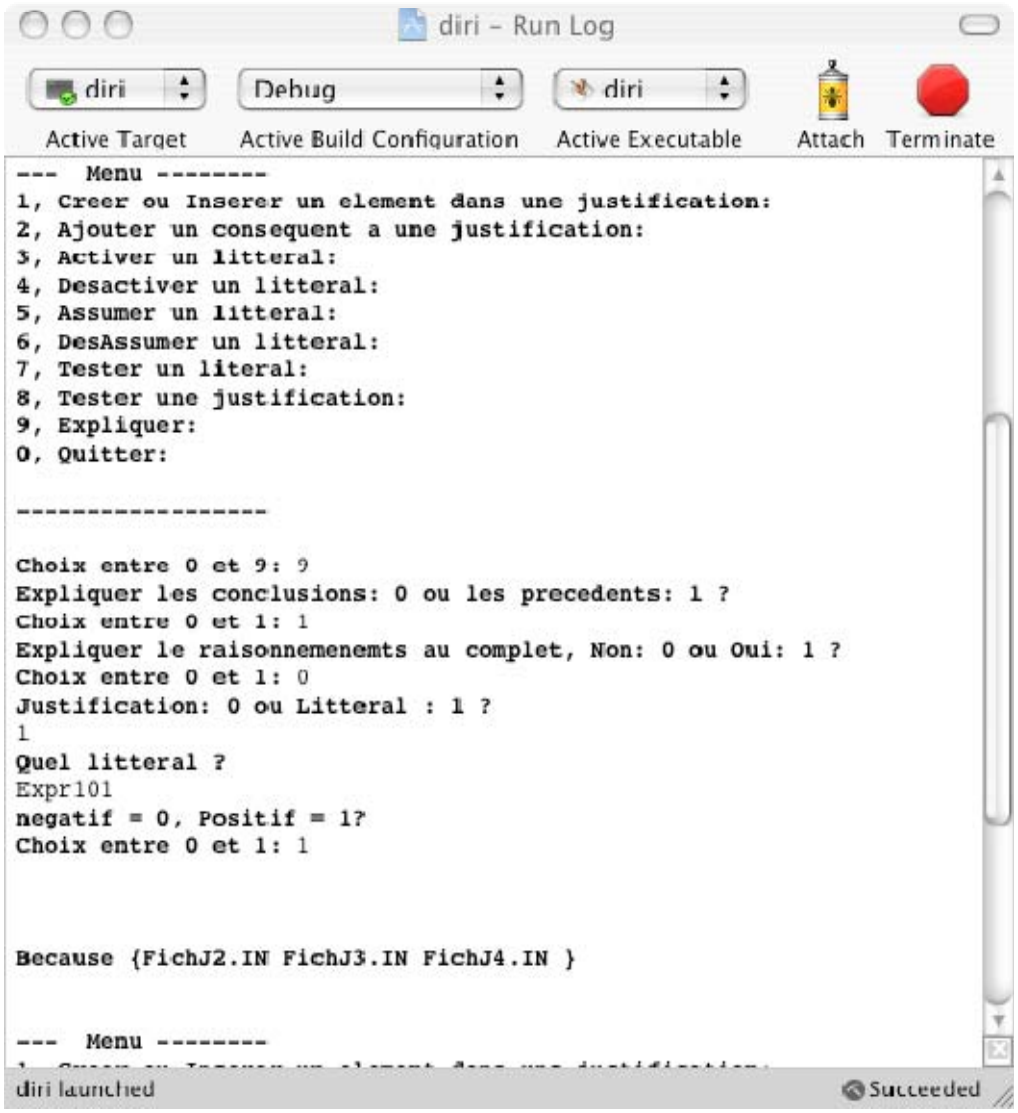


Figure 6.9 Exemple d'explication ponctuelle des précédents

Figure 6.10 Exemple d'explication ponctuelle des conclusions

6.6 Discussion des résultats

Pour tester notre système, nous avons repris des exemples triviaux décrits par Forbus et De Kleer (1993) et nous nous sommes assuré que le système produit les mêmes résultats que ceux présentés par ces auteurs. Au lieu de tester les système à l'aveuglette, nous avons sélectionné des exemples triviaux pour tester le système en ses points critiques. D'après nous, la détection et la gestion des circuits constituent le cas le plus complexe que le

système doit réussir. Ce cas touche quasiment toutes les autres fonctionnalités que le système doit accomplir notamment la propagation, l'ordre de propagation, la capacité de trouver des supports bien fondés et la génération des explications. Pour la gestion des contradictions, nous avons repris l'exemple particulier de Forbus et de Kleer (1993) qui illustre aussi les raisonnements par défaut. Tous ces exemples ont été expliqués et référencés dans le chapitre 2. Finalement, pour montrer que le système peut supporter de lourdes charges en termes quantitatifs, un exemple a été spécialement conçu. Grâce à cet exemple, nous avons montré comment le SMV peut guider le client lors de ses raisonnements, comment il a pu gérer un cache de plusieurs dizaines de milliers de nœuds et comment il peut être exploité dans la résolution d'un problème concret et d'actualité.

Pour chaque exemple, des traces d'exécution ont été produites sous formes d'explications. Nous invitons le lecteur à les consulter avec attention et à se référer au schéma du domaine complet qui figure en annexe 2.

6.7 Récapitulation

Dans ce chapitre, nous avons décrit comment notre SMV a été implémenté et comment il peut être utilisé. Pour le manipuler, le développement d’un système client minimaliste a été nécessaire. Au moyen de ce système, le SMV a été testé avec des exemples triviaux. Ces exemples ne permettent pas cependant de tester les capacités de stockage du SMV. Ils ne permettent pas non plus de montrer comment il peut être utilisé lors de la résolution d’un problème concret et actuel. Le système client fut donc transformé en un mini-moteur de recherche qui a nécessité la création d’un réseau de plusieurs milliers de nœuds.

Comme tout autre système, notre SMV présente des points forts et des points faibles que nous avons exposés. Les points forts cités découlent en grande partie de la réutilisation des bibliothèques standards et de l'emploi d'un ensemble de patrons de conception judicieusement choisis. Les points faibles sont d'ordre technique. Le système peu être encore optimisé au niveau du code.

Chapitre 7 : Conclusion

Dans la documentation scientifique, on peut distinguer trois principaux types de SMV : à base de justifications, à base logique et à base d'assomptions. Ces systèmes utilisent des structures en réseau pour enregistrer les instances d'un ensemble de règles de production et ils s'inscrivent dans un paradigme orienté listes. Dans certains de ces systèmes, la négation est représentée comme une entité à part alors que dans d'autres, la normalisation est utilisée pour éviter d'y recourir. Cependant, cette dernière approche limite le système dans ses capacités et augmente sa complexité. Pour réduire la complexité des SMV des limitations ont dû être introduites. Pour éviter ce type de problèmes nous avons adopté un paradigme objet.

Notre principal objectif était donc de proposer une approche orientée objet pour élaborer un SMV. Dans une première étape, nous avons étudié les SMV existants pour en comprendre la problématique. Puis, nous avons modélisé un SMV au niveau des connaissances. Nous avons ensuite conçu le système selon une approche orientée objet en appliquant des patrons de conception. Enfin, nous avons implémenté et testé notre système. À partir de deux exemples tirés de la documentation scientifique, nous avons pu montrer que notre système offre des fonctionnalités équivalentes à celles des SMV étudiés. De plus, nous avons utilisé notre SMV comme mini-moteur de recherche.

Bibliographie

Apple inc., *Cocoa Fundamentals Guide*, Cupertino, Californie, 2008, http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals

Ayel M. et Rousset C., *La cohérence dans les bases de connaissances*, Cépaduès Editions, Toulouse, 1990.

Ayel M. et Laurent J.P., *Validation, Verification and Test of Knowledge-Based System*, John Wiley and Sons, England, 1991.

Bellefeuille S., *Proposition d'un modèle de système d'aide à la vérification de la cohérence dans les bases de règles*, Mémoire de maîtrise, Université Laval, Québec, 2001.

Booch G., Rumbaugh J. et Jacobson I., *The unified modeling language user guide*, Addison-Wesley Professional, 1999.

Buschman F., Meunier R., Rohnert H., Sommerlad P. et Stal M., *A System of Patterns*, John Wiley and Sons, New York, 1996.

Carbonara L. et Sleeman D., Effective and Efficient Knowledge Base Refinement, *Machine learning*, Vol. 37, No. 2, pp. 143-181, Kluwer Academic Publishers, 1999.

Craw S. et Boswell R., Representing problem-solving for knowledge refinement, *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the eleventh Innovative applications of artificial intelligence*, pp. 277-234, Florida, 1999.

Chaib-Draa B., Jarras I. et Moulin B., Systèmes multiagents : Principes généraux et applications, In *Principes et architectures des systèmes multi-agents*, J.P. Briot et Y. Demazeau (éditeurs), Hermès, Lavoisier, 2001.

Cooper J.W., *Design Patterns Java Companion*, Addison Wesley, 1998. http://www.patterndepot.com/put/8/DesignJava.PDF

De Kleer J., An Assumption-based TMS, *Artificial Intelligence*, Vol. 28, pp. 127-162, 1986. http://www2.parc.com/spl/members/dekleer/Publications/An%20Assumption-Based%20TMS.pdf

De Kleer J. et William B-C., Back to backtracking: Controlling the ATMS, *Proceedings, Proceedings of the National Conference on Artificial Intelligence*, pp. 910-917, 1986. http://www2.parc.com/spl/members/dekleer/bib.htm

De Kleer J. Extending the ATMS, *Artificial Intelligence*, Vol. 28, pp. 163-196, 1986.

De Kleer J., A general labeling algorithm for assumption-Based Truth maintenance, *Proceedings of the National Conference on Artificial Intelligence*, pp. 188-192, 1988. *http://www2.parc.com/spl/members/dekleer/Publications/Back%20to%20Backtracking.pdf*

De Kleer J., A comparison of the ATMS and CSP techniques, *IJCAE*, 1989. *http://www2.parc.com/spl/members/dekleer/Publications/A%20Comparison%20of%20ATMS%20and%20CSP%20Techniques.pdf*

Doyle, J., A Truth Maintenance System, *Artificial Intelligence*, Vol. 12, No. 3, pp. 231-272, 1979.

Doyle, J., The Ins and Outs of Reason Maintenance, *IJCA*, pp. 349-351, 1983.

Eckel B., *Thinking in C++*, Prentice Hall, New Jersey, 2002.

Eckel B., *Thinking in Patterns*, Prentice Hall, New Jersey, 2003.

Fensel D. et Benjamins V-R., The Role of Assumptions in Knowledge Engineering, *International Journal of Intelligent Systems*, Vol. 13, No. 8, pp. 271-280, 1998. *http://citeseer.ist.psu.edu/cache/papers/cs/1999/ftp:zSzzSzftp.aifb.uni-karlsruhe.dezSzpubzSzmikezSzdfezSzpaperzSzassumptions.pdf/fensel98role.pdf*

Fensel D., Harmelen F-V Wolfgang R. et Teije A-T., Formal support for Development of Knowledge-Based Systems, *Information Technology Management*, Vol. 2, No. 4, 1998. *http://citeseer.ist.psu.edu/cache/papers/cs/198/http:zSzzSzwww.cs.vu.nlzSz~annettezSzpostscriptzSzITM98.pdf/fensel98formal.pdf*

Forbus K.-D., De Kleer J., *Building Problem Solvers*, The MIT Press, Cambridge, 1993.

Fowler M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

Gamma E., Helm R., Johnson R.J. et Vlissides J., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

Harmelen F-V., Applying rule-base anomalies to KADS inference structures, *Decision Support Systems*, Vol. 21, No. 4, pp. 271-280, 1998.

Iglesias C. A., Garijo M., Gonzales J. C. et Velasco R., Analysis and Design of Multi-Agent Systems using MAS-CommonKADS, In *Proceedings of the Proceedings of the National Conference on Artificial Intelligence Workshop on Agent Theories, Architectures and Languages*, Providence, USA, July 1997.

Johnson L.F. et Shapiro C., *Says Who? Incorporating Source Credibility Issues into Belief* Revision, 1999. *http://www.cs.buffalo.edu/~shapiro/Papers/*

Johnson L-F.et Shapiro C., Implementing Integrity Constraints in an Existing Belief Revision System, In Baral C. et Truszczynski M., Eds., *Proceedings of the 8th International Workshop on Non monotonic Reasoning NMR2000,* 2000. *http://www.cs.buffalo.edu/~shapiro/Papers/*

Jussien N., Relaxation de contraintes pour les problèmes dynamiques, Thèse de doctorat, Université de Rennes, 1997. *http://www.emn.fr/x-info/jussien/publications/jussien-THESIS.pdf*

Kerievsky J., *Refactoring To Patterns*, Version 0.17, industrial Logic, 2002.

Krzysztof C. et Brad A., *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*, (Microsoft .NET Development Series), 2005.

London P., *Dependency Networks as Representation for Modelling in General Problem Solvers*, Ph.D. Dissertation, Technical Report 698, College Park, MD: Department of Computer Science, University of Maryland, 1978.

Martin R-C., UML Tutorial: *Finite State Machines*, Engineering Notebook Column, C++ Report, June 1998. *http://www.objectmentor.com/resources/articles/umlfsm.pdf*

Martins J., *The Truth, the Whole Truth, and Nothing But the Truth,* Artificial Intelligence Magazine, Vol. 11, No. 5, Jan., pp. 7-25, 1990.

McAllester D.A., *A three valued maintenance system*, Massachusetts Institute Of technology, Artificial Intelligence Laboratory, Artificial Intelligence Memo 473, 1978.

Newell A,. The Knowledge Level. *Artificial Intelligence*, Vol. 18, No 1, pp. 1:87-127, 1982.

Preece A., Evaluating Verification and Validation Methods in Knowledge Engineering, In R. Roy, (Eds), *Micro-Level Knowledge Management*, Morgan-Kaufman, pp. 123-145, 2001.

Reiter R. et De Kleer J., Foundations of the Assumption-based Truth Maintenance System: Preliminary report *AAAI*, Vol. 9, pp. 183-189,1987. *http://www2.parc.com/spl/members/dekleer/Publications/Foundations%20of%20Assumption-Based%20Truth%20Maintenance%20Systems.pdf*

Riehle D., *Composite design patterns*, ACM Press, New York, USA, 1997. *http://portal.acm.org/citation.cfm?id=263739*

Riehle D., Brudermann R., Gross T. et Mätzel K.W. *Pattern Density and Role Modeling of an Object Transport Service,* ACM Computing Surveys 32, Article No. 10, 2000.

Schreiber G., Akkermans H., Anjewierden A., De Hoog R., Shadbolt N., Van de Velde W. et Wielinga B., *Knowledge Engineering and Management*, A Bradford Book, The MIT Press, Cambridge, Massachusetts, 2000.

Shapiro Stuart C., *Belief Revision and Truth Maintenance Systems: An Overview and a Proposal*, December 1998. *http://www.cs.buffalo.edu/~shapiro/Papers/*

Shapiro Stuart C., Rapaport, William J., Kandefer M., Johnson Frances L. et Goldfain A., Metacognition in SNePS, *Artificial Intelligence Magazine*, Vol. 28, No. 1, 2007. *http://www.cse.buffalo.edu/~shapiro/Papers/metacognition.pdf*

Stallman, Richard M., Sussman et Gerald J., Forward Reasoning and Dependency-Directed Backtracking In a System for Computer-Aided Circuit Analysis, *Artificial Intelligence*, Vol. 9, No. 2, pp. 135-196, 1977. *https://dspace.mit.edu/bitstream/1721.1/6255/2/AIM-380.pdf*

Studer V., Benjamins R. et Fensel D., Knowledge Engineering: Principles and methods, *Data & Knowledge Engineering*, Vol. 25, pp. 161-197, 1998. *http://citeseer.ist.psu.edu/cache/papers/cs/8679/ftp:zSzzSzftp.aifb.uni-karlsruhe.dezSzpubzSzmikezSzdfezSzpaperzSzDKE98.pdf/studer98knowledge.pdf*

William J.-B., Raphael C. Malveau, H-W., "Skip" McCormick (III) et Thomas J-M., *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley and Sons, Canada, 1998.

Wiratunga N. et Craw S., Informed Selection of Training Examples for Knowledge Refinement. In *Proceeding on 12th International Conference on knowledge Engineering and Knowledge management*, pp. 233-248, Springer-Verlag, 2000. *http://www.scms.rgu.ac.uk/staff/smc/papers/ekaw2000nw.pdf*

Annexe 1 : Feuilles de travail

Modèle de la communication	**Feuille de travail CM-1 : Description de la transaction "Ajouter nœud"**
Nom de la transaction	`"Ajouter un nœud"`
Objet d'information	L'ajout d'un nœud qui est l'objet de l'information, est opéré au sein de la sous-tâche *`"Construction du réseau".`* Cette sous-tâche est suivie de la sous-tâche *`"Changement d'états".`* Chaque nœud ajouté peut affecter l'état d'activité d'autres nœuds et ceci peut engendrer des contradictions.
Agents impliqués	Cette transaction implique dans l'ordre : - l'agent client et l'agent *`DTMSproxy;`* - l'agent *`DTMSproxy`* et l'agent `DNode;` - l'agent `DNode` (À chaque insertion d'un objet `DNode` dans l'une de listes d'un autre objet `DNode,` une opération inverse est effectuée selon le processus décrit dans la couche des tâches)`.`
Plan de communication	`"Plan de communication principal"`
Contraintes	-
spécifications des échanges d'informations	Les spécifications sont fournies dans la feuille de travail CM-2.

Modèle de la communication	**Feuille de travail CM-1 : Description de la transaction "Changer d'état"**
Nom de la transaction	"Changer d'état"
Objet d'information	Un nœud est toujours l'objet d'information lors d'un changement d'état. Cette transaction peut concerner plusieurs nœuds et elle peut être opérée récursivement au sein de la même sous-tâche *"Changement d'états"*. Elle peut être aussi initiée par la sous-tâche *"Construction du réseau".*
Agents impliqués	Cette transaction implique dans l'ordre : - l'agent client et l'agent *DTMSproxy;* - l'agent *DTMSproxy* et l'agent DLiteral; - l'agent DLiteral et l'agent DJustification (récursivement).
Plan de communication	"Plan de communication principal"
Contraintes	-
spécifications des échanges d'informations	Les spécifications sont fournies dans la feuille de travail CM-2.

Modèle de la communication	**Feuille de travail CM-1 : Description de la transaction d'informations "Expliquer"**
Nom de la transaction	*"Expliquer"*
Objet d'information	Au sein de cette transaction, un nœud est l'objet de l'information. Cette transaction concerne plusieurs agents au sein de la sous-tâche *"Fournir des explications"*.
Agents impliqués	Initialement, cette transaction implique l'agent client et l'agent *DTMSproxy.* Elle peut impliquer par la suite et dans l'ordre : - l'agent *DTMSproxy* et l'agent DLiteral; - l'agent DLiteral et l'agent DJustification (récursivement). Elle peut impliquer aussi par la suite et dans l'ordre : - l'agent *DTMSproxy* et l'agent DJustification; - l'agent DJustification et l'agent DLiteral (récursivement).
Plan de communication	"Plan de communication principal"
Contraintes	-
spécifications des échanges d'informations	Les spécifications sont fournies dans la feuille de travail CM-2.

Modèle de la communication	**Feuille de travail CM-2 : Spécification des échanges d'informations "Ajouter un nœud"**
Transaction	`"Ajouter un nœud"`
Agents impliqués	1.1 **Expéditeur** : Système client 1.2 **Destinataire** : *`DTMSproxy`* 2.1 **Expéditeur** : *`DTMSproxy`* 2.2 **Destinataire** : `DNode` 3.1 **Expéditeur** : `DNode` 3.2 **Destinataire** : `DNode`
Items d'information	Pour la création d'un nœud, le *`DTMSproxy`* reçoit du système client : - le contenu du nœud; - le signe du nœud en option; par défaut il est supposé être positif. Pour inter relier deux nœuds, il reçoit en plus : - le contenu de chacun d'eux; - le signe du littéral; - et éventuellement, l'étiquette qui doit être propagée. Un nœud doit recevoir : - le nœud à ajouter; - l'ensemble dans lequel il doit l'ajouter; - l'étiquette associée et le signe en option. 1 **Rôle** : les données échangées sont essentielles et ne servent pas de support. 2 **Forme** : les données sont transmises sous forme de chaînes de caractères, d'entiers constants et de références. 3 **Medium** : les données sont transférées directement en mémoire.
Spécifications du message	1 **Type de communication** : demande/réponse (`ask/reply`); 2 **Contenu** : nœud, (signe optionnel) et l'ensemble dans lequel il doit être inséré.
Contrôle des messages	Les objets cibles d'une transaction d'explication doivent exister au préalable.

Modèle de la communication	**Feuille de travail CM-2 : Spécification des échanges d'informations "Changer d'état"**
Transaction	`"Changer un état"`
Agents impliqués	1.1 **Expéditeur** : Système client 1.2 **Destinataire** : *`DTMSproxy`* 2.1 **Expéditeur** : *`DTMSproxy`* 2.2 **Destinataire** : `DNode` 3.1 **Expéditeur** : `DNode` 3.2 **Destinataire** : `DNode`
Items d'information	Le *`DTMSproxy`* reçoit du système client une requête pour modifier certains états d'un littéral : - le nom du littéral dont l'état doit être modifié; - la partie du nœud concernée (positive ou négative); - le type d'état à modifier. Le *`DTMSproxy`* envoie un message au littéral en question en mentionnant quel état modifier et quelle est la partie concernée. Les états qui peuvent être modifiés sont les états d'activité et les états concernant les assomptions. Un `DNode` littéral relaie le message vers les objets `DJustification` concernés et ceux-ci les relayent, à leur tour, vers d'autres objets de type `DLiteral`. Cette opération est récursive. 1 **Rôle** : les données échangées sont essentielles et ne servent pas de support. 2 **Forme** : les données sont transmises sous forme de chaînes de caractères, d'entiers constants. 3 **Medium** : les données sont transférées directement en mémoire.
Spécifications du message	1 **Type de communication** : demande/réponse *`(ask/reply)`* 2 **Contenu** : nœud, signe du nœud (par défaut c'est la partie positive), l'état devant être modifié
Contrôle des messages	Les objets cibles d'une transaction d'explication doivent exister au préalable (voir les machines d'états qui figurent en annexe 3).

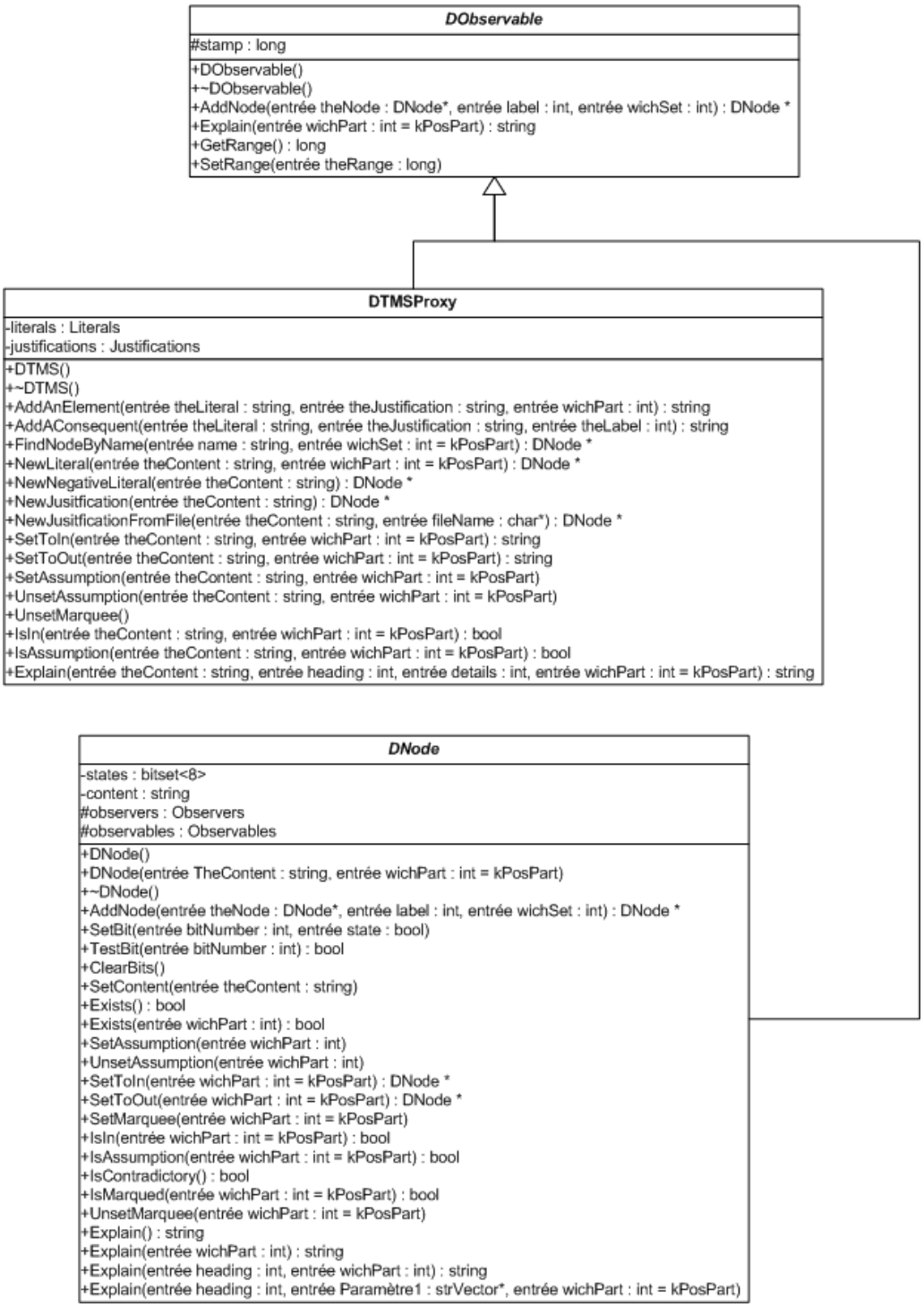
Modèle de la communication	**Feuille de travail CM-2 : Spécification des échanges d'informations "Expliquer"**
Transaction	`"Expliquer"`
Agents impliqués	1.1 **Expéditeur** : Système client 1.2 **Destinataire** : *`DTMSproxy`* 2.1 **Expéditeur** : *`DTMSproxy`* 2.2 **Destinataire** : `DNode` 3.1 **Expéditeur** : `DNode` 3.2 **Destinataire** : `DNode`
Items d'information	1 **Rôle** : les données échangées sont essentielles et ne servent pas de support. 2 **Forme** : les données sont transmises sous forme de chaînes de caractères, d'entiers constants et/ou de conteneurs dépendamment du contexte. 3 **Medium** : les données sont transférées directement en mémoire.
Spécifications du message	1 **Type de communication** : demande/réponse *`(ask/reply)`* 2 **Contenu** : nœud, signe du nœud (par défaut partie positive) et le type d'explications (détaillée ou non, par chaînage avant ou par chaînage arrière)
Contrôle des messages	Les objets cibles d'une transaction d'explication doivent exister au préalable.

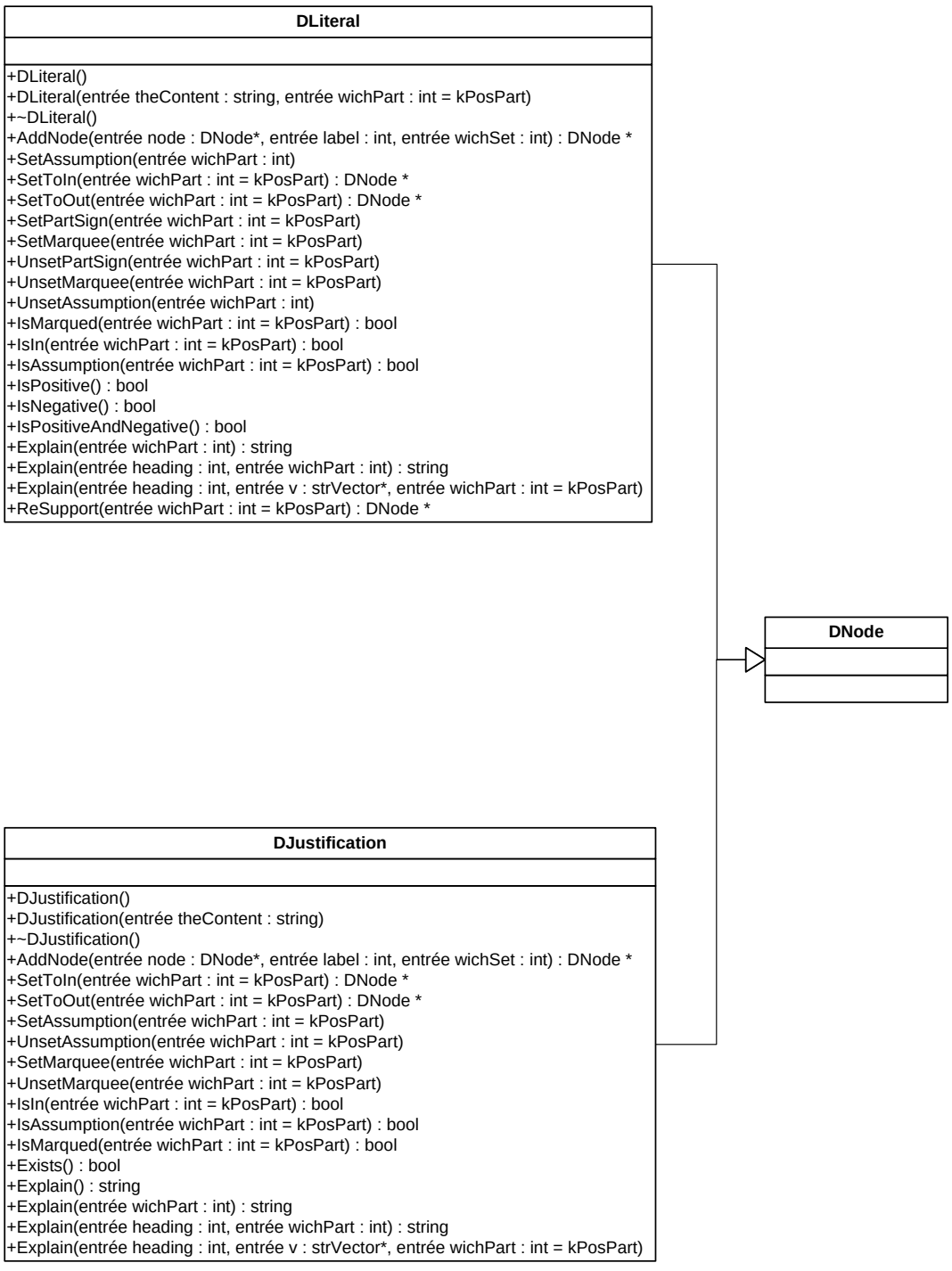
Modèle de conception	**Feuille de travail DM-1 : Architecture du système**
Sous système	Le SMV est un sous-système de la partie "modèle" d'un système client qui est responsable de fournir la partie "contrôleur", la partie "vue" et le reste.
Contrôle du modèle	Le système est contrôlé par le système client pour lequel il joue un rôle de serveur.
Décomposition	Le système est un tout.

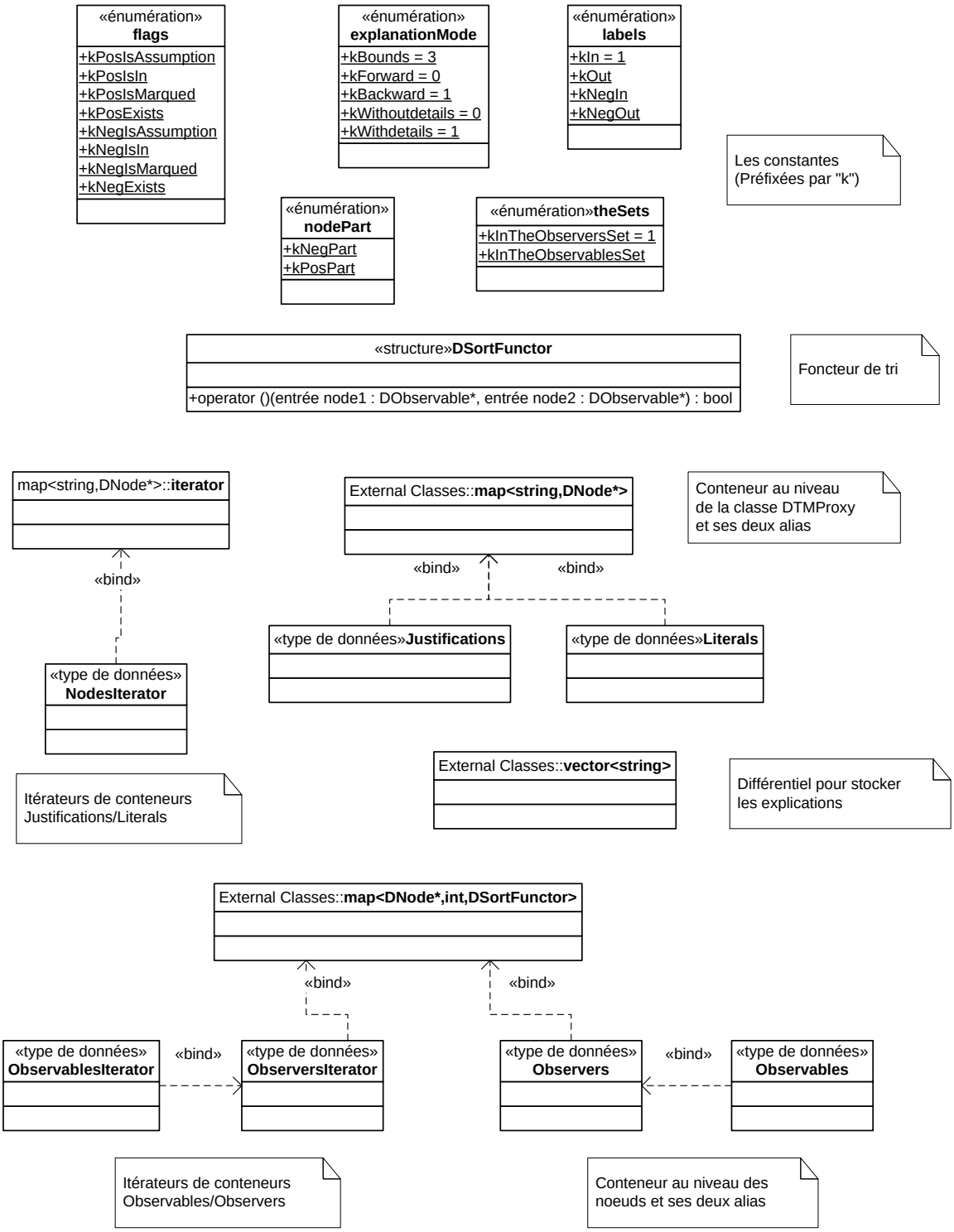
Modèle de conception	Feuille de travail DM-2 : Plateforme d'implémentation
Logiciel	"Détecteur d'Incohérence par Récursion et par Itération"
Matériel possible	Le système est indépendant de toute plateforme.
Matériel cible	Indépendant de toute plateforme (Mac Os X,/Unix, Windows).
Librairie de visualisation	Aucune. La visualisation relève du système client.
Type de langage	C++. Le système est facilement portable vers tout autre langage à objets.
Support CommonKADS	Le système est indépendant de tout autre système.

Modèle de conception	Feuille de travail DM-3 : Spécification de l'architecture
Contrôleur	Relève du système client. Le SMV ne possède n'est vue ni contrôleur. C'est un serveur.
Tâche	`"Maintien de vérité"`
Méthodes de tâches	Les méthodes suivent le paradigme orienté objet.
Méthodes d'inférences	Voir le tableau 5.1
Rôles dynamiques	`iterator`, `bitset`, `string`, `vector`, `map`
Rôles statiques	Plusieurs accesseurs et mutateurs ont été définis (annexe 2)
Base de connaissances	voir schéma du domaine final en annexe 2.
Vues	La partie "vue" relève du système client.

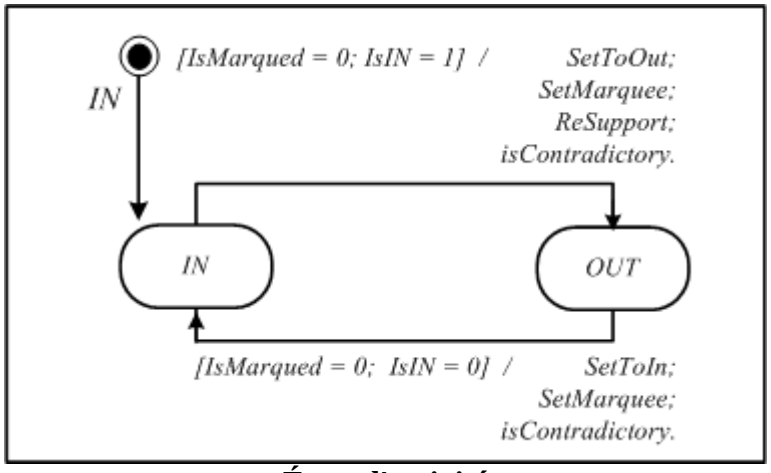
Annexe 2 : Schéma du domaine détaillé



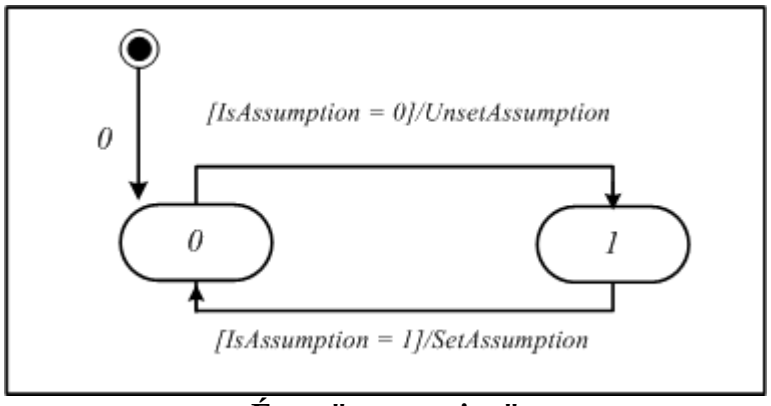

DLiteral
+DLiteral()
+DLiteral(entrée theContent : string, entrée wichPart : int = kPosPart)
+~DLiteral()
+AddNode(entrée node : DNode*, entrée label : int, entrée wichSet : int) : DNode *
+SetAssumption(entrée wichPart : int)
+SetToIn(entrée wichPart : int = kPosPart) : DNode *
+SetToOut(entrée wichPart : int = kPosPart) : DNode *
+SetPartSign(entrée wichPart : int = kPosPart)
+SetMarquee(entrée wichPart : int = kPosPart)
+UnsetPartSign(entrée wichPart : int = kPosPart)
+UnsetMarquee(entrée wichPart : int = kPosPart)
+UnsetAssumption(entrée wichPart : int)
+IsMarqued(entrée wichPart : int = kPosPart) : bool
+IsIn(entrée wichPart : int = kPosPart) : bool
+IsAssumption(entrée wichPart : int = kPosPart) : bool
+IsPositive() : bool
+IsNegative() : bool
+IsPositiveAndNegative() : bool
+Explain(entrée wichPart : int) : string
+Explain(entrée heading : int, entrée wichPart : int) : string
+Explain(entrée heading : int, entrée v : strVector*, entrée wichPart : int = kPosPart)
+ReSupport(entrée wichPart : int = kPosPart) : DNode *
DNode
DJustification
+DJustification()
+DJustification(entrée theContent : string)
+~DJustification()
+AddNode(entrée node : DNode*, entrée label : int, entrée wichSet : int) : DNode *
+SetToIn(entrée wichPart : int = kPosPart) : DNode *
+SetToOut(entrée wichPart : int = kPosPart) : DNode *
+SetAssumption(entrée wichPart : int = kPosPart)
+UnsetAssumption(entrée wichPart : int = kPosPart)
+SetMarquee(entrée wichPart : int = kPosPart)
+UnsetMarquee(entrée wichPart : int = kPosPart)
+IsIn(entrée wichPart : int = kPosPart) : bool
+IsAssumption(entrée wichPart : int = kPosPart) : bool
+IsMarqued(entrée wichPart : int = kPosPart) : bool
+Exists() : bool
+Explain() : string
+Explain(entrée wichPart : int) : string
+Explain(entrée heading : int, entrée wichPart : int) : string
+Explain(entrée heading : int, entrée v : strVector*, entrée wichPart : int = kPosPart)


«énumération»
flags
+kPosIsAssumption
+kPosIsIn
+kPosIsMarqued
+kPosExists
+kNegIsAssumption
+kNegIsIn
+kNegIsMarqued
+kNegExists
«énumération»
explanationMode
+kBounds = 3
+kForward = 0
+kBackward = 1
+kWithoutdetails = 0
+kWithdetails = 1
«énumération»
labels
+kIn = 1
+kOut
+kNegIn
+kNegOut
Les constantes
(Préfixées par "k")
«énumération»
nodePart
+kNegPart
+kPosPart
«énumération»theSets
+kInTheObserversSet = 1
+kInTheObservablesSet
«structure»DSortFunctor
+operator ()(entrée node1 : DObservable*, entrée node2 : DObservable*) : bool
Foncteur de tri
map<string,DNode*>::iterator
«bind»
«type de données»
NodesIterator
External Classes::map<string,DNode*>
«bind»
«bind»
«type de données»Justifications
«type de données»Literals
Conteneur au niveau
de la classe DTMProxy
et ses deux alias
Itérateurs de conteneurs
Justifications/Literals
External Classes::vector<string>
Différentiel pour stocker
les explications
External Classes::map<DNode*,int,DSortFunctor>
«bind»
«bind»
«type de données»
ObservablesIterator
«bind»
«type de données»
ObserversIterator
«type de données»
Observers
«bind»
«type de données»
Observables
Itérateurs de conteneurs
Observables/Observers
Conteneur au niveau des
noeuds et ses deux alias

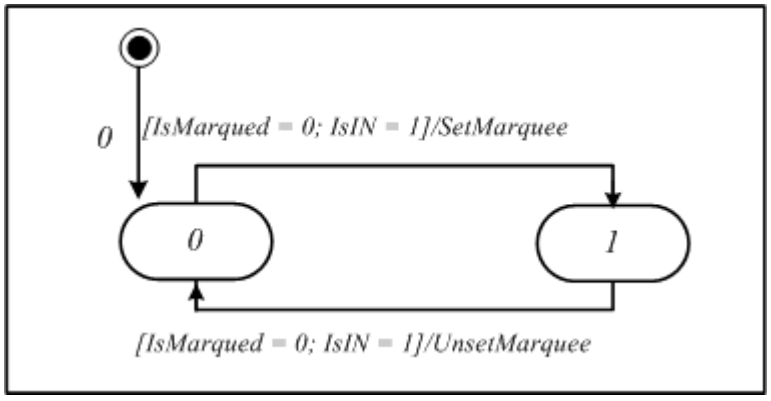
Annexe 3 : Spécifications des machines d'états



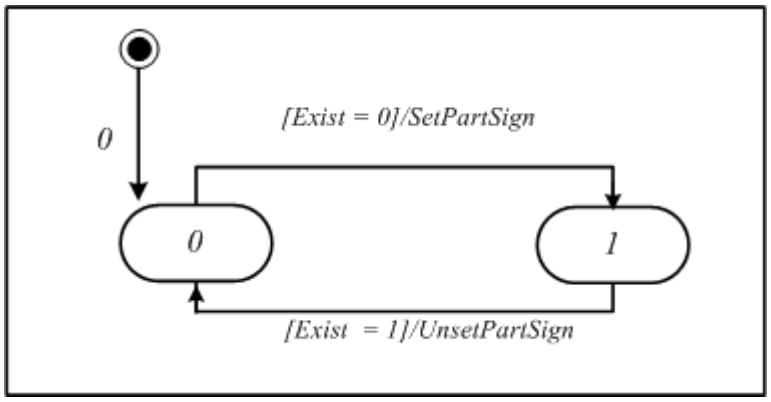
États d'activité



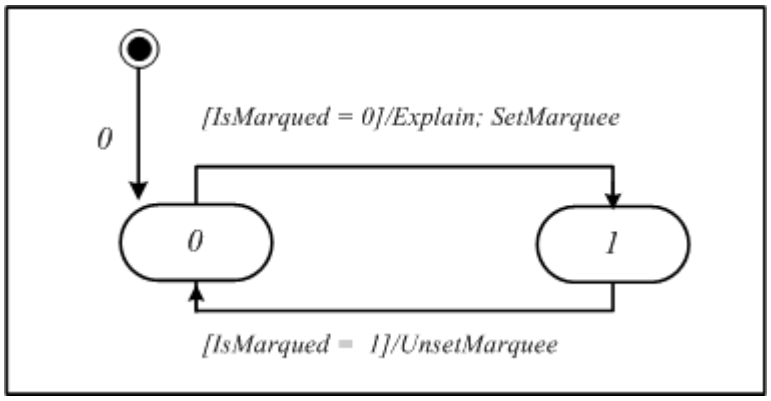
États "assomption"



États "marqué"



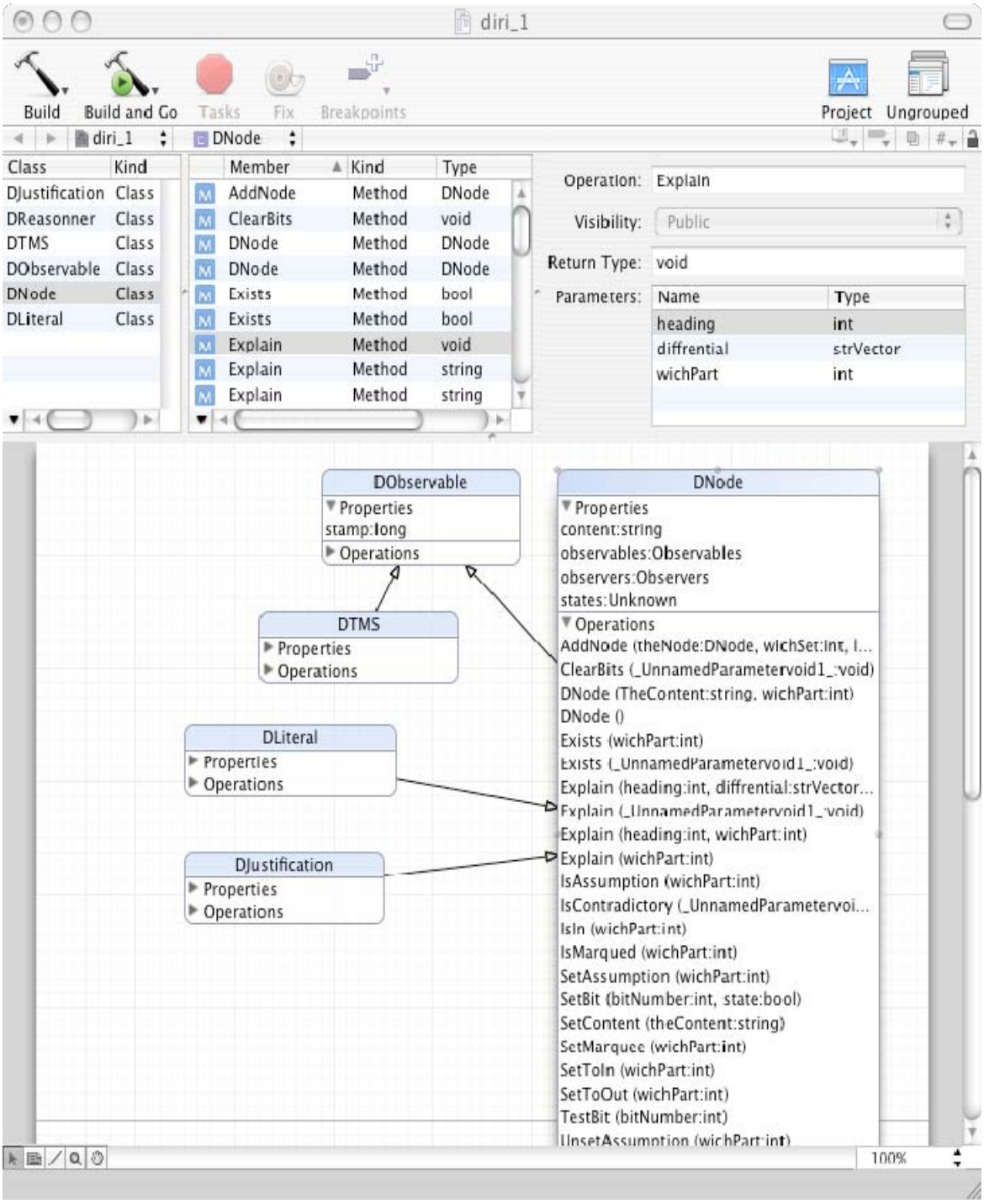
États du signe

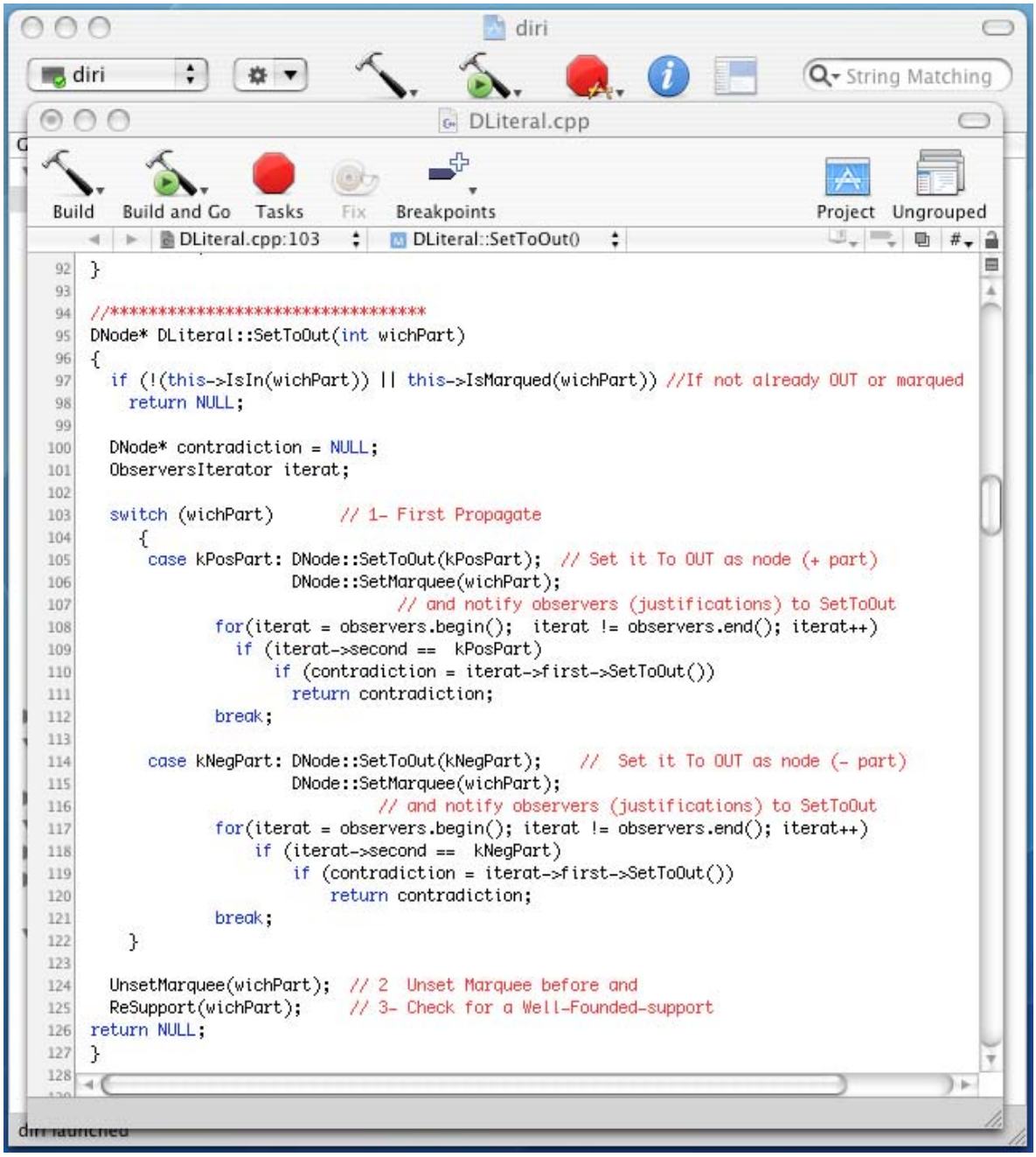


États "expliqué"

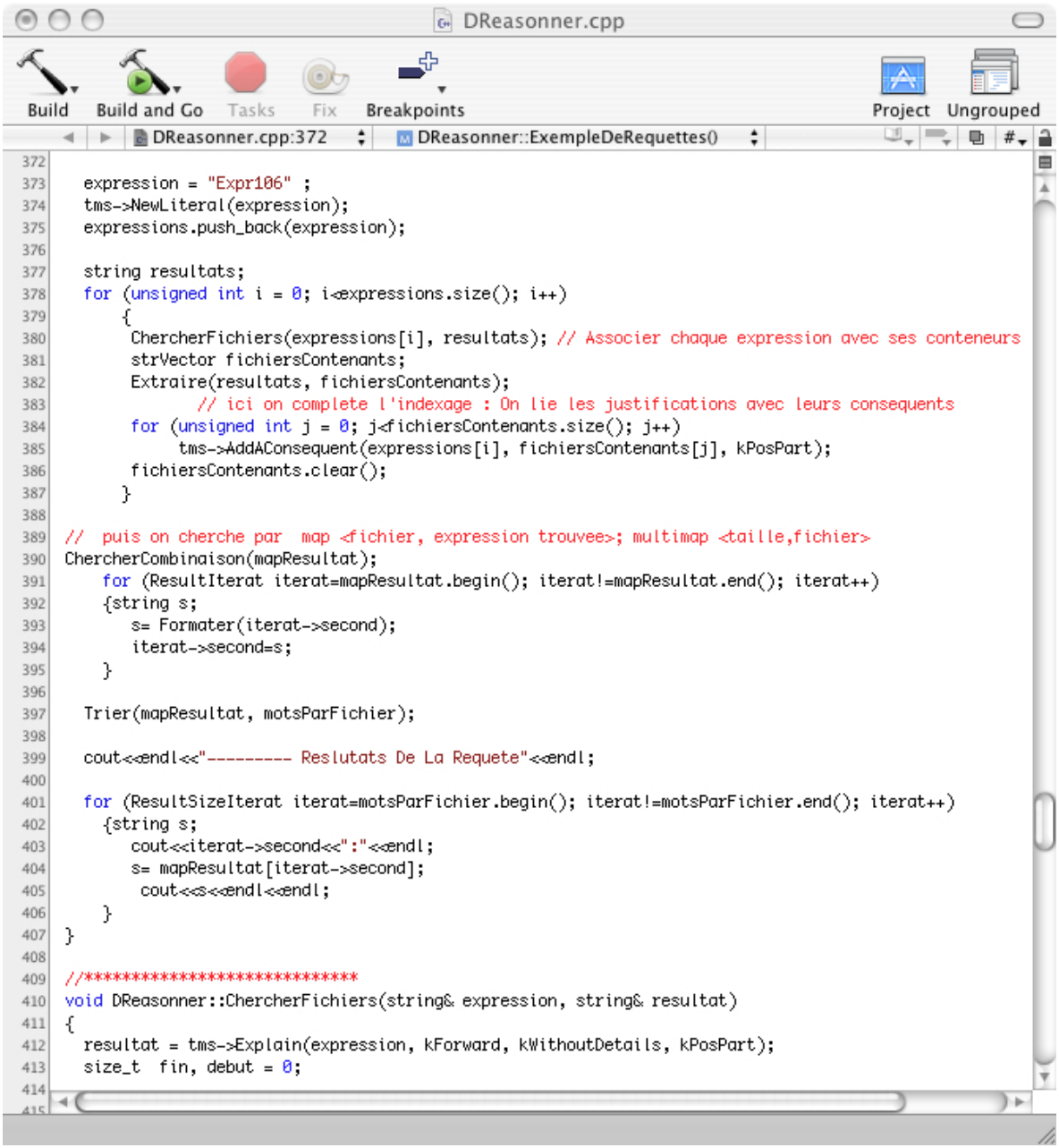
Légende: *[Conditions]/Actions* État Début

Annexe 4 : L'environnement de développement




diri
diri
String Matching
DLiteral.cpp
Build
Build and Go
Tasks
Fix
Breakpoints
Project
Ungrouped
DLiteral.cpp:103
DLiteral::SetToOut()
}
//**********
DNode* DLiteral::SetToOut(int wichPart)
{
if (!(this->IsIn(wichPart)) || this->IsMarqued(wichPart)) //If not already OUT or marqued
return NULL;
DNode* contradiction = NULL;
ObserversIterator iterat;
switch (wichPart) // 1- First Propagate
{
case kPosPart: DNode::SetToOut(kPosPart); // Set it To OUT as node (+ part)
DNode::SetMarquee(wichPart);
// and notify observers (justifications) to SetToOut
for(iterat = observers.begin(); iterat != observers.end(); iterat++)
if (iterat->second == kPosPart)
if (contradiction = iterat->first->SetToOut())
return contradiction;
break;
case kNegPart: DNode::SetToOut(kNegPart); // Set it To OUT as node (- part)
DNode::SetMarquee(wichPart);
// and notify observers (justifications) to SetToOut
for(iterat = observers.begin(); iterat != observers.end(); iterat++)
if (iterat->second == kNegPart)
if (contradiction = iterat->first->SetToOut())
return contradiction;
break;
}
UnsetMarquee(wichPart); // 2 Unset Marquee before and
ReSupport(wichPart); // 3- Check for a Well-Founded-support
return NULL;
}
diri launched

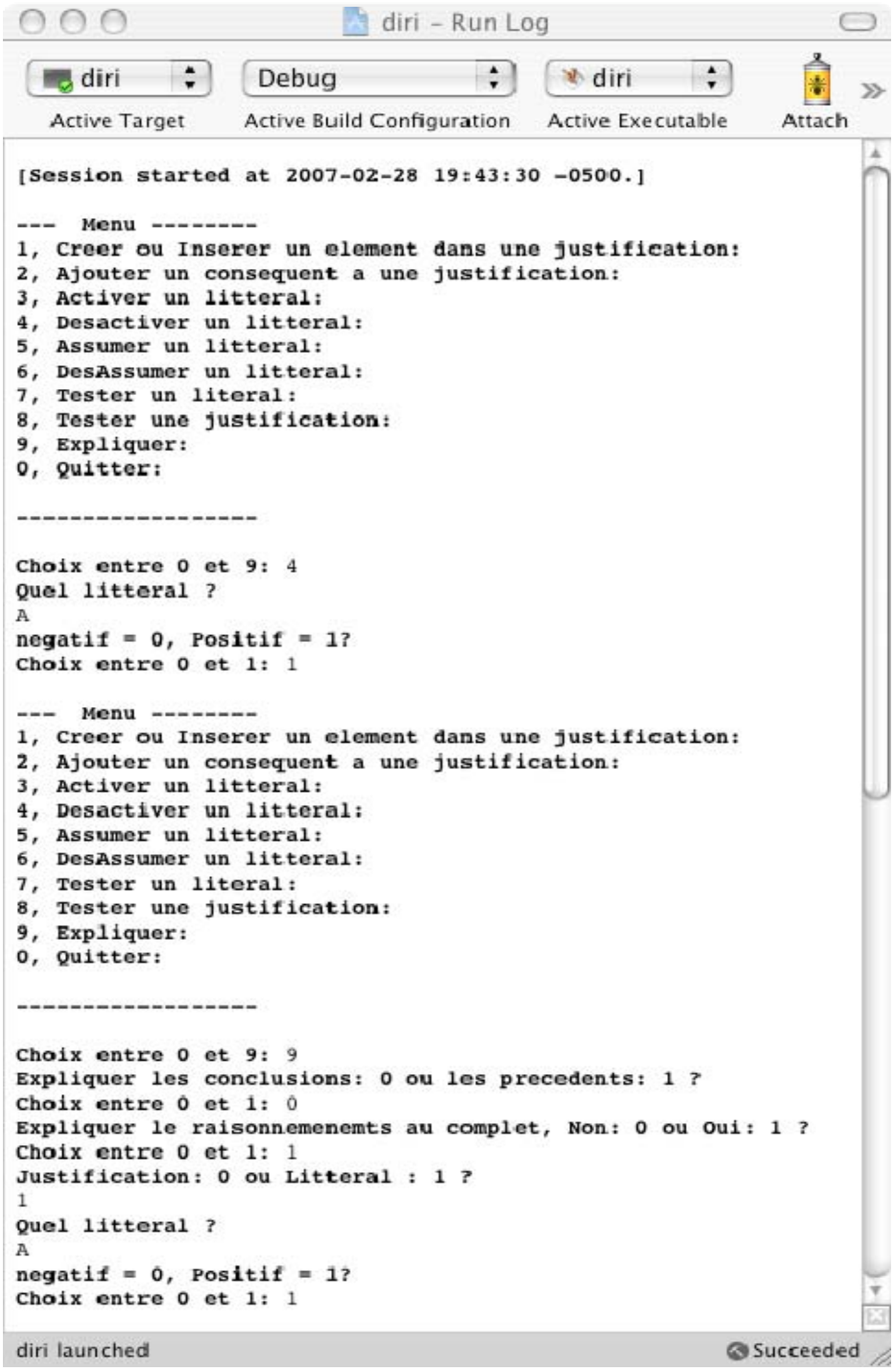
Fragment de l'implémentation du client

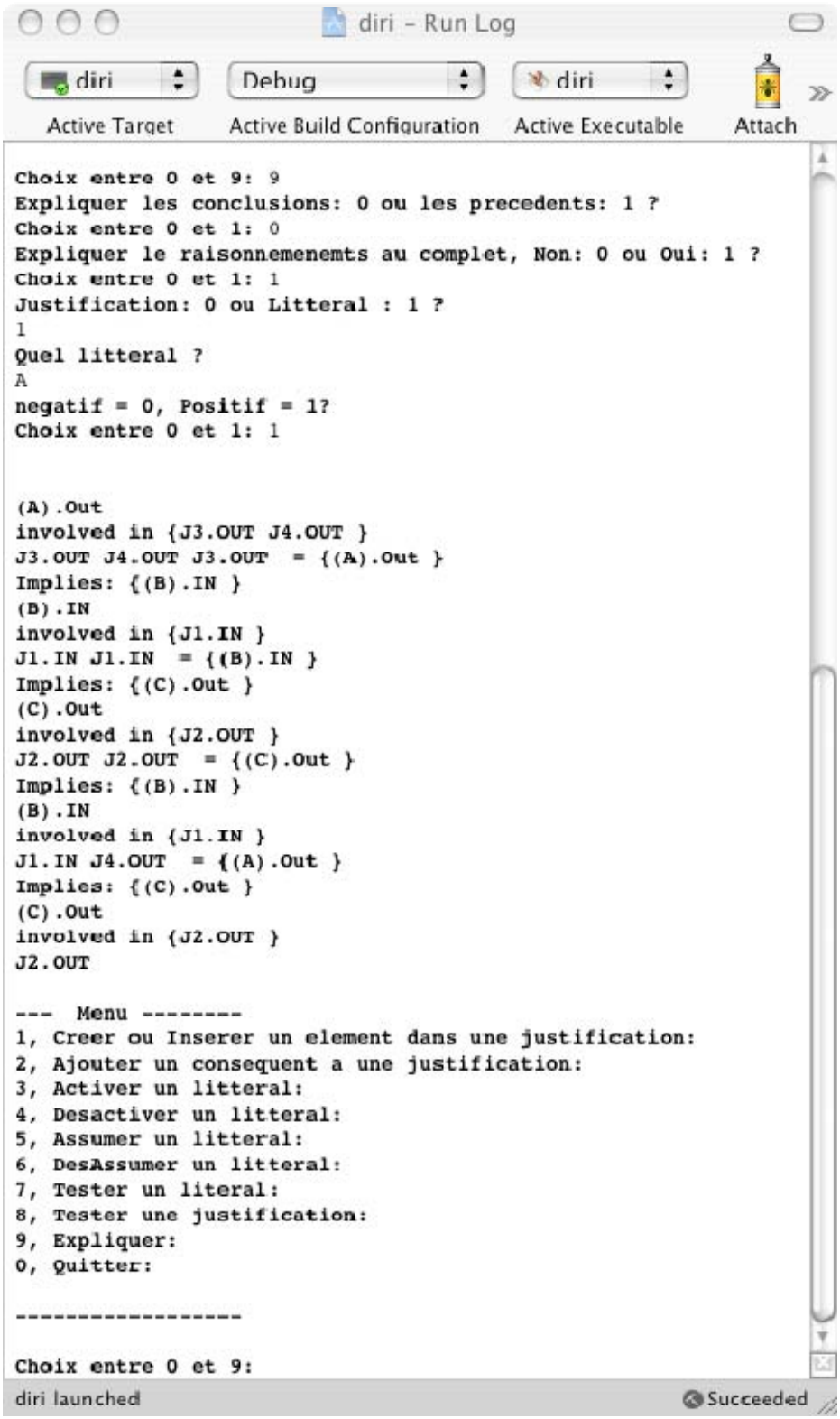



```
372
373     expression = "Expr106" ;
374     tms->NewLiteral(expression);
375     expressions.push_back(expression);
376
377     string resultats;
378     for (unsigned int i = 0; i<expressions.size(); i++)
379         {
380          ChercherFichiers(expressions[i], resultats); // Associer chaque expression avec ses conteneurs
381          strVector fichiersContenants;
382          Extraire(resultats, fichiersContenants);
383                 // ici on complete l'indexage : On lie les justifications avec leurs consequents
384          for (unsigned int j = 0; j<fichiersContenants.size(); j++)
385               tms->AddAConsequent(expressions[i], fichiersContenants[j], kPosPart);
386          fichiersContenants.clear();
387         }
388
389  //  puis on cherche par  map <fichier, expression trouvee>; multimap <taille,fichier>
390  ChercherCombinaison(mapResultat);
391      for (ResultIterat iterat=mapResultat.begin(); iterat!=mapResultat.end(); iterat++)
392      {string s;
393         s= Formater(iterat->second);
394         iterat->second=s;
395      }
396
397     Trier(mapResultat, motsParFichier);
398
399     cout<<endl<<"--------- Reslutats De La Requete"<<endl;
400
401     for (ResultSizeIterat iterat=motsParFichier.begin(); iterat!=motsParFichier.end(); iterat++)
402       {string s;
403          cout<<iterat->second<<":"<<endl;
404          s= mapResultat[iterat->second];
405           cout<<s<<endl<<endl;
406      }
407  }
408
409  //*****************************
410  void DReasonner::ChercherFichiers(string& expression, string& resultat)
411  {
412    resultat = tms->Explain(expression, kForward, kWithoutDetails, kPosPart);
413    size_t  fin, debut = 0;
414
```

Annexe 5 : Circularité, exemples d'exécution

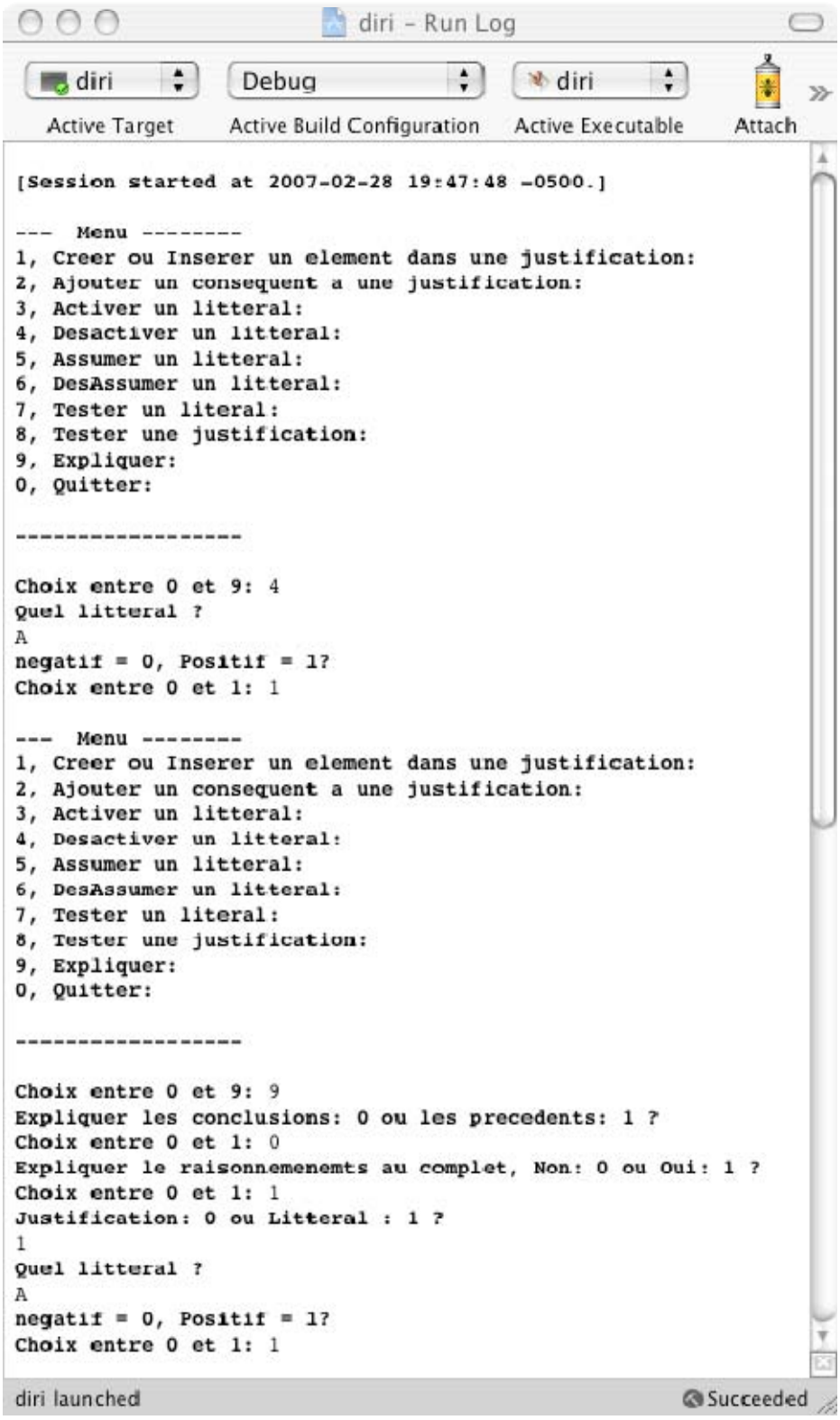
Exemple 1 de 2

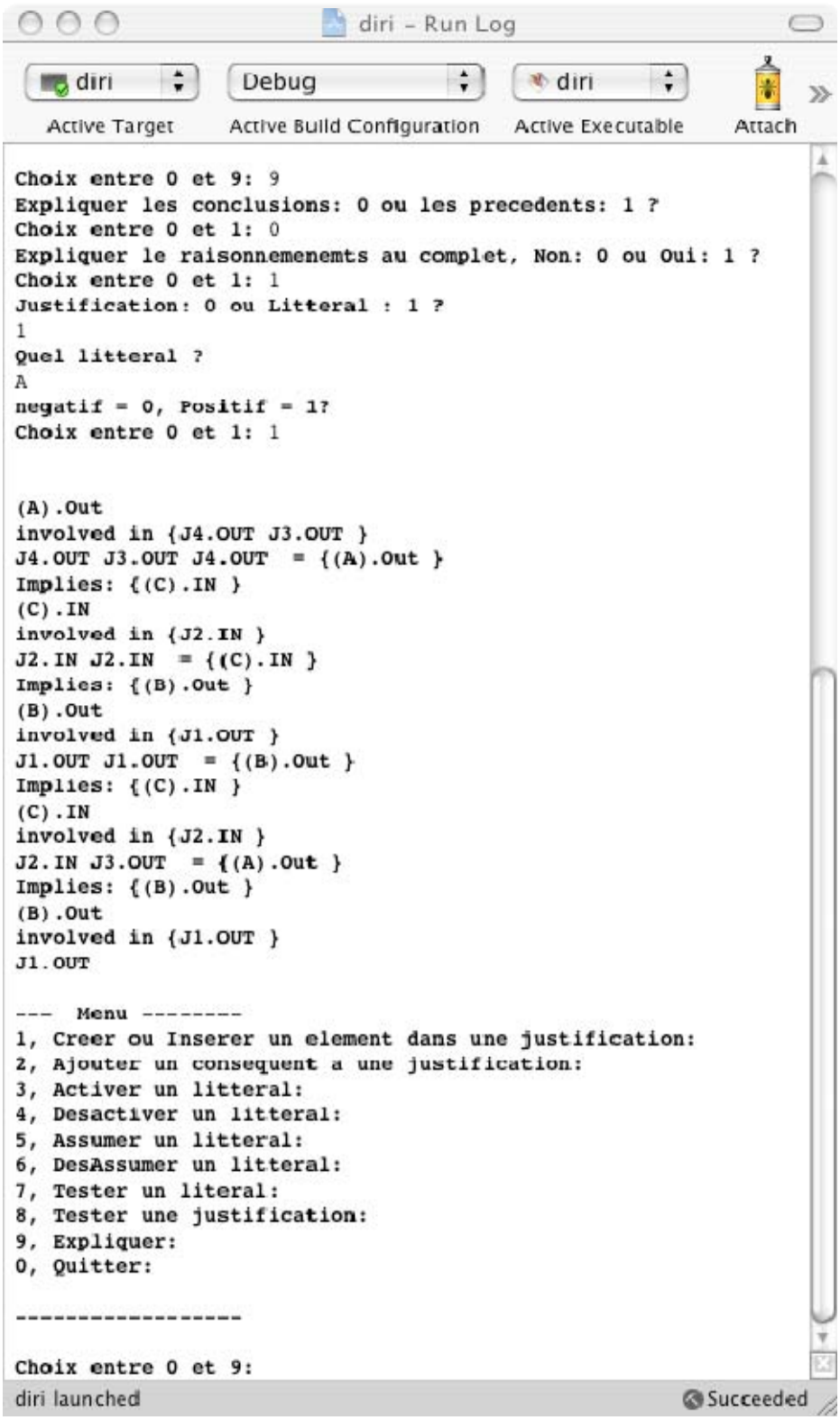



diri - Run Log
diri
Debug
diri
Active Target
Active Build Configuration
Active Executable
Attach
Choix entre 0 et 9: 9
Expliquer les conclusions: 0 ou les precedents: 1 ?
Choix entre 0 et 1: 0
Expliquer le raisonnemenemts au complet, Non: 0 ou Oui: 1 ?
Choix entre 0 et 1: 1
Justification: 0 ou Litteral : 1 ?
1
Quel litteral ?
A
negatif = 0, Positif = 1?
Choix entre 0 et 1: 1
(A).Out
involved in {J3.OUT J4.OUT }
J3.OUT J4.OUT J3.OUT = {(A).Out }
Implies: {(B).IN }
(B).IN
involved in {J1.IN }
J1.IN J1.IN = {(B).IN }
Implies: {(C).Out }
(C).Out
involved in {J2.OUT }
J2.OUT J2.OUT = {(C).Out }
Implies: {(B).IN }
(B).IN
involved in {J1.IN }
J1.IN J4.OUT = {(A).Out }
Implies: {(C).Out }
(C).Out
involved in {J2.OUT }
J2.OUT
--- Menu --------
1, Creer ou Inserer un element dans une justification:
2, Ajouter un consequent a une justification:
3, Activer un litteral:
4, Desactiver un litteral:
5, Assumer un litteral:
6, DesAssumer un litteral:
7, Tester un literal:
8, Tester une justification:
9, Expliquer:
0, Quitter:

Choix entre 0 et 9:
diri launched
Succeeded

Exemple 2 de 2




diri – Run Log
diri
Debug
diri
Active Target
Active Build Configuration
Active Executable
Attach
Choix entre 0 et 9: 9
Expliquer les conclusions: 0 ou les precedents: 1 ?
Choix entre 0 et 1: 0
Expliquer le raisonnemenemts au complet, Non: 0 ou Oui: 1 ?
Choix entre 0 et 1: 1
Justification: 0 ou Litteral : 1 ?
1
Quel litteral ?
A
negatif = 0, Positif = 1?
Choix entre 0 et 1: 1
(A).Out
involved in {J4.OUT J3.OUT }
J4.OUT J3.OUT J4.OUT = {(A).Out }
Implies: {(C).IN }
(C).IN
involved in {J2.IN }
J2.IN J2.IN = {(C).IN }
Implies: {(B).Out }
(B).Out
involved in {J1.OUT }
J1.OUT J1.OUT = {(B).Out }
Implies: {(C).IN }
(C).IN
involved in {J2.IN }
J2.IN J3.OUT = {(A).Out }
Implies: {(B).Out }
(B).Out
involved in {J1.OUT }
J1.OUT
--- Menu --------
1, Creer ou Inserer un element dans une justification:
2, Ajouter un consequent a une justification:
3, Activer un litteral:
4, Desactiver un litteral:
5, Assumer un litteral:
6, DesAssumer un litteral:
7, Tester un literal:
8, Tester une justification:
9, Expliquer:
0, Quitter:

Choix entre 0 et 9:
diri launched
Succeeded