Représentation des connaissances pour les problèmes de conception. Application à un système à base de connaissances pour la conception de réseaux informatiques : NEST.

Christine Jouve

ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT-ETIENNE

N° d' ordre : 77ID

THESE

Présentée par

Christine JOUVE

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITE DE SAINT-ETIENNE ET DE L'ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

(Spécialité : Informatique)

Représentation des connaissances pour les problèmes de conception. Application à un système à base de connaissances pour la conception de réseaux informatiques : NEST.

soutenue à SAINT-ETIENNE le 3 Septembre 1992.

Composition du jury :

Président :	Mr B. PEROCHE
Rapporteurs :	Mr J.M. FOUET
	Mr J. ROUAULT
Examinateurs :	Mme Y. AHRONOVITZ
	Mr C. BERTIN
	Mr M. CHEIN

ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT-ETIENNE

N° d' ordre : 77ID

THESE

Présentée par

Christine JOUVE

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITE DE SAINT-ETIENNE ET DE L'ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

(Spécialité : Informatique)

Représentation des connaissances pour les problèmes de conception. Application à un système à base de connaissances pour la conception de réseaux informatiques : NEST.

soutenue à SAINT-ETIENNE le 3 Septembre 1992.

Composition du jury :

Président :	Mr B. PEROCHE
Rapporteurs :	Mr J.M. FOUET
	Mr J. ROUAULT
Examinateurs :	Mme Y. AHRONOVITZ
	Mr C. BERTIN
	Mr M. CHEIN

Remerciements.

Toute ma reconnaissance et tous mes remerciements s'adressent à toutes personnes ayant accepté de juger ce travail, m'ayant aidé au cours de la réalisation de cette thèse ou tout simplement m'ayant supportée durant ces années. Ainsi je tiens à remercier :

Monsieur Jean-Marc FOUET, Professeur à l'université de Lyon I et Monsieur Jacques ROUAULT, Professeur à l'université de Grenoble, d'avoir accepté d'être rapporteur de cette thèse, d'avoir examiné attentivement mon travail et qui, par leurs nombreuses remarques m'ont permis d'améliorer ce mémoire,

Monsieur Christian BERTIN de la société ACRI pour m'avoir encadrée les premières années de ma thèse et pour sa participation à ce jury,

Monsieur Bernard PEROCHE, Professeur à l'école des Mines de Saint-Etienne, pour m'avoir accueillie au sein du département qu'il dirige, pour son soutien après le départ de Monsieur Bertin et pour sa disponibilité malgré sa surcharge de travail,

Madame Yolande AHRONOVITZ, Maître de Conférence à l'université de Saint-Etienne et Monsieur Michel CHEIN, Professeur à l'université de Montpellier pour leur présence dans ce jury,

Tous le membres du projet MMI2 pour nos bonnes relations et notre bonne collaboration et plus particulièrement Fabienne BALFROID et Françoise DARSES sans lesquelles NEST n'aurait pas vu le jour,

Tous les membres du département Informatique de l'école pour leur amitié et leur aide ponctuelle,

Le service de reprographie sans qui cette thèse n'aurait pu être éditée,

Mes parents pour m'avoir permis d'arriver jusqu'ici et pour leur confiance quant'à ma réussite, la petite soeurette pour avoir pu et pouvoir encore me défouler, Denis, pour son amour et son encouragement, le reste de ma famille, tous mes amis avec qui j'ai vécu des heures inoubliables parmi la splendeur des montagnes, Monsieur et Madame Bebronne et en mémoire de Bernard.

Sommaire.

Liste des figures.

Introduction générale.

Un des champs d'application de l'intelligence artificielle est la conception de programmes qui exhibent un comportement intelligent. Pour cela, il est nécessaire de disposer de mécanismes permettant de stocker l'information requise mais aussi permettant de simuler l'activité de raisonnement d'un humain. Je me suis ainsi intéressée aux mécanismes de représentation des connaissances. Dans un premier temps, j'essaierai d'établir l'état de l'art, le plus objectivement possible, des diverses techniques de représentation des connaissances, tout en les comparant. L'accent sera porté sur celles qui me semblent le plus appropriées au problème qui m'a été soumis : la réalisation d'un système à base de connaissances pour la conception de réseaux informatiques. La deuxième partie de cette thèse présentera en détails la réalisation de ce système : NEST (Network design Expert SysTem).

Le Projet ESPRIT II MMI^2.

Le cadre de ce travail est le projet ESPRIT II : 2474, MMI^2 (Multi-Modal Interface for Man-Machine Interaction with knowledge based system) dont l'objectif principal est la conception d'une interface homme/machine intégrant plusieurs modes de communication et couplée à un système à base de connaissances. Au sein de ce projet, un système expert particulier est développé dans le domaine de la conception de réseaux informatiques. Les différents partenaires œuvrant à la réalisation de ce projet sont : BIM - Belgique, le CRISS - Grenoble, l'INRIA - Rocquencourt, ISS - Espagne, RAL - Angleterre, l'Université de Leeds - Angleterre et ENSM - Saint-Etienne. Ce projet a débuté en Janvier 1989. Le premier prototype du système intégrant le système de gestion du dialogue [MMI^2 partners 91] a été présenté en Octobre 1990 alors que le premier démonstrateur intégrant les différents modes et le système à base de connaissances [Wilson 91] a été présenté en Octobre 1991 et lors de la conférence Esprit en Novembre 1991 à Bruxelles.

La prépondérance que prend la convivialité d'un système (qu'il soit conventionnel ou à base de connaissances) vis à vis de son utilisateur justifie qu'on y porte intérêt dans un projet. De plus, un système à base de connaissances implique généralement beaucoup d'échanges entre l'utilisateur et le système que ce soit au cours de sa résolution, lors d'acquisitions d'informations manquantes auprès de

l'utilisateur, lors de la donnée des résultats obtenus ou lors des étapes d'explication du raisonnement suivi par le système. Ceci est d'autant plus vrai s'il s'agit d'un système d'aide à la décision. Ainsi les interfaces des systèmes à base de connaissances doivent elles être très conviviales car les interactions sont nombreuses et doivent apporter des facilités pour ces échanges, par exemple une saisie graphique du plan d'un bâtiment (ce serait très fastidieux par d'autres moyens non graphiques tels que le langage naturel ou d'autres).

L'ambition du projet MMI^2 est l'intégration de plusieurs modes de communication : les langues naturelles : anglais, français et espagnol, le graphique, le gestuel et un langage de commande. Cette interface aura un comportement adapté à tout type d'utilisateur : de l'expert confirmé qui aura recours surtout au langage de commande, aux menus et au graphique, au débutant dans le domaine qui se servira principalement des moyens de communication naturels tels que le graphique et la langue naturelle. Egalement un module de modélisation de l'utilisateur est inclus dans la boîte à outils MMI^2 pour assister le gestionnaire du dialogue dans la sélection d'un mode de sortie, dans le contenu d'une réponse, dans le choix de la structure du dialogue, en fonction du type d'utilisateur.

Le développement de l'interface est indépendant de l'application. Néanmoins, pour prouver son efficacité, l'interface sera connectée à une application spécifique : un système expert de conception de réseaux informatiques que nous allons présenter en détails dans ce mémoire. Une grande variété d'utilisateurs potentiels peut être intéressée par un tel système, que ce soit des experts du domaine : commerciaux voulant soumettre leur problème au système, experts désirant comparer plusieurs solutions..., ou des personnes moins compétentes dans le domaine : un client par exemple. De plus cette application offre de nombreuses opportunités d'interaction multi-modes : saisie graphique des plans du ou des bâtiments à câbler, et apport d'information en langue naturelle ou en langage de commande et utilisation des menus, gestuel pour déplacer facilement par exemple des machines d'une pièce à une autre ...

Un système à base de connaissances pour la conception de réseaux informatiques : NEST.

Permettre aux ordinateurs de communiquer, d'échanger entre eux des données à traiter et des résultats, tel est l'objectif des réseaux d'ordinateurs. Configurer un réseau d'ordinateurs est une tâche difficile car elle ne consiste pas seulement à relier les ordinateurs entre eux au moyen de câbles mais aussi (1) à introduire des équipements techniques tels que des répéteurs, des connecteurs câble-machine, des éléments de filtrage, (2) à prévoir une décomposition en sous-réseaux dans le cas de problèmes complexes et étendus et (3) à rendre la communication possible lorsque plusieurs protocoles de communication sont utilisés par le choix de valeurs adéquates pour certains paramètres de communication ou le choix de logiciels.

Alors que les systèmes experts fleurissent dans tous les domaines, presque aucun n'apporte une aide pour la configuration de réseaux. Le plus important champ d'investigation de l'intelligence artificielle dans le domaine des télécommunications est la réalisation de systèmes experts pour le diagnostic de fautes ou la détection de pannes [Gross 88]. Parmi ces réalisations, on peut noter :

- Automated Cable Expert (ACE), développé par AT&T Bell Laboratories en 1981, est un système d'analyse automatique pour identifier et diagnostiquer les problèmes dans des boucles locales. ACE engendre des messages électroniques lorsqu'il a détecté un problème et propose une solution.

- Designet, également développé par AT&T Bell Laboratories, est un outil de simulation qui donne une représentation du trafic sur un réseau implantant la norme X.25. L'utilisateur peut créer de nouveaux liens dans le réseau ou laisser Designet suggérer une amélioration du réseau.

- NetHandler est un projet de la société SCS Organisationsberatung und Informationstechnik GmbH, Hamburg, en 1988. Sa principale tâche est l'interprétation de messages d'un réseau et non leur diagnostic. Le système en présente un condensé à l'opérateur et étudie les possibles conséquences des fautes détectées. NetHandler a été réalisé en utilisant les principes de KADS[1], une méthodologie pour le développement de systèmes à base de connaissances [Krickhahn & al 88].

Un système proche de NEST puisqu'il est lui aussi destiné à la conception de réseaux locaux est ISLAND (Intelligent System for Local Area Network Design) [Metzler & al 88]. A la différence de NEST qui est un outil automatique ne nécessitant aucune intervention de l'utilisateur pendant son processus de résolution, ISLAND travaille en collaboration avec l'utilisateur. L'architecture de ce système est proche de celle des tableaux noirs et inclut ainsi une base de connaissances, des modules experts, le dessin du réseau, un agenda et un module de contrôle.

Un autre outil dans le domaine des réseaux est un configurateur développé par Philips [Lutticke & al 89] dont l'intérêt majeur est l'interface graphique interactive.

Deux types de réseaux ont des principes de conception totalement différents : les réseaux longues distances (WAN, Wide Area Network) permettant de connecter des sites distants et les réseaux locaux (LAN, Local Area Network) pour relier des machines sur un même site. Dans un premier temps, NEST est un système dédié à la conception de réseaux locaux implantant les technologies Ethernet - TCP/IP.

Objectifs et plan de la thèse.

Cette thèse présente le résultat de mon travail tant en ce qui concerne l'étude des mécanismes de représentation des connaissances que l'utilisation de certains de ces formalismes dans la réalisation de

1. KADS est une abréviation de Knowledge Acquisition, Documentation and Structuring system. et est l'objet du projet Esprit no 1098.

systèmes à base de connaissances pour la conception. Ainsi ce mémoire est-il constitué de deux parties.

La première traite des divers formalismes de représentation des connaissances. Les formalismes les plus généraux : logique, représentation procédurale, système de production, réseaux sémantiques, systèmes à base de "théories" sont discutés dans le premier chapitre alors que d'autres méthodes font l'objet d'une étude plus approfondie car utile pour la réalisation de NEST. Ainsi les différents concepts propres aux modèles centrés objets sont-ils énoncés au chapitre deux. NEST a été réalisé à l'aide d'un langage Prolog, ProLog_by_BIM, auquel une couche orientée objets (donnant BIM_Probe) a été couplée ainsi qu'un langage de programmation par contraintes (PCL). L'objectif du travail que j'ai eu à réaliser traitant de la conception de réseaux, le dernier chapitre de cette partie cerne le problème de la conception et présente diverses approches employées dans différents systèmes dont l'objectif est la conception.

L'objectif de la deuxième partie est de présenter le système développé au cours de ces trois années de thèse et de projet Esprit. Tout d'abord, les différentes caractéristiques de l'interface multi-modes développée et couplée à NEST sont introduites. La première étape, avant de pouvoir concevoir un système à base de connaissances est l'acquisition de celles-ci auprès des experts confirmés dans le domaine concerné, en l'occurrence ceux de la société belge BIM pour notre problème. Ainsi le deuxième chapitre de cette partie est-il consacré à la description de la conception de réseaux informatiques et à la délimitation faite sur ce champ d'application qui est très vaste afin de spécifier les objectifs quant à la réalisation de NEST. Le dernier chapitre présente NEST et l'architecture adoptée pour son implémentation. Les résultats obtenus sont également discutés dans ce chapitre en ce qui concerne NEST et également l'interface multi-modes. Ainsi les intérêts de la méthode employée dans NEST sont-ils présentés et des améliorations quant à la flexibilité du système sont proposées.

En fin de cet thèse, plusieurs annexes présentent chacune une partie de ce travail de recherche. La première est la base de connaissances de l'outil développé. Elle contient donc la description de tous les éléments de base de la conception de réseaux. La seconde introduit les modules implémentés permettant à NEST de concevoir un réseaux. Un formalisme de présentation de ces modules a été défini afin de permettre au lecteur une compréhension plus aisée du mécanisme de l'outil de conception de NEST. L'annexe 3 liste les informations nécessaires à NEST pour pouvoir résoudre un problème de conception de réseau. La quatrième annexe présente le langage utilisé pour le développement de NEST : BIM_Probe. Les annexes 5 et 6 sont des listings des programmes réalisés.

Note.

L'application sur laquelle porte cette thèse : la réalisation d'un système à base de connaissances pour la conception des réseaux locaux a été développée en étroite collaboration, principalement avec :

- Fabienne Balfroid de la société BIM, avec laquelle j'ai travaillé au développement de NEST,

- Françoise Darses de l'INRIA, qui, par ses recherches en psychologie cognitive, a permis une bonne acquisition des connaissances et nous a aidées dans nos réflexions tout au long de la réalisation de cet outil de conception.

Première partie :

Représentation des

connaissances.

La réalisation d'un système à base de connaissances implique la modélisation des connaissances dans le domaine considéré. Le problème qui se pose est de trouver des structures informatiques qui permettront non seulement le stockage de ces connaissances mais aussi l'utilisation correcte de celles-ci. Ainsi les structures informatiques choisies ne peuvent-elles être dissociées de leurs modes d'utilisation. Barr et Feigenbaum donnent la définition suivante : une représentation des connaissances est une combinaison de structures de données et de procédures interprétatives qui, utilisées correctement dans un programme, conduiront à un comportement intelligent [Barr & al 81].

Le choix d'une bonne représentation des connaissances n'est pas aisé puisqu'aucun formalisme universel n'existe ; au contraire, plusieurs sont disponibles tels que : les modèles logiques, les procédures, les règles de production, les réseaux sémantiques, les systèmes à base de "théories", le modèle orientée objet, ... L'objectif de cette partie est donc de présenter l'état de l'art des divers formalismes de représentation, tout en mettant l'accent sur certains points semblant prometteurs pour la réalisation de NEST. Ainsi le deuxième chapitre présentera-t-il les modèles orientés objets. Le dernier chapitre, après avoir introduit l'activité de conception fera le point sur les différents systèmes permettant une telle résolution et me semblant intéressants.

CHAPITRE 1

Représentation des Connaissances - Généralités.

Un des premiers pas vers des outils de développement de mécanismes de raisonnement a certainement été réalisé par Frege lorsqu'il a introduit la logique mathématique à la fin du 19ème siècle. Quant à l'idée qu'un raisonnement puisse être mené par une machine, elle peut être attribuée à Turing qui publia en 1950, l'article "Computing Machinery and Intelligence" qui débute ainsi : "Les machines peuvent-elles penser ?" [Turing 50].

C'est également dans les années 50 (1956) que le terme d'intelligence artificielle est proposé par John Mc Carthy. Ainsi cherche-t-on à faire résoudre par une machine des problèmes qui, résolus par un humain, demandent de l'intelligence. Et en 1969, Mc Carthy propose un manipulateur de faits capable de répondre à certaines questions concernant son domaine de représentation ; ce qui aujourd'hui est communément appelé : un système fondé sur la connaissance. Ainsi est né l'ancêtre de tout système expert. Actuellement, de nombreux systèmes experts ont été conçus dans différents domaines mais malheureusement très peu sont des systèmes opérationnels (ils sont plutôt restés à l'état de prototype). Pour chacun d'eux un formalisme de représentation des connaissances du domaine concerné a été choisi parmi les différents formalimes actuellement à la disposition des concepteurs.

Afin de sélectionner un formalisme adéquat de représentations des connaissances, il est intéressant de cerner les divers types de connaissances que l'on sera amené à représenter et de délimiter les problèmes à résoudre pour avoir une représentation adaptée permettant de bien modéliser les données du monde réel et les capacités de raisonnement des experts humains. Les deux premiers paragraphes de ce chapitre traiteront ainsi de ces deux aspects alors que le dernier fera un survol des principaux formalismes de représentation des connaissances.

1.1 Les différents types de connaissances qu'on peut être amené à représenter.

Un système à base de connaissances est principalement constitué de deux éléments : une base de connaissances et un moteur d'inférence. La base de connaissances regroupe toutes les connaissances relatives au domaine d'application considéré. C'est dans cet ensemble de connaissances que le moteur d'inférence ira puiser les informations qui lui sont nécessaires pour résoudre un problème particulier.

La phase de représentation des connaissances consiste à modéliser les connaissances connues des experts et à les réunir dans la base de connaissances. Cette opération s'effectue en identifiant des granules de connaissances, entités abstraites contenant une certaine quantité de connaissances et qui seront implémentées dans des structures de données adéquates. La représentation des connaissances peut ainsi être définie comme étant l'ensemble des techniques qui permettent la manipulation des fragments de connaissance.

Il est intéressant de tenter de classifier les granules de connaissance que le concepteur d'un SBC[1] aura à manipuler lors de la phase de représentation. D'après [Bittencourt 88], plusieurs critères de classification sont possibles : (1) le sens lié à la valeur logique de chaque élément de connaissance, c'est à dire son statut épistémique (valeur de vérité) et son statut assertionnel (degré de confiance), (2) l'utilisation à laquelle un élément de connaissance est destiné, (3) les dimensions, critère applicable à un ensemble d'éléments de connaissance qui se réfère aux caractéristiques de la connaissance liées au fait que, pour les traiter, il faut stocker et manipuler cette connaissance. Un critère qui semble plus digne d'attention assurant cette classification est celui qui reflète les différents types de connaissances. Ces différents types sont les suivants [Laurière 82] :

- *éléments de base, objets du monde réel* : Ce sont des perceptions immédiates du domaine à représenter. Cette information n'est pas remise en cause.
- *assertions* et *définitions* sur les objets de base : Les connaissances sont généralement pensées en tant que faits sur des objets. Par exemple," un oiseau a des ailes", "Titi est un oiseau", "l'eau de mer est salée" sont des faits qui pourront être insérés dans un système à base de connaissances. Elles sont a priori considérées comme sûres.
- *concepts* ou *abstractions* : Ils permettent le regroupement ou la généralisation d'objets du domaine étudié.
- *relations* : Elles expriment des propriétés élémentaires des éléments de base ou des relations de cause à effet entre concepts. Ces relations sont plus ou moins vraisemblables, plus ou moins corrélées à une situation donnée.
- *théorèmes* et *règles* : Ce sont des connaissances sûres, généralement extraites de théories et de livres. Ce type de connaissances est associé à des règles expertes qui stipulent leur utilisation, ce qui est particulièrement nécessaire pour les théorèmes.
- *algorithmes de résolution* : Ils permettent d'accomplir certaines tâches par exécution dans un ordre fixe d'une suite d'actions mémorisées en bloc. Ce type de connaissance est limité à des cas très particuliers principalement pour le traitement numérique de l'information. Il ne concerne pas réellement le domaine de l'intelligence artificielle qui est des-

1. système à base de connaissances

tiné à l'informatisation de fonctions pour lesquelles le seul algorithme connu est combinatoire.

- *stratégies* et *heuristiques* : Ce sont des connaissances empiriques reflétant les stratégies de résolution acquises par les experts humains par expérience. Elles permettent d'inférer des actions à envisager dans une situation précise.
- *méta-connaissances* : connaissances sur la connaissance, sur ce qui est déjà connu. Par exemple, des méta-connaissances peuvent exprimer des connaissances sur : (1) la prolongation ou l'origine de certaines connaissances, (2) la fiabilité de certaines informations, (3) la relative importance de faits spécifiques, (4) la performance des experts humains au sens cognitif, c'est à dire leurs forces ou faiblesses, leurs degrés d'expertise, etc.

1.2 Les problèmes liés à la représentation des connaissances.

Avant de décrire les principales méthodes de représentation des connaissances, il est intéressant d'étudier les divers problèmes auxquels conduit cette discipline et qui devront être traités par ces formalismes.

Problème 1. L'exception.

Des lois générales se dégagent mais sont souvent contredites par quelques exceptions. Ainsi par exemple, "Les oiseaux peuvent voler" est une loi générale caractérisant l'élément de base oiseau. Il n'en reste pas moins que les manchots, les autruches ou tout autre oiseau qui n'a plus d'ailes, ne volent pas. Par conséquent, il est toujours difficile, si ce n'est impossible, de généraliser le comportement d'objets du monde réel appartenant à une même classe. Au niveau de la représentation se pose le problème de savoir s'il faut représenter la loi générale séparément de ses exceptions. Si oui, comment garantir que celle-ci ne sera pas appliquée justement dans les cas où elle est annulée par une exception ? Sinon, est-il raisonnable de supposer que les lois générales se présentent pourvues de la liste complète de leurs exceptions ?

Problème 2. L'univers en évolution.

Dans un SBC, des informations sont créées, inférées, modifiées ou détruites. Certaines sont en relation avec de nouvelles informations ; il s'en suit donc la mise à jour des connaissances rendues périmées par ces modifications. Par exemple, un changement de pression provoque une modification de la température et/ou du volume pour que l'équation des gaz parfaits soit satisfaite. D'autres informations changent de type. Par exemple, à sa majorité une personne acquiert les droits et les obligations d'une personne majeure. D'autre part, les connaissances sur un univers étudié changent au fur et à mesure que l'univers évolue du fait d'une continuelle recherche d'améliorations des techniques ou matériaux utilisés. Le domaine de la conception de réseaux informatiques est une bonne illustration de cette évolutivité ; de nouveaux composants de réseaux apparaissent continuellement sur le marché de même que

de nouvelles techniques d'implantation de réseaux. Va ainsi se poser le problème de la mise à jour des connaissances. Il va falloir être capable de déterminer quelles sont les connaissances obsolètes et déterminer s'il est nécessaire de les détruire donc de les perdre ou de les garder car bien qu'obsolètes elle n'en restent pas moins vraies dans un certain contexte.

Problème 3. Modalité des connaissances.

Toutes les connaissances n'ont pas le même statut. Lors de l'étude du problème précédent, un premier type de statut a été révélé : *valide* ou *périmée*. Un autre critère est la possibilité de modification des connaissances. Leur statut peut alors être soit *intangible*, soit *modifiable*. Un troisième type de statut repose sur le mode d'acquisition de la connaissance : soit c'est une *donnée*, soit elle a été *déduite* par inférence. Certaines connaissances sont *certaines*. C'est particulièrement le cas des éléments de base, des relations sur ces derniers, des concepts, des théorèmes et des algorithmes de résolution. D'autres sont *incertaines*. Ce sont des heuristiques, des stratégies utilisées par les experts humains, l'expression d'un point de vue (par exemple : je crois que ...). La connaissance du statut est nécessaire; on ne peut utiliser la même stratégie dans le cas d'informations certaines ou dans le cas d'incertitude, par exemple.

Problème 4. Conservation des ambiguïtés.

Vouloir représenter la réalité impose une réduction d'information afin d'avoir des systèmes plus efficaces car n'ayant pas à manipuler toute une série d'informations inutiles. Mais il faudra faire un choix adéquat pour cette réduction d'informations qui n'empêche pas le système de fonctionner correctement ; celui-ci devra être capable de raisonner sans cette information. D'autre part, par exemple le granule de connaissance : *"la température est trop élevée"* introduit une ambiguïté au niveau de la signification de "trop élevée". En effet, cette information dépend du contexte dans lequel se situe la règle. Ainsi par exemple "trop élevée" signifie "supérieur à mille degrés" pour un réacteur alors qu'elle est équivalente à "supérieur à zéro degré" pour un canon à neige. Une bonne représentation doit permettre au système de poursuivre son raisonnement en traitant ces ambiguïtés. Mises à part les informations incomplètes, un deuxième type d'ambiguïté que le système devra pouvoir manipuler est constitué des informations partiellement résolues. Il devra également être capable d'inférer de nouveaux faits et donc de s'acheminer vers son but malgré ce type d'ambiguïté.

Problème 5. Informations incomplètes, incertaines ou implicites.

Deux principaux problèmes surviennent du fait d'avoir à utiliser ce type de connaissance : leur modélisation et leur traitement. Le fait d'avoir des connaissances incertaines implique d'être capable d'introduire cette incertitude quant à leur véracité (généralement par l'affectation de coefficient à chaque granule de connaissance reflétant leur degré de véracité). Même si certaines informations sont manquantes car implicites ou incomplètes, le système doit être capable d'inférer tout en ayant lui-même un raisonnement imprécis et en fournissant en sortie des résultats "incertains", pondérés par des estimations de l'incertitude.

Problème 6. Continuité, proximité, transitivité.

Le modèle de représentation des connaissances choisi doit pouvoir concilier les caractéristiques des modèles discrets (choix vrai ou faux) et celles des modèles continus ou quasi-continus. Ainsi est

satisfait le paradoxe du sorite exprimant que la connaissance du genre : "un tas de sable auquel on ôte un grain de sable reste un tas de sable" est vraie mais un autre mécanisme devra être introduit permettant d'exprimer que cette règle devient fausse si elle est employée itérativement un trop grand nombre de fois.

Problème 7. Quantificateurs.
Les deux quantificateurs reconnus par la logique classique : universel ($\forall$) et existentiel ($\exists$) ne suffisent pas pour représenter tout type de connaissance et en particulier la majorité, la plupart, généralement, souvent...

Problème 8. Connaissances temporelles et spatiales.
Comment représenter de telles connaissances ? Qu'est-ce qu'un instant ? un temps ? un jour ?... Qu'est qu'un lieu ? trois coordonnées ? une ville ?... Que ce soit pour les connaissances temporelles ou spatiales, il est difficile de fixer une fois pour toutes le degré de finesse que ce soit pour une unité de temps ou pour des coordonnées. Souvent des informations précises du genre : l'objet A est en contact avec l'objet B, doivent pouvoir être simultanément utilisées avec des connaissances vagues tels que A et B sont situés quelque part à gauche de C.

1.3 Les principaux formalismes de représentation des connaissances.

Il existe de nombreux formalismes de représentation des connaissances [Barr & al 81 Chapitre III], qui sont plus ou moins bien adaptés au traitement des problèmes qui viennent d'être énumérés. Ce paragraphe se veut une présentation succincte des formalismes principaux de représentation des connaissances tels que la logique, les procédures, les systèmes de production, les réseaux sémantiques, le formalisme à base de "théories". A chaque fois, nous donnons leurs avantages et leurs inconvénients. D'autres chapitres qui suivent ont pour objectif d'introduire plus en détails certains formalismes qui seront utilisés pour la réalisation de NEST et donc ne sont pas présentés dans cette section.

Mais, tout d'abord, deux définitions sont à rappeler pour une meilleure compréhension de certaines remarques : la cohérence et la complétude.

Un système est cohérent si les actions qu'il exécute sont toujours les conséquences de conditions vérifiées.

Un groupe donné de règles d'inférence est dit complet pour la déduction si, quel que soit un ensemble de formules bien formulées, toutes leurs conséquences logiques peuvent être dérivées à partir d'elles comme des théorèmes (c'est à dire par un nombre fini d'applications de règles du groupe) [Farreny & al 87].

1.3.1 La logique.

Présentation :

La logique fut le premier formalisme de représentation des connaissances en intelligence artificielle (si l'on excepte les langages de programmation). Et aujourd'hui encore, l'approche classique pour représenter des connaissances du monde réel reste la logique formelle. Cette représentation est identique à celle obtenue par les psychologues avec l'analyse prédicative. En effet, son utilisation a déjà été étudiée dans la représentation des connaissances chez l'être humain [Israel 83].

En logique, seule la valeur de vérité d'une proposition importe. Cinq opérateurs logiques sont à la disposition d'un utilisateur éventuel : et ($\wedge$), ou ($\vee$,)non ($\sim$), l'implication ($\Longrightarrow$) et l'équivalence ($\Longleftrightarrow$). Ces opérateurs utilisés dans des propositions constituent le calcul propositionnel qui permet d'exprimer des faits tels que : "Titi est un oiseau". Des règles de calcul des valeurs de vérité permettent la composition de propositions. La première règle d'inférence est le modus ponens :

si X est un théorème et si X $\Longrightarrow$ Y est un théorème alors Y est un théorème

De plus, pour pouvoir parler d'objets, établir des relations entre ceux-ci, le calcul des prédicats (extension du calcul propositionnel) apporte la possibilité de manipuler des variables et des quantificateurs ($\forall$ et $\exists$). Au lieu de ne se préoccuper que de la valeur de vérité d'une proposition, des états pour des objets spécifiques sont décrits par l'intermédiaire de prédicats. Ainsi la règle "tout oiseau a des ailes" s'écrit à l'aide de deux prédicats, oiseau et a_des_ailes et une variable, x.

R1 : $\forall x$ *[oiseau (x)* $\Longrightarrow$ *a_des_ailes (x)]*

Le mécanisme est celui de l'inférence. Des règles d'inférence permettent à partir de faits donnés de déduire de nouveaux faits et ainsi de démontrer des buts.

Par exemple si la règle, R2 : $\forall x$ *[robin (x)* $\Longrightarrow$ *oiseau (x)]* est ajouté à la précédente et le fait, *F : robin (titi)* alors le système déduira de R2 *oiseau (titi)* puis *a_des_ailes (titi)* en utilisant R1.

Ainsi un système fondé sur la logique est-il constitué par un ensemble de formules et un ensemble de règles d'inférence. Ces dernières apportent la structure déductive à tout système fondé sur la logique puiqu'elles sont capables d'engendrer de nouveaux théorèmes à partir d'un ensemble de formules. A chaque formule est associée une valeur de vérité exprimant qu'une formule est vraie ou fausse. Le processus de résolution est constitué d'une première étape : le filtrage. Il permet d'obtenir un sous-ensemble de formules méritant d'être comparée. La deuxième opération consiste à unifier les formules ainsi obtenues avec le but à résoudre par comparaison de leurs termes. Ainsi toute nouvelle formule obtenue par inférence est vraie puisqu'elle est prouvée à partir de prémisses déjà connues comme étant vraies. C'est une des raisons qui fait de la logique un outil puissant, fiable et apprécié.

Le meilleur exemple de formalisme informatique fondé sur la logique est le langage Prolog [Colmerauer 83] créé par Alain Colmerauer en s'appuyant sur la logique du premier ordre. SNARK [Laurière 84] système élaboré par Laurière en 1982, utilise aussi la logique. Ces systèmes sont seulement deux exemples d'utilisation de la logique mais bien d'autres pourraient être cités.

Avantages :

Pour la logique du premier ordre, il a été prouvé qu'il existe des ensembles de règles (théorèmes et règles d'inférence) qui sont à la fois corrects et complets. Ce résultat est la base de la puissance de la logique comme formalisme de représentation des connaissances. D'autre part, les arguments en faveur de la logique sont les suivants :

- La logique dispose de solides bases théoriques, pouvant s'étendre à de nouvelles utilisations du fait du développement actuel de logiques dites non-classiques telles que les logiques non-monotones, multivaluées, modales [Kayser 84].
- Le naturel avec lequel les éléments de connaissance peuvent être exprimés sous forme de formules logiques ; on peut exprimer un fait sans se soucier de ses manipulations.
- La flexibilité et la modularité dues à l'indépendance des assertions résultant du traitement déclaratif de la connaissance qui facilite la modification et l'élargissement d'une base logique de connaissances.

Inconvénients :

- Un des intérêts de la logique est que la cohérence et la complétude sont garanties mais cette caractéristique peut être, dans certains cas, un inconvénient car la logique est une représentation très formelle et mathématique. Elle ne permet pas, par exemple, une manipulation aisée des informations incertaines ou incomplètes.
- Aucune possibilité n'est offerte pour le contrôle, c'est à dire pour le choix des éléments de connaissance adéquats à employer dans une situation donnée. Le système doit le plus souvent travailler de façon exhaustive et manipuler un grand nombre d'hypothèses pour trouver celles utilisables. Contre la logique subsiste ainsi toujours l'argument de l'inefficacité due à l'explosion combinatoire. Si la base de connaissances devient trop importante, il n'est plus possible de l'utiliser de façon performante.

1.3.2 La représentation procédurale.

Présentation :

Le terme représentation procédurale est principalement utilisé en opposition avec les méthodes dites déclaratives, généralement associées à la logique. En fait, la représentation procédurale combine à la fois des assertions (faits connus) et des procédures, suites d'instructions capables de déduire de nouveaux faits. Ces procédures ont pour fonction soit de réaliser des manipulations spécifiques au domaine d'application, soit de contrôler le processus de raisonnement ; elles sont alors associées à une représentation déclarative. Cette méthode de représentation des connaissances varie par rapport aux programmes classiques dans le sens où plusieurs modes [Vignard 86] peuvent être employés pour appeler une procédure :

- l'appel direct : Un programme principal appelle les procédures de façon "classique".
- l'attachement procédural : La procédure est attachée à une zone de données d'une source de connaissance. A chaque accès à cette zone de données, la procédure est exécutée.

- le démon : L'exécution d'une procédure P est lancée si une condition C est satisfaite lors du déroulement du processus global du système. Ce démon est, par exemple, une instruction du type : démon (P, C).

- l'appel dirigé par les schémas (pattern-matching) : Pour chaque programme, un schéma décrit la tâche qu'il réalise. Le programme principal utilise ce schéma pour appeler le programme concerné, en spécifiant tout d'abord le but à réaliser puis en cherchant une correspondance entre le but ainsi défini et un schéma. Par exemple, le schéma correspondant au but "Situe_a" est : (situe_a personne objet) dont le programme permet de déplacer une personne près d'un objet. Cette technique permet une décomposition naturelle d'un problème pour atteindre le but recherché mais donne un système très difficile à modifier et à étendre.

PLANNER [Hewitt 72] est l'exemple le plus connu implémentant des techniques de représentation procédurale. Il utilise la notion d'appel dirigé par les schémas.

Avantages :

- Un premier avantage est la facilité apportée par ce formalisme pour représenter les connaissances de nature heuristique. C'est le cas, par exemple, pour savoir si un théorème doit être appliqué en chaînage avant ou arrière, pour déterminer quelle connaissance doit être employée dans une situation donnée ou pour définir quel sous-but doit être tenté en premier, ...

- De ce premier avantage découle que le raisonnement des systèmes procéduraux est parfaitement guidé : ils ne peuvent pas utiliser des connaissances inadéquates ou suivre un mauvais cheminement, sauf s'ils sont mal programmés évidemment.

- Ce raisonnement bien dirigé évite une recherche, éventuellement longue et coûteuse, entre les diverses actions possibles comme c'est le cas, par exemple, pour les systèmes de production (plusieurs règles activables pour résoudre un même but).

Inconvénients :

- Cette représentation ne permet pas de construire un système flexible puisque les informations sont spécifiées avec un mode prédéfini d'utilisation. L' utilisation des informations est trop déterministe.

- Elle n'est pas modulaire et par conséquent ne facilite pas l'extension ou la modification du système.

- Les systèmes procéduraux ne sont pas toujours cohérents. Par exemple, l'utilisation du raisonnement par défaut peut introduire une incohérence dans le cas de connaissances incomplètes. De même, beaucoup de systèmes procéduraux ne sont pas complets, c'est à dire que dans certains cas, le système peut connaître tous les faits nécessaires pour accomplir un certain but mais ne pas être assez puissant pour réaliser les déductions souhaitables.

- Un tel système peut difficilement expliquer le raisonnement qu'il a suivi car il ne contient pas de connaissances sur son mode de raisonnement et sur ses connaissances.

1.3.3 Les systèmes de production.

Présentation :

Dans de nombreux systèmes experts, tels que MYCIN [Shortliffe 76], spécialisé dans le diagnostic et la prescription des infections bactériennes du sang, PROSPECTOR [Hart & al 78], système pour l'a-

nalyse de dépôts minéraux en vue d'évaluer l'intérêt d'un site pour la prospection minière ou R1 [McDermott & al 82], utilisé par Digital Equipment Corporation pour la configuration des ordinateurs Vax, ou bien d'autres encore, la plupart des connaissances sont représentées par des règles de production.

Un système de production utilise comme représentation des connaissances les règles de production et le modus ponens comme mécanisme de raisonnement.

La partie gauche d'une règle de production exprime les caractéristiques d'une situation pour lesquelles il est approprié d'activer la partie droite de la règle. Ainsi un système de production est composé de règles de production dont la syntaxe est :

SI <Condition> ALORS <Action>. Par exemple : *SI le feu est rouge ALORS vous devez vous arrêter.*

La condition d'une règle de production peut contenir une ou plusieurs conditions appelées les prémisses ou les antécédents de la règle. De même, la conclusion peut être multiple, c'est à dire que la règle peut avoir une ou plusieurs conséquences. Une règle de production décrit donc les actions à réaliser si l'ensemble des conditions est vérifié. Chaque règle est indépendante des autres parce qu'elle est la description d'une "réalité" élémentaire plutôt qu'une suite d'actions. Par contre, elles peuvent déclencher l'activation d'une autre règle si l'une des prémisses ne peut être vérifiée par un fait. Un système de production n'active jamais les règles suivant un ordre fixé.

Généralement, un système de production contient au minimum trois modules :

- une base de connaissances, ensemble de règles de production pouvant contenir des variables. Ces dernières seront instanciées lors du processus de raisonnement lorsque le système tentera de mettre en accord les règles avec les faits connus. C'est l'unification.
- un espace de travail, mémoire où se situent les données d'un problème à traiter, les buts, les résultats intermédiaires ou finaux obtenus au cours du processus de raisonnement, stockés sous la forme de faits.
- un interpréteur de règles ou moteur d'inférence ou encore système cognitif qui contrôle l'activité du système par un cycle à trois phases. La première phase est la sélection d'un ensemble de règles dont la condition est satisfaite par l'état actuel de l'espace de travail. Cette étape s'appelle le filtrage et recherche s'il existe un jeu de substitution de variables rendant deux formules logiques identiques. La deuxième phase consiste à choisir parmi les règles exécutables obtenues à l'étape précédente, un sous-ensemble restreint d'une ou plusieurs règles qui seront effectivement exécutées. C'est la résolution des conflits. Enfin à la dernière étape a lieu l'exécution proprement dite des règles choisies à la phase deux, provoquant généralement l'introduction ou la suppression d'informations dans l'espace de travail. Selon le système de production choisi, différents modes d'invocation des règles existent. En chaînage avant, les prémisses des règles sont examinées ; si elles sont satisfaites, la règle est activée engendrant ainsi de nouvelles connaissances. Ce processus est itéré jusqu'à ce qu'aucune inférence ne satisfasse un but ou qu'il y ait épuisement des inférences exécutables. Ce mécanisme permet d'arriver à une conclusion inconnue a priori (diagnostic) alors qu'en chaînage arrière un but est fixé et le système va tenter de le résoudre. Cet objectif fixé, le système examine les règles concluant sur ce but et vérifie si elles sont satisfaites. Des sous-buts

peuvent apparaître lorsqu'une règle peut être unifiée avec la prémisse d'une règle concluant sur le but initial ou un autre sous-but. En fin de résolution, soit tous les sous-buts ont été testés et ont conduit à un échec, soit les prémisses d'une règle concluant sur le but initial ont pu être vérifiées, conduisant à un succès. Un autre type d'invocation des règles est le mode qui combine simultanément les deux chaînages. Il est ainsi appelé le chaînage mixte.

Les systèmes de production sont une méthode générale qui est particulièrement appropriée quand la connaissance est décomposable en une série d'actions comme c'est le cas dans les systèmes experts. Mais cette représentation ne convient qu'à certains domaines d'application comme le précisent Davis et King [Davis & al 77]. La connaissance doit être diffuse, constituée de nombreux faits (par exemple, en médecine), c'est à dire opposée aux domaines dans lesquels se dégage une théorie concise et unifiée (par exemple, en physique). Les processus doivent pouvoir être représentés par une série d'actions indépendantes. Les règles de production doivent être indépendantes les unes des autres. La connaissance doit être facilement séparable de son mode d'utilisation. Enfin, les tâches à résoudre ne doivent pas faire appel à de nombreux sous-buts fortement reliés entre eux.

Avantages :

- Les règles de production permettent de représenter la connaissance sous forme de petits modules indépendants. Cette modularité, facilite l'ajout et la modification des éléments de connaissance. De plus, la connaissance sur le domaine est séparée de son mode de manipulation et peut être donnée en vrac.
- L'uniformité de la représentation facilite la compréhension de la connaissance stockée.
- Le naturel avec lequel les règles condition-action peuvent être utilisées par le spécialiste pour exprimer son savoir fait que les systèmes de production sont très employés. Le raisonnement humain se formalise assez facilement sous forme de règles de type antécédent-conséquence. En effet, des études faites en psychologie ont montré que l'homme exprime habituellement sa connaissance en disant ce qu'il faut faire dans une situation donnée [Clancey 83].
- Avec ce formalisme, il est possible de gérer des informations incertaines. C'est typiquement le cas de MYCIN [Buchanan & al 1985] qui autorise ce type d'informations par l'introduction de coefficient de certitude pondérant la véracité de chaque règle. De plus, de tels systèmes peuvent raisonner avec incertitude.
- Les systèmes de production apporte des facilités pour fournir des explications. Cela est dû à la fois aux règles et à la structure de contrôle.

Inconvénients :

- La représentation sous forme de règles de production ne s'adapte pas à tous les domaines d'applications comme nous l'avons vu précédement.
- Elle répond très mal au besoin de représenter les éléments de base du domaine d'application.
- L'inefficacité d'exécution, due à la modularité et à l'uniformité de la représentation, principalement en ce qui concerne les séquences d'actions, peut devenir importante d'autant plus si le nombre de règles est grand.
- Enfin il est difficile de suivre le flux de contrôle du système. Ceci est dû au fait que les règles ne peuvent pas s'invoquer directement mais sont contraintes de communiquer à travers la mémoire de travail.

1.3.4 Les réseaux sémantiques.

Présentation :

A l'origine, les réseaux sémantiques ont été développés par Quillian [Quillian 68] à partir de travaux faits sur la modélisation en psychologie de la mémorisation associative des êtres humains. Dans son modèle, les concepts sont représentés par des noeuds et les relations entre ces concepts par des arêtes. Un réseau sémantique est ainsi une méthode pour représenter, de façon déclarative, des relations entre entités. Il est construit à partir de fragments qui sont des relations de la forme : relation (entité 1, entité 2).

Par exemple : est_un (chien, mammifère) représente le fait : le chien est un mammifère.

Un réseau sémantique est donc composé d'un ensemble de noeuds représentant des objets, des ensembles d'objets, des concepts, des événements ou des situations et d'un ensemble de liens entre ces noeuds exprimant les relations binaires qui les unissent. Il est surtout intéressant de représenter les liens entre ces noeuds, ceci est généralement fait en utilisant des prédicats simples ou structurés.

Par exemple, la phrase : *McEnroe a joué hier contre Connors au tennis à Wimbledon* peut être représentée par le réseau sémantique suivant :

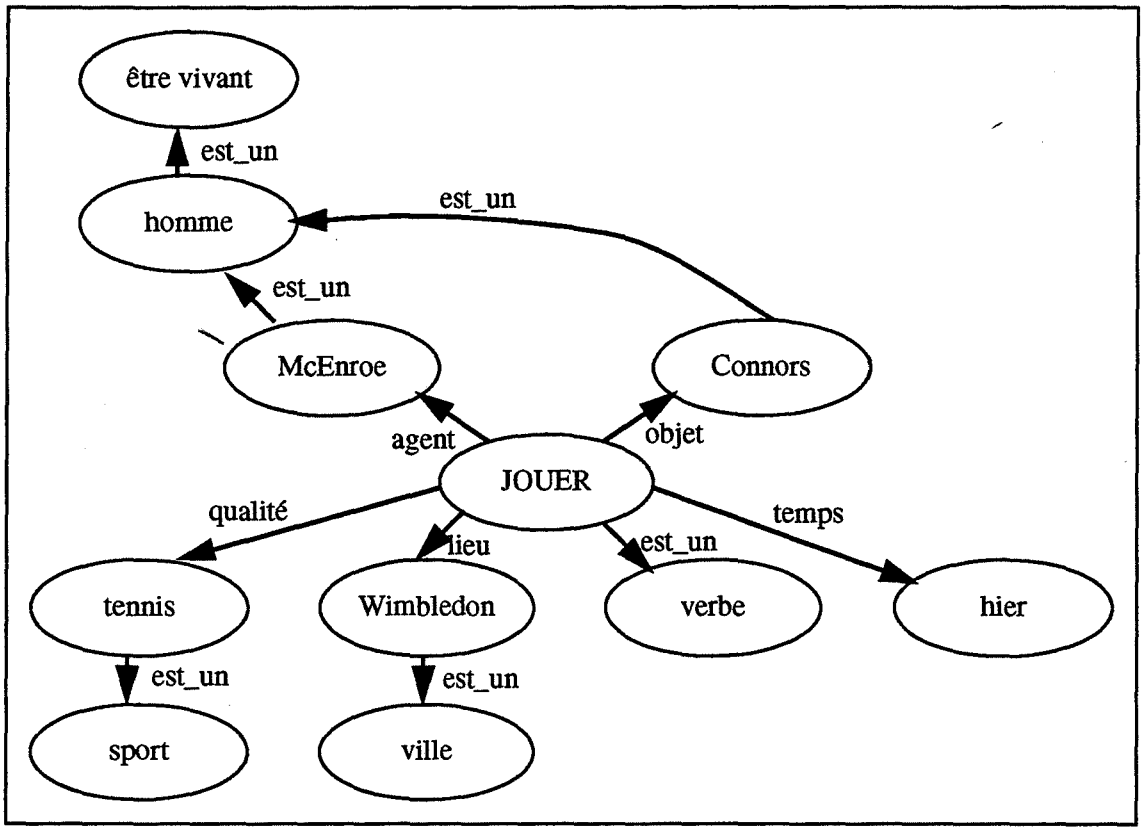
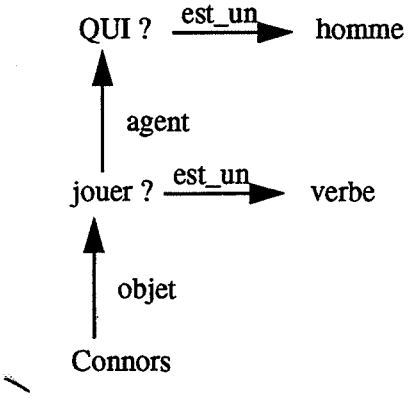


figure 1 : Un exemple simple de réseau sémantique.

La notion de réseau sémantique est proche de celle de graphe étiqueté. Alors que chaque graphe conceptuel représente une seule formule logique, un réseau sémantique représente non seulement une collection de formules mais aussi décrit leurs connexions mutuelles [Thayse & al 90].

Les avantages de cette représentation sont d'une part qu'un élément n'apparait qu'une seule fois dans la description de plusieurs concepts (par exemple homme dans le réseau à la figure 1) et d'autre part, que de nombreuses relations ne sont pas exprimées ; les réseaux sémantiques permettent l'héritage des propriétés. Dans l'exemple figure 1, McEnroe hérite de la propriété : est_un être vivant. Les propriétés d'un noeud, c'est à dire les relations binaires auxquelles il est lié, peuvent être héritées par d'autres noeuds si elles sont accessibles à partir du noeud en question par un chemin constitué par des arêtes spéciales (par exemple du type est_un). Ainsi un réseau sémantique autorise des déductions à l'aide d'inférences par héritage. Ce type d'héritage permet de représenter de façon naturelle des domaines ayant des taxonomies compliquées ou une organisation complexe.

Le mécanisme de raisonnement le plus utilisé dans les réseaux sémantiques est l'appariement (ou filtrage). Ainsi, dans l'exemple précédent, la question : *Qui a joué contre Connors ?* sera construite par le système à l'aide du filtre suivant :



Le système essaie alors d'identifier dans sa base de connaissances une valeur pour chacune des deux inconnues, QUI ? et jouer ? de telle manière que le filtre ci-dessus soit instancié tout en correspondant à une situation vraie.

Le système le plus connu implémentant des réseaux sémantiques est CASNET (Causal ASsociation NETwork, connu aussi sous le nom de GLAUCOMA) [Weiss & al 78] qui est un système expert d'aide au diagnostic et à la thérapeutique des glaucomes (maladies de l'oeil). Dans le domaine de la prospection minière, PROSPECTOR [Hart & al 78] utilise à la fois des règles de production et des réseaux sémantiques qui permettent de représenter les règles par des liens de type antécédent-conséquence. Les travaux de Schank [Schank 72], en compréhension du langage naturel, ont également été d'un grand apport pour les réseaux sémantiques.

Avantages :

- Les réseaux sémantiques sont faciles à lire, à comprendre du fait de l'organisation des connaissances permise par le mécanisme d'héritage. De plus, l'emploi du "pattern-maching" comme mécanisme de raisonnement apporte une facilité d'utilisation.
- Ils sont capables de gérer correctement les exceptions.

Inconvénients :

- Il manque une sémantique formelle et une terminologie standard aux réseaux sémantiques. Certains types de connaissance (procédurales ou quantifiées) ne peuvent pas être représentés. Par exemple, ce modèle est incapable de traiter la quantification mis en jeu dans la phrase : "*tout parent aime ses propres enfants*".
- Du fait de la simplicité structurelle des noeuds, un réseau sémantique est complexe dès qu'il contient beaucoup d'informations. Il est ainsi d'une manipulation délicate et difficile à étendre ou à modifier. La modularité apparente n'est pas un réel avantage.
- Il n'existe pas encore de procédures qui manipulent rapidement ces réseaux.
- Enfin, les systèmes qui utilisent ce formalisme ont des difficultés pour fournir des explications sur leur raisonnement.

1.3.5 Formalisme à base de théories.

Avant d'introduire le formalisme à base de théories, il est intéressant d'examiner le sens usuel de ce vocable. A une théorie est associée un langage dans lequel sont exprimés des énoncés. Le langage doit être suffisamment riche et notamment contenir une notion de négation. Une série de règles permet de distinguer un énoncé d'un assemblage de symboles. D'autres règles permettent d'isoler, parmi les énoncés, certains énoncés de base qu'on appellera : axiomes. Ces axiomes sont considérés comme "vrais". A partir de ces axiomes, d'autres énoncés pourront être déduits. [Thayse & al 90]

Le calcul des prédicats peut être vu comme un équivalent formel de la notion de théorie. Il comporte un langage et des règles permettant de distinguer les énoncés (formules) des simples assemblage de symboles. Une théorie fondée sur le calcul des prédicats est généralement appelées : théorie du premier ordre.

Dans la plupart des cas, une théorie est élaborée pour rendre compte d'une certaine réalité (physique, chimie, politique, astrologie, etc). Le but de la théorie est alors de déduire, à partir d'un ensemble d'énoncés vrais pris comme axiomes, tous énoncés vrais concernant la réalité. Pour l'intelligence artificielle, une théorie constitue la base de connaissances d'un système opérant dans un univers du discours considéré [Haton & al 91].

Un formalisme à base de théorie a été implémenté dans le système EPSILON [Murcan 91]. EPSILON est un gestionnaire de connaissances reposant sur la logique. Dans ce système, chaque entité conceptuelle est modélisée par un concept unique : la théorie. Une base de connaissances EPSILON est formée d'un ensemble de théories, entités hétérogènes qui peuvent être reliés entre elles. Chaque théorie appartient à une classe spécifique qui définit le mécanisme d'inférence, les opérations pour manipu-

ler les théories de cette classe et un ensemble d'outils commun à toutes les théories appartenant à cette classe. Un mécanisme appelé lien permet aux théories de dialoguer entre elles. Un dictionnaire de la base de connaissances contient les informations sur les théories et sur les liens :

- pour les théories :
  - son nom,
  - un indicateur spécifiant si la théorie définit ou non une classe,
  - le nom de la classe à laquelle appartient la théorie.
- pour les liens :
  - le nom du lien,
  - les noms des deux théories qu'elle met en relation.

Fournir une solution basée sur la programmation logique aux problèmes de représentation et de manipulation de connaissances constitue le but d'EPSILON. Les concepts de théorie, classe, et lien donnent à la programmation logique les avantages de la modularité et permettent de travailler avec différents formalismes de représentation des connaissances et différentes formes de raisonnement.

1.4 Représentation des connaissances et système à base de connaissances.

Un système à base de connaissances est généralement au moins composé de quatre principaux modules qui sont (1) un système cognitif, dont la fonction la plus importante est de résoudre les problèmes qui sont posés au système, (2) une base de connaissances pour le stockage des connaissances du domaine considéré acquises auprès des experts compétents, (3) une interface assurant un dialogue convivial entre utilisateur et système, (4) un module explicatif donnant le raisonnement suivi par le système pour aboutir à certaines solutions et fournissant d'autre types d'explication. De plus, toutes les opérations effectuées par le système cognitif ont lieu au sein d'un espace de travail où sont stockés, entre autres, les données du problème et les résultats intermédiaires. Il contient donc l'état courant du système. Concevoir un système à base de connaissances implique, après la phase d'acquisition des connaissances, d'introduire celles-ci dans l'architecture décrite ci-dessus. La figure 2 présente cette opération de transcription des connaissances acquises dans un système à base de connaissances.

De plus, cette figure nous montre que, suivant les connaissances à représenter un formalisme sera choisi et que celui-ci va influencer le choix des méthodologies utilisées principalement pour trois des composantes d'un système à base de connaissances.

1. La base de connaissances.

On peut employer un des formalismes présentés dans le paragraphe 1.3. Un autre formalisme possible est celui des objets, que ce soit avec l'aide d'un langage de classes, de frames ou encore d'acteurs. Le chapitre deux présentera ce formalisme.

2. Le système cognitif.

Le rôle du système cognitif est de raisonner en utilisant la base de connaissance, l'espace de travail, les connaissances qu'il contient. Ainsi il est hautement dépendant du formalisme choisi pour la représentation des connaissances. Son processus de résolution se déroule en cinq phases :

- la sélection de l'activité à exécuter dans l'agenda des actions en attente,
- la génération, l'exécution de l'activité sélectionnée,
- la mise à jour de l'agenda des actions à exécuter du fait de la création de nouveaux faits à l'étape précédente,
- l'élagage pour éliminer les cas sans intérêt,
- et enfin la terminaison détermine si le processus doit être continué ou si le problème a été résolu.

Le système cognitif doit donc également implémenter des techniques de recherche dans un espace de possibilités afin de réduire le nombre d'opérations qu'il devra exécuter.

3. L'espace de travail.

En général, le concepteur n'a pas vraiment le choix d'une technique car le choix fait sur la représentation des connaissances et sur le processus de raisonnement limite énormément la représentation de l'espace de travail. En fait, il comprend des variables globales (données d'entrées, but, valeurs calculées), un agenda listant les actions à exécuter par la suite et un historique de la résolution, c'est à dire le chemin suivi pour arriver à l'état courant du système. Par exemple, dans un système de production, l'espace de travail contient l'état du système avec un agenda implicite : l'ensemble des règles qui peuvent être appliquées, obtenu lors de la résolution des confits. Dans un système à base de connaissances dont l'architecture choisie est le tableau noir, un "tableau" représente l'espace de travail qui sert d'intermédiaire entre les différentes sources de connaissances et le système cognitif.

Tous les systèmes experts nécessitent le choix d'une représentation des connaissances. Chacun des formalismes présentés dans ce paragraphe offre des avantages et des inconvénients. En fin de cette première partie, un tableau récapitulatif permettra de faire le point entre ces différents formalismes et ceux présentés dans les chapitres suivants en fonction des diverses caractéristiques nécessaires pour bien représenter la connaissance.

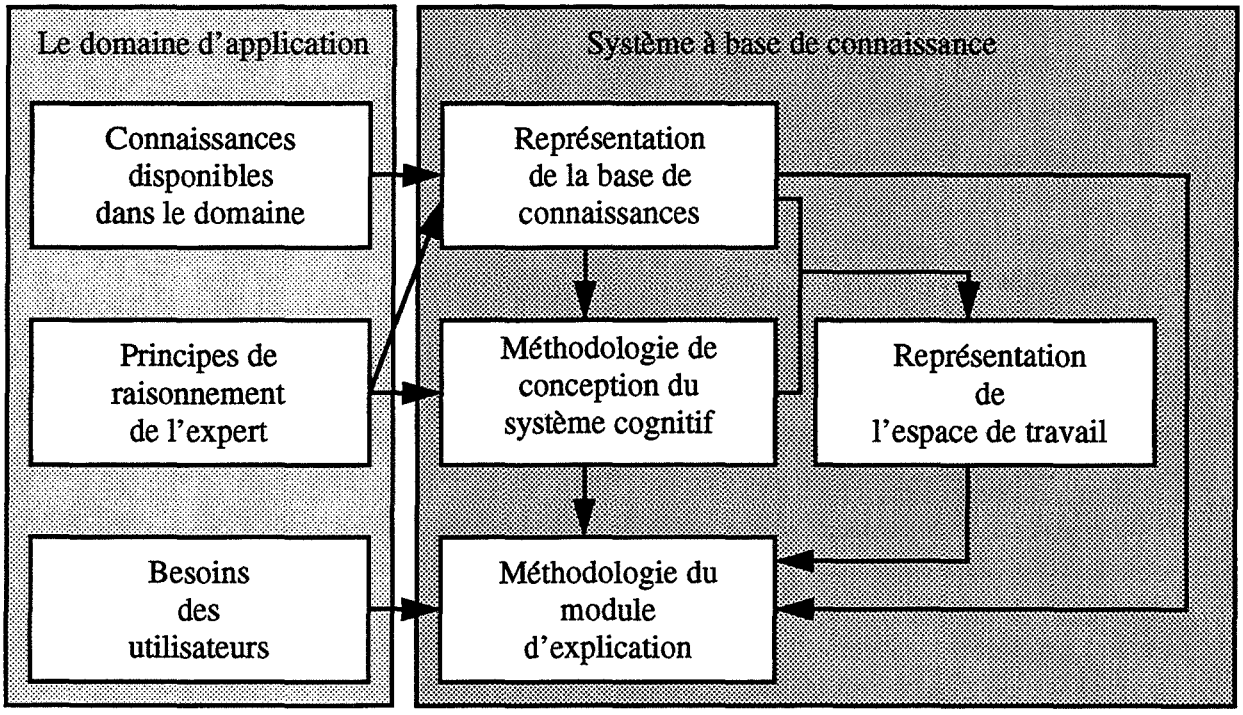


figure 2 : Les Interactions lors de la représentation du domaine de connaissance dans un système à base de connaissances.

CHAPITRE 2

Programmation orientée objets.

Tant dans le domaine de l'intelligence artificielle que dans celui du génie logiciel, la programmation orientée objets connaît, à l'heure actuelle, un succès croissant car elle répond à certains besoins actuels : facilité pour tester, pour améliorer, pour réutiliser et pour maintenir.

Au premier abord, la programmation par objet est surprenante bien que tendant à représenter des entités et des mécanismes proches des phénomènes physiques. Pour un développeur habitué à des méthodes conventionnelles, la programmation orientée objets est source de changements majeurs dans la manière de conceptualiser et de développer les programmes. En effet, les programmeurs classiques ont tendance à penser un programme comme une série d'actions à exécuter tandis qu'en programmation orientée objets, l'accent est mis sur les structures de données qui décrivent le problème. Cette programmation est dirigée par les données : pour traiter une application, le programmeur commence par définir les types d'objets appropriés avec leurs opérations spécifiques. Chaque entité manipulée dans le programme est un représentant d'un de ces types (instance). L'univers de l'application est composé d'un ensemble d'objets qui détiennent, chacun pour sa part, les clés de leur comportement.

Le concept clé de la programmation orientée objets résulte du fait qu'une collection de données et les opérations qui sont normalement réalisées sur ces dernières sont très liées et doivent être traitées dans une seule entité plutôt que séparément. Alors que dans l'informatique traditionnelle, procédures et structures de données sont indépendamment implémentées, un système orienté objet n'utilise qu'un seul concept : l'objet qui englobe les deux.

J. Ferber [Ferber 87] distingue trois grandes familles de langages à objets, chacune privilégiant un point de vue de la notion d'objet :

- le point de vue structurel : l'objet est un type de données, qui définit un modèle pour la structure de ses représentants physiques et un ensemble d'opérations applicables à cette structure. Ce sont les langages de classes. Simula et Smalltalk-80 en sont historiquement les chefs de file.
- le point de vue conceptuel : l'objet est une unité de connaissances, représentant le prototype d'un concept. Cette deuxième famille regroupe les langages de frames[1].

1. traduit quelquefois par *cadre, trame* ou *schéma.* L'usage du terme anglo-saxon est maintenant communément admis.

- le point de vue acteur : l'objet est une entité autonome et active, qui se reproduit par copie.

Une quatrième catégorie regroupe les langages hybrides dont les caractéristiques sont empruntées à l'une ou l'autre des trois familles présentées ci-dessus. La plupart du temps, ces langages comprennent une composante de programmation logique parce qu'il est parfois difficile de représenter dans un formalisme objet des connaissances qui se traduisent plus naturellement en formules logiques ou encore en règles de production. LOOPS [Bobrow & al 83] développé au dessus de Lisp, en est très certainement le précurseur.

Dans ce chapitre, après un bref historique, je m'intéresserai plus particulièrement aux langages de classes, en introduisant les concepts et mécanismes de base et en les comparant aux autres types de langages (à base de frames, d'acteurs). Un paragraphe traitera de l'utilisation des objets dans les systèmes à base de connaissances. Enfin une dernière section développera les avantages d'une telle programmation.

2.1 Un peu d'histoire.

Le langage Simula 67, conçu par Ole-Johan Dahl et Krysten Nygaard en 1967 [Dahl & al 70], est à l'origine des langages orientés objet. Simula introduit le terme d'objet pour qualifier un appel de procédure en attente d'exécution. Un objet est ainsi une instance d'une procédure dont l'appel est suspendu et la notion de classe permet d'allouer dynamiquement les processus envisagés tout en associant à chaque classe des règles de comportement. A l'exception de l'héritage multiple, tous les principaux concepts de la programmation orientée objets y étaient définis. Mais jusque vers 1981, les méthodes à objet sont restées connues seulement d'une minorité car elles apparurent porteuses de concepts réputés à l'époque difficiles (liste, appel par valeur, récursion, file d'attente, classe ...). Il a fallu l'apparition du langage Smalltalk [Goldberg & al 83] pour qu'elles soient connues d'un plus large public. En effet, Smalltalk a su adapter ses concepts à des applications pratiques et grand-public hautement interactives. L'apport principal de Smalltalk est l'intégration des concepts orientés objet à un puissant environnement de développement et à un support graphique de très haute qualité. C'est aussi en 1981 que certains langages ont apporté leur contribution à la programmation orientée centrés objets comme Actors par les travaux effectués sur l'envoi de messages et Flavors pour l'héritage multiple. De même, la théorie sur les "frames" et son implémentation dans des langages tels que KRL (1977), FRL (1977), UNITS (1979), KEE (1985) furent d'un grand intérêt pour le modèle centré objets.

Ces dernières années de nombreux langages et systèmes sont apparus. Beaucoup sont des extensions de langages déjà existants auxquels des notions de la programmation par objets ont été ajoutées. Je citerais ainsi le langage C étendu à C++ d'AT&T [Stroustrup 86] et à Objective-C de la société PPI ou la dernière version de Pascal (Apple a développé Object Pascal [Tesler 85]). Ces langages n'offrent pas l'héritage multiple. Des extensions de Lisp sont également disponibles comme Loops de Xerox, Flavors du MIT ou Ceyx de l'INRIA. Enfin le langage Eiffel hérité de Simula et de l'évolution des

idées développées en génie logiciel pour améliorer la qualité du logiciel apporte flexibilité, réutilisabilité, fiabilité et efficacité [Meyer 87].

Actuellement, on peut dénombrer une cinquantaine de langages de ce type, mais la plupart ont une distribution très limitée. La figure 3 situe dans le temps les langages les plus répandus et leur langage origine (les langages orientées objets apparaissent dans les boîtes en gras).

Volontairement, dans cette énumération ne figurent pas les langages ADA et Modula-2, pourtant généralement qualifiés d'orientés objet. Bien que permettant une programmation modulaire et offrant les propriétés d'abstraction de données et d'encapsulation, ils n'utilisent pas la notion d'objet proprement dite et n'implantent pas le mécanisme d'héritage.

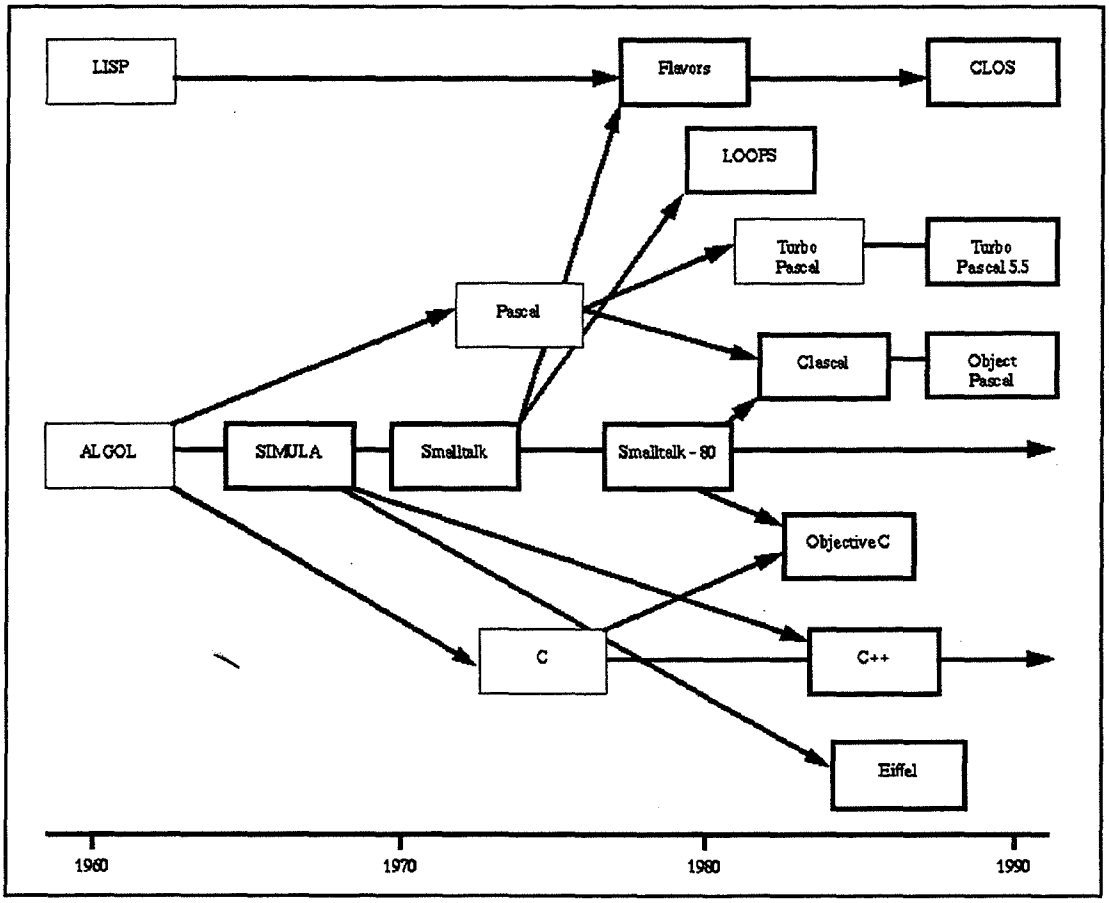


figure 3 : Quelques langages orientés objet à travers les âges.

2.2 Les concepts généraux.

2.2.1 Les mots clé.

Très peu de concepts sont nécessaires pour la bonne compréhension de n'importe quel langage orienté objet. En effet, quelques mots-clés suffisent à caractériser la programmation orientée objets : objet, attribut, méthode, classe, métaclasse, instance, message et démon.

2.2.1.1 Les notions d'objet, d'attribut, de méthode.

Le concept fondamental de la programmation orientée objets est l'idée d'objet. Un objet est une unité de programmation auto-suffisante dans le sens où elle possède ses propres données locales et les procédures de traitement de ces dernières.

Les langages de programmation "classique" (procédurale) introduisent une dichotomie entre les données et les opérations qui peuvent être effectuées sur ces dernières. L'approche orientée objet supprime ce découpage en "encapsulant" données et fonctions d'utilisation au sein d'une même entité autonome, l'objet. Ainsi un objet est constitué d'un ensemble d'attributs, propriétés caractérisant l'objet et d'un ensemble de procédures, les méthodes décrivant ses actions potentielles, son comportement.

Par exemple, la définition d'un objet "machine" peut être faite par la donnée des attributs et méthodes suivants :

Machines
attributs :
serveur_d_application : Liste de Machines qui servent cette machine.
serveur_pour_les_machines_sans_disque : Serveur de cette machine si elle est sans disque.
est_un_serveur : Booléen affecté à vrai si c'est un serveur, à faux sinon.
sorties : Liste des types de ses sorties.
méthodes :
deplacer : pour déplacer une machine dans une autre pièce.
detruire_un_lien : pour déconnecter une machine du réseau.
ajouter_un_lien : pour connecter une machine à un connecteur vide.
a_un_disque : permet de connaître le disque connecté à la machine s'il existe.

exemple 1 : L'objet Machines.

2.2.1.2 Les notions de classe, de métaclasse, d'instance.

Notion de classe.

L'idée est de regrouper les objets obéissant à la même sémantique, donc partageant le même ensemble de propriétés, mais dont les attributs peuvent avoir des valeurs distinctes. Ainsi une classe est

une structure de données générique qui décrit les informations relatives à un ensemble d'objets.On appelle donc Classe le moule qui permettra de définir de nouveaux objets, instances de cette classe. Les classes sont des abstractions, des gabarits qui décrivent le format que toute instance créée à partir de cette classe aura.

Au niveau de l'implémentation, ce mécanisme de classe permet d'une part de décrire une fois pour toutes les caractéristiques propres à un ensemble d'objets dans une classe, d'autre part de localiser les procédures au niveau des classes tout en autorisant un partage du code.

Notion d'instance.

Une instance est un des représentants "physiques" d'une classe, représentant pour lequel les attributs définis dans la classe seront affectés à une valeur. Son comportement est défini par les méthodes de la classe. L'exemple 2 donne deux représentants de la classe Machines définie à l'exemple 1. Leurs propriétés ont été affectées à une valeur. Il est possible de leur appliquer les méthodes définies dans la classe Machines telles que les déplacer, tester si elles ont un disque. Ainsi a-t-on pu définir deux objets différents mais ayant la même sémantique, l'objet Machines décrivant de manière générale tout type de machine.

Sparc_SLC_1
attributs :
serveur_d_application : [Sparc_SLC_3, Sun_386i_1]
serveur_pour_les_machines_sans_disque : Sparc_SLC_3
est_un_serveur : faux
sorties : [[AUIF1]]

Sparc_SLC_2
attributs :
serveur_d_application : [Sparc_SLC_3, Sun_386i_1]
serveur_pour_les_machines_sans_disque : Sparc_SLC_3
est_un_serveur : faux
sorties : [[AUIF1]]

exemple 2 : Deux instances de la classe Machines.

Une instanciation est une création dynamique d'un objet dont les caractéristiques sont définies dans une classe. On appelle généralement "in" le lien qui unit l'instance à sa classe. Une instance est une occurrence spécifique du monde réel alors qu'une classe en est une abstraction.

Notion de métaclasse.

De même que pour les instances, les classes peuvent être regroupées au sein d'une même métaclasse. Une classe est alors une instance d'une métaclasse. Ainsi les métaclasses sont garantes de l'ho-

mogénéité du langage et lui apportent la réflexivité : le système est ainsi capable d'avoir un rapport de cause à effet avec lui-même car les métaclasses décrivent la connaissance que le système a de lui-même. Par ailleurs, elles permettent au concepteur du langage d'engendrer le noyau à partir duquel sera défini tout objet. Une métaclasse peut être utilisée pour particulariser le comportement d'un groupe de classes en la dotant de méthodes particulières. Cette possibilité permet par exemple, de spécifier comment imprimer le contenu de toute classe ou de redéfinir la fonction de création des instances et ainsi masquer celle définie dans la classe mère de toute classe.

Par contre elles n'offrent guère d'intérêt pour le génie logiciel qui considère une classe plutôt comme étant une spécification du comportement d'un ensemble d'objets qu'un objet à part entière. Leur existence est même perçue comme un handicap du fait qu'elles autorisent une création dynamique d'objets et rendent le typage statique donc la compilation impossible. Ainsi, la notion de métaclasse n'existe pas dans le langage Eiffel, par exemple.

On vient de faire la distinction entre les notions d'instance, de classe et de métaclasse ; en fait, tout est objet. Le terme objet est employé comme étant générique.

2.2.1.3 Les notions de message, méthode, démon.

L'objet apparait comme un système autonome détenteur d'un ensemble de connaissances intrinsèques mais il peut aussi avoir à communiquer avec le monde extérieur par l'intermédiaire d'un protocole : la transmission de message.

Envoyer un message permet d'activer une des fonctions attachées à l'objet receveur. Trois éléments sont nécessaires à la constitution d'un message : l'objet receveur, le nom de la méthode et les arguments à transmettre si la méthode le nécessite. Ainsi la syntaxe d'un message est :

<Objet-Receveur> <Méthode-sélecteur> <Liste_des_Arguments>

L'objet receveur peut être omis dans le cas où un objet fait appel à l'une de ses propres méthodes.

Les méthodes définies dans l'objet sont ainsi déclenchées par les messages envoyés par une méthode d'un autre objet ou par un programme externe.

Dans l'exemple ci-dessous, le calcul du coût total d'un réseau est une addition du prix de chacune des composantes du réseau : câbles, éléments de connexion, composants additionnels (software ou hardware) pour les boîtes de connexion. Outre la nécessité de demander à chaque objet faisant partie du réseau la valeur de son attribut coût, la méthode cout_du_reseau doit envoyer un message pour connaître le coût de chacun des câbles qui prend en compte sa longueur et son coût d'installation. Ainsi cout_du_reseau envoie un message à chaque objet de type Cables composant le réseau (en BIM_Probe cet envoi peut se faire par l'instruction send).

Cables
attributs : ...
méthodes : ...
cout_du_cable : pour calculer le coût du câble en fonction de son coût d'installation et de sa longueur
...
Reseaux
attributs : ...
méthodes : ...
cout_du_reseau : ...
..., send (cable_cost (_cable, _valeur)), ...

exemple 3 : transmission de message.

Alors que les méthodes sont des procédures ou fonctions attachées à un objet et activées par l'intermédiaire d'un message, le démon est un type spécial de procédure attachée à un attribut. Il est déclenché lors d'un accès ou d'une modification de la valeur de cet attribut. Tous les langages orientés objet ne supportent pas cette notion de démon. Dans l'exemple ci-dessus, au lieu de faire appel à une méthode de la classe Cables pour le calcul du coût d'un câble, il est possible d'avoir un attribut coût auquel est associée une procédure qui sera activée à chaque accès à cette valeur. Cette deuxième méthode est préférable dans ce cas puisqu'elle apporte une plus grande homogénéité dans le calcul du coût du réseau.

2.2.2 Les principaux mécanismes propres à la programmation orientée objets.

2.2.2.1 L'héritage et l'héritage multiple ---> la notion de hiérarchie.

La description d'objets du monde réel fait ressortir généralement des caractéristiques communes à plusieurs entités. Afin d'éviter les redondances d'informations, l'héritage permet de factoriser les propriétés communes à plusieurs classes, en les fournissant dans une classe supérieure. Le mécanisme d'héritage a été introduit pour permettre à un objet d'hériter des attributs ou des méthodes d'un autre objet. C'est un moyen de partage de connaissances entre objets.

Ainsi par exemple, on admet facilement que la classe des humains a des caractéristiques communes avec celle des mammifères. Il y a inclusion. On décrira alors ces caractéristiques dans la classe des mammifères et celles propres à l'homme dans la classe des humains et on créera un lien "isa" indiquant que la classe des humains hérite de la classe des mammifères.

Si les objets sont organisés sous forme d'arbre, ils sont hiérarchiquement structurés des classes plus générales aux classes plus spécialisées. Une classe peut par conséquent posséder des sous-classes et une ou plusieurs super-classes. Le mécanisme d'héritage est une recherche de bas en haut à travers la hiérarchie des objets, de propriétés définies dans des classes d'ordre plus élevé. L'objet sommet de

l'arbre est la *racine.* Généralement, cette classe racine est appelée Object ou Root. Tout objet créé est descendant de cette racine et donc en hérite. Deux autres objets sont également généralement prédéfinis dans les langages orientés objet : Class et Metaclass.

On parle de *parent* d'un objet pour qualifier tout objet qui est au dessus de lui dans la hiérarchie. Le parent direct est celui qui à un lien "isa" direct. De même on parle des *enfants* pour citer les objets inférieurs et qui lui sont liés. Un objet hérite de tous ses parents.

Dans beaucoup de systèmes orientés objet, l'héritage multiple (une même classe hérite de plusieurs super-classes) est permis mais son utilisation reste difficile parce qu'elle requiert de résoudre les conflits pouvant exister en cas d'attribut ou de méthode de même nom hérité de plusieurs parents.

La technique de l'héritage est la clé principale apportant la facilité de développement lorsqu'un langage orienté objet est utilisé car elle évite de recréer des champs identiques pour plusieurs objets. Il suffit de s'assurer de la programmation des bons liens de parenté entre objets.

2.2.2.2 Spécialisation - Généralisation.

Une classe peut être conçue à partir d'observations du monde réel ; c'est alors une *classe concrète.* Ou c'est une *classe abstraite* ; elle exprime alors un concept et/ou elle a été construite dans le but de fournir des caractéristiques héritables.

Les classes abstraites peuvent être induites de classes concrètes par *généralisation.*

Les classes concrètes peuvent provenir d'autres classes par *spécialisation.*

En fait, une classe doit être considérée comme un réservoir de connaissances à partir duquel il est possible de définir d'autres classes plus spécifiques. Ces nouvelles classes contiennent de nouvelles propriétés ou méthodes ou respécifient une ou plusieurs propriétés ou méthodes déjà définies dans leur(s) super-classe(s). Cette opération s'appelle la spécialisation d'une classe. Les connaissances les plus générales sont ainsi mises en commun dans des classes qui sont ensuite spécialisées lors de la définition de sous-classes successives contenant des connaissances de plus en plus spécifiques.

2.2.2.3 Encapsulation - masquage de l'information.

L'encapsulation exprime le fait qu'un objet est une entité complète par elle-même qui "encapsule" à la fois données et méthodes. Aucun objet ou programme extérieur n'a la possibilité d'accéder ou de modifier les attributs d'un objet sans faire appel à une des méthodes de cet objet. Il y a donc masquage de l'information.

2.2.2.4 Polymorphisme.

Littéralement, le polymorphisme est la capacité pour une entité de prendre plusieurs formes. Dans les langages orientés objets, toute variable désignant un objet est potentiellement polymorphe puisqu'elle peut désigner au cours d'une exécution plusieurs objets de types différents, grâce à l'héritage.

Pour clôturer ce paragraphe et le résumer, la figure ci-dessous présente le stéréotype d'un objet.

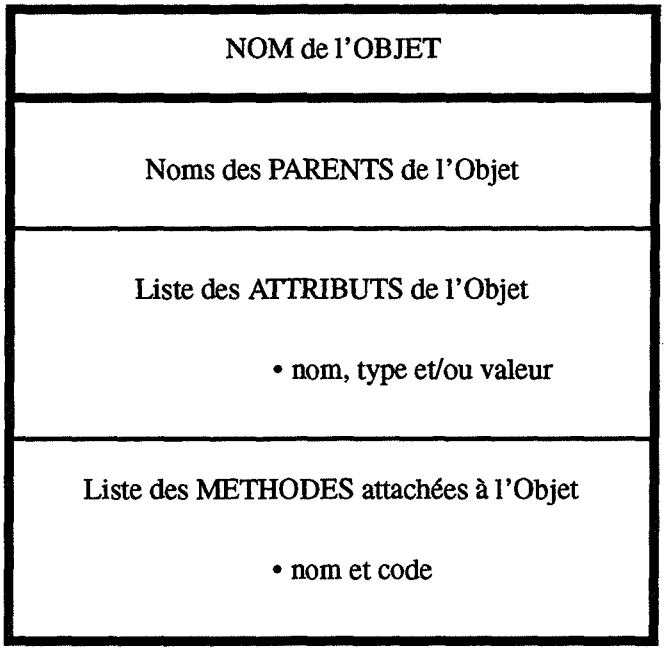


figure 4 : Un diagramme conventionnel de l'objet.

2.3 Etude des mécanismes de frame et d'acteur et comparaison avec les langages de classes.

2.3.1 Les frames.

La structure de frame a été imaginée par M. Minsky [Minsky 75] pour des études en psychologie cognitive. La théorie de Minsky est fondée sur une observation : *pour analyser une situation nouvelle, un humain consulte des structures élémentaires qui intègrent les situations auxquelles il a déjà été confronté. Il choisit la structure qui lui semble la plus proche de la situation courante et la modifie éventuellement pour la faire correspondre à cette nouvelle situation.* Ainsi un frame est une structure de données permettant de représenter une situation stéréotypée. Minsky jugeait que les noeuds d'un réseau sémantique ou que les prédicats logiques sont des granules de connaissances trop petites et trop peu structurées pour prendre en compte l'aspect factuel et l'aspect procédural du raisonnement. Les frames sont des unités de connaissances complètement structurées, de granularité plus importante. Aujourd'hui les structures appelées communément frames respectent encore l'esprit de la définition de Minsky, mais leur forme a beaucoup évolué. Leur conception repose sur une structure à trois niveaux : frame - attribut - facette. Les différentes propriétés d'un frame sont décrites par ses attributs qui eux-même possèdent des facettes. Celles-ci servent à décrire la nature de l'information que l'attribut con-

tient mais aussi à préciser comment la calculer ou l'utiliser. Elles peuvent être soit déclaratives (elles associent alors des valeurs aux attributs), soit procédurales ; on les appelle alors des réflexes. Les réflexes sont activés lors des accès aux valeurs des attributs.

DATE

attributs	*facettes*
MOIS	NOM
JOUR	ENTIER (INTERVALLE 1 31)
ANNEE	ENTIER (POUR REMPLIR SUPPOSER 1992)
JOUR_SEMAINE	(MEMBRE (DIMANCHE, ..., SAMEDI)) (QUAND INSTANCIE TROUVER_DATE_DEPUIS_JOUR) (POUR REMPLIR TROUVER_JOUR)

exemple 4 : le frame DATE.

Les langages de frames et de classes présentent de nombreux points communs :

- Tous deux décomposent l'univers en termes d'objets qui encapsulent données et procédures. Un frame apparait comme un objet particulier sans action sémantique mais dont l'état est déterminé par un ensemble de champs dont les valeurs sont, soit obtenues directement, soit calculées par activation d'une fonction (réflexe lancé lors d'un accès explicite au champ).
- Ils sont organisés en une hiérarchie d'héritage. Dans l'ensemble des frames, tout objet est un représentant des frames dont il est issu et un générateur de frames plus spécialisés. Le lien sorte_de symbolise le lien d'héritage entre frames.

Par contre, ils diffèrent par deux aspects fondamentaux :

- La hiérarchie des classes est inspirée du modèle des ensembles. En effet, elle peut être modélisée par des ensembles abstraits inclus les uns dans les autres. Une classe étant le modèle permettant la création de tous les objets (instances) appartenant à cet ensemble. Par contre, les langages de frames reposent sur la théorie des prototypes déduite d'observations faites par des psychologues sur les processus de raisonnement des êtres humains. *Ceux-ci ont tendance à identifier une famille d'objets et à mener des raisonnements sur ses membres en faisant référence à un objet précis, typique de la famille* [Rosch 75]. Ainsi un concept décrivant une catégorie d'individus peut être matérialisé par l'un d'eux, généralement le plus connu et appelé prototype. Une nouvelle instance est par conséquent liée à un prototype et les valeurs de ses attributs sont celles données pour ce prototype sauf si elles se voient explicitement affecter une autre valeur. Le modèle des

ensembles n'est pas dans tous les cas un bon modèle de représentation du monde réel. En particulier, les exceptions ou la représentation des connaissances incomplètes (ce qui est souvent le cas puisque la liste des caractéristiques essentielles est très difficile à fixer a priori) sont difficilement modélisables par des ensembles. Le modèle des ensembles et donc les langages de classes sont souvent trop souples pour être utilisé en représentation des connaissances et il est donc préférable d'avoir recours à une autre approche ; celle des frames convient mieux.

- Des méthodes sont associées à chaque classe pour décrire son comportement et sont activées par l'envoi de message explicite. En revanche, les frames possèdent des opérations attachées à leurs attributs, les réflexes, activés par effet de bord lors des accès aux attributs.

2.3.2 Les acteurs.

La structure de contrôle des acteurs est issue de travaux menés par C. Hewitt [Hewitt & al 73]. A l'origine, le modèle d'acteur a été développé pour représenter des connaissances de manière distribuée, sous la forme d'une société d'experts coopérant les uns avec les autres. Un acteur est une particule de connaissance potentiellement active qui communique avec les autres acteurs par envoi de messages. Son comportement est défini par un script où est décrit la manière dont il réagit aux événements provoqués par les autres acteurs. Les connaissances et le comportement d'un système développé avec un langage d'acteurs sont diffusés parmi les différents acteurs qui effectuent des tâches en parallèle, de manière indépendante. Le modèle est donc réparti.

Tout en décrivant plus en détails ce qu'est un acteur, intéressons nous à sa comparaison avec les classes:

- Comme pour les classes, un acteur regroupe à la fois (1) des données locales, appelées *accointances* qui sont les autres acteurs que cet acteur connaît directement, (2) des actions que l'acteur peut avoir à faire suivant les événements qui surviennent. Son comportement n'est pas défini comme pour les classes par des méthodes mais par un script unique. Ce dernier filtre les messages reçus par l'acteur et active la partie appropriée du comportement. Comme pour les classes, les acteurs encapsulent données et procédures.
- Un autre type de donnée que l'acteur possède est ses *intentions* qui décrivent ses actions potentielles, ses capacités. Il connaît les intentions de ses accointances mais non la manière dont elles sont implantées. Ainsi la notion classique d'abstraction de données est également satisfaites pour les langages d'acteurs : un acteur est défini par ses capacités et non par sa représentation physique.
- Comme pour les langages de classes, ceux d'acteurs sont des systèmes homogènes ; l'acteur y est l'unique entité.
- Par contre, contrairement aux classes, les acteurs ne sont pas organisés hiérarchiquement. Un équivalent du mécanisme d'héritage est défini par délégation de messages à un

autre acteur quand l'un d'eux ne peut pas le traiter. Chaque acteur connaît un autre acteur, appelé son "proxy" dans Act 1 [Lieberman 87], à qui il peut déléguer les messages auxquels il ne sait pas répondre. L'acteur "proxy" prend alors en charge la tâche qui lui a été déléguée. Comme l'héritage, elle constitue un moyen de partager des connaissances et des comportements [Stein 87]. Un acteur peut avoir un comportement général et être le "proxy" d'autres acteurs au comportement plus spécifique. La délégation paraît cependant plus souple que l'héritage de classes [Masini & al 90]. Un acteur peut choisir dynamiquement de nouveaux "proxies", alors que le partage de la connaissance établi par le graphe d'héritage d'une classe est fixé statiquement.

- Une classe décrit les propriétés communes à un ensemble d'instances, ces dernières étant créées par moulage de la classe. Par contre, un acteur est créé soit par une copie rigoureusement identique, soit par une copie différentielle, c'est à dire spécialisée par adjonction de nouvelles caractéristiques. Aucun lien ne rattache un acteur à celui à partir duquel il a été conçu.
- Pour ce qui est de l'envoi de message d'un acteur à un autre, il diffère du mécanisme des classes du fait qu'un acteur intermédiaire : le messager est utilisé pour transmettre un message d'un acteur émetteur à une cible (voir figure 5). Le messager est un acteur particulier dit *primitif* qui n'a pas besoin de passer par l'intermédiaire d'un messager pour réaliser son objectif, heureusement. Une de ses accointances est un acteur enveloppe qui contient le message à transmettre. A la réception d'un message, l'acteur cible exécute l'opération qui lui est demandé et retourne le résultat. Un message contient une *continuation* qui désigne l'acteur auquel doit être transmis ce résultat. Celui-ci peut être transmis à tout autre acteur que celui qui a émis le message. On parle de communication asynchrone pour qualifier le fait qu'un acteur ayant envoyé un message n'est pas bloqué et peut continuer son activité même s'il n'a pas reçu la réponse à son message. Inversement, une classe n'a aucun rôle actif mais détient le comportement de ses instances qui sont activées par envoi de messages synchrones.

En conclusion, notons que le parallélisme intrinsèque du modèle bouleverse quelque peu les habitudes de la programmation. D'autre part, les langages d'acteurs restent pour l'instant peu diffusés et sont loin d'être des produits finis de qualité comparable à celle de certains langages orientés objets (Smalltalk-80, par exemple). Mais l'arrivée des architectures parallèles et des réseaux de processeurs leur promet un bel avenir.

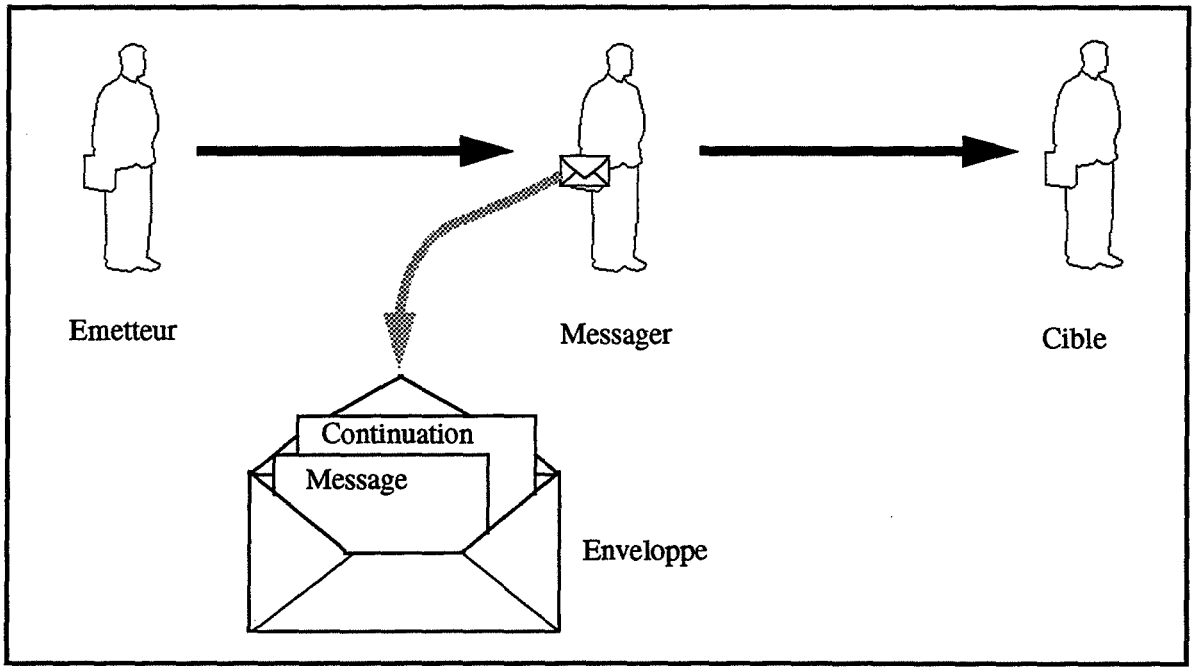


figure 5 : L'envoi de message entre acteurs.

2.4 Utilisation de l'approche objet dans les systèmes experts.

2.4.1 Modèles centrés objets et systèmes experts.

Comme un système expert est un logiciel qui résout des problèmes ou fait des recommandations en utilisant des connaissances symboliques et en raisonnant à partir de faits, il est parfois intéressant de faire appel à la notion d'objet pour modéliser ceux-ci.

En étudiant les systèmes experts existants, il apparait que les approches utilisées sont :

- des langages ou des générateurs de systèmes experts à base de règles,
- des langages ou des générateurs de systèmes experts à base de règles auquels sont ajoutées quelques caractéristiques des modèles centrés objets ou de frames (exemple: OPS5),
- des langages ou des générateurs de systèmes experts avec des règles et des frames ou des fonctionnalités limitées des modèles centrés objets (exemple : Hybrid I),
- des langages ou des générateurs de systèmes experts avec des règles et un langage orienté objet (exemple : Hybrid II).

Typiquement, les systèmes experts ont des structures de données ou des modèles plus riches et plus complexes que des programmes plus conventionnels, incluant ceux à base d'objets. Les objets

peuvent être utiles pour stocker et modéliser les connaissances nécessaires à l'implantation de tout système expert.

Lorsque les possibilités d'un modèle orienté objets sont exploitées dans un système expert en plus des règles, les différents types de connaissances sont implantés en utilisant les mécanismes suivants :

types de connaissances	modélisation
éléments de base, concepts	objets
inférences	règles, démons, héritage
contrôle procédural	système procédural, envoi de messages (encapsulation)
interface	objets, démons

Les différentes technologies présentées par la figure 6 peuvent être employées pour la conception d'un système expert, selon le mode de programmation utilisé.

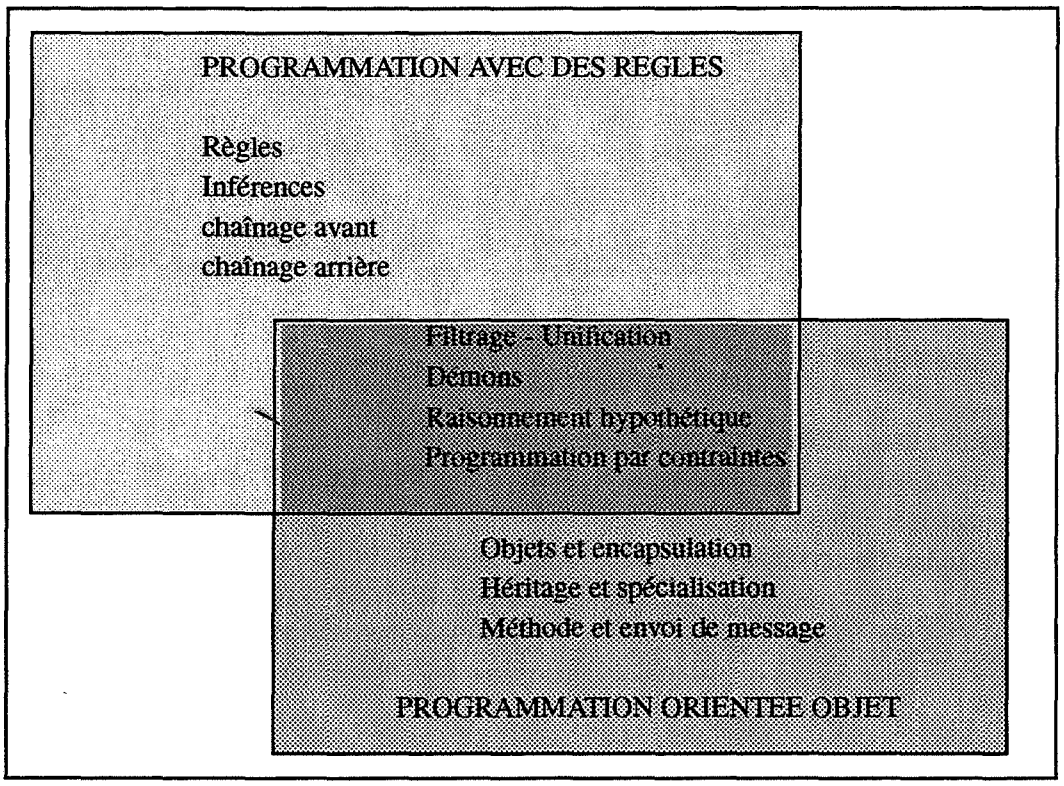


figure 6 : Technologies utilisées pour le développement des systèmes experts.

Le modèle centré objets est particulièrement utilisé pour le développement des interfaces. Comme le montre la figure ci-dessous extraite de [Harmon & al 90], elle est aussi particulièrement adaptée au

problème de planification/ordonnancement ; plus la tâche devient d'une complexité et d'une taille élevées, plus l'approche orientée objets est intéressante.

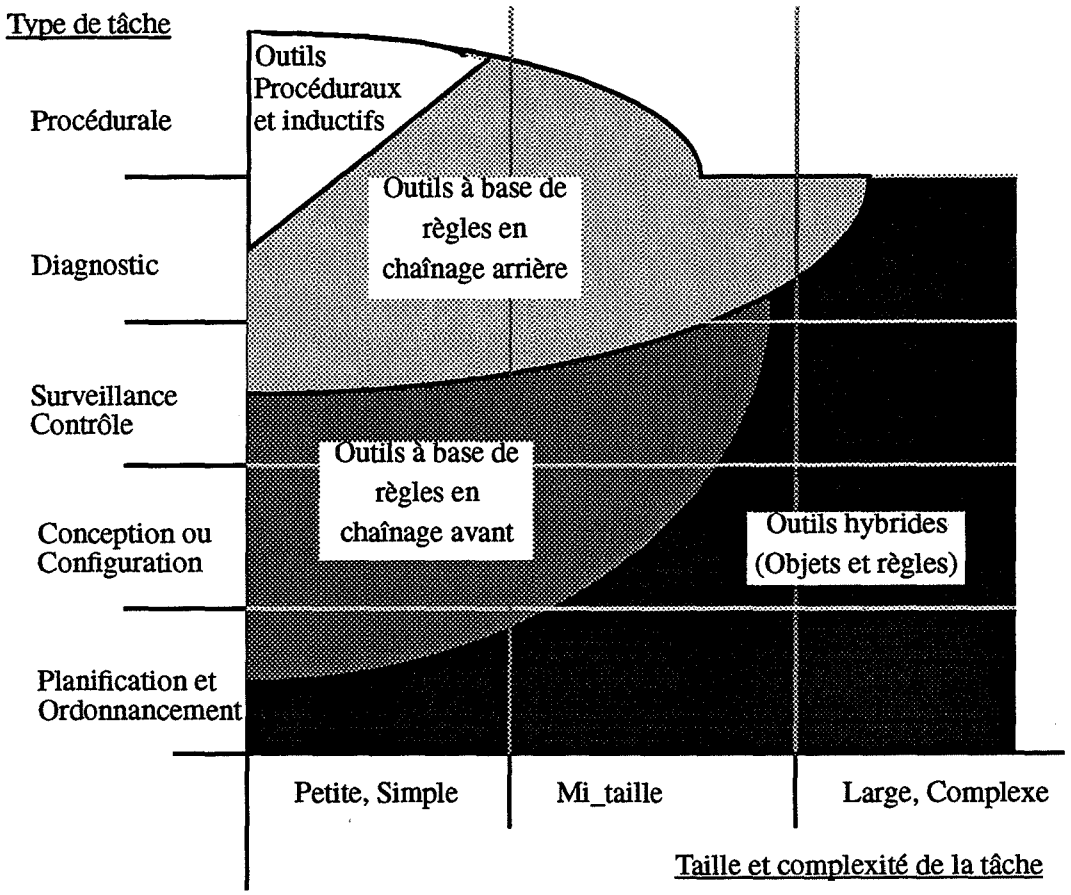


figure 7 : Technologies utilisées en fonction du type et de la complexité de la tâche à réaliser.

2.4.2 Les représentations hybrides.

Les capacités des modèles centrés objets sont très intéressantes pour l'implémentation des concepts d'un domaine particulier dans un système expert mais elles ne sont pas suffisantes. Des possibilités permettant d'écrire des règles doivent être aussi permises pour faciliter l'expression des assertions de type relationnel, des règles de décisions, etc. Plutôt que de rester prisonnier d'un formalisme unique, il est commode d'en faire coexister plusieurs dans un même système. Les systèmes à base de connaissances pouvant nécessiter plusieurs formalismes pour représenter les divers types de connaissances qu'ils doivent manipuler, il est parfois nécessaire d'avoir recours à un langage hybride, c'est à dire un langage intégrant plusieurs outils de programmation dans un même environnement : par exemple, procédures, classes, frames, règles de production, etc.

Deux écoles se sont principalement développées autour des deux langages favoris de l'intelligence artificielle, Lisp et Prolog, afin d'y intégrer les concepts du modèle centré objets. LOOPS [Bobrow & al 83] a été un précurseur dans ce domaine et constitue le chef de file de l'école Lisp. Pour Prolog, c'est surtout le projet japonais de machines de cinquième génération [Albert 85] qui est le moteur de cette école. BIM_probe [BIM 90], au dessus de Prolog_by_BIM, et OBJLOG [Dugerdil 88], à partir de Prolog II sont des exemples de langage implémentant une couche objet au dessus de Prolog.

Un langage hybride est d'abord un modèle centré objets, construit à partir d'un langage de programmation fonctionnelle (Lisp) ou logique (Prolog). S'il est fondé sur le modèle des classes, il est généralement enrichi de fonctionnalités empruntées aux frames, et vice-versa : par exemple, les variables d'instances des classes sont pourvues de facettes ou les frames sont dotés de comportements décrits par des méthodes. Un outil de programmation logique, moteur d'inférences de type Prolog ou règles de production, complète les possibilités du langage. Ainsi le programmeur a à sa disposition différents styles de programmation en n'utilisant qu'un seul langage, par exemple, modèle centré objet, programmation dirigée par les accès et programmation logique.

2.5 Intérêts et inconvénients de l'approche objet.

Le premier postulat sur lequel reposent ces techniques est que la meilleure façon de décrire les systèmes informatiques est de se fonder, non pas sur ce qu'ils font, mais sur les classes d'objets qu'ils manipulent. Une série de qualités que nous allons énumérer ci-dessous explique le succès encouru par ce style de programmation :

- *interactivité :* Le fait que ces langages sont interprétés, même si certains disposent d'un compilateur, provoque une haute *interactivité* : toute expression entrée est immédiatement exécutée pour produire un résultat aussitôt analysable et utilisable.
- *uniformité* des entités manipulées : tout est objet.
- *décentralisation* dans le sens où la connaissance est distribuée parmi les différents objets et non plus détenue dans une base de données globale.
- *modularité,* l'unité de base étant l'objet.
- *modifiabilité* apportée par la modularité : Même si un langage comme Pascal permet un bon découpage d'une application en procédures, le moindre changement de la structuration des données peut entraîner de profonds bouleversements dans l'organisation de ces procédures. La notion d'encapsulation pallie cet inconvénient. L'univers est structuré en objets dont il est possible de changer ou d'ajouter une définition avec un minimum d'interactions sur les autres.

- *facilité à tester, à manipuler.*
- *flexibilité* parce que c'est une méthode de programmation apportant modularité et modifiabilité.

- *généricité* apportée par l'abstraction de données. La *généricité* du modèle objet est sa faculté de pouvoir créer plusieurs entités physiques, les instances, à partir d'un même modèle abstrait, la classe. Un identificateur peut désigner des actions sémantiquement différentes définies par des champs ou des méthodes d'objets différents. Ainsi des objets différents sont susceptibles de répondre chacun à leur manière aux mêmes messages. Par exemple, il est agréable de désigner par un symbole unique, +, l'addition d'entiers, la concaténation de chaînes de caractères, l'union de listes, etc. Une méthode différente sera définie dans chaque classe (entier, chaînes de caractères, liste).
- *extensibilité* : Un programme écrit en utilisant l'approche objets est extensible dans le sens où il est possible d'ajouter de nouveaux objets mais aussi de nouvelles définitions d'attributs ou de méthodes.
- *réutilisabilité :* Il est facile de créer des bibliothèques d'objets que tout programmeur pourra réutiliser, soit pour construire des objets de même type, soit via le mécanisme de spécialisation pour créer de nouveaux objets plus spécifiques. Le souci de réutilisabilité est satisfait puisque tout est organisé, dans un langage orienté objets, autour d'une structure de données complète et non pas assuré par une fonction unique.
- *lisibilité / compréhensibilité* : L'encapsulation, la modularité, la décentralisation, l'uniformité et la généricité renforcent la lisibilité des programmes.

Représenter les connaissances sous forme d'objet apparait beaucoup plus naturel que toute autre méthode de représentation des connaissances. Ainsi s'explique l'engouement suscité actuellement pour la programmation orienté objet. Si les différents avantages de cette approche par les objets peuvent être facilement trouvés dans la littérature, il est bien difficile, voir impossible de trouver un auteur émettant des critiques ou citant des inconvénients à une telle méthode de programmation. Faut-il en déduire, pour autant, que le modèle centré objets est la méthode idéale que tout programmeur aura intérêt à utiliser ? A cette question, je donnerai une réponse mitigée, pensant que certains problèmes s'adaptent plus à une approche procédurale. En effet, lorsque le nombre d'objet dans un domaine étudié n'est pas trop important, l'approche procédurale est peut-être plus adaptée surtout s'il est possible d'écrire des algorithmes "facilement". Cela dit, l'approche orientée objets présente d'autres problèmes :

- Le premier auquel sera confronté un concepteur désirant employer l'approche orientée objets, est celui de la détermination des objets utiles pour résoudre son problème. Or aucune méthode bien établie n'existe pour l'aider dans cette tâche si ce n'est celle consistant à écrire ce qu'il veut réaliser et à en extraire les noms pour trouver les classes et les verbes pour les méthodes. Cette technique est assez simpliste et ne peut aboutir qu'à des résultats grossiers comme le signale Meyer [Meyer 88] ; en particulier, elle engendre de très nombreuses classes. D'autre part, Meyer fait la remarque suivante : "la création de classes inutiles est une erreur fréquente, mais il n'existe pas de méthode pour éviter cette erreur". Il conseille d'obtenir les classes fondamentales d'une application directement à partir du monde réel que l'on tente de modéliser. Aux classes des objets physiques correspondent ainsi, les classes logicielles. Ces classes étant définies, elles doivent

être organisées sous forme de hiérarchie en utilisant correctement les mécanisme d'héritage. Un fois que le programmeur les aura définies, il devra les décrire, établir les relations entre ces objets et développer des programmes pour les utiliser. Là encore aucune méthode ne pourra le seconder. Ayant enfin réussi à décrire son problème sous forme d'objets, il va devoir s'assurer que l'objectif désiré par son programme est bien atteint et donc le tester. Et là encore (troisième fois) comment va-t-il faire ? Ce ne sont sûrement pas les méthodes inexistantes qui vont venir à son secours. Ainsi, même si un concepteur potentiel dispose d'un nombre invraisemblable de langages dits orientés objets (ce qui peut, d'ailleurs, être aussi un problème quand on se voit confronter au choix de l'un d'eux), aucune méthodologie ne pourra venir à sa rescousse pour l'aider dans sa tâche.

- D'autre part, la représentation objets n'est pas homogène et déclarative du fait de l'insertion fréquente dans les objets d'appels de programmes.
- De plus, elle ne présente pas de mécanisme de filtrage qui pourrait permettre [Vignard 86], par exemple : d'offrir des techniques d'inférences, de manipuler des notions imprécises et incomplètes, d'avoir des notions d'importance entre les éléments, de manipuler des objets sous différents aspects, d'utiliser la sémantique des objets, c'est à dire leur contenu plutôt que de se satisfaire de leur nom, d'avoir un raisonnement nuancé, etc.
- Les langages orientés objets sont aussi accusé d'entraîner une surcharge du système non négligeable principalement à cause du mécanisme de communication par messages.

Alors le modèle orienté objets est-il une panacée pour résoudre tous problèmes informatiquement? Certes, il présente de nombreux avantages très intéressants du point de vue qualité d'un logiciel mais il requiert du concepteur une certaine expérience dans ce type de programmation pour combler le manque en méthodes d'implémentation de l'approche orientée objet.

CHAPITRE 3

Quelques méthodes de représentation des connaissances pour des systèmes de conception.

Concevoir est une des activités de la résolution de problème proposant l'un des plus grands défis aux ingénieurs. Cette tâche nécessite à la fois une grande quantité de connaissances spécifiques du domaine d'application et d'importantes aptitudes à résoudre les problèmes. La conception peut être définie comme étant une activité créatrice englobant diverses techniques de résolution de problèmes et nécessitant des facultés de jugement pour faire des choix parmi ces techniques. C'est par conséquent une activité complexe et malheureusement l'intelligence artificielle a relativement peu de théories permettant d'implémenter un système à base connaissances résolvant des problèmes de conception. Plusieurs systèmes à base de connaissances pour la conception ont tout de même été construits mais chacun d'eux est souvent resté très spécifique à son domaine d'application et aucune théorie générale n'en a découlé. Parmi ces systèmes (souvent restés à l'état de prototype) peuvent être cités tout d'abord R1, un système commercialisé à base de règles pour la configuration des ordinateurs VAX [McDermott 82], PRIDE (Pinch Roll transport Interactive Design environment/Expert) [Mittal & al 86] qui se concentre sur un type particulier de système de transport de papier, celui qui utilise les rouleaux à pincement pour déplacer le papier, VEXED [Mitchell & al 85] et son expansion VEXED+ [Norton & al 88], destinés à la conception de circuits VLSI, etc.

Avant de tenter d'établir un aperçu des diverses techniques employées dans différents systèmes dont l'objectif est la conception, un premier paragraphe introduira ce qu'est l'activité de conception afin de pouvoir mieux cerner comment construire un tel système.

3.1 L'activité de conception.

Comprendre le processus de conception a longtemps été un objectif pour les ingénieurs, les architectes et bien d'autres. Une telle compréhension peut conduire à une conception meilleure, à une exécution plus rapide et à de grands progrès dans le sens d'une confrontation à des besoins réels. Mais surtout pour celui qui veut tenter d'informatiser ce processus, sa compréhension est bien évidemment nécessaire ; c'est pourquoi avant d'aborder les différentes techniques utilisées pour résoudre ce problème de conception, nous allons étudier le mécanisme de conception des experts humains.

3.1.1 La conception et le diagnostic.

Ces deux types de problèmes (la conception et le diagnostic) sont parfois complémentaires du fait que le processus de conception comprend généralement des phases de diagnostic des solutions obtenues.

L'objectif dominant des systèmes experts a longtemps été celui du diagnostic, correspondant à la pondération et la classification de modèles pour évaluer une situation qui est soit anormale (comme dans le cas des maladies ou des fautes) ou améliorable (comme pour la prospection minière). Ainsi un système capable de diagnostiquer un système conçu opère en évaluant, critiquant, recommandant des corrections et/ou proposant une nouvelle solution. L'objectif de la conception est totalement différent. En effet, son but est de développer un système ou un objet répondant à certaines spécifications données initialement. Mais lors du déroulement du processus de conception, des tests sont nécessaires afin de diagnostiquer l'état courant de la ou des solutions dans le but de s'assurer que les contraintes et les spécifications du système à concevoir sont vérifiées. De ce fait, le processus de conception a affaire aux mêmes contraintes que le diagnostic dans le sens où les tests nécessaires peuvent être coûteux, imprécis et difficiles à sélectionner.

Un troisième type de processus est celui apportant une assistance interactive. Typiquement ce mécanisme est la combinaison des deux autres types de processus : le diagnostic et la conception.

3.1.2 Les caractéristiques de la conception.

L'étude des caractéristiques propres à la conception va aider à mieux cerner le problème auquel un informaticien est confronté lorsque son objectif est de vouloir simuler informatiquement une telle activité. Par la suite, elle permettra d'établir plus facilement le processus de conception. Parmi ces caractéristiques, certaines ont un caractère cognitif d'autres expérimental.

Connaissances cognitives :

- Premièrement, la conception est un problème extrêmement mal structuré [Simon 73]. Le but d'un système à concevoir est généralement mal défini. La tâche de conception peut être vue comme le développement d'une représentation de plus en plus détaillée du but à atteindre. Ceci peut être fait par des modifications successives de la représentation de l'objet à concevoir, sans avoir recours à une méthode définie assurant ces changements. Tout de même, les composants à partir desquels la construction d'un système peut être faite doivent être connus ainsi que leurs propriétés et leurs inter-relations. La sélection et la connexion de ceux-ci sont des opérations importantes en conception comme le sont aussi d'ailleurs la déduction et le test des propriétés. Dès qu'une proposition est énoncée, elle doit être testée pour s'assurer de sa consistance avec les spécifications du système à construire.
- La conception requiert une énorme quantité de connaissances (connaissances du domaine mais aussi capacités de jugement, expérience, tests d'acceptabilité des solutions, ...) et des capacités à raisonner logiquement et intuitivement à partir de ces connaissances pour aboutir à des conséquences pratiques.

- Même si le processus de conception est mal structuré, un système reflète la structure ou la fonctionnalité de ce qui est en train d'être conçu. La représentation d'un système pourra aussi être structurée par la donnée de différents objets le composant ainsi que par les propriétés de ces objets et leurs inter-relations.
- Pour réduire l'espace théoriquement large d'un problème à une taille maniable, le concepteur décompose généralement le problème en sous-problèmes d'une complexité moindre, ne résolvant jamais un problème dans sa totalité en un seul but.
- Pendant la phase de conception, des restrictions variées, des contraintes techniques, doivent être vérifiées à des points appropriés du processus de conception.

Informations expérimentales :

- Comme l'activité de conception est un problème ouvert, c'est à dire que de nombreuses solutions sont possibles et satisfont les contraintes spécifiées, le choix du système le meilleur devient réellement un défi. Il n'y a pas une solution unique à un problème de conception comme le prouve le fait suivant : selon le concepteur, différents résultats à un même problème peuvent être trouvés, chacun d'eux étant aussi correct qu'un autre. Ainsi habituellement plusieurs solutions peuvent satisfaire de manière acceptable les prérequis et les contraintes, chacune d'elles présentant des avantages et des inconvénients. Ce concept d'acceptabilité est particulièrement intéressant et crucial mais n'est pas facile à déterminer. Les facteurs d'acceptation d'une solution considérés par les concepteurs sont nombreux et variés, incluant des critères techniques, économiques et personnels et ont des interactions complexes. De plus, parmi plusieurs solutions acceptables, des critères de choix permettant de déterminer une seule solution doivent être définis. Le critère usuel pour sélectionner le "meilleur" système est de minimiser le poids ou le coût de la structure. Les connaissances d'un concepteur expérimenté ne sont généralement pas suffisantes pour produire la structure minimum au sens poids/coût, surtout quand la structure est large et a beaucoup de composantes. Ainsi, apparait un besoin d'introduire des optimisations mathématiques dans le processus de conception.
- Le processus de conception demande de prendre beaucoup de décisions fondées sur des règles apprises sur le tas, des heuristiques et des expériences antérieures. Intuition, jugement et expérience doivent être utilisés pour sélectionner les valeurs exactes des paramètres de la conception. La conception est réellement une activité humaine ; le rôle personnel du concepteur est primordial.
- Des plans permettant la réalisation d'une partie d'un système sont souvent utilisés dans des situations particulières. De tels plans résultent d'une utilisation répétée de planifications et de validation de ces planifications. De plus, les concepteurs ont tendance à sélectionner des ensembles de solutions déjà connues et bien maîtrisées. Ainsi une part significative de l'activité de conception est composée de tâches de routine.
- Différents états de la solution en évolution sont atteints avant d'avoir un ultime système. L'activité de conception nécessite souvent une ou plusieurs phases de brouillon suivies du système proprement dit.

3.1.3 Les connaissances nécessaires pour la conception.

Quelle que soit la tâche à réaliser, une approche basée sur la connaissance nécessite la réponse à la question : Quelle connaissance (faits ou capacités de raisonnement) est utilisée par les experts humains pour résoudre cette tâche ? Ce savoir est dépendant du domaine d'application et de la tâche à réaliser mais il est tout de même possible de distinguer diverses catégories parmi les connaissances nécessaires pour tout type de problème de conception :

- *la modélisation des objets du domaine :*
  - connaissances syntaxiques : représentation des relations syntaxiques entre symboles (objets),
  - connaissances sémantiques : concernant la signification des objets et leur relations avec les autres objets,
- *les règles-tactiques de conception :* Certaines tâches sont répétitives et/ou bien connues et ont des méthodes de résolution éprouvées qui peuvent être traduites sous forme de règles.
- *les règles de contrôle :* permettent d'ordonnancer le processus de conception, de déterminer quelle action réaliser, de tester l'acceptabilité d'une solution ou d'un éléments de solution.

3.1.4 Le processus de conception.

Les décisions prises par les concepteurs sont de trois types :

- *des décisions concernant le processus de conception :* ce sont des décisions de planification permettant de choisir quoi faire postérieurement, comment réaliser une tâche sélectionnée, quand faire des tests ou non, quelle(s) ressource(s) activer, etc.
- *des décisions techniques* pour satisfaire les besoins spécifiques et techniques tels que la sélection d'un équipement, la détermination d'un processus de fabrication, la définition d'une taille ou d'une forme, etc.
- *des décisions d'acceptation* pour déterminer si une solution candidate ou une partie du système global satisfait les prérequis et les contraintes imposés initialement.

Le processus de conception peut être modélisé par une interaction récursive de ces trois types de décision, comme représenté sur la figure 8 par le cycle de base du processus de conception. Dans cette figure, les ressources nécessaires à chacune des étapes de ce cycle sont également introduites. Parmi ces ressources, une distinction est faite entre les connaissances opératoires (boîtes en gris) et celles spécifiques d'un problème particulier.

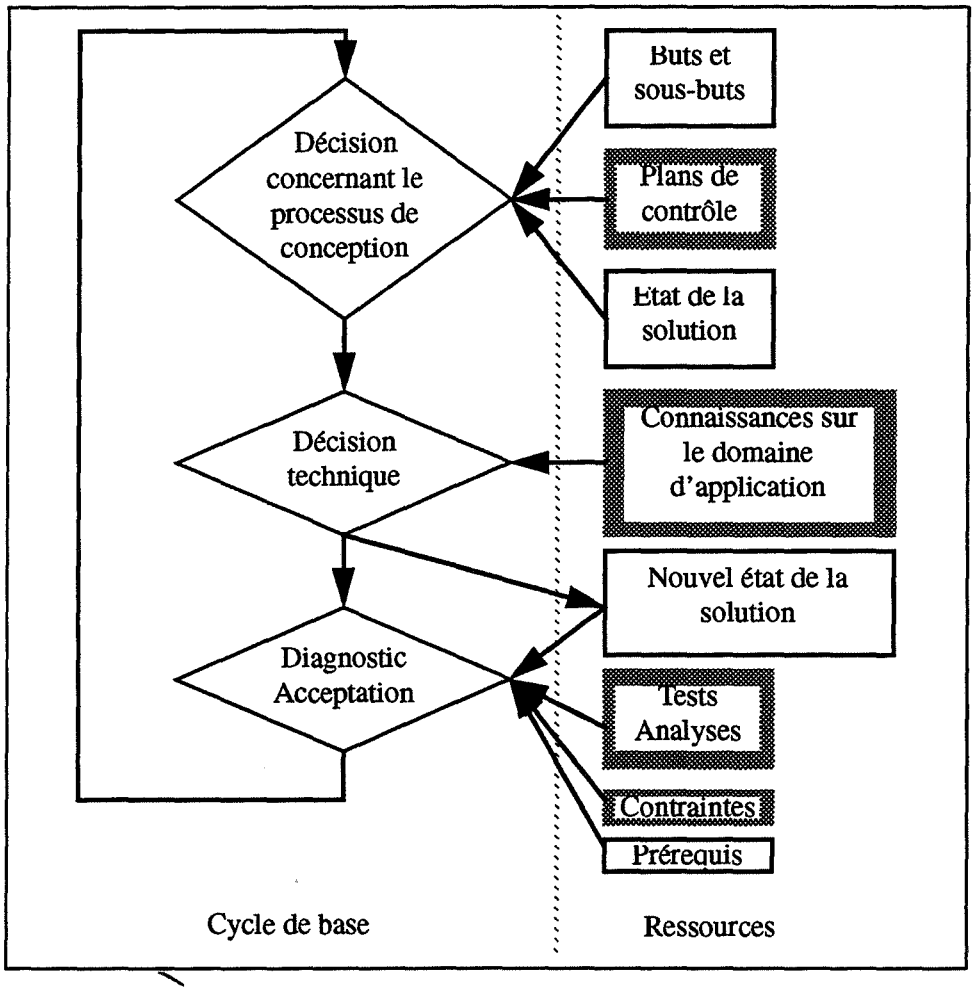


figure 8 : Schématisation du processus de conception.

Pendant le développement d'un système, l'acheminement vers une solution finale par le processus de conception peut être vu comme la génération de nouveaux états de la solution à partir des états courants. Ainsi, un système peut être interprété dans une représentation des états de la solution comme un état initial qui est transformé par l'intermédiaire de connaissances expertes en une série d'états intermédiaires jusqu'à l'obtention d'une ou plusieurs solutions finales. La figure 9 montre cette transformation de la ou des solution(s). Les transitions entre chaque état sont des processus utilisés pour passer d'un état à un autre, conduisant ainsi à considérer de nouveaux candidats comme solutions potentielles du problème de conception. A chaque création d'un nouvel état, celui-ci doit être vérifié pour identifier s'il satisfait toutes les contraintes, sinon il est annulé ou certaines contraintes sont relâchées. A la fin du processus de conception plusieurs solutions acceptables peuvent être trouvées.

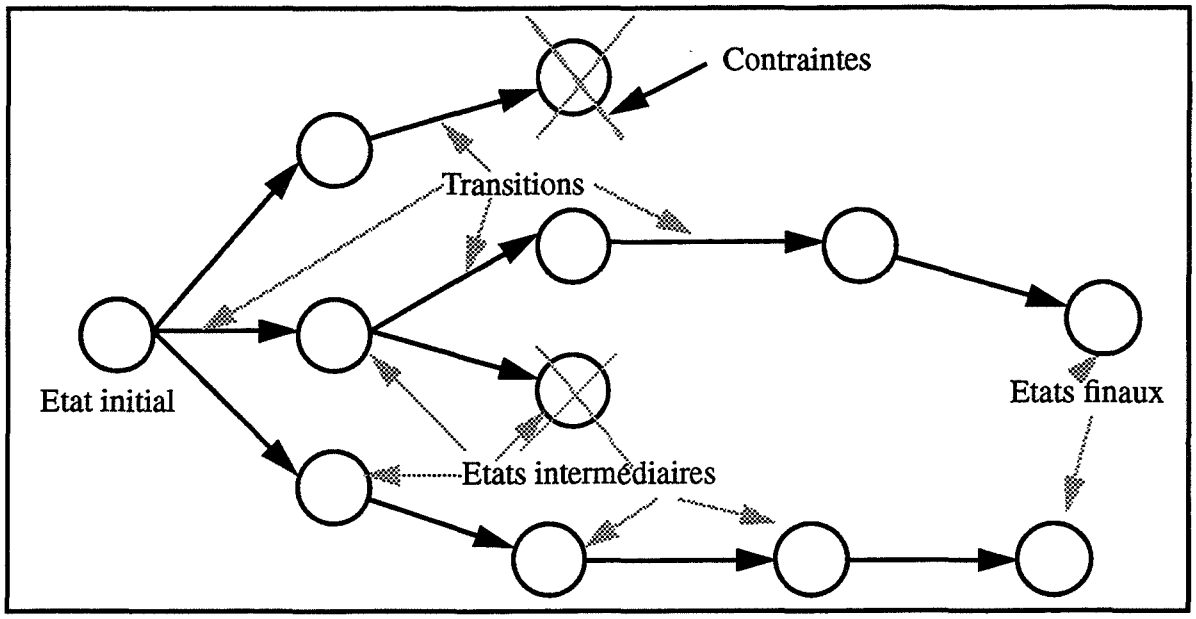


figure 9 : Les différents états de la solution d'un problème de conception.

Ainsi différents états de la solution, de plus en plus détaillés, sont atteints avant d'arriver à une solution finale, en passant par :

- un système préliminaire, après avoir fait la synthèse des besoins fonctionnels,
- plusieurs systèmes de plus en plus détaillés,
- l'analyse et l'optimisation, la vérification et l'évaluation de tous les aspects du système afin de tester si on a affaire à une solution finale ou s'il faut l'améliorer.

3.1.5 Les composantes d'un système à concevoir.

Les principales composantes d'un système ont été résumées dans [Rychener 88] et sont présentées ci-dessous.

- *Données :*
  - les spécifications de l'objet ou du système désiré en donnant ses caractéristiques, ses fonctions, ses contraintes, le budget, etc,
  - les composants possibles du système ou de l'objet désiré, leurs propriétés, leurs inter-relations,
  - les tests standards d'analyse des (sous-)systèmes et composants qui sont proposés ou conçus,
- *Objectif :*
  - produire un objet ou un système qui satisfait les spécifications données,
- *Contraintes :*

- les tests peuvent être nombreux et difficiles à sélectionner,
- les tests peuvent être coûteux (en temps et en argent),
- les test peuvent être peu fiables ou imprécis,

• *Opérations :*

- faire la synthèse des données initiales,
- sélectionner et spécifier les composants du système à réaliser et ses particularités,
- inférer les propriétés du système désiré à partir des spécifications données et du résultat de l'opération précédente,
- mettre en relation les composants dans des (sous-) systèmes,
- vérifier les spécifications (caractéristiques, contraintes, coût, ...),
- établir puis réaliser des tests analytiques pour prédire le comportement du processus de résolution,
- faire évoluer et mettre à jour le système dessiné en utilisant des retours-arrière à partir des tests, en enregistrant les raisons pour lesquelles les décisions ont été prises et les inter-dépendances entre les particularités du système et en s'assurant de la satisfaction des contraintes,
- créer, représenter et utiliser de nouveaux composants et de nouvelles contraintes,
- observer les procédures efficaces connues pour la réalisation de sous-tâches (tâches de routine, actions déjà planifiées par les experts humains, etc),
- appliquer des procédures d'optimisation,
- considérer les critères non économiques tels que la sûreté, la protection de l'environnement, l'esthétique.

Plusieurs méthodes ont été employées, plus ou moins proches les unes des autres, pour la réalisation de systèmes à base de connaissances dont l'objectif est la conception d'un système. Le paragraphe suivant va les présenter.

3.2 Les méthodes.

La conception peut être interprétée, dans une première approche, comme étant l'application de méthodes connues de conception permettant de réaliser un système préliminaire suivi de son évaluation afin de déterminer s'il satisfait à toutes les contraintes; tant que l'une d'elles n'est pas vérifiée, ce processus est itéré. Dixon et Simmons ont implanté une telle stratégie [Simmons & al 84]. Leur approche consiste essentiellement en l'utilisation du modèle : engendrer puis tester, dans lequel les contraintes sont principalement employées pour tester les solutions développées. Or ce mécanisme se révèle d'une très grande inefficacité car il nécessite souvent de engendrer un grand nombre de solutions (si ce n'est toutes) envisageables avant d'en obtenir une correcte. Un autre système HI-RISE [Maher & al 85] pour la réalisation de systèmes structurels de hauts bâtiments utilise aussi cette méthode de génération puis

test. D'autres systèmes ont également été conçus qui apportent une meilleure représentation des connaissances par le biais de hiérarchisations et de planifications. Comme les contraintes à vérifier sont nombreuses en conception, il est important de bien les maîtriser et de les employer de manière efficace. D'autre part, l'architecture tableau noir semble également très bien convenir à de tels problèmes. Ces différentes approches sont discutées dans les paragraphes qui suivent.

3.2.1 Hiérarchisation et planification.

La hiérarchisation de l'activité de conception n'est pas faite parce qu'un système est intrinsèquement hiérarchique mais parce que les humains l'utilisent pour diminuer la complexité [Brown & al 86]. En effet, une manière de résoudre un système complexe de l'être humain est de le diviser en sous-problèmes qui peuvent être étudiés séparément. Ainsi la complexité du problème global est-elle réduite à celle des sous-problèmes. De même, plusieurs systèmes à base de connaissances ont appliqué ce principe. Par exemple, VEXED, pour la conception de circuits VLSI [Mitchell & al 85], redéfinit le modèle du circuit complet en sous-modèles représentant des blocs du circuit pour les principales fonctions de celui-ci. Chaque sous-modèle peut à son tour être redéfini en sous-sous-modèles jusqu'à trouver une méthode simple qui permette de le réaliser, c'est à dire principalement de déterminer ses paramètres. Ainsi une hiérarchie est-elle développée dont les niveaux les plus hauts traitent des aspects généraux alors que les niveaux inférieurs traitent d'aspects plus spécifiques.

Brown et Chandrasekaran [Brown & al 86], tout en mettant en oeuvre la caractéristique hiérarchique de l'activité de conception, apportent une amélioration en décomposant la connaissance en différents agents. En effet, leur objectif était de développer un langage générique et une architecture capables de supporter l'activité de résolution de système dans le cas d'un domaine d'application ayant des routines fixées, c'est à dire que le concepteur fait sa sélection parmi une série d'alternatives bien connues. Un langage DSPL (Design Specialists and Plans Language) en découle et un système AIR-CYL a été construit pour tester la méthode. Mais revenons à leur méthode. Elle consiste en une communauté d'agents hiérarchiquement organisés. Plusieurs types d'agents existent dans leur structure. Tout d'abord des **spécialistes** ont chacun la charge d'une certaine tâche. Par exemple, cette tâche peut être de dessiner une partie majeure du système à concevoir. Pour réaliser sa tâche, un spécialiste utilise un répertoire de plans. Les actions des spécialistes sont ainsi de choisir un de ces plans et de diriger les spécialistes à des niveaux d'abstraction inférieurs. Un échec peut être source de différentes actions : choisir un autre plan, transférer le contrôle à un autre spécialiste parent, ... Le deuxième type d'agent, que nous avons d'ailleurs entrevu, est le **plan**. Un plan représente une méthode pour concevoir une partie d'un système. Il spécifie l'ordre d'invocation des divers agents utilisés dans sa méthode. Les éléments d'un plan sont les **étapes** à suivre (troisième type d'agent), l'ordonnancement entre celles-ci, comment conduire chaque étape, comment détecter les erreurs et des suggestions sur la manière de les supprimer. A chaque étape, une décision est prise, principalement sur la valeur des paramètres concernés. Les étapes sont regroupées en **tâches** (quatrièmes type d'agent). Celles-ci ont le rôle de concevoir une section du système fonctionnellement, structurellement, logiquement cohérente. Enfin le dernier type d'agent est celui de **contrainte** qui est une inter-dépendance entre les différents paramètres du système.

Le système PRIDE (Pinch Roll transport Interactive Design environment/Expert) [Mittal & al 86], emploie une méthode très proche de celle de Brown et Chandrasekaran. Ce système se concentre sur un type particulier de transport de papier, celui qui utilise les rouleaux à pincement pour déplacer le pa-

pier. Dans ce système, la connaissance est également décomposée en différents modules assez proches de ceux de Brown et Chandrasekaran bien qu'exprimés en des termes différents. Ainsi le processus de conception est vu comme une décomposition en sous-buts d'un **but** initial plutôt que des spécialistes appelant des spécialistes plus spécifiques et à des niveaux inférieurs. Le premier but émis est par conséquent le sommet d'une hiérarchie de sous-buts qui vont permettre de réaliser ce but. Chaque but spécifie explicitement les paramètres du système dont il est responsable et ceux dont il dépend, définissant ainsi un espace d'un système partiel. Un but peut être réalisé aussi par l'activation d'un **plan**, méthode de contrôle spécifiant la série de sous-buts à réaliser pour exécuter ce but et déterminant l'ensemble des paramètres associés au but concerné. Pour un même but, différents plans peuvent être activés comme pour les spécialistes de Brown et Chandrasekaran. Un but peut également être réalisé par l'activation d'une **méthode** qui va affecter les valeurs des paramètres de ce but et déterminer le comportement de certains composants du système. Le rôle des méthodes est donc d'engendrer des systèmes partiels de la solution totale. Ainsi un but peut être réalisé soit en le décomposant et en résolvant ses sous-buts, soit en appliquant une méthode. Enfin des **contraintes** sur les paramètres du système sont attachées aux buts. La majorité de l'ordonnancement des tâches provient des contraintes et particulièrement de là où elles sont attachées.

Mittal et ses collègues apportent une amélioration à la méthode employée pour PRIDE. En effet, ils implémentent un meilleur processus de résolution que le modèle "engendrer puis tester" en introduisant des modules appelés conseils. Ceux-ci sont des processus qu'il est conseillé d'exécuter dans le cas d'un échec d'une contrainte. Les conseils sont attachés aux contraintes et sont activés quand une contrainte échoue. Deux types de conseil existent. Le premier analyse l'expression de la contrainte ayant conduit à un échec et détermine quels paramètres peuvent être modifiés et comment satisfaire la contrainte. Le deuxième type de conseils est utilisé dans le cas où la connaissance sur les paramètres qui sont le plus facilement modifiables est connue. Le système peut alors déterminer comment changer ces paramètres pour que la contrainte soit satisfaite.

Ce type de méthode, fondé sur la décomposition d'un problème en une hiérarchie de sous-problèmes, nécessite que le problème de conception à résoudre soit bien structuré, ce qui, a priori, est loin d'être toujours le cas [Simon 73]. En effet la conception relève plutôt, en grande partie, de connaissances incertaines, incomplètes et de problèmes mal formalisés. De plus le mécanisme employé réclame la possibilité de bien structurer la connaissance sous forme de différents plans et de méthodes. Par contre un point positif de cette architecture est le relatif opportunisme permis par le processus de résolution. En effet celui-ci n'est pas réalisé avec un ordre strict et fixé par programmation des séquences successives mais par un choix fait pour chaque sous-problèmes parmi un ensemble de méthodes ou de plans alternatifs.

3.2.2 Vers une meilleure utilisation des contraintes.

Le problème de la conception, comme nous l'avons déjà vu, est généralement décomposable en sous-problèmes relativement indépendants même si cette décomposition n'est pas toujours évidente. C'est cette dépendance entre les sous-problèmes qui est la clé de l'approche par postage de contraintes proposée ci-après. Quand les sous-problèmes ne réagissent pas réciproquement, ils peuvent être résolus indépendamment. Or l'expérience acquise en résolution de problèmes ces dernières années a montré que cette situation idéale est inhabituelle lors de la résolution de problèmes dans le monde réel. Au contraire, une interaction entre les sous-problèmes apparait même dans les cas les plus simples. L'ac-

ceptation de ce fait par les chercheurs a provoqué de leur part un intérêt pour se concentrer sur la nature de ces inter-actions et sur la façon dont elles peuvent être prises en compte dans la résolution des problèmes de conception. Ce paragraphe va principalement s'appuyer sur les travaux conduits par Stefick sur la planification avec contraintes [Stefick 81]. Un système appelé MOLGEN illustre sa théorie dans le domaine de la génétique moléculaire. L'objectif de MOLGEN est d'engendrer un plan de manipulations génétiques en vue de construire une entité biologique donnée.

Une solution pour gérer les interactions entre les sous-problèmes est de les minimiser en trouvant une décomposition du problème global en sous-problèmes interagissant faiblement entre eux mais ce n'est pas toujours possible. Une autre approche est d'utiliser ces contraintes comme une aide pour la résolution. Dans le paragraphe précédent, les systèmes présentés utilisaient les contraintes pour tester si les solutions trouvées (partielles ou finales) satisfaisaient aux contraintes. Une meilleure approche est d'utiliser les contraintes pour guider l'élaboration des solutions.

Ainsi Stefick considère-t-il également l'activité de conception comme divisible en sous-problèmes mais, pour lui, la clé principale de la conception est le fait que ces sous-problèmes inter-agissent et que ces inter-actions doivent être utilisées pour guider le processus de résolution. Ces interactions sont modélisées par des contraintes dont l'utilisation peut être :

- *élagage de possibilités:* élimination des cas ne vérifiant pas une contrainte.
- *description partielle et soumissions :* l'objectif est de différer les décisions aussi longtemps que possible, en soumettant au système de nouvelles contraintes.
- *moyen de communication :* moyen d'expression des interactions entre sous-problèmes.

L'approche de Stefick nommée "postage de contraintes" est principalement un mariage des idées issues de la planification hiérarchique et de la satisfaction de contraintes. Différents plans seront donc activés pendant le processus de résolution et les contraintes permettent d'exprimer les relations entre les différentes variables des plans.

Plusieurs opérations peuvent être réalisées sur les contraintes :

- *la formulation de contraintes :* soumission de nouvelles contraintes. Un planificateur peut procéder hiérarchiquement en formulant des contraintes dont le niveau de détails s'accroit lorsque la conception progresse.
- *la propagation de contraintes :* création de nouvelles contraintes à partir d'anciennes dans un plan. Cette opération apporte un moyen de communication entre les sous-problèmes à différents niveaux de détails. Ainsi sont propagées à travers la hiérarchie les exigences d'autres parties du système. L'objectif est de différer les décisions aussi longtemps que possible en créant de nouvelles contraintes.
- *la satisfaction des contraintes :* consiste à trouver les valeurs des variables tout en vérifiant les contraintes dont elles dépendent.

Une approche pourrait être de sélectionner dès que possible les valeurs des objets afin de satisfaire une contrainte. Hors, ces variables peuvent intervenir dans d'autres contraintes rencontrées plus tard dans le processus de conception et l'on court le risque que ces nouvelles contraintes ne soient pas vérifiées quand elles seront rencontrées plus tard dans le processus de résolution. Comme beaucoup de combinaisons sont possibles quant aux valeurs des variables, MOLGEN garde ces options ouvertes. C'est à dire que plutôt que d'affecter dès que possible une valeur à ces variables, il formule une contrainte qui sera prise en compte ultérieurement lors d'un processus de satisfaction de contraintes. En

plaçant une contrainte, MOLGEN rend explicite une exigence sur des variables et peut ainsi assurer la combinaison de contraintes lors du choix d'une valeur pour une variable. En effet, cette sélection est faite parmi l'intersection des ensembles des solutions satisfaisant chaque contrainte. Les décisions sont donc différées aussi longtemps que possible, jusqu'à ce qu'il soit vraiment nécessaire d'en prendre une.

Nous venons de décrire l'idée générale de l'approche par postage de contraintes. Nous allons maintenant voir comment celle-ci a été implémentée dans MOLGEN. En effet, le processus de résolution de ce système est intéressant du point de vue de l'emploi des contraintes, mais aussi de celui de la planification des tâches à réaliser. MOLGEN travaille en comparant des buts, en trouvant des différences et en choisissant des opérateurs pour réduire ces différences. Comme d'autres programmes utilisant la planification, MOLGEN planifie sa résolution hiérarchiquement en débutant par un plan abstrait qu'il raffine par des plans de plus en plus spécifiques. MOLGEN étend les travaux sur la planification hiérarchique en incluant une structure de contrôle divisée en plusieurs niveaux permettant la méta-planification. L'organisation du résolveur de problèmes est ainsi partitionnée en trois niveaux distincts (appelés aussi espaces). Le dernier niveau représente les connaissances spécifiques au domaine d'application et permet donc de résoudre des sous-problèmes à un niveau de détail élevé. Les plans sont construits à ce niveau. L'espace de dessin, au-dessus du précédent, fournit un répertoire explicite d' opérateurs pour planifier hiérarchiquement. L'idée organisationnelle qui sous-tend cet espace est que la planification hiérarchique peut être interprétée comme l'action d'opérations sur des contraintes. A ce niveau, les décisions sont prises quant aux choix des plans. Enfin, l'espace le plus élevé dans cette hiérarchie est la stratégie qui dicte les décisions de conception en créant et en exécutant des tâches dans l'espace de dessin. En particulier, la décision sur le style de planification est prise à ce niveau. Il peut être de deux types : **le moindre engagement** ou **le raisonnement par heuristiques**. Durant le cycle du moindre engagement, MOLGEN envoie un message à l'espace de dessin pour lui demander de suggérer une tâche à exécuter. Cette tâche comprend (1) la proposition d'un but après avoir noté les différences entre l'état courant de la solution et le but courant, (2) le raffinement d'un objet ou d'un opérateur et (3) la formulation d'une nouvelle contrainte. MOLGEN peut échouer dans la recherche d'une tâche. Dans ce cas, il suspend cette étape et se concentre sur une autre. Ce cycle est appelé le moindre engagement car MOLGEN se compromet rarement dans une étape d'un plan qui devra être abandonnée plus tard lors du développement de ce plan. Si MOLGEN ne peut pas résoudre une étape en satisfaisant les exigences de ce cycle, il passe au mode de raisonnement par heuristiques.

Le processus proposé par Stefick n'est autre que celui de "engendrer puis tester", mais dont l'efficacité est améliorée par la recherche de solutions en appliquant dès que possible des contraintes. De cette façon, les branches correspondant à des plans possibles mais ne vérifiant pas les contraintes sont éliminées le plus tôt possible, n'étendant pas ainsi l'effort de calcul à des raisonnements inutiles. De plus, quand les contraintes peuvent être propagées à travers des solutions partielles à un sous-problème, elles permettent au planificateur d'anticiper les interactions et effectivement d'élaguer les choix possibles sans les expliciter.

La puissance de l'approche par postage de contraintes provient principalement de deux propriétés. La première est son habileté à planifier hiérarchiquement tout en introduisant de nouvelles contraintes et de nouvelles variables. La seconde est la capacité à anticiper les interférences entre les sous-problèmes en utilisant la propagation des contraintes et en éliminant les solutions inutiles le plus tôt possible.

Malheureusement, MOLGEN n'est pas capable de reconnaître les équivalences logiques des contraintes. Alors que MOLGEN est apte à propager les contraintes qu'il crée, il n'a pas la capacité de

propager des variations logiquement équivalentes à ces contraintes. En effet, les opérateurs dédiés à la propagation de contraintes sont strictement des équivalences syntaxiques des contraintes. Une deuxième limitation de MOLGEN est qu'il n'utilise pas les contraintes pour décrire les différents processus utilisés lors de la résolution. Le contraintes ne portent que sur la spécification d'objets du système. Enfin MOLGEN n'utilise pas de méta-contraintes qui pourraient porter sur les plans ou sur le processus de planification.

3.2.3 L'architecture tableau noir.

Avant de considérer l'utilisation de cette approche pour résoudre des problèmes de conception, un premier paragraphe présente l'architecture tableau noire.

3.2.3.1 Présentation.

Historiquement, le premier système implémentant une architecture tableau noir a été HEARSAY-II, un système pour la compréhension du langage parlé qui a été développé entre 1971 et 1976 [Errman & al 80]. D'autres tels systèmes bien connus sont aussi HASP/SIAP [Nii & al 82], un interprète de signaux sonores continus captés par les sonars, dérivé de HEARSAY-II, CRYSALIS [Terry 83], un identificateur de la structure des protéines, TRICERO [Williams & al 84], un moniteur pour le contrôle de l'espace aérien, OPM [Hayes-Roth 85], pour la planification ... De plus, le besoin d'outils basés sur cette architecture a provoqué la naissance de systèmes génériques tels que HEARSAY-III [Erman & al 81] généralisation de HEARSAY-II, AGE [Aiello 83] inspiré de HASP/SIAP et de CRYSALIS, BB-1 obtenu à partir du mécanisme de contrôle de OPM, BLOBS [Middleton & al 85] un système générique qui intégre à la fois des notions des modèles centrés objets et un raisonnement temporel et enfin ATOME [Laasri & al 88] issu des travaux réalisés sur CRYSALIS, HASP/SIAP, AGE et BB-1.

Plus qu'un formalisme de représentation des connaissances, l'approche tableau noir définit une architecture spécifique, destinée à des systèmes d'intelligence artificielle. Néanmoins, ce chapitre s'insère bien dans cette partie : cette architecture assure le stockage des connaissances dans divers modules appelés les sources de connaissances et l'utilisation de cette connaissance par l'intermédiaire de l'interaction des sources de connaissance sur un espace de travail, le tableau noir, est sous le contrôle d'un module spécialisé dans cette tâche. Or la fonction d'une représentation des connaissances n'est autre que de permettre la mémorisation et l'utilisation des connaissances. Enfin, cette approche semble très prometteuse pour la résolution de problèmes et plus particulièrement pour ceux de conception.

La base de l'approche tableau noir est la volonté de vouloir diviser le problème global en sous-tâches indépendantes. Pour une application particulière, le concepteur définit un espace des solutions et les connaissances nécessaires pour trouver la solution. Ces connaissances sont partitionnées en diverses sources de connaissances spécialisées pour accomplir certaines sous-tâches. L'espace des solutions, quant à lui, est décomposé en régions appelées les niveaux du tableau noir, contenant des solutions partielles ou intermédiaires. Ces régions sont généralement conçues dans le but de mémoriser les informations utiles ou produites lors de la réalisation d'une sous-tâche. Par exemple, dans HEARSAY-II [Errman & al 80] sont présents entre autre les niveaux "syllabe" et "mot" car des procédures permettent d'inférer des mots par assemblage de syllabes ou de construire des syllabes à partir des mots.

La méthode "tableau noir" fournit donc un cadre de travail pour organiser les connaissances en les partitionnant en plusieurs sources de connaissances et une stratégie pour utiliser ces connaissances répondant ainsi aux questions : quelle source de connaissance doit être utilisée, quand et comment ? Ce type d'architecture permet d'avoir un modèle de raisonnement plus ou moins opportuniste, résolvant un but fixé par incrémentation de l'espace des solutions et en utilisant une source spécialisée de connaissance à chaque étape.

L'architecture tableau noir est constituée généralement des trois éléments suivants :

- Les *sources de connaissances* (ou spécialistes) :

  Ce sont des processus indépendants qui coopèrent dans le but de résoudre un problème. Chacune d'elles est un expert dans un sous-domaine du domaine étudié. Elle aide à parvenir au résultat par modification du tableau noir. Leurs actions sont : la création ou la modification d'éléments solutions sur le tableau. Chaque source de connaissance est responsable de savoir quand elle peut contribuer au processus de résolution. En effet, des préconditions indiquent la condition qui doit être satisfaite avant l'activation du corps de la source de connaissance en question. En conséquence, leur représentation est de la forme : condition - action ou précondition - corps. Les sources de connaissances sont représentées généralement par des procédures et/ou des paquets de règles ou parfois par des assertions logiques.

- Le *tableau noir* (l'espace des solutions) :

  Toutes les solutions possibles, qu'elles soient partielles ou finales, sont enregistrées sur une base de données globale et structurée : le tableau noir. Ainsi ce dernier représente l'état courant de la solution. Les sources de connaissance, et elles seulement, produisent des changements sur le tableau noir, conduisant incrémentalement à une solution finale ou à un ensemble de solutions acceptables. Ainsi l'interaction entre les diverses sources de connaissance a lieu par l'intermédiaire de changements sur cette structure. La solution est construite étape par étape, à chaque activation d'une source de connaissance. Le tableau noir est constitué d'objets qui sont soit des données initiales, soit des solutions partielles, soit des alternatives, soit des solutions finales. Les objets sur le tableau noir sont hiérarchiquement organisés en niveaux d'analyse des solutions. Ainsi il permet de considérer la solution avec plusieurs niveaux de détails.

- La *composante de contrôle* :

  Cette composante gère l'activité de résolution des problèmes posés au système. Comme cette résolution se fait par l'intermédiaire de changements sur le tableau noir provoqués par les sources de connaissance, le rôle principal du module de contrôle est de déterminer quelle source de connaissance activable, c'est à dire dont la condition est vérifiée, va être exécutée prochainement. Il gére aussi les changements sur le tableau et décide quelle action accomplir ensuite. Pour cela, il choisit un point de focalisation qui peut être aussi bien de décider quelle source de connaissance activer prochainement que de choisir le problème à traiter et son contexte d'invocation sur le tableau noir ou une combinaison des deux, c'est à dire quelle source de connaissance employer et sur quel objet du tableau noir. La plupart des systèmes implémentant une architecture tableau noir n'utilisent qu'une de ces trois approches. Généralement, pour accomplir sa tâche, le module de

contrôle maintient à jour un agenda des sources de connaissance activables en attente [Hayes-Roth 83]. La résolution d'un problème est réalisée par cette composante de contrôle en exécutant un cycle de base qui est le suivant :

- Choix d'un point de focalisation.
- Suivant la nature de l'information contenue dans ce point de focalisation, le module de contrôle prépare l'exécution d'une source de connaissances.
  - Si c'est une source de connaissance, un ou plusieurs objets sur le tableau sont choisis pour servir de contexte d'invocation de la source de connaissance (approche dirigée par la connaissance).
  - Si c'est un objet du tableau, une source de connaissance est sélectionnée qui pourra réaliser cet objet (approche dirigée par les événements).
  - Si c'est les deux à la fois, la source de connaissance est prête pour exécution.
- Exécution de la source de connaissance provoquant :
  - la génération d'événements sur le tableau noir,
  - la recherche des nouvelles sources de connaissances susceptibles d'être activées du fait du nouvel état de la solution,
  - la mise à jour de l'agenda.

En conséquence, la séquence d'invocation des sources de connaissance est dynamique et opportuniste et non fixée et préprogrammée. Généralement, une source de connaissance a pour rôle d'indiquer quand le processus de résolution est terminé : une solution acceptable a été trouvée, ou le système est arrêté à cause d'un manque d'informations.

3.2.3.2 Utilisation de l'architecture tableau noir pour résoudre des problèmes de conception.

Revenons toujours à la même idée énonçant que l'activité de conception peut être vue comme la décomposition d'un problème global en sous-problèmes tout en s'assurant de la bonne inter-connexion entre les diverses sous-parties du systèmes total, obtenues par la résolution des sous-problèmes. L'architecture tableau noir semble très bien refléter cette tactique. En effet, différentes sources de connaissance peuvent détenir le savoir sur le moyen de réaliser une portion du système, vérifier avant d'agir si des contraintes sont satisfaites pour pouvoir intervenir et enfin participer à la réalisation du système par une action sur le tableau noir. De plus l'activité de conception est généralement mal structurée par nature, c'est à dire qu'un chemin pour la résolution ne peut pas être spécifié a priori. A chaque étape de la conception, diverses solutions se présentent. Le choix de l'une d'entre elles doit être guidé localement. Ainsi une approche assurant un certain opportunisme dans le processus de résolution est-elle nécessaire pour bien modéliser l'activité de conception. Or des techniques sont disponibles, et en particulier l'architecture tableau noir complétée par des connaissances spécifiques aux domaines d'application permet de traiter ces problèmes mal structurés. La division du tableau noir en différents niveaux d'abstraction permet aux éléments de chaque niveau d'être des abstractions d'éléments situés à des niveaux plus bas. Cette partition des connaissances est non seulement naturelle mais aussi utile pour le problème de conception du fait qu'une solution partielle (un ensemble d'hypothèses) à un niveau d'abstraction peut être utilisée pour contraindre la recherche à des niveaux adjacents.

Deux exemples de systèmes pour la conception implémentant une telle approche vont être présentés. Le premier est DECADE (Design Expert for CAtalyst DEvelopment), un prototype de système expert pour la sélection de catalyseurs [B.Alcantara & al 88]. A partir de la spécification d'une réaction, ce système tente de proposer un ensemble de matériaux ayant un haut degré de probabilité d'être de bons catalyseurs pour la réaction désirée et les conditions suivant lesquelles les catalyseurs proposés devront opérer. DECADE permet une représentation hybride des connaissances par la manipulation aussi bien de règles de production que de procédures et d'objets. Pour cela, deux langages sont utilisés: OPS5 pour les règles de production et SRL, extension de FranzLISP, pour les connaissances déclaratives car il permet une représentation très sophistiquée des concepts et de leur relations tout en assurant un mécanisme d'héritage. Comme tout système à architecture tableau noir, DECADE est composé de plusieurs sources de connaissances et d'un tableau noir. Le mécanisme de contrôle (schématisé par la figure 10) se déroule par l'intermédiaire d'une source de connaissance spécialisée : la focalisation de l'attention. Dès que l'utilisateur a sélectionné le problème, une description du but à résoudre est introduite sur le tableau noir. Toute source de connaissance qui a accès au tableau et qui peut résoudre ce type de but peut proposer une estimation pour la solution. La focalisation de l'attention attend que toutes les estimations possibles soient soumises, puis elle les évalue en assignant une priorité à chacune des sources de connaissance. Celle ayant la priorité la plus élevée est activée. Trois éventualités sont alors possibles.

- (1) La source de connaissance ainsi sélectionnée résout le problème complètement à travers le tableau noir et le contrôle est retourné à l'utilisateur si c'est lui qui a fait la requête. Dans le cas où une source de connaissance avait demandé la résolution d'un sous-but, c'est la focalisation d'attention qui prend la main. Celle-ci désigne alors le nouveau but à traiter : le parent du sous-but.

- (2) La source de connaissance nécessite d'autres résultats partiels pour pouvoir résoudre le problème. Un sous-but est alors soumis qui sera traité comme tous les autres buts et en particulier comme le but initial. La seule différence est qu'après résolution de ce sous-but, une solution finale ne sera jamais obtenue mais le contrôle sera passé à la source de connaissance ayant fait cette requête. Un pointeur permet de relier le sous-but à son parent.

(3) Si la source de connaissance ne peut pas résoudre le problème, un échec apparait. Le contrôle est alors rendu à la focalisation de l'attention. Cette dernière active une autre source de connaissance, s'il en reste, ayant la priorité la meilleure parmi celles ayant proposé une estimation de la solution. Dans le cas où aucune source de connaissance (SC) n'est en attente d'activation, la main est rendue à l'utilisateur.

Une autre approche intéressante à base de tableau noir pour la conception est proposée par Dixon et Simmons [Simmons & al 84]. Celle-ci est axée sur l'observation suivante : dans de nombreux cas, le processus de conception peut être considéré comme composé de deux phases. La première permet d' obtenir un système initial. Ensuite des itérations sont faites, composées d'une évaluation suivie d'une modification de la solution courante du système, jusqu'à l'obtention d'une solution acceptable. Ainsi chacune des fonctions suivantes : réalisation du système initial, évaluation, décision sur l'acceptabilité de la solution et modification, sont implémentées par des sources de connaissance. Un cinquième type de source de connaissance régit le contrôle tandis qu'un sixième type s'occupe des échanges avec l'utilisateur (voir figure 11).

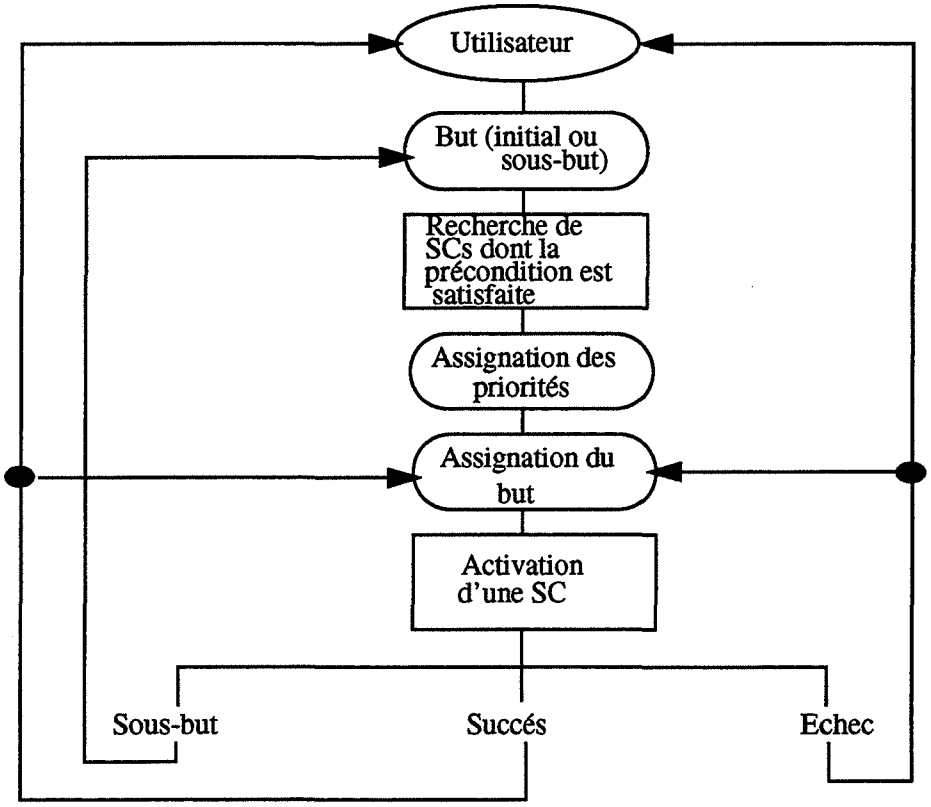


figure 10 : Le mécanisme de contrôle de DECADE.

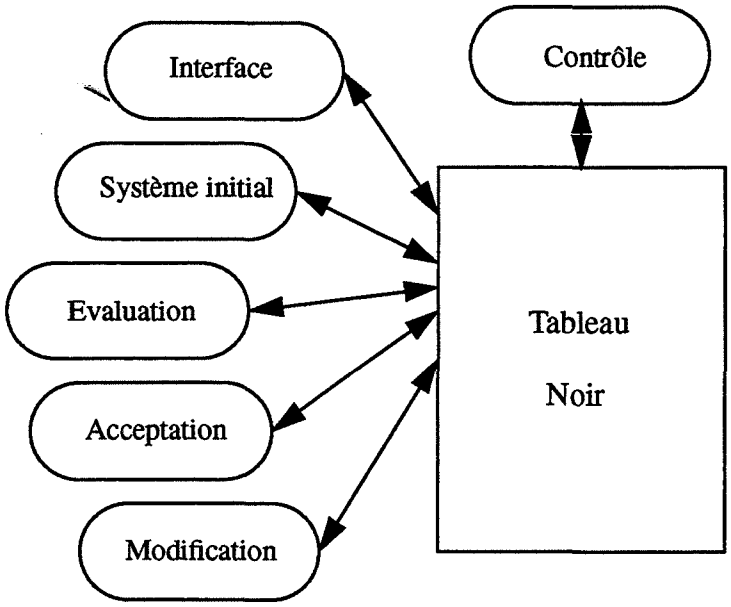


figure 11 : L'architecture de Dixon et Simmons.

Les stratégies adoptées pour la réalisation d'un système initial s'inspirent principalement de l'expérience des experts du domaine et sont réalisées par exécution d'une ou d'une combinaison des opérations suivantes :

- création d'un système initial à partir de systèmes effectués dans le passé et ayant des prérequis similaires,
- création d'un système initial minimal et grossier,
- création d'un système initial qui satisfait seulement une petite partie des contraintes ou des spécifications les plus critiques,
- création d'un système initial, prototype très général.

Un principe important de l'architecture proposée est que les résultats de l'évaluation et de l'acceptation peuvent guider le processus de modification du système courant. Ainsi la source de connaissance de l'évaluation est capable d'interpréter l'information fournie par les différentes analyses et rend cette interprétation accessible, via le tableau noir, aux sources d'acceptation et de modification. De même l'acceptation n'est pas seulement un jugement binaire car en cas de négation, l'étape modification du système a besoin de savoir pourquoi elle n'a pas été acceptée.

3.2.3.3 Forces et faiblesses de l'architecture tableau noir.

L'architecture tableau noir fournit un mécanisme dont le comportement au niveau du contrôle peut être qualifié d'intelligent. En effet, ce contrôle ne résoud pas simplement les problèmes mais il possède des informations sur comment résoudre les problèmes, pourquoi réaliser une action plutôt qu'une autre et quelle action il est préférable d'activer plus tard. Il utilise cette connaissance pour adapter son comportement en fonction des situations dynamiques de résolution de problème auxquelles il se trouve confronté.

Plusieurs raisons font qu'une approche tableau noir peut être préférée à une autre pour la résolution de problème. D'une part, la modularité de la représentation, que ce soit au niveau des sources de connaissances, des divers niveaux du tableau noir, de la structure de contrôle et la séparation entre les concepts liés au domaine et ceux liés au contrôle conduit à un système évolutif, lisible, concis, structuré, fiable et facile à tester et à maintenir. D'autre part, les principales forces de cette architecture tableau noir sont aussi sa grande flexibilité dans le comportement du système conduisant à des solutions opportunistes et sa robustesse. Comme le prouve le nombre d'applications implémentant ce modèle (voir introduction et [Engelmore & al 88]), cette architecture est adaptée à divers domaines d'application. Elle est particulièrement adaptée pour l'intégration d'informations disparates (différentes sortes de connaissances) et lorsqu'apparaissent divers sous-groupes de connaissances spécialisés.

Cependant, comme l'objectif est de diviser le problème global en sous-problèmes, la manière dont on réalise cette décomposition provoque beaucoup de différences quant à la clarté de l'approche, la rapidité avec laquelle la solution est trouvée, les ressources nécessaires et même la capacité à résoudre un problème. D'autre part, le développement et la mise au point de systèmes fondés sur le modèle du tableau noir comme HEARSAY-II, ... est difficile à comprendre et à maîtriser et demande, par conséquent, un effort considérable étant donné la complexité du contrôle. Des générateurs de systèmes implémentant l'architecture tableau noir (HEARSAY-III, AGE, ...) offrent un gain de temps considérable dans la mise en oeuvre d'un tel système parce qu'ils apportent des outils prêts à être utilisés. Par contre, ils n'apportent pas vraiment une aide pour la compréhension de la résolution de problèmes très complexes et du processus de coopération entre plusieurs agents chargés de résoudre ensemble ces

problèmes. Bien que destinés à fournir des outils généraux et donc indépendants du domaine d'application, ces générateurs se sont avérés orientés vers des types d'applications assez restreints, étant donné que chacun d'eux impose un contrôle uniforme pour la résolution de problème [Laasri & al 88].

Si quelqu'un se fixe le but de construire un système ayant une haute performance, l'architecture tableau noir est très certainement inappropriée car sa plus grande faiblesse est sûrement son coût en stockage d'informations et en temps de calcul. Certes, un tel système peut bénéficier de mécanismes de contrôle "intelligents" mais au détriment de son efficacité. En conséquence, l'architecture tableau noir offre un environnement extrêmement flexible et modulaire pour le développement expérimental de systèmes d'intelligence artificielle mais s'avère inefficace pour un système d'application final.

Cette méthode a été testée dans quatre domaines différents en Mécanique. Du fait que le mécanisme est du type engendrer puis tester et du fait d'un rapide accroissement des possibilités de conception dès que la taille du problème augmente, cette approche est bien adaptée à des petits problèmes ou à des systèmes décomposables en plusieurs composants à concevoir. Pour des problèmes plus complexes, Dixon et Simmons proposent d'effectuer la démarche de manière hiérarchique ; c'est à dire que le système est conduit à travers une série de niveaux dont la spécificité s'accroit. A chaque niveau, des sous-problèmes sont définis et sont passés à des niveaux inférieurs, combinant ainsi l'approche tableau noir et la hiérarchisation de la résolution.

En conclusion de ce chapitre, j'aimerais noter que, quelle que soit l'architecture employée, elle est toujours fondée sur une seule remarque, qui est la suivante : l'activité de conception peut être vue comme la décomposition d'un problème global en sous-problèmes tout en s'assurant de la bonne inter-connexion entre les diverses sous-parties du système total, obtenues par la résolution des sous-problèmes. Différentes contraintes et spécifications sont vérifiées à des points stratégiques du processus de résolution afin de s'assurer du cheminement de la résolution vers une solution correcte. Dans le cas contraire, ce cheminement est réajusté. Le degré d'efficacité des différents systèmes en conception provient en grande partie de l'utilisation faite de ces contraintes. Des améliorations du modèle "engendrer puis tester" ont été présentées dans ce chapitre. L'une d'elles porte sur l'émission de conseils de résolution dès qu'une contrainte n'est pas satisfaite. Dans le cas de MOLGEN, les contraintes permettent l'élagage de l'ensemble des alternatives et apportent un mécanisme de communication entre les différentes parties du système alors que le postage de celles-ci autorise de différer le plus longtemps possible la prise de décision. Enfin, dans l'architecture tableau noir, les contraintes sont vérifiées avant l'activation d'une source de connaissances assurant ainsi l'existence d'une solution courante satisfaisante.

Conclusion de la première partie.

Différentes méthodes de représentation des connaissances ont été présentées dans cette partie. Il est bien difficile de les comparer, d'autant plus que certaines impliquent aussi une architecture particulière du système les implémentant. Néanmoins, une classification peut être établie en fonction des fonctionnalités nécessaires pour avoir une représentation résolvant les problèmes auxquels on peut être confronté lors du choix de l'une d'entre elles (voir le premier chapitre). La figure ci-après présente cette classification s'inspirant de classifications similaires [Pinson 81], [Vignard 86]. Chaque formalisme de représentation des connaissances est classifié comme répondant de manière bonne, moyenne ou mauvaise à une fonctionnalité. Cette classification est nécessairement subjective, étant donné que les méthodes considérées ne sont pas, elles-mêmes, définies précisément.

La logique présente le grand avantage d'être à la base de théories bien définies dont les limites sont bien connues. Cependant, certaines connaissances sont difficilement représentables avec ce formalisme, comme par exemple les exceptions. L'indépendance entre les formules logiques (ou règles) offre l'avantage de garder la modularité sémantique aussi bien que syntaxique de la base de connaissances. Ceci est très important pour les systèmes experts dont la base de connaissances est développée incrémentalement. Par contre, il n'est pas toujours évident de déterminer l'impact provoqué par l'ajout d'une nouvelle formule (règle) à la base de connaissances. Comment s'assurer qu'elle ne provoque pas une contradiction avec une autre règle ?

La représentation procédurale apporte des facilités pour l'écriture des algorithmes mais, par contre, elle permet difficilement de représenter les heuristiques, l'expérience des experts humains. En quelque sorte, la plupart des connaissances nécessaires à la réalisation d'un système expert sont laborieusement traduisibles en procédures. Par contre, cette représentation est souvent utilisée pour la réalisation des mécanismes de contrôle.

Les systèmes de production ont longtemps offert et offrent encore un formalisme bien adapté à la représentation des connaissances incluant leur traitement, comme le prouve le nombre de systèmes experts utilisant cette technique.

Conclusion de la première partie.

Les réseaux sémantiques apportent une bonne organisation des connaissances mais il manque une théorie permettant de les standardiser. D'autre part, un réseau sémantique devient vite complexe et difficilement manipulable dès qu'il doit contenir beaucoup d'informations.

L'avantage du modèle centré objets est que les caractéristiques descriptives et procédurales d'un objet sont directement associées à cet objet. Le bénéfice de cette possibilité sont la réutilisabilité, la testabilité, la maintenabilité, l'extensibilité, la robustesse. De plus, ce formalisme est bien approprié pour exprimer des hiérarchies, l'héritage de propriétés, les exceptions, les valeurs par défaut alors que ces possibilités sont difficilement traitées par les autres méthodes.

Il est parfois difficile de représenter un ensemble de connaissances sur un domaine particulier à l'aide d'un seul formalisme, par exemple uniquement des procédures ou des règles ou encore des objets. L'utilisation simultanée de plusieurs modes de représentation des connaissances peut faciliter la tâche du concepteur, d'autant plus que la connaissance sur un domaine particulier est très souvent hétérogène. Ainsi de nouveaux types de langages sont-ils apparus : les langages hybrides mettant à la disposition des utilisateurs plusieurs formalismes de représentation des connaissances.

Différentes méthodes pour simuler l'activité de conception ont été implantées, mais relativement peu de théories sont à la disposition du concepteur dans ce domaine.

Ayant ainsi fait le point sur les différents mécanismes actuellement à la disposition de tout concepteur de systèmes à base de connaissances, le choix d'un formalisme adapté à la conception peut être envisagé et particulièrement dans le cas de la conception de réseaux informatiques.

L. Logique

P. Procédure

S.P. Système de production

R.S. Réseau Sémantique

P.O.O. Programmation Orienté Objets

	L.	P.	S. P.	R. S.	P. O. O.
Organisation	Mauvais	Mauvais	Moyen	Bon	Bon
Théorie disponible	Bon	Moyen	Moyen	Moyen	Moyen
Représentation de connaissances déclaratives	Bon	Mauvais	Bon	Bon	Bon
Représentation de connaissances procédurales	Mauvais	Bon	Moyen	Mauvais	Bon
Représentation de méta-connaissances	Moyen	Moyen	Bon	Moyen	Bon
Représentation des heuristiques	Moyen	Mauvais	Bon	Moyen	Moyen
Représentation de l'incertain et de l'imprécis	Mauvais	Mauvais	Moyen	Mauvais	Moyen
Représentation des exceptions	Mauvais	Mauvais	Moyen	Mauvais	Moyen
Cohérence	Moyen	Mauvais	Moyen	Moyen	Moyen
Modularité	Bon	Mauvais	Bon	Moyen	Bon
Modifiabilité	Bon	Mauvais	Bon	Moyen	Moyen
Extensibilité	Bon	Mauvais	Bon	Moyen	Bon
Simplicité, lisibilité	Moyen	Moyen	Bon	Moyen	Moyen
Facilité d'utilisation	Moyen	Mauvais	Bon	Moyen	Moyen
Explication	Moyen	Mauvais	Bon	Moyen	Moyen
Efficace (temps d'exécution)	Mauvais	Bon	Moyen	Mauvais	Moyen
Taille des fragments	Petit	Gros	Moyen	Petit	Gros

figure 12 : Comparaison des méthodes de représentation des connaissances.

Conclusion de la première partie.

Deuxième partie : Un système à base de connaissances d'aide à la conception de réseaux informatiques, NEST.

Le réalisation d'un système à base de connaissances implique les quatre étapes suivantes : (1) l'acquisition des connaissances auprès des experts du domaine concerné, (2) le choix d'une représentation adéquate, (3) la formalisation des connaissances et (4) la validation du système. Ces quatre étapes sont le reflet de l'ossature de cette partie.

Mais tout d'abord, étant donné que cette réalisation a été effectuée au sein d'un projet dont l'objectif est une interface pour systèmes à base de connaissances, un premier chapitre présentera cette interface.

Le second chapitre introduit le domaine d'application : la conception de réseaux informatiques. Ce domaine étant très vaste, il sera élagué.

Un troisième chapitre présente la réalisation de NEST. Après avoir introduit l'architecture de NEST., les différents éléments le constituant seront décrits. Ce chapitre traitera aussi des résultats obtenus. Comme NEST est couplée à une interface prometteuse puisque multi-modes, ses avantages pour un système à base de connaissances seront mis en évidence. Malgré l'effort déployé pour l'implémentation de NEST, tout n'est pas parfait. Aussi des améliorations sont proposées. Enfin, il montre l'apport de la réalisation de NEST pour la recherche en intelligence artificielle et plus particulièrement pour celle des systèmes à base de connaissances pour la conception, en comparant la méthode employée à celles développées au chapitre 4 de la première partie.

CHAPITRE 1 Le projet Esprit MMI^2.

Le système à base de connaissances qui est l'objet de cette partie a été développé dans le cadre d'un projet Esprit dont l'objectif est une interface multi-modes pour des systèmes à base de connaissances. Nous allons donc présenter ici ce projet, pour bien situer le système à base de connaissances sur lequel nous avons travaillé.

L'objectif principal du projet Esprit II MMI^2 est la conception d'une boîte à outils apportant des possibilités d'interfaçage multi-modes pour un système à base de connaissances. Différents modes de communication : les langues naturelles anglaise, française et espagnole, le graphique, le gestuel et un langage de commande, sont ainsi intégrés de telle sorte qu'il est permis de mixer les modes au cours d'une seule interaction. Pour plus de détails sur cette interface, le lecteur pourra se référer à [Binot & al 90] ou [Wilson 91].

1.1 Architecture de l' interface MMI^2.

L' architecture de cette interface est fondée sur la notion d' "expert". Un expert est un module autonome dans le sens où il possède ses propres structures de données, dont le but est d'accomplir une tâche spécifique telle que, par exemple, assurer le dialogue en langue naturelle, modéliser l'utilisateur, gérer le contexte du dialogue. Ainsi neuf experts ont été définis qui gravitent autour d'un module central appelé le contrôleur de dialogue. Celui-ci contrôle le bon déroulement de l'interaction entre les différents modes, l'acheminement des entrées utilisateur (données ou questions) vers les modules concernés, les sorties du système en vue d'une demande de compléments d'information ou pour répondre à une requête de l'utilisateur. Chaque requête doit passer par ce module de gestion du dialogue pour être ensuite aiguillée vers l'expert approprié. Cependant, il est possible, selon les besoins, que les experts communiquent directement entre eux sans passer par cet expert. C'est, en particulier, le cas de l'expert du domaine lorsqu'il détecte un manque d'information pour pouvoir activer une action du système à base de connaissances. Il communique directement avec l'expert de planification de la communication pour lui demander d'initialiser un sous-dialogue pour acquérir auprès de l'utilisateur l'information manquante par l'intermédiaire de questions.

L'architecture de cette interface, présentée sur la figure 13, est par conséquent très modulaire. Un langage interne, appelé CMR pour Common Meaning Representation, a été défini pour permettre à ces modules de communiquer entre eux.

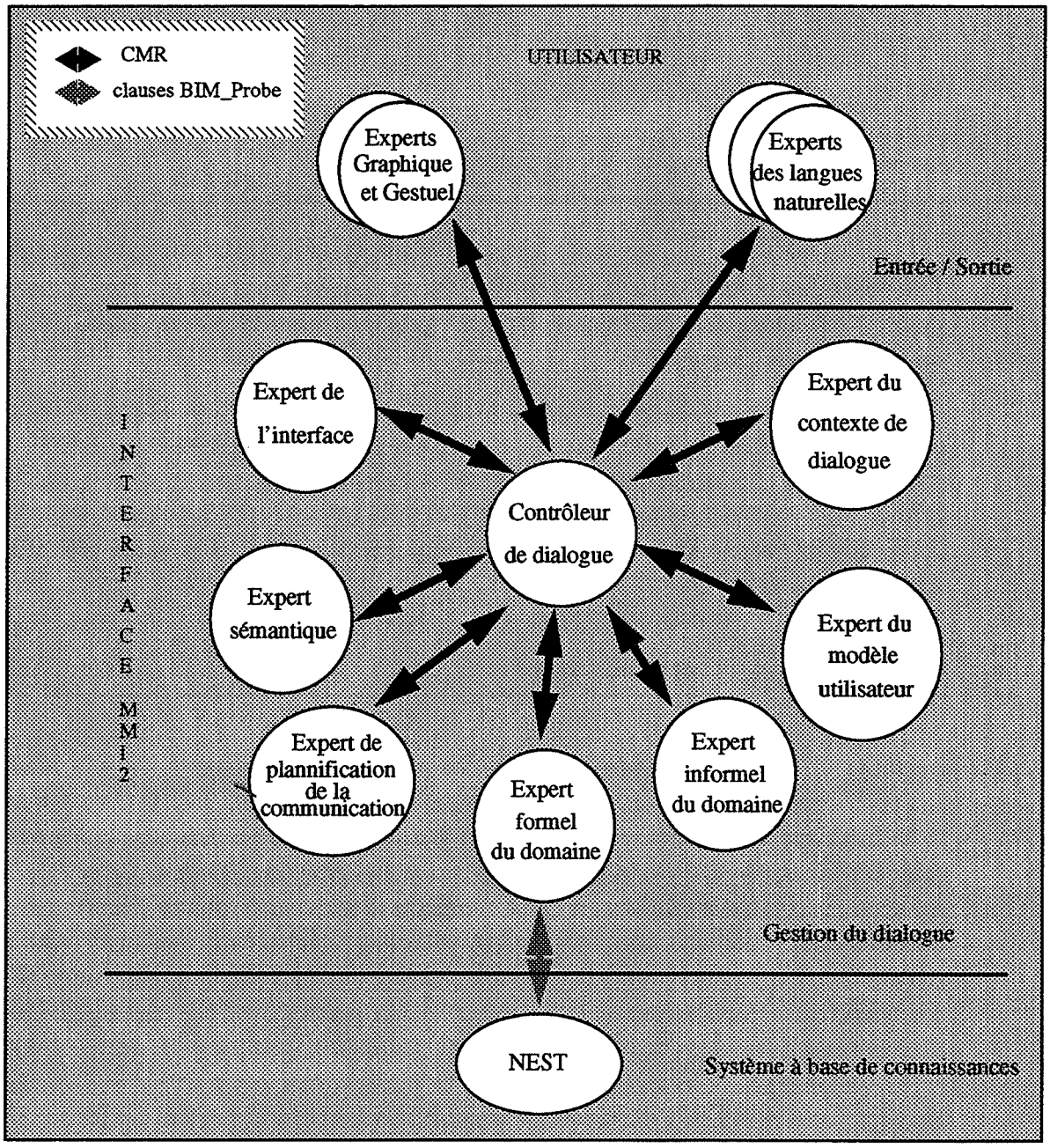


figure 13 : L'architecture de l'interface MMI2.

Plusieurs experts assurent les échanges entre l'utilisateur et le système suivant le ou les modes de communication employé(s). Trois experts des langues naturelles gèrent le français, l'anglais ou l'espa-

gnol. L'expert graphique fournit des outils pour la description de bâtiments, d'étages, de pièces, d'éléments de réseaux... mais aussi pour la représentation des résultats que ce soit pour le réseau conçu par NEST ou pour d'autres résultats puisque des histogrammes, des graphiques, des camemberts peuvent présenter plus explicitement certains résultats. Un expert gestuel permet des requêtes par l'intermédiaire de gestes relevés avec la souris sur la fenêtre graphique. L'utilisateur peut également utiliser le langage de commande. Les autres experts ont la charge de s'assurer de la gestion du dialogue et enfin l'un d'eux (l'expert formel du domaine) transmet les échanges entre l'application (NEST) et l'interface. Tous dialoguent en CMR sauf l'application et l'expert formel du domaine qui, eux utilisent le langage BIM_Probe.

Nous présenterons dans les paragraphes qui suivent les divers experts de cette interface avant de décrire le langage interne utilisé par MMI^2.

1.2 Les experts des langages naturels.

Trois langues sont utilisables pour dialoguer avec l'interface : le français, l'espagnol et l'anglais. Ces modes permettent également de poser des questions ou de répondre à l'utilisateur en langage naturel. L'implémentation de chacun de ces modes s'est faite indépendamment puisque chaque langue a des caractéristiques qui lui sont propres. Cependant ils permettent tous de saisir des phrases en langue naturelle, de les interpréter et de les traduire en expressions CMR. Leurs structures sont proches les unes des autres puisque tous ces modes, à partir des caractères tapés au clavier, tentent d'interpréter correctement la phrase ainsi saisie. C'est un outil morphologique qui effectue cette interprétation et, en sortie de cet outil, plusieurs alternatives sont proposées pondérées par des facteurs de probabilité. Un analyseur traite syntaxiquement la branche de poids le plus fort afin de préparer la dernière étape de traduction de la phrase initiale en expressions CMR. D'autres modules peuvent également être inclus dans leur architecture. Trois systèmes indépendants et d'architectures diverses ont été développés parce que ces trois langues présentent des caractéristiques différentes et requièrent des traitements différents.

1.2.1 Le français.

L' expert pour le français est décomposé en trois modules :

- l'analyseur morphologique : les chaînes de caractères tapées par l'utilisateur sont converties en une ou plusieurs séquences, composées de paires de la forme (entrée lexicale, catégorie grammaticale + valeurs de variables), correspondant à une phrase [CRISS 91]. La sortie de cet analyseur est une liste de séquences ordonnées de la plus probable à la moins probable.

- l'analyseur syntaxique : à partir de la liste fournie par l'analyseur morphologique, il tente de construire la structure syntaxique de la phrase. Ensuite, il fournit une structure fonctionnelle de la séquence afin de préparer l'étape suivante.

- le module sémantique : opère un certain nombre de transformations sur la structure fonctionnelle puis la transforme en expression CMR à l'aide d'opérations linguistiques et logiques.

1.2.2 L'espagnol.

Cet expert est décomposé en cinq modules [Pérez & al 91] :

- le composant morphologique, MORFEO, qui est lui même constitué de trois éléments : un outil morphologique, un analyseur morphologique et un filtre. L'expert de l'interface traduit l'événement d'entrée en une chaîne de caractères qui est transmise à l'outil morphologique. Celui-ci a pour but de préparer cette chaîne en vue de son analyse. Pour cela, il opère un pré-traitement de cette chaîne : transformation ou suppression de certains signes, etc. L'analyseur analyse les mots de la phrase et les classifie morphologiquement. Il assigne des lemmes qui pointent sur l'information lexicale des mots et fournit des solutions pour certains phénomènes. A partir des traitements faits par l'analyseur, le filtre ordonne les solutions possibles de l'interprétation de la chaîne de caractères initiale pour n'en passer qu'une, celle ayant la plus forte probabilité, au lexique.

- le lexique, LEX : il ajoute aux structures de données dynamiques des informations lexicales concernant les aspects syntaxique et sémantique nécessaires à l'analyseur.

- l'analyseur d' espagnol, SP : la sortie du composant morphologique est réduite à un ensemble de groupes élémentaires. Puis ces groupes sont combinés successivement suivant une procédure de gauche à droite et de bas en haut. Finalement, il remplit des termes non spécifique tels que des structures sans type, des sujets nuls. La sortie de cet analyseur est une formule descriptive fonctionnelle (FD).

- le traducteur en CMR, LOGIC : comme son nom l' indique, il traduit les FDs en expressions CMR.

- le générateur d'espagnol, GENIUS : permet d'engendrer en espagnol des réponses à l' utilisateur venant du système ou des questions.

1.2.3 L'anglais.

Ce module est extrait du système Loqui [Ostler 89], une interface en langue naturelle anglaise pour des bases de données relationnelles, qui a été portée au domaine de l'application de MMI^2. De ce fait, un nouveau lexique et des interfaces de communication avec les autres modules de MMI^2 ont dû être définis. Ce module est aussi composé d'un outil morphologique utilisant le lexique, d'un analyseur traitant les information venant de l'outil précédemment cité et d'un traducteur en expressions CMR.

1.3 Les experts graphique et gestuel.

Un autre mode de dialogue est à la disposition de l'utilisateur et du système : le graphique. Cet expert apporte des facilités pour représenter les plans des bâtiments à câbler et les réseaux passant dans cette structure mais aussi pour la présentation des résultats fournis par le système ou pour poser des questions gestuelles au système. Dans l'architecture de la figure 15, on retrouve différents modules fournissant ces facilités :

- l'outil de dessin permet la saisie des plans de bâtiments et le dessin des éléments d'un réseau. Une copie d'écran de la fenêtre de l'outil de dessin est présentée sur la figure 48 de la page 152. Le plan du bâtiment a été introduit par l'utilisateur au moyen des utilitaires de cet outil et la réponse de NEST à la requête de conception de réseau a été visualisée grâce à cet outil.
- des outils de représentation graphique assurent la sortie de résultats sous forme de camemberts, d'histogrammes, de tables ou de courbes de dispersion. La figure 49 de la page 158 est un exemple de représentation à l'aide de camemberts du coût des machines d'un réseau.
- l'outil gestuel. Cet outil est capable d'interpréter certains gestes de l'utilisateur comme étant des commandes. La figure 14 donne quelques exemples de symboles du mode gestuel et leur interprétation.

L'architecture de cet expert graphique présente deux niveaux (voir figure 15). Le premier est constitué des outils graphiques interactifs listés ci-dessus. Le second est celui du gestionnaire graphique qui assure le dialogue en langage CMR entre le niveau 1 de l'expert graphique et les autres experts de MMI[2]. En effet, quand le mode graphique est choisi, le gestionnaire graphique décide quel outil graphique sera activé pour visualiser l'information et transforme les expressions CMR en un ensemble d'actions à exécuter par l'outil choisi. Inversement, il transforme les informations venant des outils graphiques en expressions CMR qui seront envoyées vers le contrôleur de dialogue [Ben Amara & al 92].

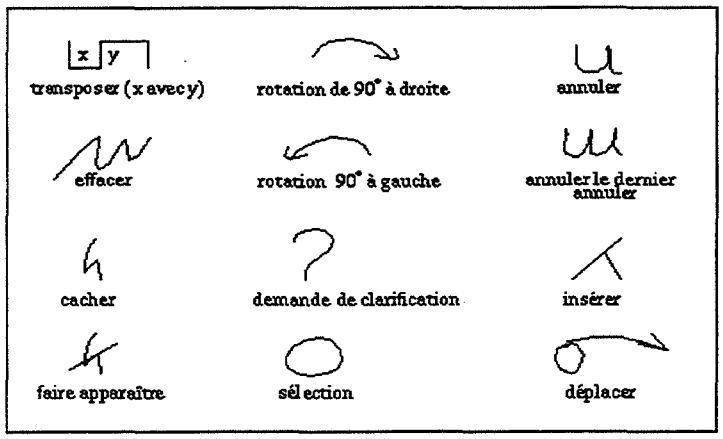


figure 14 : Quelques symboles du mode gestuel.

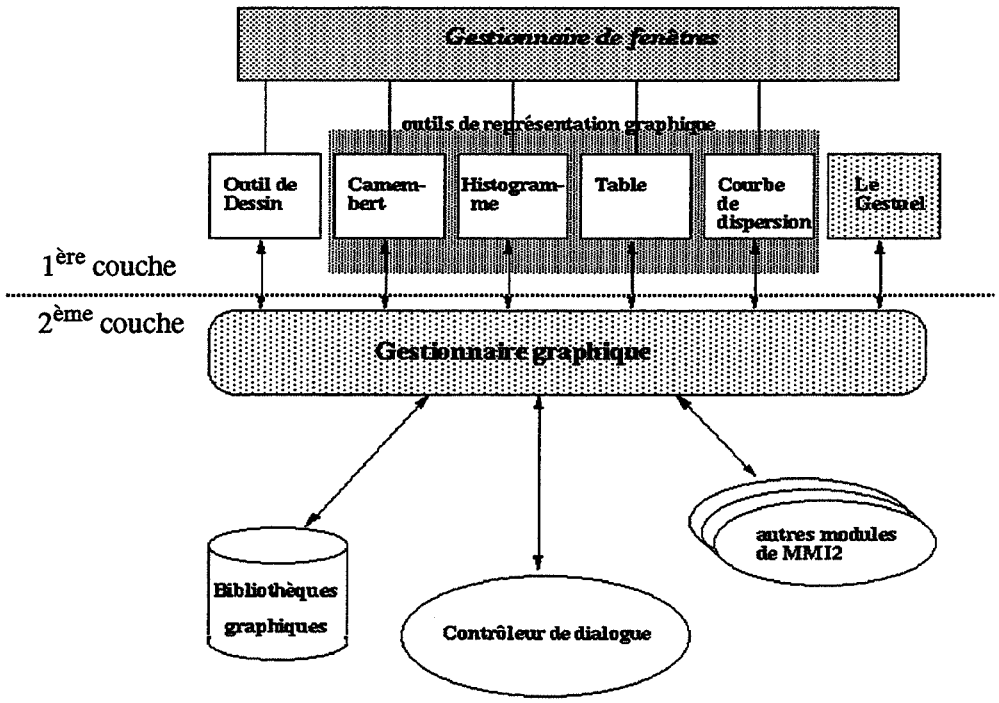


figure 15 : Architecture de l'expert graphique.

1.4 Le langage de commande.

Une interface nécessite une variété de commandes. Dans MMI^2, elles sont divisées en plusieurs groupes :

- des méta-commandes, par exemple, pour accéder aux commandes UNIX ou encore pour la création de "macros",
- des opérations sur des fichiers, pour des sauvegardes éventuelles dans un fichier, etc,
- des commandes spécifiques à l'utilisateur, pour changer d'utilisateur en cours de session, changer de langage ou modifier le nom d'une commande, etc,
- des commandes pour le dialogue (répéter une commande antérieure, par exemple),
- des commandes pour une requête d'information, pour demander une explication (help),
- des actions graphiques (ajouter un objet dans une pièce, par exemple),
- des commandes pour le dessin du réseau, par exemple pour les calculs de coût.

Généralement, le langage de commande est choisi comme un moyen d'interaction avec un système puissant et efficace par les utilisateurs expérimentés. Pourtant, dès que le nombre de modes est important, l'utilisation de ce langage est moins fréquente. Dans MMI^2, certaines commandes peuvent être exécutées en utilisant aussi bien le graphique ou le gestuel que le langage de commande. C'est le cas, par exemple de la commande "move" permettant de déplacer un objet graphique. Cette caractéristique

permet à l'utilisateur de faire exécuter des commandes sans avoir à passer dans le mode propre au langage de commande. Il peut le faire en restant dans le mode courant.

1.5 L'expert de l'interface.

L'expert de l'interface a cinq rôles :

- Son rôle principal est le contrôle du flux de CMR entre le contrôleur de dialogue et les différents modes cités précédemment. Il rassemble les expressions CMR produites par chacun des modes de MMI^2 lorsque l'utilisateur interagit avec le système puis les passe au contrôleur de dialogue. L'objectif est de séparer clairement les modes de la fonction de contrôle du dialogue. Cet expert transmet aussi les réponses du système au mode concerné.
- Il est responsable de l'évaluation formelle des paquets de CMR qu'il reçoit lorsque le contenu de ces paquets relève plus des connaissances de l' interface que de celles de l'application.
- Il doit gérer une fenêtre texte pour permettre à l'utilisateur d'entrer du texte ou au système de répondre.
- Il contrôle les différentes fenêtres : pour le graphique, pour le langage de commande, pour les langues naturelles et quand nécessaire pour les histogrammes, graphes ... Pour cela, il mémorise la position des fenêtres utilisées par l'interface MMI^2 et crée de nouvelles localisations lorsque de nouvelles fenêtres sont nécessaires.
- Il fournit la boucle principale de bas niveau qui permet le contrôle des événements provoqués par l'utilisateur. Il contrôle aussi si ces événements sont bien transmis au module de l'interface concerné par leur interprétation.

1.6 L'expert sémantique.

L'expert sémantique aide d'une part à la traduction des interventions de l'utilisateur en des expressions CMR et d'autre part à l'interprétation des expressions CMR pour accéder aux différents aspects du dialogue.

La nécessité dans MMI^2, d'un tel expert est justifiée par deux remarques : (1) la CMR ne doit ni être exprimée en langue naturelle, ni contenir des mots de la langue naturelle et (2) la CMR ne doit pas être exprimée directement en termes d'objets et de prédicats dépendants de l'application. Ainsi un formalisme a été défini : le type sémantique. Ce type peut être lié soit à des objets (personne, machine, etc), soit à des actions (création, déplacement, etc), soit à des situations (être connecté, etc). L'expert sémantique associe aux types sémantiques des rôles ou des propriétés qui sont utiles pour interpréter les entrées de l'utilisateur et détecter les incohérences et les formulations inadaptées sans avoir à accéder à l'application.

Pour des raisons d'indépendance de l'interface par rapport à l'application, les instances créées lors d'une session sont partiellement dupliquées dans l'interface et dans NEST. Le rôle principal de cet expert est de faire la jonction entre les mots standard transformés par MMI^2 et ceux transformés dans l'application. Dans MMI^2, ces mots standard contiennent une étiquette. Ces étiquettes sont attribuées par l'expert sémantique pour représenter les objets manipulés par l'interface, les commandes, les objets graphiques et le sens des mots représentatifs pour les trois langues naturelles, légalisant ainsi le vocabulaire du formalisme de la CMR. Par exemple, lorsque l'utilisateur introduit un Sun4, une référence de discours associé au type de la machine est créée (par exemple : *SPARCS (sparcs34)*). Un exemple pour une relation est le suivant : *connected_to (DISK (disk35), SUN3 (sparcs34), CABLE (cable36))* indiquant que le disque *disk35* est connecté à la machine *sparcs34* par le câble *cable36*. La référence de discours d'une relation est dénotée par son type et ses arguments. Les mots de l'application sont représentés au moyen de termes pertinents pour l'application et par le sens des mots relatifs aux trois langues naturelles.

Les fonctionnalités de l'expert sémantique sont les suivantes :

- mettre en correspondance les mots de l'application et ceux de MMI^2,
- définir des étiquettes communes du formalisme de la CMR pour la représentation des entrées et des sorties des différents modes,
- établir des correspondances conceptuelles entre les commandes MMI^2, les objets graphiques et les étiquettes définies par cet expert,
- offrir des facilités générales par des prédicats assurant la résolution de phénomènes spécifiques au dialogue multimodes tels que les ambiguïtés, les présuppositions implicites, les anaphores associatives et les références,
- aider à construire les contextes complexes nécessaires à la communication et au raisonnement de l'interface et du système à base de connaissances à partir des concepts simples et des formules de CMR,
- aider à construire un dialogue coopératif et susceptible de réparer des échecs de certaines commandes.

1.7 L'expert du contexte de dialogue.

L'expert du contexte du dialogue enregistre les informations toutes les interactions venant de l'utilisateur ou du système et apporte des fonctions pour l'extraction ou l'ajout d'informations dans l'enregistrement de l'historique du contexte du dialogue.

L'idée maîtresse est que chaque interaction, ou chaque *discours* entre l' utilisateur et l'interface, quel que soit le mode utilisé, prend place dans un *monde de discours*. Chaque objet mentionné au cours d'une interaction doit exister dans le *monde de discours* où il est appelé *référence de discours*.

Actuellement, cet expert est utilisé seulement pour la résolution des anaphores. Quand une expression CMR non résolue, c'est à dire contenant une ou plusieurs expressions anaphoriques, est transmise

au contrôleur de dialogue, celui-ci fait appel à l'expert du contexte du dialogue pour fournir l'information contextuelle nécessaire à la résolution de l'expression CMR. L'exemple suivant d'un dialogue dans MMI^2 illustre le problème à résoudre.

```
1 u : Déplacer le Terminal_X_1 dans la pièce Room1
2 s : ok
3 u : Quel est le coût du Sun360_1 ?
4 s : X francs
5 u : Détruire ces deux machines
```

Pour répondre à cette requête, l'interface doit trouver les deux machines auxquelles l'utilisateur fait référence. Pour cela, le contrôleur du dialogue fait appel à l'expert du contexte du dialogue pour retrouver les deux derniers antécédents machines dans l'historique du contexte du dialogue.

Pour représenter tous les aspects du dialogue, des sous-dialogues, des démarches, des échanges, etc, des arborescences de nœuds ont été choisies pour structures de données. Ces structures de données peuvent être vues comme un arbre n-aire où chaque nœud terminal représente une démarche ou un échange et les autres nœuds un dialogue ou un sous-dialogue. Seuls les nœuds de dialogue peuvent avoir un espace de contexte de discours. Les autres nœuds héritent du contexte de discours du nœud père le plus proche. Pour plus de renseignements sur l'expert du contexte du dialogue, le lecteur pourra se référer à [Francony 91].

1.8 L'expert du modèle utilisateur.

L'expert du modèle utilisateur acquiert les informations sur l'utilisateur, ce qui va permettre par la suite au système de fournir des réponses adaptées au degré de compétence de l'utilisateur. L'implémentation du modèle utilisateur est inspiré de GUMS (General User Modeling Shell) [Finin 89].

Plusieurs sortes de connaissances sur l'utilisateur sont mémorisées :

- les connaissances générales de l'utilisateur concernant le domaine de l'application, la conception de réseau actuellement. Pour cela, cet expert répond à la question : quels objets du domaine connaît l'utilisateur ?
- les connaissances de l'utilisateur concernant l'interface MMI^2.
- les préférences de l'utilisateur en ce qui concerne le système MMI^2 (par exemple, préférence pour l'affichage des prix des machines d'un réseau à l'aide d'un histogramme).

L'information sur l'utilisateur courant est dérivée du dialogue et est stockée, de manière permanente, entre les sessions. Des règles heuristiques basées sur les travaux de [Grice 75], [Kass 87], [Bunt 89] et sur des expériences de simulation d'un système par un expert humain [Chappel & al 91] permettent d'extraire du dialogue l'information sur l'utilisateur. A partir de cette information, il est possible

de catégoriser l'utilisateur comme étant d'un certain type. Des classes stéréotypes d'utilisateur sont définies dans l'expert du modèle utilisateur et des informations complémentaires pour chaque type d'utilisateur sont données dans ces classes. Ces informations ont été obtenues par des suppositions faites sur les connaissances et les comportements des différents types d'utilisateurs. Le modèle choisi est un modèle orienté objets. A chaque classe d'utilisateurs est associée une description générale des connaissances qu'est censé posséder chaque membre de la classe. De plus, chaque utilisateur possède une description individuelle qui contient des informations qui lui sont propres, comme ses préférences, ses capacités, ses croyances. Des sous-modules appelés spécialistes permettent la mise à jour et l'extraction des informations utiles à partir de ces structures de données. La hiérarchie des types d'utilisateur assure l'héritage multiple entre différents stéréotypes. Le lecteur pourra se référer à [Chappel & al 91] pour plus d'informations sur l'expert du modèle utilisateur.

1.9 Les experts du domaine.

L'expert du domaine est divisé en deux modules : l'expert formel et l'expert informel.

Le premier évalue les paquets de CMR qu'il reçoit afin de fournir la sémantique formelle des entrées faites par l'utilisateur et de déterminer s'il a affaire à une question ou à une assertion pour le système à base de connaissances. Ainsi il établit la communication avec l'application. Il peut donc être vu comme l'interface entre le contrôleur de dialogue et l'application pour l'interprétation formelle des questions ou assertions venant de l'utilisateur.

Le second module a pour rôle de rendre le dialogue coopératif par identification des buts, par comparaison avec des plans établis pour les différentes tâches du système et par vérification de prérequis. Si ces derniers sont satisfaits, il fait une demande d'exécution auprès de l'expert du domaine formel qui la transmet au système à base de connaissances. Sinon c'est l'expert de planification de la communication qui est sollicité pour initialiser un sous-dialogue.

Voici un exemple pour mieux comprendre. Supposons que l'utilisateur entre la requête : Ajouter une station de travail au réseau. Celle-ci correspond à un plan de tâche dont les prérequis spécifient qu'il est nécessaire de connaître le type et la localisation dans le bâtiment de la station de travail. Un sous-dialogue, présenté à la figure 16, doit être initialisé par l'expert de planification de la communication pour demander à l'utilisateur le type et la localisation de la station.

Un autre plan de tâche permet d'activer la requête de conception de réseau auprès de NEST. Les prérequis pour ce plan sont listés en annexe 3. D'autre plans de tâches correspondent aux différents analyses d'un réseau que NEST peut effectuer (voir paragraphe 3.3 de la page 130).

Pour identifier un but, le module informel inspecte le contenu d'une expression complète plutôt que la formule atomique fournie par l'expert formel du domaine et prend en compte les rôles de l'entrée de l'utilisateur et de la sortie du système dans le dialogue général. Cet expert a pour rôle principal

d'identifier les erreurs sur les conditions qui doivent être vérifiées avant d'entrer en contact avec l'application. Le principe de la méthode de détection de ces erreurs est l'utilisation de connaissances du domaine de l'application et de métaconnaissances sur le système sous la forme de *plans de tâche.* Ceux-ci définissent, pour diverses tâches du domaine d'application, des informations prérequises et des contraintes sur l'état courant des données issues des interactions de l'utilisateur avec le système. Un plan de tâche est donc un modèle du type d'informations à fournir à l'application dans le but de spécifier un problème bien formulé. Il peut être vu comme une sorte de squelette pour un problème typique résolvable par l'application, qui sera progressivement instancié au cours du dialogue avec l'utilisateur. Un exemple d'informations prérequises pour le problème de la conception de réseaux est donné en annexe 3. L'identification de problèmes permet de retarder la requête faite auprès du système à base de connaissances jusqu'à avoir les informations nécessaires à son activation. Il permet, d'autre part, à l'interface de prendre l'initiative dans le dialogue tout en poursuivant un but venant de l'utilisateur.

1.10 L'expert de planification de la communication.

Le rôle de cet expert est de façonner la sortie du système, par sélection d'un mode de sortie et par action sur le contenu de cette sortie en collaboration avec l'expert du contexte et l'expert de modélisation de l'utilisateur. Cet expert travaille au moyen de *plans de communication.* Un tel plan est une structure permettant de guider le dialogue avec l'utilisateur. Il travaille en corrélation avec les plans de tâche de l'expert du domaine. Il existe au minimum un plan de communication pour chaque plan de tâche. D'autres plans de communication sont définis pour d'autres parties du dialogue ou pour des sous-dialogues. L'expert de planification de la communication détermine, tout d'abord, quel plan de communication (c_plan) activer, puis le mode avec lequel le dialogue avec l'utilisateur sera établi (qui peut changer au cours d'un même dialogue ou même être multiple). Selon le c_plan activé, il capture ensuite les informations requises et construit chaque argument (a_role) dont est composé le c_plan concerné. Un exemple de sous-dialogue à l'initiative de cet expert est donné par la figure 16. L'utilisateur désire ajouter une station de travail au réseau. Or il est nécessaire pour cela de connaître le type de cette station pour pouvoir la créer.

dialogue	dialogue principal	sous-dialogue	
1 u: Ajouter une station de travail au réseau.	requête		1
2 s: Je suis désolé.		regret	2
3 s: La localisation et le type de la station de travail ne sont pas spécifiés.		notification	3
4 s: Tout ajout d'un station de travail requière la spécification de sa localisation et de son type.		justification	4
5 s: quel est le type de la station de travail?		interrogation	5
6 u: Sun3/60.		réponse	6
7 s: Quelle est sa localisation ?		interrogation	7
8 u: <utilise le graphique pour spécifier la localisation x>		réponse	8
9 s: <ajoute la Sun3/60 à la localisation x>	<action>		9
10 s : ok.	remerciement		10

figure 16 : Le traitement du plan de communication "Ajouter une station de travail au réseau".

1.11 Le contrôleur de dialogue.

L'expert central de l'architecture de l'interface MMI^2 est le contrôleur de dialogue. Il a pour rôle d'assurer la communication entre les différents experts de cette architecture. Une entrée de l'utilisateur lui parvient, traduite en langage CMR par l'un des modes de l'interface. Inversement, les résultats sont transmis par ce module à un des modes sous forme d'expressions CMR.

Une classification des entrées de l'utilisateur est faite par le contrôleur de dialogue. Pour cela, les abréviations suivantes sont utilisées : U pour utilisateur, S pour système, w pour veut (wants), k pour sait (knows). Ainsi :

- (Uwk,P) signifie : l'utilisateur pose la question P.
- (UwSk,P) : l'utilisateur veut dire P au système.
- (Uw,P) : l'utilisateur veut lancer la commande P.

Une telle paire constitue une *attitude* de l'utilisateur alors qu'une expression CMR est appelée un désir de l'utilisateur. La classification des entrées de l'utilisateur est faite en fonction de leur contenu et aussi en fonction de la sortie précédente du système. Aux questions est assigné l'attitude Uwk, aux commandes Uw et aux assertions UwSk. L' entrée de l'utilisateur est placée dans le *contexte du dialogue* qui n'est autre qu' une pile. Le contrôleur de dialogue introduit les désirs de l'utilisateur dans cette pile et décide des actions à accomplir en décrémentant la pile et en activant le premier désir qu'il y trouve. Si par exemple, le désir est de la forme (Uwk,P), le contrôleur va, par l'intermédiaire de l'expert du domaine puis peut être via l'application, répondre à la question posée par l' utilisateur.

Le contrôleur a aussi une classification pour les sorties du système :

- (SwUk,P) : le système dit P à l'utilisateur.
- (Swk,P) : le système pose la question P à l'utilisateur.
- (Sw,P) : le système exécute la commande P.

Comme pour les désirs venant de l'utilisateur, ceux du système sont introduit dans la pile du contexte de dialogue.

1.12 Le langage de communication interne à l'interface : la CMR.

Un langage interne a dû être défini pour permettre le dialogue entre les différents experts de l'interface MMI^2 : la CMR (Common Meaning Representation). Ce langage a pour objectif de fournir des facilités d'expression des actes de communication, c'est à dire des actes pouvant être représentés par l'un des modes de communication, et aussi fournir une structure de données que le système puisse manipuler. Ce langage est dérivé de la logique du premier ordre à laquelle quelques extensions on été ajoutées telles que le quantificateur "the" assurant la possibilité d'utiliser des termes génériques, les quantificateurs numériques : exactement, moins de, au plus, etc.

Une expression CMR contient quatre types d'informations : un contenu propositionnel, une forme logique, une force élocutionnaire (c_force) et des annotations. Une expression CMR est aussi une structure de données qui peut inclure les informations additionnelles suivantes : état du processus, mode, temps de l'action, présuposition de l'utilisateur, erreurs de l'utilisateur et d'autres informations syntaxiques [Ben Amara 92]. Pour plus d'informations sur ce langage, le lecteur pourra se référer à [D2 89].

CHAPITRE 2

Acquisition des connaissances pour la conception de réseaux.

2.1 Enoncé du problème.

Permettre aux ordinateurs de communiquer entre eux, d'échanger des données à traiter et des résultats de calcul, tel est l'objectif principal des réseaux informatiques. Le mariage de l'informatique et des télécommunications (d'où le nom de télé-informatique) est un point de passage quasi obligé lorsqu'on veut interconnecter des machines, surtout entre sites éloignés. En effet, les réseaux téléphoniques ou plus spécifiques tels que Transpac, les lignes spécialisées louées aux PTT, les faisceaux hertziens ou encore les satellites apportent une infrastructure déjà en place. Il est nécessaire de s'adapter à l'existant, bon ou mauvais, et de se plier au monopole exercé de fait par France Télécom en France.

L'intérêt de regrouper des machines sous forme de réseaux est multiple :

- Partager des ressources en rendant accessibles à chaque membre du réseau divers programmes, données ou équipements.
- Assurer une grande fiabilité en introduisant de la redondance dans le réseau ; par exemple par la possibilité d'avoir plusieurs (au moins deux) chemins d'accès entre deux machines du réseau, par copie d'un même fichier sur plusieurs disques ou par utilisation de plusieurs unités centrales permettant de prendre le relais si l'une d'elles tombe en panne.
- Réduire les coûts : les micro-ordinateurs ont un ratio prix/performance avantageux alors que les grandes installations apportent plus de rapidité, environ dix fois plus, mais sont mille fois plus chères. Un bon compromis est d'avoir des micros personnels puissants et plusieurs serveurs de fichiers, le tout sur un réseau.
- Apporter un moyen de communication entre personnes séparées par de longues distances. Par exemple, pour l'écriture d'un rapport en commun à l'aide du courrier électronique.
- Permettre l'accès à des programmes ou à des bases de données distantes.

Le domaine de la conception des réseaux informatiques est très large, mais peut être séparé en deux catégories distinctes : les réseaux longues distances (WAN, Wide Area Network) pour connecter des sites distants et les réseaux locaux (LAN, Local Area Network) pour relier des machines sur un même site donc couvrant des distances relativement courtes. Tous deux (LAN et WAN) sont complémentaires puisqu'un réseau longue distance assure la communication entre divers réseaux locaux. En

effet, un WAN peut être vu comme un graphe dont les nœuds sont des LANs, comme le montre la figure 17. Concevoir un WAN ou un LAN sont des problèmes différents sur le plan pratique. Les équipements utilisés sont différents, de même que les règles de configuration. D'ailleurs, les concepteurs de WANs n'étudient pas les détails des LANs qu' ils incluent mais en font des abstractions en les considérant comme de simples noeuds d'un WAN. De même, les créateurs de LANs doivent considérer le WAN auquel leur réseau local sera connecté mais ils le font uniquement en choisissant une boîte de connexion appropriée au WAN.

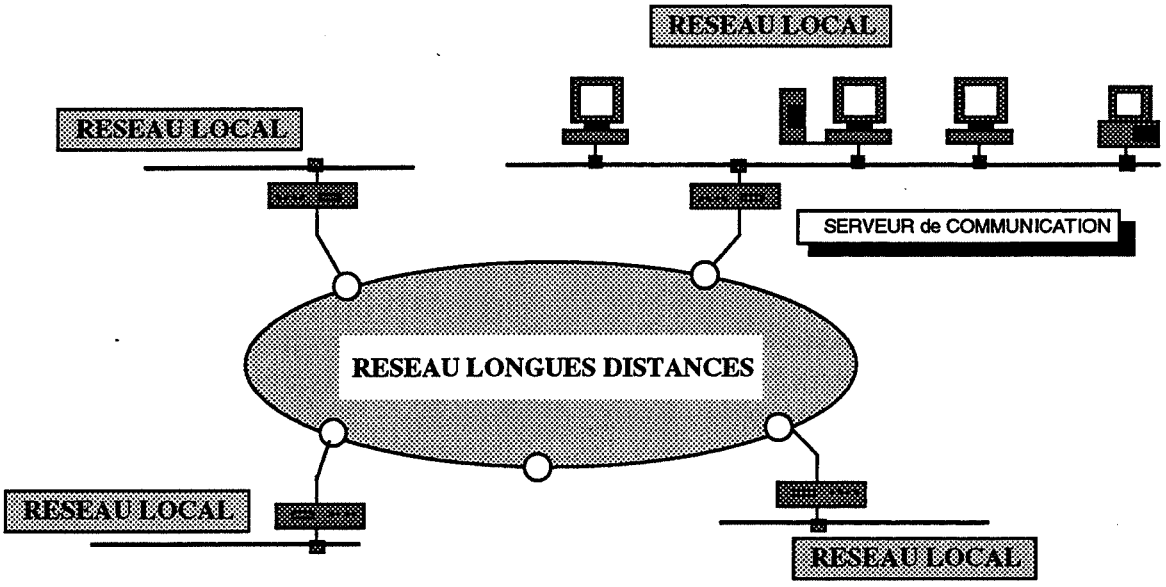


figure 17 : Interconnexion des réseaux longues distances (WAN) et des réseaux locaux (LAN).

Permettre à deux systèmes de dialoguer implique bien évidemment de les relier par un support physique qui sera branché sur des ordinateurs. A priori, rien de bien compliqué et pourtant les écueils sont nombreux, rendant la tâche de conception de réseau difficile à traiter. Voici une liste des écueils que nous avons rencontrés :

- Les données issues d'un équipement informatique ne peuvent pas rester sous leur forme d'origine pour être transmises sur un support quelconque à un autre équipement. Ainsi, chaque ordinateur doit comprendre un adaptateur assurant la conversion dans les deux sens, lors de l'émission ou lors de la réception d'un message.

- Il ne suffit pas à deux systèmes informatiques de s'être entretenus sur les modalités matérielles du dialogue pour pouvoir communiquer. Un accord doit se faire sur les procédures à suivre lors du transfert de données et assurer qu'ils donnent le même sens aux informations qu'ils échangent autrement dit qu'ils parlent le même langage au niveau logiciel cette fois. Par conséquent, il faut définir la forme sous laquelle les données seront transmises : bit par bit, par paquets de bits ..., décider à quel moment il est possible d'émettre (émetteurs et récepteurs doivent être prêts) et surveiller la transmission du message en s'assurant que les données sont entièrement parvenues à leur destinataire. Toute transmission est donc effectuée suivant un protocole précis.

- Les données transmises sont numériques et même binaires. Lorsque la transmission de ces données doit être réalisée via des voies conçues initialement pour le transport de signaux analogiques tels que le réseau téléphonique commuté par exemple, elle doit être précédée par une étape de modulation transformant du numérique en de l'analogique. De même, à la réception du message, une étape inverse est nécessaire : la démodulation. Les modems permettent ces opérations.
- Les besoins sont extrêmement diversifiés : du minitel sur lequel on tape un caractère de temps à autre au transfert d'énormes fichiers. De plus, un ordinateur peut être connecté de multiples manières : par des lignes directes, par des modems et les lignes téléphoniques, par des réseaux locaux, ..., chacune d'elles ayant ses propres caractéristiques. Pour les réseaux locaux également, les façons de les réaliser sont nombreuses, comme nous le verrons au chapitre 1.1.2.
- Les plus importants fabricants d'informatique ont chacun d'eux développé leur propres ordinateurs et surtout possèdent leurs propres méthodes et équipements d'interconnexion de leurs machines. Heureusement, des normes ont été édictées, mais il n'en reste pas moins encore quelques divergences entre ces constructeurs, ceci ne facilitant pas la tâche des concepteurs de réseaux.
- Les techniques évoluent très vite. Les nouveaux produits entrant sur le marché des ordinateurs et celui des technologies de communication progressent tous deux.

2.1.1 Quelques mots sur les technologies à la disposition des concepteurs de WANs.

France Télécom met à la disposition d'utilisateurs potentiels de réseaux longues distances plusieurs possibilités.

Le réseau téléphonique, plus précisément appelé le réseau téléphonique commuté (RTC) utilise une technique dite de commutation de circuits. Cette dernière permet à la demande d'un utilisateur potentiel d'établir une liaison temporaire par un circuit qui lui est réservé pendant toute la durée de la communication. Les centraux téléphoniques jouent le rôle de commutateurs en aiguillant l'appel sur une ligne libre. L'inconvénient de ce moyen de communication est qu'il est coûteux dès qu'il s'agit de transmettre de grandes quantités d'information et de surcroît lent. De plus, le taux d'erreur moyen n'est pas négligeable : le rapport entre le nombre de bits erronés et le nombre de bits émis est de 10^{-4}. Nécessitant de plus l'usage de modems, le réseau téléphonique est mal adapté à la transmission de données numériques. D'ailleurs, il n'a pas été conçu à cette fin.

En revanche, le réseau Transpac, opérationnel depuis 1978, a été créé pour ce type d'opérations. Il fait appel à des ordinateurs spécialisés assurant les fonctions de concentration et de commutation des données. Transpac est doté de X25 définissant l'interface entre le réseau et les équipements informatiques. Son mode de transmission des données est la communication par paquets. Comme son nom l'indique, les données sont transmises par paquets de bits. Avant l'envoi d'un message, un paquet de reconnaissance est envoyé sur le réseau pour repérer les chemins de transmission les moins encombrés. Ce réseau est bien plus rapide (48 Kbits par seconde) et plus fiable (le taux d'erreur est de l'ordre de un bit pour 10^{-9} paquets transmis) que le réseau téléphonique. Le coût d'utilisation de Transpac est déterminé non pas en fonction de la distance mais en fonction du volume d'information transmis à un débit donné.

D'autres moyens de transmission sont également mis à la disposition des utilisateurs potentiels par France Télécom tels que Transcom, transmission à moyen débit (64Kbits/s) sur liaisons commutées sur le RTC, Transdyn, transmission avec une large gamme de débits sur liaisons commutées à base du réseau Télécom-1 (satellite géo-stationnaire) et Transfix, transmission sur liaisons permanentes mais d'utilisation moins fréquente et plus spécifiques, Numéris (RNIS), réseau numérique à intégration de services pour des liaisons bidirectionnelles symétriques commutées [Dicenet 88]. Il est également possible de louer auprès de France Télécom des lignes spécialisées.

En fait, parmi les réseaux longues distances, plusieurs catégories peuvent être distinguées :

- *les réseaux publics* (tels que Transpac) *ou privés* (Tynnet, Telenet) *de transmission de données*, fondés le plus souvent sur la norme X25 du CCITT et généralement connectés entre eux au plan international.
- *Les réseaux privés* qui offrent en particulier à leur usagers des applications de transfert de fichiers et de messagerie. Le transport de données se fait par des liaisons spécialisées et par satellites ou via des réseaux publics de données. Par exemple, EDF-GDF a son propre réseau : RETINA.
- *Les réseaux d'organismes de recherche* tels que ARPANET aux USA, JANET en Angeterre, DFN en Allemagne, REUNIR en France, RARE en Europe, etc.
- *Les réseaux internationaux de messagerie* : EARN, BITNET, CSNET, FNET, USENET, etc.

2.1.2 Les technologies à la disposition des concepteurs de réseaux locaux.

Tous les grands constructeurs de systèmes informatiques tels que IBM, Digital Equipment, Unisys, Helwett-Packard, Xerox, Bull, ont développé leur propre architecture réseau, tout en définissant les moyens et les protocoles de communication pour leurs machines. A la fin des années 70, chaque machine, en fonction de sa marque, parlait un dialecte qui lui était propre. Ainsi une machine IBM était capable de dialoguer avec sa consoeur d'IBM mais pas avec celle d'un autre fabriquant. Ce problème est parfois résolu par l'ajout de passerelles (gateway), sorte d'interface jouant le rôle de traducteur. Avec l'arrivée des micro-ordinateurs) au début des années 80, le problème précédent s'est renforcé et existe encore même si c'est à un degré moindre. Les organismes de normalisation ont dû très tôt se pencher sur l'épineux problème des communications entre matériels hétérogènes. De cette tentative de normalisation est ressorti le besoin de distinguer différents niveaux allant du plus bas traitant des caractéristiques physiques de la connexion des équipements (brochage des connecteurs, valeurs des tensions) aux niveaux les plus hauts concernés par les protocoles de communication et les logiciels nécessaires à la présentation et à la compréhension des données. Le résultat est donc un modèle en sept couches présenté figure 18. Cette organisation stratifiée est désormais respectée par un grand nombre de fabricants de matériels informatiques. L'objet de chaque couche est d'offrir certains services aux couches plus hautes. Comme son nom l'indique, la couche de base (**physique**) définit le support physique pour le transport de données. La seconde couche concerne le contrôle des données transmises, c'est la couche **liaison**. Afin que les systèmes communiquent au travers d'un réseau, il a fallu définir une couche **réseau** chargée du routage des informations pour s'assurer du bon transit des messages au

travers des différents systèmes intermédiaires. La quatrième couche contrôle le transport des données entre deux systèmes d'où son nom de couche de **transport**. La couche suivante régule les messages de façon à alterner les phases d'émission et de réception, il s'agit de la couche **session**. La présentation des messages est prise en compte par la couche **présentation** qui "enveloppe" le message tandis que la couche **application** délivre de l'enveloppe le contenu du message. Pour avoir une description plus détaillée de ces septs couches, le lecteur pourra se référer à [Robin 88] ou à [Tanenbaum 90].

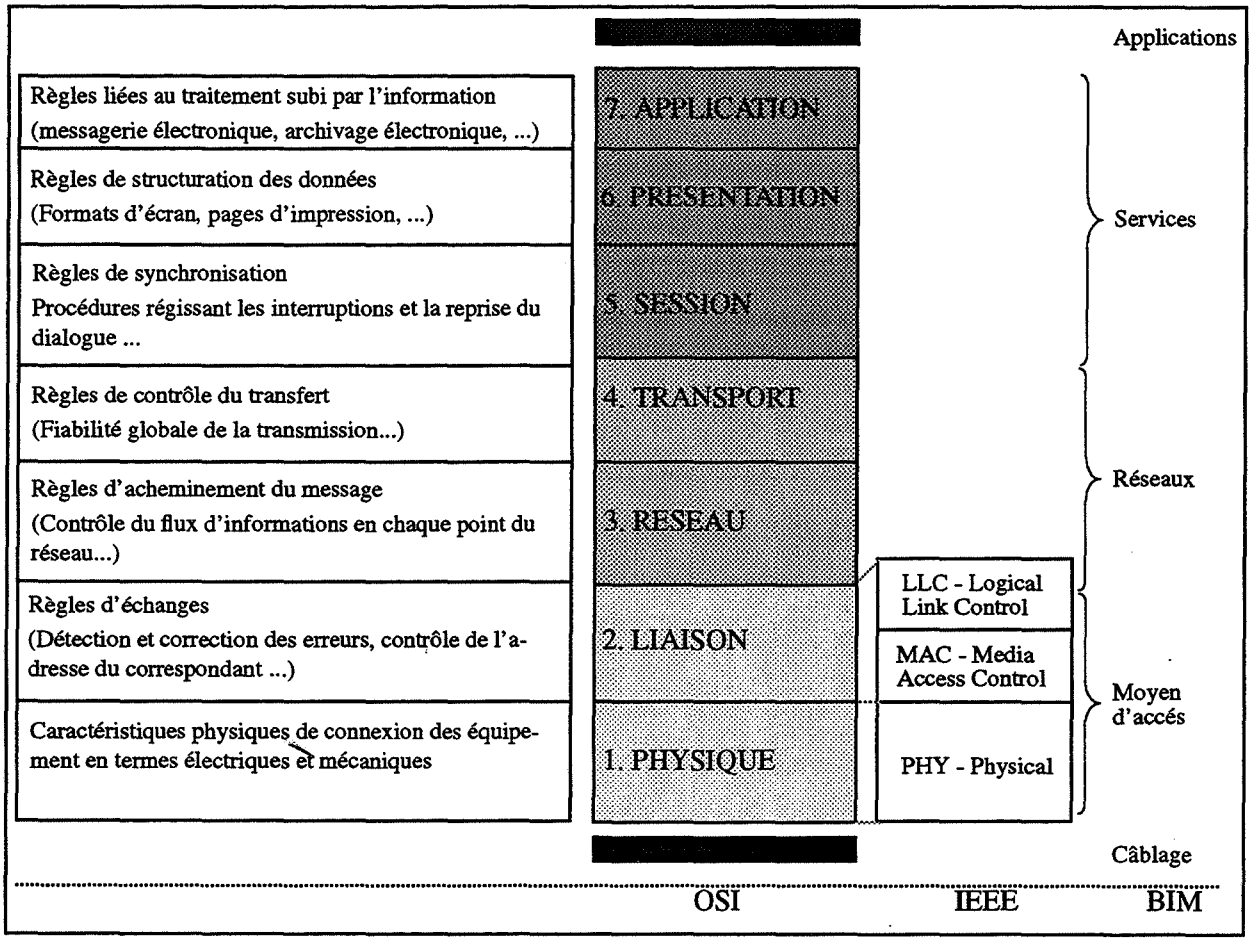


figure 18 : Couches fondamentales du modèle OSI (Open Standard Interconnection).

Pour les experts de BIM, le modèle OSI (Open Standard Interconnection) doit être réinterprété en distinguant cinq niveaux fondamentaux correspondant à une fonction majeure.

- Câblage : choix d'un type de câble (coaxial, paire torsadée, fibre optique, etc) et de la transmission (bande de base ou large bande).
- Accès : CSMA/CD, Token Ring, Token Bus, ...
- Réseau : connexionless, connexionfull, type de routage, ...

- Services : tous les services de base qui servent à construire des applications.
- Applications : les applications proprement dites.

Malgré cette norme, le besoin d'interfaces entre matériels de constructeurs différents est encore présent même si celles-ci sont plus simples à développer et à mettre en oeuvre. Le modèle OSI n'est pas une structure excessivement figée. Dans le cadre des réseaux locaux, par exemple, deux grandes catégories peuvent être distinguées : les réseaux dits Ethernet (répondant à la norme IEEE 802.3 reconnue par l'ISO) et les réseaux "à passage de jetons" (norme IEEE 802.4 ou 802.5).

Choisir un réseau local et son interconnexion à un réseau étendu suppose une sélection pour chacune des caractéristiques suivantes :

- Une topologie : les machines peuvent être connectées linéairement (bus) ou en étoile ou encore en anneau.
- Un support : pour le câblage (type de câble et de transmission).
- Une méthode d'accès au réseau : pour autoriser la communication simultanée de plusieurs équipements sur une même ligne. Par exemple : CSMA/CD, passage de jetons, etc.
- Un protocole de communication : SNA, DECNET, TCP/IP, OSI, etc.

Les paragraphes suivants vont présenter en détail ces différents points.

2.1.2.1 Les différentes topologies de réseaux locaux.

Il existe trois principales topologies pour l'interconnexion des points d'accès au réseau, présentées sur la figure ci-dessous. Elles peuvent bien entendu être combinées entre elles.

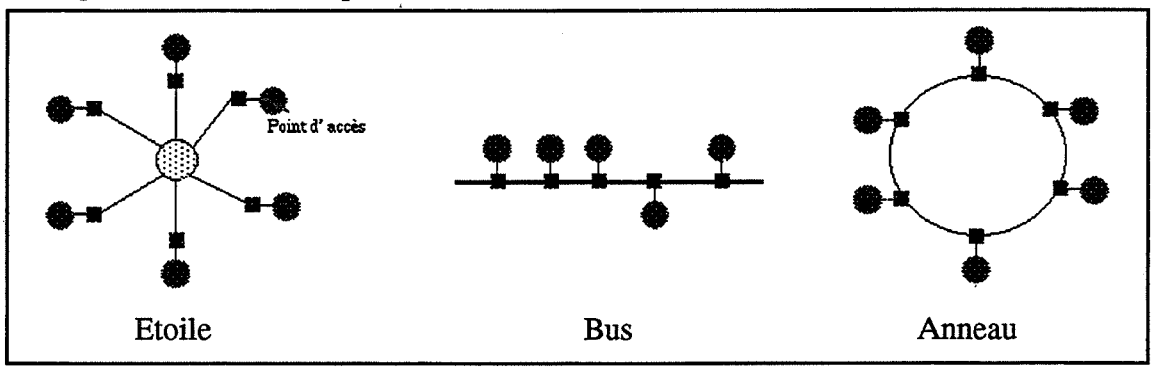


figure 19 : Topologie des réseaux.

La topologie en étoile assure un transfert d'information généralement en point à point. Du fait de sa centralisation, cette structure est fragile et peu souple. De plus, la longueur de câblage est limitée. Par contre, son exploitation en est facilitée et il est commode d'ajouter ou de retirer des machines sur un tel réseau. Les centraux téléphoniques utilisent cette technique.

Lorsque les machines sont reliées par un bus, elles sont connectées en multipoint et les informations émises sont diffusées sur le réseau et captées par chacune d'entre elles. Dans ce cadre de réseau, la longueur de câblage est minimale mais son exploitation est plus délicate du fait de la décentralisation des équipements. Dès que la charge augmente, les performances diminuent. De plus, il est difficile de détecter les coupures dans un tel réseau.

Dans un anneau, l'information circule dans un seul sens sur la boucle. Il autorise de grandes distances du fait de la regénération du signal à chaque noeud. Par contre, cette structure est fragilisée par la possibilité de rupture de la boucle. Il faut également prévoir, contrairement au cas de bus, la possibilité de retrait explicite des messages émis sur la boucle. Cette topologie nécessite une station de supervision centralisée ou distribuée.

2.1.2.2 Le câblage.

Plusieurs types de câble sont à la disposition des concepteurs de réseaux : coaxial (fin ou épais), paire torsadée, fibre optique, etc. Ils sont comparés dans le tableau ci-dessous.

Type de câble	Avantages	Inconvénients	Débit
Paire torsadée	* très répandue * bon marché * facile à mettre en oeuvre	* sensible aux perturbations éléctro-magnétiques	* faible : Mbits/s sur courtes distances
Câble coaxial type Ethernet : thin ou thick	* faible atténuation	* difficile à poser * coûteux * sensible aux perturbations éléctro-magnétiques	* élevé : 10 Mbits/s
Fibre optique	* atténuation supérieure à celle des coaxiaux, permettant de grandes longueurs sans répéteur * totale immunité vis à vis des perturbations électro-magnétiques	* coûteux * ne permet pas la réalisation directe d'un réseau en bus multipoint	* très élevé : de quelque 10 Mbits/s à 1 Gbits/s
Ondes radio-électriques	* distances allant de quelques centaines de mètres à quelques kilomètres	* coûteux * réglementation	* élevé

figure 20 : Comparaison des différents types de câbles.

Pour la transmission, une distinction est à faire entre celle à bande de base et celle à large bande. La bande de base consiste à appliquer directement sur le support de transmission le signal de données

ou signal en bande de base, sans aucune modulation particulière. C'est ce mode qui est employé pour les réseaux Ethernet. La large bande, quant à elle, consiste à moduler les signaux de base selon toute une gamme de fréquences. Ceci permet le transport simultané de plusieurs signaux, en utilisant des techniques de multiplexage fréquentiel, autorisant ainsi plusieurs communications à la fois. Ce type de transmission est fréquemment utilisé sur câble coaxial ainsi que sur fibre optique.

2.1.2.3 Les méthodes d'accès au réseau.

Sont présentées ci-après les deux principales méthodes d'accès.

1. CSMA/CD (Carrier Sense Multiple Access/Collision Detection).

Lorsque la méthode d'accès est CSMA, toutes les stations sont à l'écoute. Si une station reconnaît qu'un message lui est adressé, elle se cale sur la fréquence du câble et établit la connexion avec l'émetteur. Si deux stations émettent en même temps, une collision survient. Chaque station va alors reémettre après un temps aléatoire, en principe différent pour chacune d'elles. Pour limiter les collisions, on impose à la station voulant émettre d'écouter pour vérifier qu'il n'y a pas déjà une transmission en cours avant de commencer à envoyer ses données. De plus, si la détection des collisions (CD) est ajoutée à cette méthode CSMA, une station qui émet est à l'écoute continuellement. Si elle n'entend que son message, tout va bien. Sinon (brouillage), il y a une collision, elle interrompt son émission qu'elle reprendra plus tard.

Cette méthode est efficace quand le temps de propagation est court par rapport à la transmission d'un paquet sur le réseau et quand le réseau fonctionne à faible charge.

Mais il est impossible de garantir un temps de réponse, ce qui peut s'avérer inacceptable dans certains cas et en particulier pour les applications "temps réel". De plus, les performances chutent quand la charge du réseau est trop importante.

2. Méthode à jeton.

Pour les réseaux à passage de jetons, la méthode consiste à faire circuler un jeton sur la ligne reliant les ordinateurs. Lorsqu'un noeud désire émettre, il doit attendre d'être traversé par un jeton libre. Il commence alors à émettre tout d'abord en changeant l'état de ce jeton, le rendant occupé. Ensuite il envoie son message en indiquant l'adresse de son correspondant. Ce message transmis est lu par tous les noeuds du réseau et le destinataire le copie dans sa mémoire, tout en le laissant faire le tour de l'anneau pour repasser par son émetteur, pour confirmer la bonne réception. Le noeud émetteur remet alors le jeton à l'état libre et retire son message. Le principal promoteur de cette méthode est IBM avec son anneau nommé Token Ring.

Contrairement à la méthode CSMA/CD, avec celle à jeton, le temps de réponse maximum est connu. Ceci. autorise les applications dites temps réel telles que le transport de la voix ou le contrôle de processus. Les réseaux implémentant cette technique offrent également un bon comportement sous forte charge.

Par contre, le temps d'accès au réseau peut être long et nécessite de garantir l'unicité et la présence du jeton. Si une station est en panne, le jeton peut être perdu si c'est justement celle-ci qui le possède. Un station de supervision est nécessaire pour en recréer un, évitant ainsi une regénération multiple par les différentes stations.

2.1.2.4 Les protocoles de communication.

Pour le choix des protocoles de haut niveau, plusieurs possibilités sont offertes.

Tout d'abord, les protocoles spécialisés proposés par un constructeur sont nombreux. Par exemple, SNA : Systems Network Architecture d'IBM, DNA : Digital Network Architecture par Digital Equipment, DSA : Distributed System Architecure de Bull ou encore DECnet de DEC. Un protocole de ce type est aujourd'hui encore limité à un sous-ensemble d'équipements homogènes d'un même fabriquant.

Des protocoles internationaux sont également disponibles. C'est le cas de OSI de l'ISO. Si les protocoles OSI doivent s'imposer à terme, suite à une très grande lenteur de la normalisation de protocoles complexes, il est illusoire aujourd'hui d'envisager une solution OSI offrant une gamme très large de services avec un fonctionnement stable et éprouvé.

Enfin des protocoles ouverts offrent une large gamme de services, de bonnes performance et une voie de migration vers OSI. C'est le cas de TCP/IP (Transmission Control Protocol/ Internet Protocol). Un autre protocole ouvert est XNS. XNS est le protocole de communication de Xerox sur ses réseaux Ethernet. C'est une évolution de TCP/IP. Si XNS est un protocole de qualité offrant de très bonnes performances, il présente le désavantage fondamental de n'offrir au niveau du standard aucun service de base parfaitement normé. Ce protocole est de plus limité aux LANs.

Lorsque l'on parle de TCP/IP, on se réfère en fait à une hiérarchie de protocoles résumables ainsi :

- IP est grossièrement l'équivalent du niveau 3 (réseau) des couches OSI. Il assure l'interconnexion de différents réseaux du type Arpanet, Ethernet, X25, ...
- TCP, quant à lui, est l'équivalent du niveau transport de l'OSI. Sa tâche est d'assurer le contrôle fiable des liaisons entre émetteurs et destinataires. TCP permet le transport séquencé de longueur indéterminée. Un protocole plus simple UDP est possible à ce niveau. UDP permet le transport à haut débit de datagrammes.

Au-dessus de TCP, se situent diverses applications : TELNET pour l'accès interactif aux machines, FTP pour le transfert de fichiers, SMTP pour la messagerie, des services UNIX tels que : rlogin, rsh, rcp, rwho, etc et des services avancés "ouverts" comme : NFS, YP, RPC, etc.

Enfin, TCP/IP est utilisable sur tout type de réseau allant des LANs aux WANs, y compris les réseaux de PC par l'intermédiaire de PC-NFS.

2.1.2.5 Solutions existantes.

Le marché offre actuellement les solutions suivantes :

Solution	Méthode d'accès	Vitesse Mbs	Avantages	Inconvénients	Caractéristiques
Ethernet	CSMA/CD IEEE 802.3	10	* standard *homologation OSI * protocole prouvé et robuste * bien conçu pour le transfert de grands "paquets" à haut débit * un des meilleurs rapports prix/performance * grand nombre d'options de câblage (coaxial, paires torsadées, fibre optique, onde)		* topologie en bus
AppleTalk	CSMA/CA	0.23	* coût particulièrement bas	* lent * n'autorise qu'un petit nombre de noeuds * impensable de l'utiliser pour la réalisation d'une architecture réseau de type client/serveur	
StarLAN	CSMA/CD IEEE 802.3	1		* conçu principalement pour fonctionner sur des systèmes de câblage en paires torsadées non blindées d'AT&T	* version à 1 Mbits/s d'Ethernet * configuration en étoile dont la longueur des branches est au maximum de 250 m
Token Ring	Technique à jeton IEEE 802.5	4 (16)		* 2.5 fois plus lent qu'Ethernet * prix moyen de connexion 33% plus coûteux qu'Ethernet * concevable uniquement dans un environnement IBM homogène * déconseillé dans le cas d'un réseau hétérogène	* conçu par IBM * topologie en anneau

Solution	Méthode d'accès	Vitesse Mbs	Avantages	Inconviénients	Caractéristiques
Token Bus	IEEE 802.4	5/10		* trop d'options au niveau 1 * une très forte dépendance par rapport au type de médium * une limitation aux applications du type "Process control"	* développé par General Motors * topologie en bus
FDDI Fiber Distributed Data Interface	Méthode à jeton	100	* standard des réseaux locaux à haute vitesse * possibilité de gérer 500 stations avec une interdistance de 2 km *redondance du réseau du fait du double anneau * basculement automatique sur le réseau de secours en cas de panne	* coûteux	* anneau * câblage en fibre optique double paire

2.2 Spécification de l'application.

2.2.1 Délimitation.

Comme le domaine de la conception de réseaux est très large, il est nécessaire de le restreindre à une taille acceptable pour pouvoir l'implémenter dans un système à base de connaissances. La première restriction porte sur les deux types de réseaux, WAN et LAN puisque les méthodes de conception sont différentes. NEST ne permettra que la conception de réseaux locaux. Ce choix étant fait, le domaine est grandement réduit mais il n'en reste pas moins encore difficilement traitable surtout dans le cadre d'un prototype. En fait, la conception de réseaux locaux est constituée de deux principales tâches.

La première est la configuration physique d'un réseau qui consiste à déterminer et localiser géographiquement dans les bâtiments tous les composants physiques nécessaires à la conception d'un réseau tels que les câbles, les boîtes de connexion, les éléments de filtrage, les petits connecteurs, etc. De nombreuses contraintes sont à respecter telles que l'architecture du (ou des) bâtiment (s), les besoins du client : machines à connecter et leur localisation, les applications qui devront "tourner sur" le réseau, le budget, parfois la description d'un réseau ou d'un sous-réseau préexistant et enfin les limites techniques imposées par les équipements actuellement à la disposition du concepteur de réseaux.

La deuxième tâche englobée dans la conception de réseau est la configuration logique qui consiste à choisir un protocole, à déterminer la valeur des paramètres, etc. L'objectif de cette configuration est de permettre aux ordinateurs de communiquer, de faire tourner des application de façon satisfaisante, c'est à dire principalement avec un temps de réponse court. Pour cette configuration, des contraintes doivent aussi être vérifiées telles que les caractéristiques des différents ordinateurs, les applications qui devront tourner, les limites techniques des équipements.

Cette seconde tâche est partiellement résolue par le choix d'une méthode d'accès et d'un protocole d'interconnexion. Nous nous limiterons, de fait, à la configuration physique d'un réseau. Par contre, les critères de qualité tels que une bonne vitesse de transmission des données, une absence de surcharge pour assurer un trafic faible ou tout au moins localisé aux sous-réseaux, des possibilités d'extension du réseau réalisé, etc, seront vérifiées.

Diverses méthodes d'accès sont à la disposition des concepteurs de réseaux : Ethernet, Token Ring, Appletalk, FDDI, ... De même que plusieurs protocoles d'interconnexion sont disponibles : SNA, DECNET, TCP/IP, OSI, ... Le choix pour NEST s'est porté sur Ethernet et TCP/IP que les experts interrogés lors de l'acquisition des connaissances utilisent. Ethernet est le plus répandu, peut tourner sur tout type de média (coaxial, paire torsadée, fibre optique) et offre de bonnes performances (10 Mbps). TCP/IP est un protocole éprouvé qui offre aujourd'hui une très large gamme de services de base. Toutes les nouvelles machines disponibles sur le marché ont une implémentation de TCP/IP ou en disposent en option.

2.2.2 Fonctionnalités.

Plusieurs fonctionnalités sont souhaitables quant à la réalisation de NEST. Cette section s'appuie sur le rapport [Balfroid 89b] qui a été écrit en début de projet afin de décrire les spécifications de l'application.

1. Un outil de conception.

Cet outil est évidemment le coeur du projet. L'objectif est de concevoir informatiquement un ou plusieurs réseaux respectant les contraintes (voir paragraphe 1.3.2. pour la description de ces contraintes) tant techniques qu'issues des exigences d'un client. Ce système devra être capable d'incrémenter un dessin de réseau déjà existant. Mais il doit être aussi apte à concevoir un réseau "en partant de rien", c'est à dire sans réseau préexistant.

Les experts ne considèrent pas toujours toutes les contraintes lors de la résolution d'un problème. En effet, certaines ont une importance plus grande que d'autres et doivent donc impérativement être satisfaites même si parfois c'est au détriment d'autres. C'est principalement le cas des contraintes techniques imposées par les équipements physiques ou par la technologie Ethernet. Par exemple, lors d'emploi de câble coaxial fin (thin), la longueur des segments de câble ne doit pas dépasser, impérativement, 185 mètres. Par contre, par exemple, la contrainte du budget peut être parfois détournée au

profit d'une amélioration de la qualité du réseau. Le système devra introduire cette caractéristique dans son raisonnement.

2. Un outil d'analyse.

Cet outil, comme son nom l'indique, doit permettre d'analyser un réseau afin de tester s'il satisfait aux contraintes, principalement celles imposées par les diverses technologies. Ainsi doit-il être apte à juger la qualité d'un réseau et à fournir une évaluation critique indiquant les avantages aussi bien que les inconvénients de ce réseau.

3. Un outil interactif.

L'objectif ici est de permettre une interaction entre l'utilisateur et le système principalement en autorisant l'utilisateur à modifier le réseau proposé. Le système tirera des conséquences de ces changements. Mis à part une aide à la compréhension des réseaux, cette possibilité peut aider dans la prédiction des effets d'une extension future d'un réseau.

4. Un outil d'enregistrement.

Cet outil devra assurer la sauvegarde des problèmes résolus par NEST. Ces enregistrements seront utiles pour la réalisation des dossiers clients et la reconsidération de ces réseaux pour résoudre de nouveaux problèmes à partir de ces réseaux. Pour faire ces sauvegardes, il est nécessaire de délimiter les informations à sauvegarder et de répondre à la question : faut-il seulement enregistrer le réseau proposé ou aussi la manière de raisonner du système pour parvenir à ce réseau ?

5. Un générateur de rapport.

Les experts doivent produire des documents pour leurs clients. Ainsi serait-il utile pour eux que le système soit capable d'écrire des rapports décrivant le réseau développé ainsi qu'une approximation des coûts.

6. Un outil didactique.

Cet outil devra apporter un mécanisme d'explication du raisonnement suivi par le système tout en respectant le modèle utilisateur, c'est à dire que les explications devront être adaptées à l'utilisateur, que ce soit un non-initié au domaine des réseaux ou un expert confirmé. En posant des questions au système sur comment ou pourquoi une décision a été prise, l'utilisateur peut acquérir certaines connaissances du domaine.

A partir des spécifications souhaitables pour le système, énoncées ci-dessus, un plan de travail a été établi. Ce dernier fixe une première étape qui est la réalisation d'une maquette pour l'analyse d'un réseau physique donné. La deuxième phase est la réalisation d'un premier prototype qui s'achève au bout de ces trois premières années du contrat du projet Esprit MMI^2. Ce prototype est une extension de la maquette et présente donc les deux possibilités : l'analyse d'un réseau et la conception de réseaux à partir des spécifications des besoins d'un client : plan d'un ou plusieurs bâtiments, localisation des machines, budget, etc.

2.3 Acquisition des connaissances auprès des experts de BIM.

Très peu de connaissances peuvent être extraites d'ouvrages écrits pour la configuration physique de réseaux Ethernet - TCP/IP. Nous avons donc dû interroger des experts dans le domaine, en l'occurrence ceux de la société belge BIM.

Plusieurs méthodes ont été employées afin d'obtenir les connaissances auprès des experts de BIM. Tout d'abord, des interviews ont permis de se faire une première idée de l'activité de conception de réseaux. L'analyse de cas déjà traités a permis aux experts de nous présenter des problèmes qu'ils avaient résolus dans le passé. Cette méthode n'est pas la meilleure pour assurer une bonne diffusion de leur activité car ils expliquent généralement la solution retenue et non les alternatives auxquelles ils avaient peut-être pensé avant de se fixer sur une solution finale. Par contre, l'observation de sessions de conception sur des problèmes actuels permet de pallier cet inconvénient. F. Darses a menée une autre expérience : la comparaison de l'activité de conception de réseaux de plusieurs personnes ayant un degré de compétence différent dans le domaine (débutants, novices et experts). Enfin une analyse de la classification des problèmes de conception de réseaux a été conduite. [Darses & al 90] décrit en détails ces différentes méthodes ainsi que les résultats obtenus par l'emploi de ces dernières.

A partir de cette phase d'acquisition des connaissances, les éléments de base nécessaires à la conception de réseau ont pu être extraits. De plus, elle a montré que la conception de réseaux implique la vérification de nombreuses contraintes et qu'elle relève d'une tâche planifiée.

2.3.1 Les éléments de base.

Un réseau est constitué de différents éléments techniques tels que des câbles, des boîtes de connexion, des connecteurs, des ordinateurs. D'autre part, la topologie des bâtiments est nécessaire aux concepteurs de réseau pour évaluer l'étendue du problème, déterminer l'architecture du réseau, choisir les différents composants physiques en fonction des dimensions, etc. Ces deux types d'éléments de base sont listés ci-dessous. Seules les classes principales sont notées. En effet il serait trop long de fournir ici une liste exhaustive de ces éléments. Pour plus de précision sur le contenu de chaque classe le lecteur pourra se référer à l'annexe 1, la base de données de NEST, où tous les éléments sont présentés avec leurs caractéristiques.

•Les éléments techniques :

- les machines,
- les imprimantes,
- les serveurs de terminaux,
- les modems,
- les terminaux,
- les passerelles (Gateways[1]),

1. Je prie le lecteur de m'excuser pour avoir conservé certains termes anglais pour certaines boîtes de connexion.

- les hubs,
- les fan-outs,
- les bridges,
- les T,
- les thinnet-tap-système,
- les splitter-boxes,
- les prises,
- les liens U,
- les logiciels,
- les mémoires,
- le réseau.
- les répéteurs,
- les routeurs,
- les câbles,
- les transceivers,
- les barrettes (Barrels),
- les commutateurs (Switches),
- les petits connecteurs,
- les terminateurs,
- les disques,
- les cartes,

• La structure du site :

- les bâtiments,
- les pièces,
- les murs,
- les gaines techniques,
- les étages,
- les couloirs,
- les départements,
- les montées verticales (escaliers, ascenseurs ou gaines techniques).

2.3.2 Les contraintes.

De nombreuses contraintes doivent être vérifiées afin de s'assurer qu'un réseau est "correct". Ces différentes contraintes ont été mises en évidence par l'acquisition des connaissances et peuvent être classifiées en plusieurs sous-groupes.

2.3.2.1 Les exigences d'un client.

Un client fournit aux experts les différentes données de son problème telles que l'information sur le ou les bâtiments, les informations sur le réseau désiré et parfois sur le réseau préexistant et enfin son budget. Ces différentes contraintes sont détaillées ci-après.

L'information sur les bâtiments :

- la forme et la structure de chacun des bâtiments du site à câbler,
- le nombre d'étages,
- la dimension des pièces,
- la localisation des couloirs,
- la localisation des gaines techniques que ce soit sur le plan horizontal ou vertical ou encore les gaines inter-bâtiments,
- les montées d'escalier ou d'ascenseur,

- les contraintes environementales telles que les environnements perturbés.

L'information sur le réseau :

- les machines, leur type et leur localisation géographique,
- les prises existantes et leur localisation,
- les applications qui devront tourner sur le réseau,
- les serveurs avec leur localisation, leurs fonctionnalités et les machines qu'ils servent,
- les unités professionnelles ou les zones à isoler,
- les performances attendues (vitesse, délai de réponse),
- les exigences technologiques afin d'imposer par exemple l'emploi d'un type de câble.

N.B. : Certaines contraintes ne seront pas toujours présentes dans tous les problèmes à traiter et ont une importance moins grande que d'autres.

La contrainte du budget : Il est préférable de ne pas dépasser le budget alloué par un client.

2.3.2.2 Les contraintes techniques.

L'activité de conception de réseaux est nécessairement contrainte par les limites techniques des équipements disponibles sur le marché. Ce paragraphe correspond aux différentes contraintes testées par l'outil d'analyse de réseaux de NEST et elles ont été présentées par F. Bafroid dans [Darses & al 90]. Elles seront également prises en compte par NEST, lors de la conception de réseaux.

Contraintes concernées par tout type de composant de réseau :

- être dans un environnement autorisé (rayons X, magnétisme, etc),
- être connecté à un ou plusieurs composants ayant le même type de connecteur mais de sexe opposé.

Contraintes sur les ordinateurs :

- être connecté à un élément connectable à une machine (T, transceiver, ...),
- avoir un nombre de cartes inférieur ou égal au nombre de cartes autorisé sur ce type de machine,
- les machines sans disque doivent être reliées à un serveur,
- les serveurs doivent avoir un disque,
- les machines et leurs serveurs doivent être sur le même sous-réseau,
- un même serveur ne doit pas servir trop de machines.

Conseils pour les machines :

- présence du serveur et des machines qu'il sert sur le même segment,

- bonne distribution des machines sans disque sur les différents serveurs,
- présence des machines d'une même unité professionnelle dans un même sous-réseau.

Contraintes sur les boîtes de connexion :

- être connectées à un élément connectable à une boîte (T, transceiver, ...).

Contraintes sur les hubs, les répéteurs multi-ports et les fan-outs :

- être au moins connecté à deux éléments de réseau.

Contraintes sur les fan-outs :

- pas de fan-outs successifs.

Contraintes sur les concentrateurs et les routeurs :

- être au moins connecté à deux éléments de réseau,
- avoir un nombre de cartes connectées inférieur ou égal au nombre de cartes autorisées sur ce type d'équipement,
- avoir les extensions logicielles et matérielles requises.

Contraintes sur les Splitter-boxes :

- contenir un nombre d'épissures inférieur ou égal au nombre autorisé.

Contraintes sur les patch-panels et les prises :

- connecter des éléments deux à deux.

Contraintes sur tout type de câble :

- ne pas passer à travers des murs non perforables.

Contraintes sur les câbles coaxiaux :

- avoir une longueur supérieure ou égale à la longueur minimale permise.

Contraintes sur les câbles coaxiaux épais (thick) :

- ne pas passer dans une gaine technique horizontale.

Contraintes sur les drop cables :

- ne pas avoir une longueur supérieure à la longueur maximale permise.

Contraintes sur les fibres optiques :

- contenir un nombre de fibres optiques inférieur ou égal au nombre autorisé pour ce type de câble.

Contraintes sur les logiciels ou matériels additionnels :

- être installé sur des composants sur lesquels ils peuvent l'être,

Contraintes sur les managers pour les bridges Retix :

- un seul par réseau,
- installables seulement si des Retix M sont installés sur le réseau.

Contraintes sur les segments de câble :

- ne pas contenir de boucle,
- avoir une longueur inférieure ou égale à la longueur maximale autorisée,
- le nombre de boîtes de connexion ou de machines sur le segment doit être inférieur ou égal au nombre maximal permis,
- pour chaque connexion fibre optique, avoir exactement deux jonctions à travers le même câble en fibre optique.

Contraintes sur les sous-réseaux :

- ne pas contenir de boucle,
- ne pas contenir plus de cinq segments dont deux liens.

Contraintes sur les réseaux :

- s'il n'y a pas de routeur ou de bridge implantant l'algorithme MST (Minimum Spanning Tree), le réseau ne doit pas contenir de boucle.

2.3.2.3 Les critères de qualité.

Des exigences sur la qualité des solutions proposées doivent être respectées : un moindre coût, une haute extensibilité, un trafic faible, un délai de transmission court, etc.

2.3.3 Quelques observations sur l'activité de conception de réseaux.

Il apparait clairement que l'activité de conception de réseaux relève de la planification d'une tâche. En effet, les experts planifient la configuration de réseaux en la considérant comme un but général à réaliser que l'on peut décomposer en plusieurs sous-buts. Leur première étape est de déterminer une architecture globale que pourra avoir le réseau en cours de construction. Cette étape regroupe les sous-buts suivants : détermination du nombre d'épines dorsales[2] nécessaires et de leur localisation, détermination des points de connexion entre ces épines dorsales et les éventuels sous-réseaux. La deuxième phase consiste à concevoir ces sous-réseaux et est également décomposables en plusieurs sous-sous-buts comme le montre le graphe ci-après. Ces différents buts interagissent entre eux, c'est à dire que le choix d'une solution pour l'un d'eux peut avoir des conséquences sur un autre. Par exemple, la sélection d'une localisation pour un câble dans une gaine technique a des conséquences sur le type de câble puisque du coaxial épais (thick) ne doit pas passer dans ce type de localisation. Ces interactions entre buts sont introduites dans le graphe par des simples flèches alors que la décomposition d'un but en sous-buts est représentée par des flèches en grisé. Une dernière étape pour la conception de réseau est de s'assurer de la bonne jonction entre les montées verticales des câbles et les câblages caculés sur le

2. Ce terme est couramment appelé backbone en conception de réseaux.

plan horizontal. Enfin une phase d'optimisation et de raffinement de la solution est parfois entreprise. Elle peut, par exemple, amener à proposer à un client des équipements additionnels pour assurer la redondance du réseau.

Ainsi la conception de réseaux locaux peut être vue comme une succession de trois ou quatre étapes, chacune d'elles étant accomplie par la résolution de plusieurs sous-buts. De plus, de nombreuses contraintes se sont révélées lors de l'acquisition des connaissances témoignant des interactions entre les différents buts à réaliser.

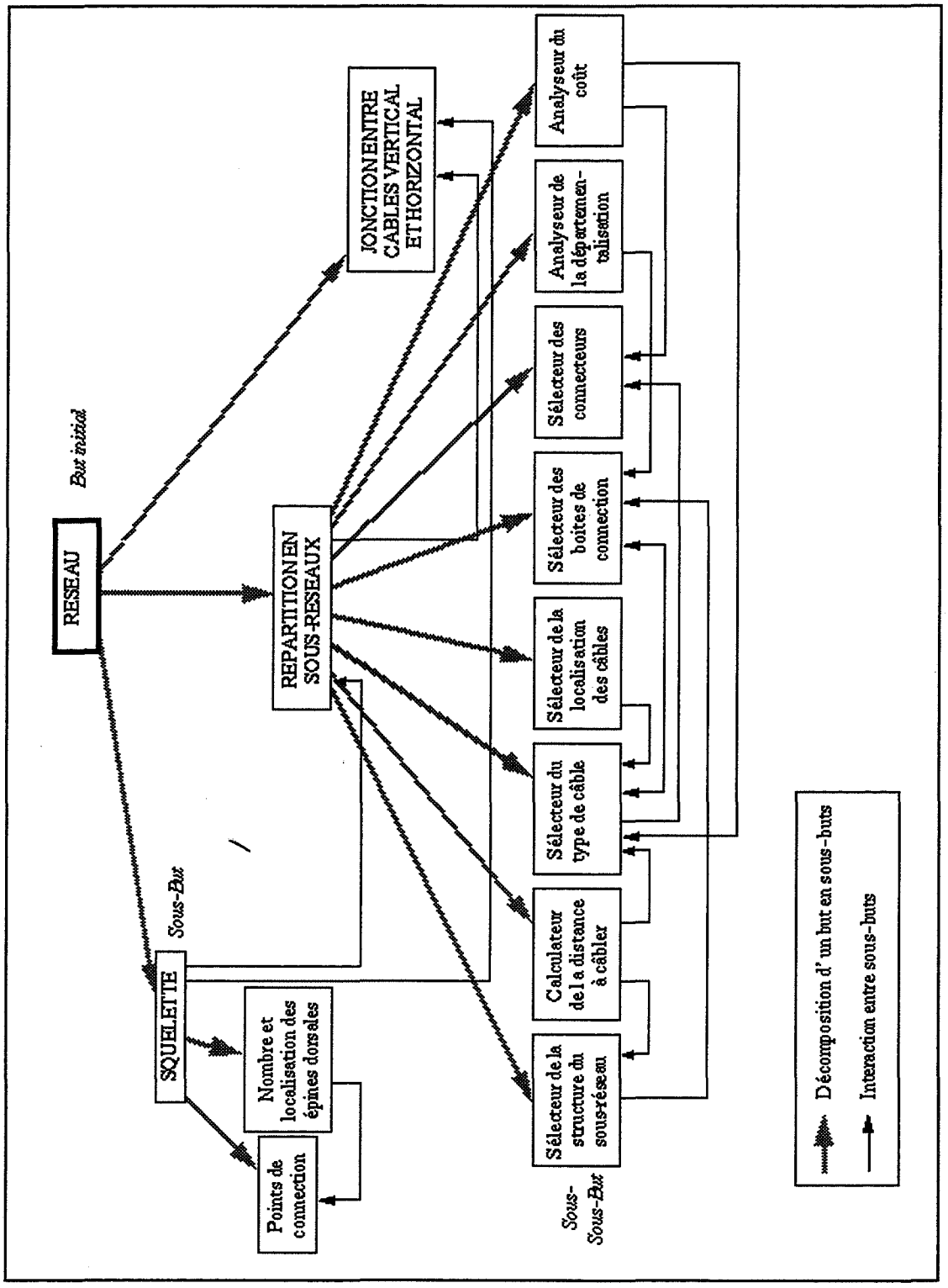


figure 21 : Conception d'un réseau : relations entre les buts.

CHAPITRE 3

Réalisation de NEST.

Avant d'entrer dans les détails de cette réalisation et plus particulièrement de l'outil de conception de réseaux, introduisons l'architecture générale de NEST.

3.1 Organisation de NEST.

3.1.1 L'architecture de NEST.

Comme NEST est constitué de deux outils distincts : un pour l'analyse d'un réseau et l'autre pour la conception, ces deux éléments vont se retrouver dans l'architecture présentée par la figure 22. Tous deux travaillent à partir de la définition du même concept. Ainsi la base de connaissances est commune à ces deux outils.

Les divers composants de cette architecture sont donc :

- la base de connaissances dont l'organisation sera amplement décrite dans les paragraphes suivants,
- un outil assurant l'analyse d'un réseau donné par l'utilisateur au moyen de l'interface graphique ou conçu par NEST,
- un outil résolvant le problème de la conception de réseau. Dès que les spécifications du site à câbler et des exigences de l'utilisateurs (budget, type de câble souhaité, etc) sont achevées, une requête de conception peut être envoyée au système. En retour, celui-ci va créer les objets nécessaires à la réalisation d'un réseau tels qu'une entité réseau, des câbles d'un type défini, des boîtes de connexions et des connecteurs. Ce réseau étant achevé, une requête d'analyse pourra être lancée une première fois et pourquoi pas, relancée après d'éventuels changements effectués par l'utilisateur du système.

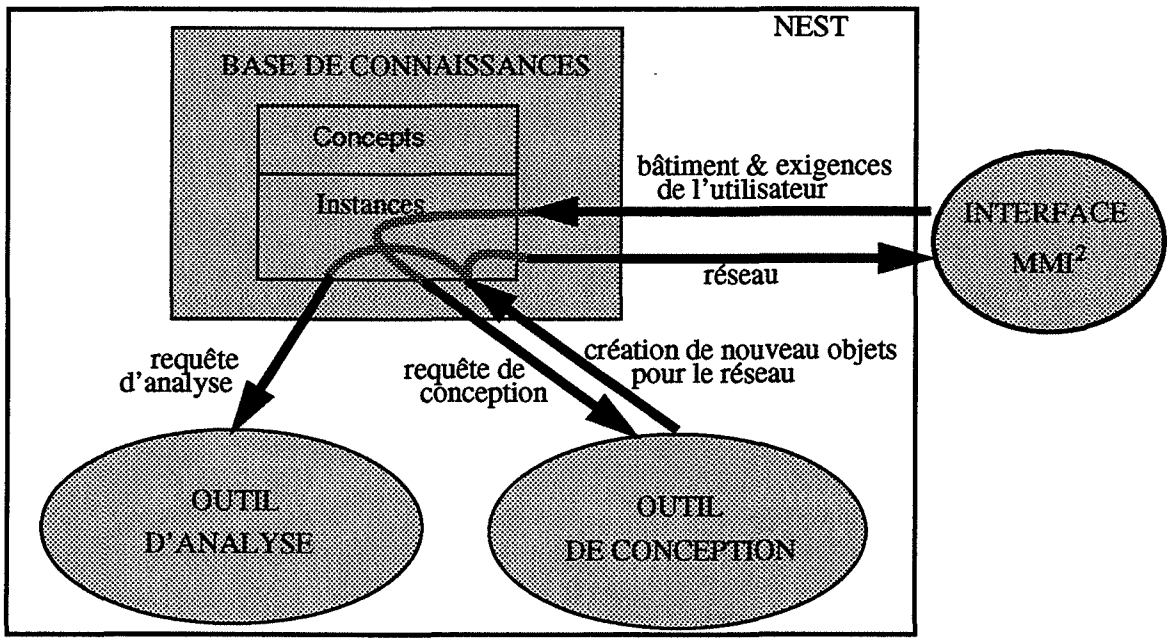


figure 22 : Architecture de NEST.

3.1.2 Répartition du travail entre les différents partenaires.

Comme déjà mentionné en introduction, NEST a été développé en collaboration avec F. Balfroid (BIM) et F. Darses (INRIA). Etablir une répartition du travail effectué n'est pas chose aisée étant donné que les réflexions, les décisions sur les points clés de la réalisation de NEST ont été prises en commun. A partir de ces concertations, l'implémentation de NEST a pu avoir lieu.

L'outil de conception de réseaux a été implémenté[1] en collaboration avec F. Balfroid. Parallèlement à de nombreuses discussions concernant le développement de cet outil, une répartition du travail a été établi. Celle-ci est mentionnée tout au long du paragraphe 3.4.2 page 135 : les modules réalisés par F. Balfroid sont suivi de FB sinon de CJ. De plus, les listings des programmes Prolog que j'ai réalisés se trouvent en annexe 6.

L'outil d'analyse de réseaux a été développé par F. Balfroid.

Pour la modélisation des éléments de base relevant du domaine d'application, je me suis intéressée à la représentation de la topographique des bâtiments alors que F. Balfroid a modélisé les différents éléments pouvant constituer un réseau (machines, boîtes de connexion, câbles, etc).

1. F. Darses, ici encore, étant une précieuse aide dans nos réflexions conduites tout au long de cette réalisation.

3.2 Modélisation des éléments de base.

Formaliser le domaine de la conception de réseaux locaux implique la modélisation de très nombreuses connaissances. Deux types de connaissances peuvent être distingués : les éléments de base et les connaissances opératoires apportant l'information permettant de résoudre des problèmes de conception de réseaux ; tous deux ont été présentés au chapitre un, deuxième partie, paragraphe 2.3. Ce paragraphe (3.2) présentera la modélisation des concepts alors que le paragraphe 3.4 montrera l'organisation des connaissances opératoires.

3.2.1 Avantages du modèle centré objets et de BIM_Probe pour modéliser les éléments de base.

Modéliser des connaissances nécessite le choix d'un langage le plus adapté au problème à traiter. Le langage que nous avions à notre disposition est BIM_Probe [BIM 90] qui est une couche orientée objets au dessus de ProLog_by_BIM, couplée à un langage de programmation par contraintes (PCL). BIM_Probe implémente les concepts de base des modèles centrés objets. Il autorise les hiérarchies de méta-classes, de classes et d'instances. Ces objets sont définis par des attributs qui peuvent être soit des propriétés, définitions d'un état statique de l'objet, soit des contraintes, exigences qu'un objet doit satisfaire, soit des méthodes attachées à cet objet, procédures qui définissent le comportement de cet objet. Ces attributs sont eux-mêmes décrits par des facettes. Les objets peuvent être organisés sous forme de hiérarchies "ISA" avec héritage multiple, héritage d'attributs, spécialisation d'attributs et attachement procédural. Un mécanisme puissant de BIM_Probe est la vérification de la consistance d'une base de données quand elle est construite ou modifiée [DeZegher 89]. Les contraintes système aussi bien que celles définies par le créateur de la base de données doivent être satisfaites pour que la base soit déclarée consistante. L'annexe 4 présente ce langage.

L'approche orientée objets est bien appropriée à la modélisation du contexte de la conception de réseaux parce que les concepts englobés dans ce domaine sont facilement décomposables en entités bien définies comme un réseau en noeuds et liens entre ceux-ci, comme un bâtiment en étages, pièces, murs et gaines techniques. D'autre part, le besoin d'organiser ce domaine en différentes hiérarchies et de permettre l'héritage de propriétés et méthodes est très grand, comme nous pourrons le constater dans les paragraphes suivants. Or le modèle centré objets est particulièrement bien appropriée pour représenter des hiérarchies d'objets et assurer l'héritage d'attributs et de méthodes. Je ne veux pas retracer ici les différents avantages de l'approche objets qui ont déjà été énoncés au paragraphe 2.5 page 50 de la première partie. Bien entendu, ils restent vrais pour l'utilisation du langage BIM_Probe. Par contre, d'autres avantages relevant du langage employé et plus particulièrement du langage de programmation par contraintes (PCL) sont à noter. (Ces avantages se retrouvent aussi dans certains langages orientés objets).

- Le premier avantage est dû au fait que la valeur d'une propriété peut être obtenue non seulement parce qu'elle a été affectée par l'utilisateur ou par le concepteur de la base de connaissances mais aussi par dérivation. Cette fonctionnalité permet en particulier de ne pas avoir à rentrer des données redondantes lors de la définition d'un problème à soumettre au système. Par exemple, une pièce est caractérisée, entre autre, par la donnée de l'étage auquel elle appartient, par l'intermédiaire de la propriété *set_in_floor*. De même, un étage contient une liste de pièces. Plutôt que d'avoir à donner deux fois la même information, une dérivation est introduite dans la classe *Floors*[1] pour la propriété *list_of_rooms* faisant la recherche des pièces appartenant à cet étage.

- Un autre intérêt de BIM_Probe est la possibilité de poser des contraintes sur les propriétés d'un objet. En effet, BIM_Probe permet la formulation de contraintes par des attributs spécifiques attachés aux objets. Celles-ci peuvent être intra-objets ou inter-objets. Des contraintes intra-objet ont été employées, entre autre, pour s'assurer que la valeur affectée à une propriété fait partie d'une liste donnée afin que l'utilisateur n'entre pas une valeur linguistiquement proche mais que le système expert ne sera pas capable d'interpréter, par exemple, Xrays au lieu de X_ray pour spécifier un environnement perturbé dans une pièce. Nous avons également défini d'autres contraintes dans le but de réduire les erreurs d'entrée des données ou pour vérifier que les composants satisfont à la technologie Ethernet. Par exemple, les connecteurs de boîte peuvent être spécifiques à un type de câble. Ainsi un *Thinnet_tap_systems* est destiné à la connexion d'une boîte et d'un câble coaxial fin. Or une contrainte sur cette propriété *for_cable_type* est que le type de câble donné doit appartenir à la classe des *Segment_cables*, c'est à dire à une des classes : *Thick_cables*, *Thin_cables*, *OF_cables* or *Twisted_cables*.

- Un troisième avantage de BIM_Probe est la possibilité d'écrire des exceptions. Par exemple, une exception a été définie pour la classe *Twisted_cables* pour redéfinir la propriété d'exception *min_max_length*.

- D'autre part, la modification d'une définition d'une propriété ou d'une méthode peut être effectuée au niveau d'une sous-classe. Ce mécanisme a été grandement employé lors la réalisation de la base de connaissances. Dans l'annexe 1, ce mécanisme pourra être repéré par l'emploi fréquent du terme : *modproperty* permettant la modification d'une propriété (celle-ci pouvant concerner uniquement la modification d'une facette de cette propriété).

- Enfin, des valeurs par défaut peuvent être associées à une propriété qui sera prise en compte si cette valeur n'a pas été explicitement affectée.

En contrepartie de ces avantages, BIM_Probe présente quelques inconvénients. Etant encore une version alpha, il n'est pas impossible de trouver des bugs (ce qui nous est arrivé pour le prédicat "exist-propvalue" vérifiant si une valeur existe et qui n'avait pas le comportement attendu). Le langage PCL se révèle très lent. J'emet cette observation à partir de la constatation effectuée lors des tests de mes programmes : les commandes "send" ne permettant d'effectuer une opération en PCL prennent un certain temps. De plus, la manière de programmer avec BIM_Probe la plus efficace est assez rebutante puisqu'elle implique l'écriture de faits tels que :

`putobject (dt, Networks, ISA, Class).` pour introduire un objet ou

1. La modélisation des éléments de base de la conception de réseaux de NEST est à la disposition du lecteur en annexe 1 où la définition de toutes les classes est donnée ainsi que leurs attributs.

`putslotdef (dt, Networks, cables, property, [type = List (Cables), derivation = (all (_cables where _cables is_in Cables & _cables#part-tof = this))]).` pour définir un attribut d'un objet. Un lecteur désirant comprendre la hiérarchie des objets et leurs attributs définis à partir de cette notion aura sûrement bien du mal.

3.2.2 Les différentes hiérarchies.

L'acquisition des connaissances a fait apparaître deux catégories de concepts à modéliser : la topologie des bâtiments au travers desquels le réseau devra être installé et les différents équipements à la disposition des concepteurs pour construire un réseau, y compris les machines à connecter sur le réseau.

Les éléments de base sont organisés sous forme de différentes hiérarchies reflétant la structure du domaine de connaissances et la fonctionnalité des différentes entités de ce domaine. En effet, l'acquisition des connaissances a montré que ce domaine est décomposable en diverses entités bien définies telles que des machines, des câbles, des boîtes de connexion, des connecteurs, etc, pour un réseau et des bâtiments, des étages, des pièces, des gaines techniques, etc, pour la description d'un site à câbler, chacun d'eux étant caractérisé par un ensemble de propriétés. Ces propriétés, de même que parfois les valeurs de ces propriétés, peuvent être communes à plusieurs entités. C'est le cas par exemple de la propriété *installable_on* donnant la liste des objets sur lesquels l'entité concernée peut être installée. Cette propriété est définie à la classe *Additional_components* définissant les caractéristiques communes à tout élément additionnel sur un autre composant de réseau. Elle est donc héritée par tout matériel additionnel tels que des cartes, des mémoires, des disques ou par tout logiciel. Bien d'autres exemples pourraient être cités. Le lecteur pourra consulter l'annexe 1 qui liste toutes les classes de la base de connaissances ainsi que leurs propriétés pour se faire une meilleure idée de la manière dont ce mécanisme d'héritage d'attributs a été utilisé.

Définir des hiérarchies d'objets rassemblant des propriétés communes à plusieurs objets enfants de cette classe permet de traiter les aspects conceptuels aux niveaux les plus élevés de la hiérarchie (Network_components, Boxes, Buidling_components, etc) alors que les feuilles de cet arbre concernent des équipement physiques particuliers (Retix2244, ...) ou des entités spécifiques telles que des gaines techniques verticales, des murs, etc. Ainsi en descendant dans la hiérarchie, les notions et les propriétés ont été successivement introduites allant des plus abstraites aux plus concrètes. Cette hiérarchisation de la base de connaissances permet d'ajouter, de modifier ou de détruire, très facilement, des entités feuilles de la hiérarchie. Ceci assure ainsi la possibilité de mettre à jour la base de connaissances facilement au fur et à mesure que de nouveaux équipements apparaissent sur le marché de la conception de réseaux locaux sans avoir à faire des changements remettant en cause toute la hiérarchie. En effet, un des principaux problèmes qui doit être traité pour obtenir une représentation adéquate des connaissances techniques de la conception de réseaux locaux est celui dû au fait que les entités de ce domaine changent fréquemment à cause d'une évolution constante des technologies de réseaux. Ainsi

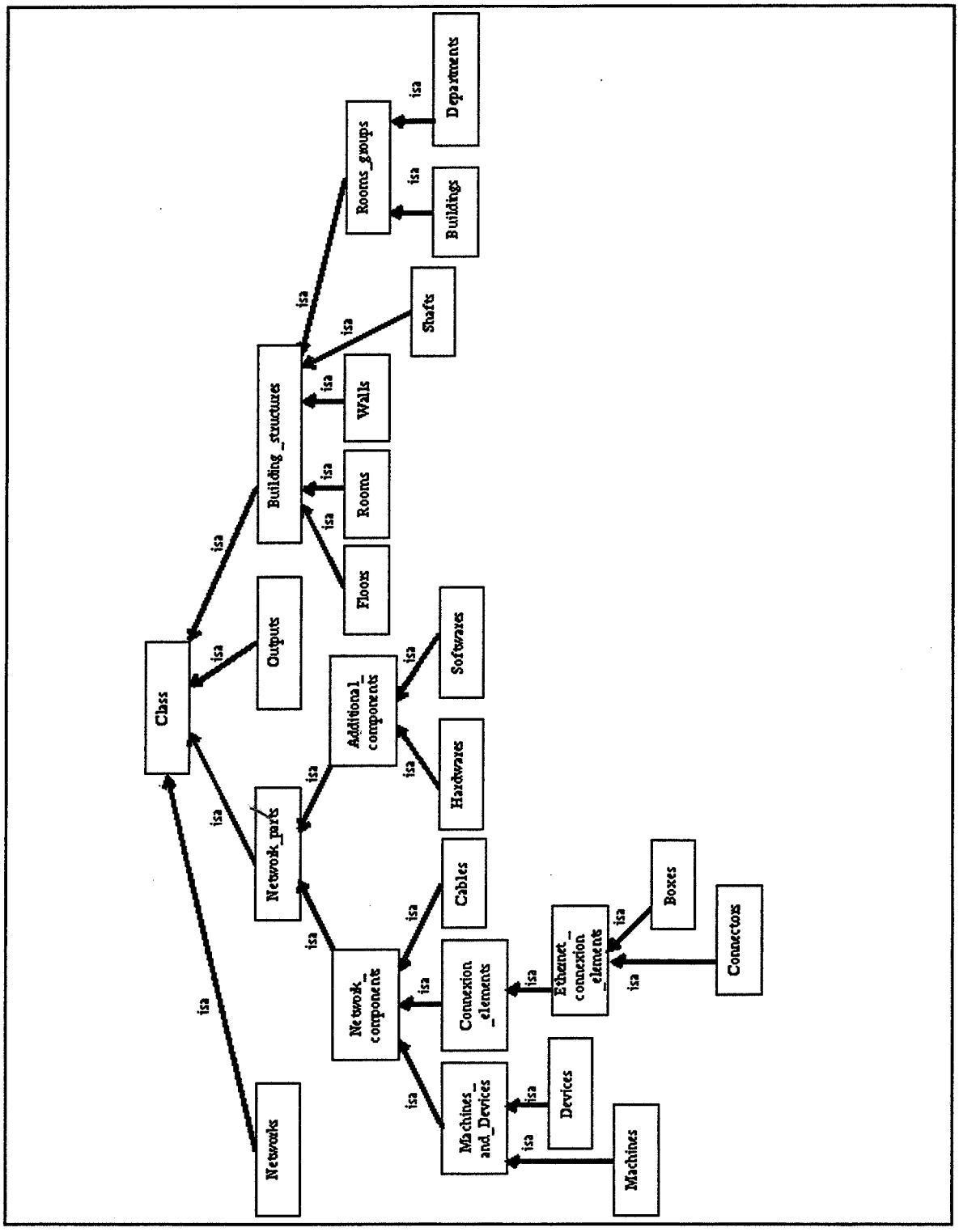


figure 23 : Hiérarchie générale des éléments de base implémentée dans NEST.

priorité a été accordée au développement d'une représentation des connaissances hautement flexible. La hiérarchisation de la base de connaissances est une réponse à ce besoin d'une grande flexibilité.

La figure 23 reflète les niveaux les plus hauts de la hiérarchie. Comme nous pouvons le voir sur cette figure, les aspects conceptuels de la conception de réseaux ont été décomposés en plusieurs super-classes. La classe Networks permet à NEST, par l'intermédiaire du mécanisme d'instanciation de représenter toute solution à un problème particulier. La sous-hiérarchie Network_parts décrit les connaissances techniques du domaine telles que les machines, les câbles, les connecteurs, les éléments additionnels qui peuvent être par exemple des disques, des mémoires mais aussi des logiciels, les divers composants de réseau tels que les répéteurs, les routeurs, les fan-outs, ... Les informations relatives à la topographie des bâtiments (étages, pièces, murs, gaines techniques, faux-plafonds, ...) et organisationnelles (départements, ...) sont représentées dans la sous-hiérarchie Building_structures. Outputs est la sous-hiérarchie de tous les types de sorties possibles des entités d'un réseau auxquels NEST sera confronté au cours de sa résolution puisqu'il doit s'assurer de la bonne connectivité des constituants du réseau considéré.

Les différentes sous-hiérarchies de ces super-classes sont introduites dans les paragraphes suivants sous forme d'arbres car ce format me semble plus parlant et plus agréable à consulter que des listes exhaustives de tous les élément de chaque sous-hiérarchie.

3.2.2.1 Networks.

Aucune sous-classe n'est définie pour cet objet Networks. L'objectif de cette classe est d'assurer une représentation de tout réseau à concevoir ou préexistant. Un réseau est constitué d'un ensemble d'équipements techniques : des câbles, des éléments de connexion, des machines, des composants additionnels logiciels ou matériels sur des machines ou sur des boîtes de connexion. Chacune de ces listes est une propriété de la classe des réseaux (voir figure 24). Les valeurs correspondant à ces propriétés sont obtenues par calcul par l'intermédiaire d'une dérivation. En fait, pour chaque élément d'un réseau, la propriété *part_of* est affectée au réseau dès que cette entité est créée. Pour le calcul des différentes listes des éléments d'un réseau, la dérivation correspondante va donc rechercher la valeur de part_of de chaque entité du type désiré (*Cables, Connexion_elements, Machines_and_devices,* etc) et tester si elle est égale à ce réseau. Des méthodes sont également associées à cette classe. La méthode *design* est celle qui conçoit un réseau alors que les autres méthodes sont principalement concernées par l'analyse d'un réseau donné par l'utilisateur ou conçu par NEST. Ces méthodes d'analyse sont présentées en détails dans le paragraphe 3.3 page 130. La méthode de conception est exposée dans le paragraphe 3.4 page 133.

```
object([
    class : Networks,
      isa : Class,
      slots,
        property : cables, [ type = List(Cables), card = 0 - U, derivation = ( all _4126 where _4126
          is_in Cables & _4126 # partof = this) ],
        property : connexion_elements, [ type = List(Connexion_elements), card = 0 - U, derivation
          = ( all _4071 where _4071 is_in Connexion_elements & _4071 # partof = this) ],
        property : machines_and_devices, [ type = List(Machines_and_devices), card = 0 - U, deri-
          vation = ( all _4181 where _4181 is_in Machines_and_devices & _4181 # partof = this) ],
        property : additional_components, [ type = List(Additional_components), card = 0 - U, der-
          ivation = ( all _4337 where _4337 is_in Additional_components & _4337 # partof = this) ],
        property : additional_components_on_boxes, [ type = List(Additional_components), card =
          0 - U, derivation = ( all _3943 where _3943 is_in Additional_components & _3943 #
          partof = this & prolog( not dt_installed_on_machine_type( _3943 ,Machines_and_devic-
          es))) ],
        property : additional_components_on_machines_and_devices, [ type = List(Additional_-
          components), card = 0 - U, derivation = ( all _4008 where _4008 is_in Additional_compo-
          nents & _4008 # partof = this & prolog(dt_installed_on_machine_type( _4008
          ,Machines_and_devices))) ],
        property : average_cable_installation_cost, [ type = List, card = 0 - U, default = nil ],
        constraint : average_cable_installation_cost_constraint, definition = value_inst_cost_con-
          st(this # average_cable_installation_cost), category = invariant ],
        property : installation_cost_option,[ type = String, presence = mandatory ],
        constraint : installation_cost_option_constraint, [ definition = (this # installation_cost_op-
          tion = oneof [ @ network, @ building, @ department]), category = invariant ],
        method : design, [ definition = design / 1, category = Prolog ],
        method : connexion_problems, [ definition = connexion_problems / 2, category = Prolog ],
        method : cost, [ definition = network_cost / 2, category = Prolog ],
        method : analyze_client_server_relation, [ definition = analyze_client_server_relation / 2,
          category = Prolog ],
        method : analyze_departmentalisation, [ definition = analyze_departmentalisation / 2, cate-
          gory = Prolog ],
        method : analyze_extensibility, [ definition = analyze_extensibility / 2, category = Prolog ],
        method : components_list_for_cost_report, [ definition = components_list_for_cost_report
          / 2, category = Prolog ],
        method : components_list_for_display, [ definition = components_list_for_display / 2, cate-
          gory = Prolog ],
        method : delayed_network_constraint_satisfaction, [ definition = delayed_network_con-
          straint_satisfaction / 2, category = Prolog ],
        method : delete, [ definition = delete_network / 1, category = Prolog ],
        method : machines_cost, [ definition = machines_cost / 2, category = Prolog ],
endslots,
endclass
]).
```

figure 24 : Définition de la classe Networks.

3.2.2.2 Network_parts.

La classe Network_parts regroupe les définitions de tous les équipements techniques à la disposition des experts pour construire un réseau. Ainsi plusieurs sous-hiérarchies la composent : les câbles, les boîtes de connexion, les connecteurs et les machines dont la super-classe est *Network_components* et les éléments additionnels sur des machines ou sur des boîtes de connexion, qu'ils soient matériels ou logiciels (*Additional_components*).

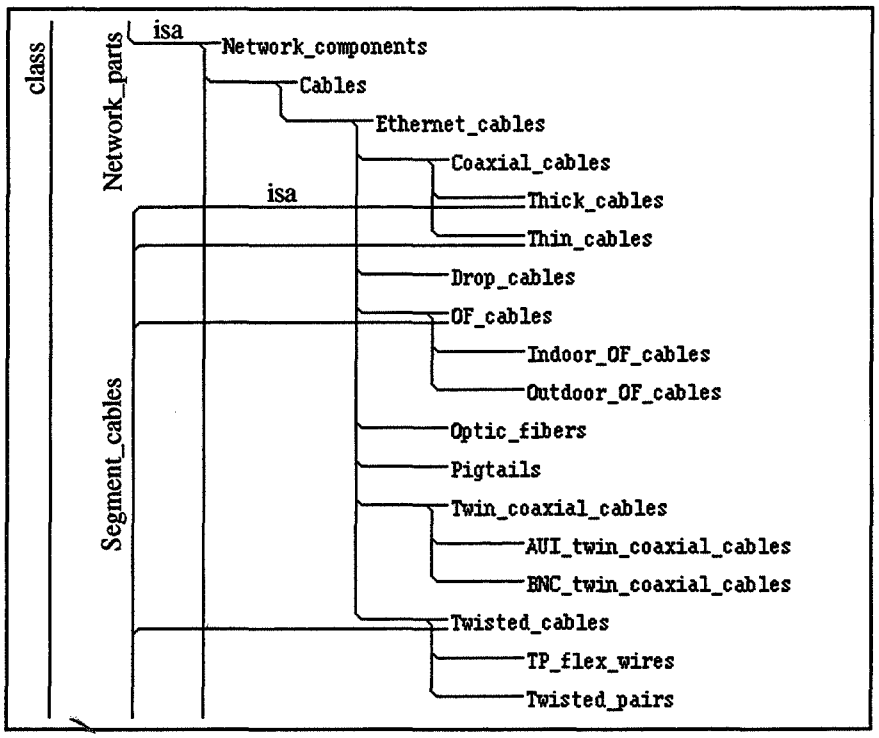


figure 25 : Les câbles.

Une sous-classe *Ethernet_cables* de *Cables* a été définie pour permettre d'étendre le système, si besoin est, à d'autres technologies qu'Ethernet.

Un segment de câble peut être constitué, du fait de la représentation interne choisie, par un ensemble de câbles, ceux existant entre deux boîtes de connexion (par exemple entre deux répéteurs). Des contraintes spécifiques aux segments de câble doivent être vérifiées lors de l'analyse d'un réseau suivant le type de câble. Par exemple, la longueur maximale entre deux répéteurs pour du câble coaxial épais (*Thick_cables*) est de 500 mètres alors qu'elle est de 185 mètres pour du coaxial fin (*Thin_cables*). La propriété *max_inter_repeaters_length* est définie au niveau de la super-classe *Coaxial_cables* et les valeurs spécifiques (500 et 185) sont données dans *Thick_cables* et *Thin_cables*. Pour permettre de faire le lien entre un segment de câble et son type, les classes *Thick_cables*, *Thin_cables*, *OF_cables*, *Twisted_cables* héritent également de la classe *Segment_cables*.

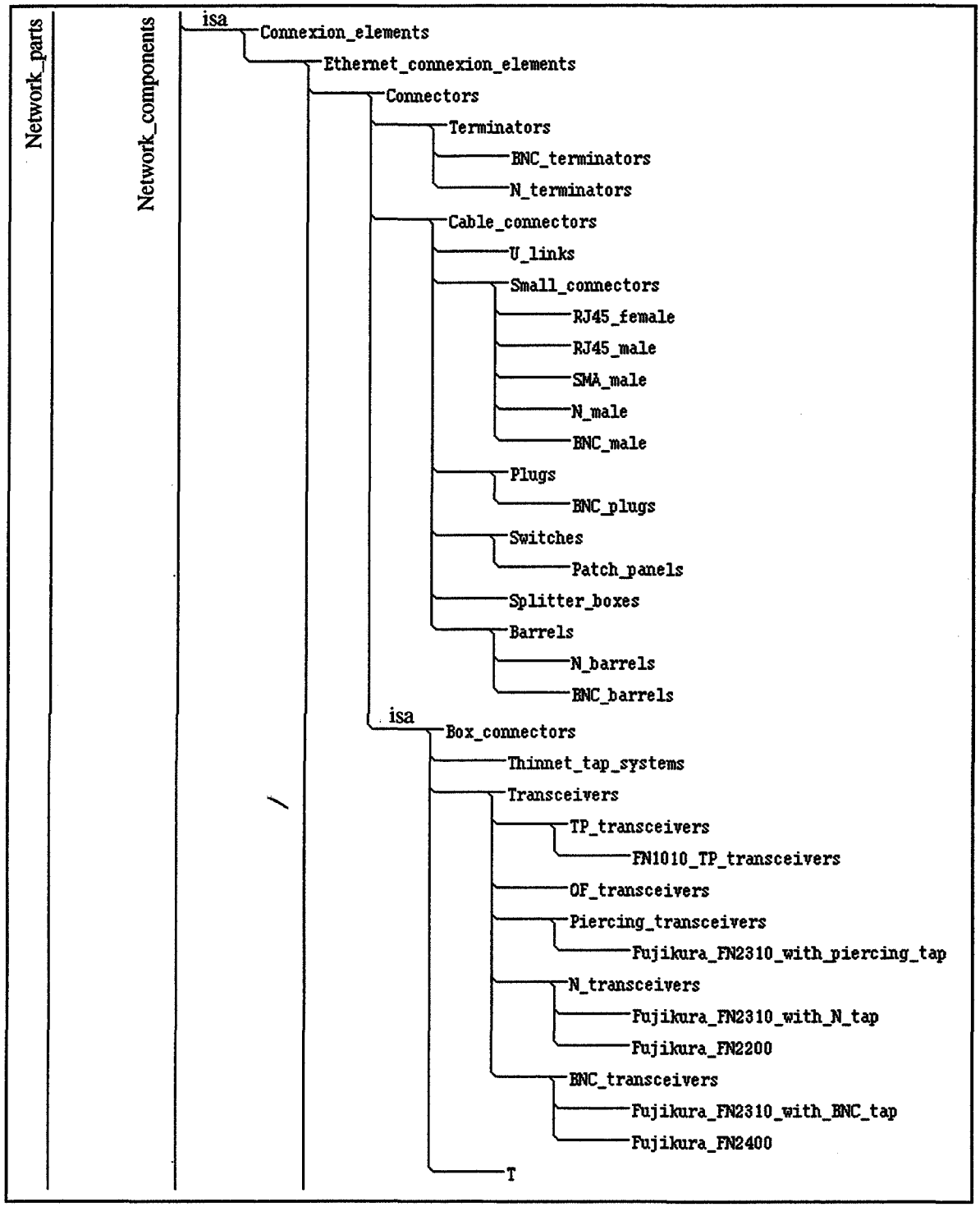


figure 26 : Les connecteurs.

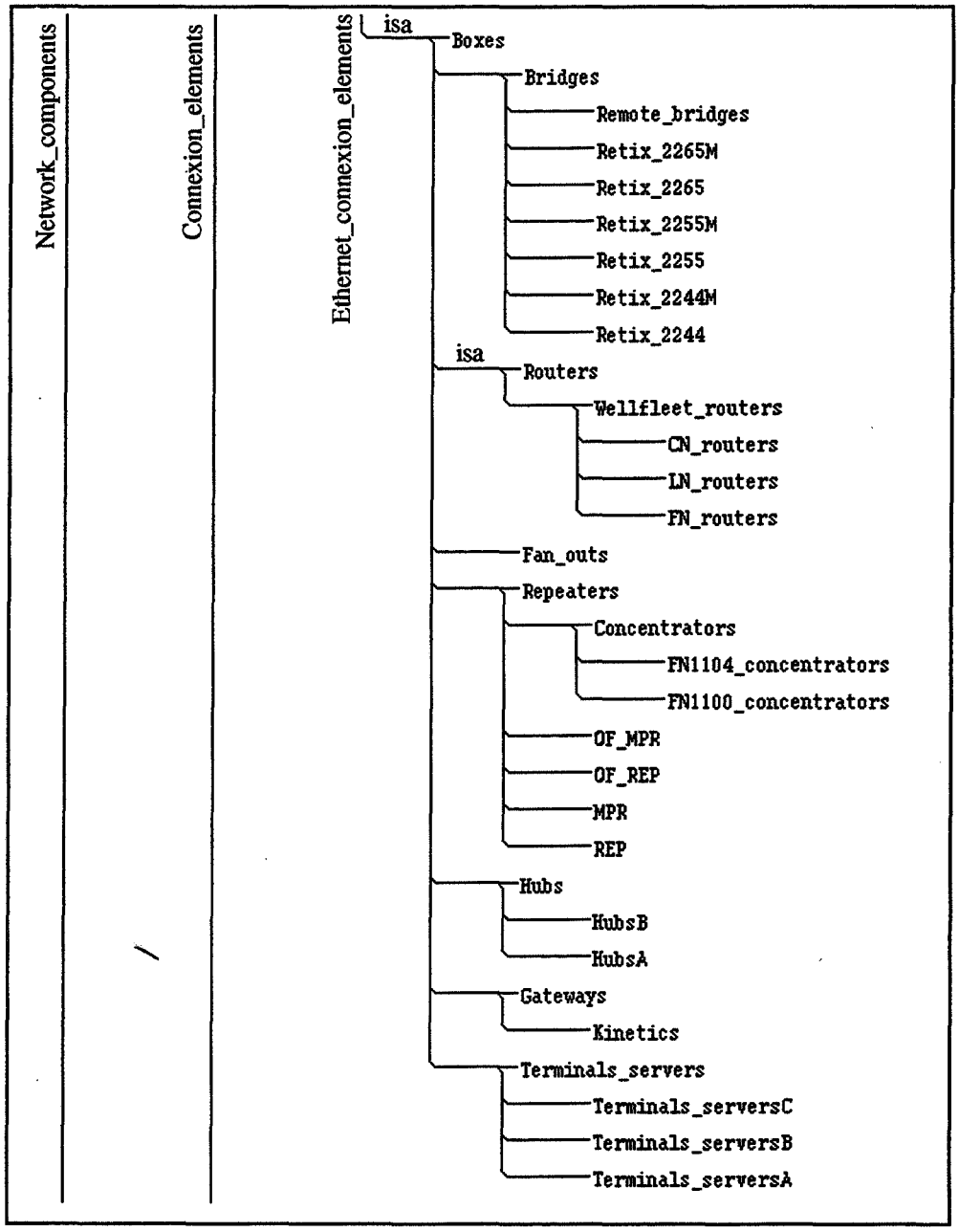


figure 27 : Les boîtes de connexion.

Alors que la hiérarchie des connecteurs (figure 26) liste les terminateurs et les connecteurs directement installables en bout de câble et ceux installables sur les boîtes de connexions, l'arbre de la figure 27 introduit tous les équipements à la disposition des concepteurs de réseaux locaux.

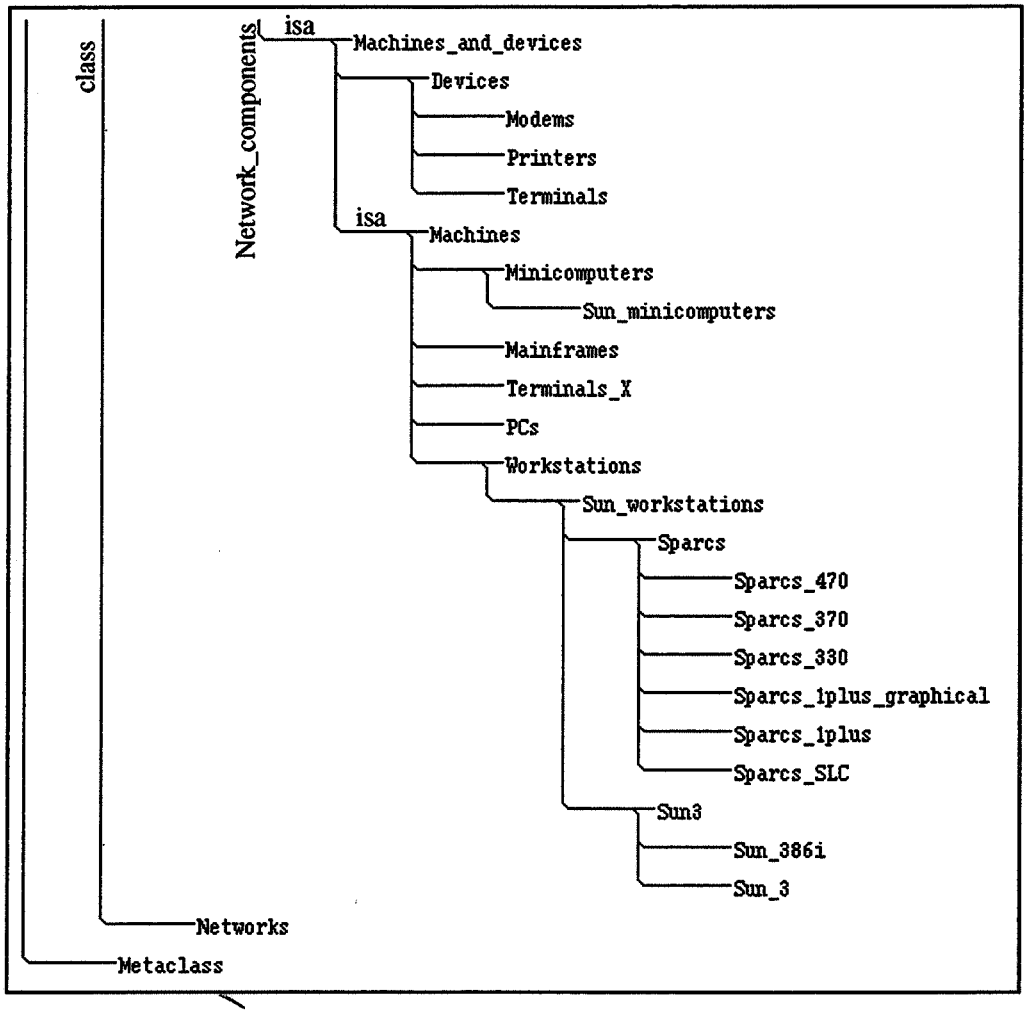


figure 28 : Les machines

Alors que la super-classe *Machines* contient toute propriété commune à tout type de machine, cinq sous-classes : *Minicomputers*, *Mainframes*, *Terminals_X*, *PCs* et *Workstations* décrivent des catégories spécifiques ayant un mode de connexion approprié au réseau. Les *Mainframes* sont généralement situés dans la salle informatique. Si celle-ci n'a pas été spécifiée, la salle informatique sera considérée comme étant celle où se trouve le ou les *Mainframes*, s'ils existent. Les machines auxquelles sont traditionnellement confrontées les experts de BIM font partie de la famille Sun ; c'est de ce fait, principalement cette famille que l'on retrouve dans la hiérarchie des machines.

Une machine est caractérisée par : (1) la liste des machines qui la servent au niveau applications, d'où la propriété : *application_servers* ; (2) si elle est sans disque, son serveur doit être défini dans *diskless_server* ; (3) si, par contre, c'est un serveur, la valeur de is_a_server devra être "vrai", sa valeur par défaut étant "faux" ; (4) le nombre maximum de machines sans disque qu'elle peut servir doit être spécifié dans *max_diskless_served* ; (5) si cette machine peut accepter des cartes, la liste des types de

ces cartes et du nombre correspondant de cartes possibles doit être donnée dans *max_cards_number*. En fait, cette dernière propriété est définie au niveau de la super-classe *Machines* et sa valeur est affectée dans ses sous-classe. Deux méthodes sont aussi associées à cette classe : *define_network_server* pour définir une machine comme étant un serveur pour toutes les machines appartenant au même réseau et *compute_cards_outputs* pour le calcul du nombre de sorties d'une machine.

Les deux hiérarchies ci-après listent les éléments qu'il est possible d'ajouter sur des machines ou sur des boîtes de connexion. La première représente les entités matérielles alors que la deuxième est pour les logiciels.

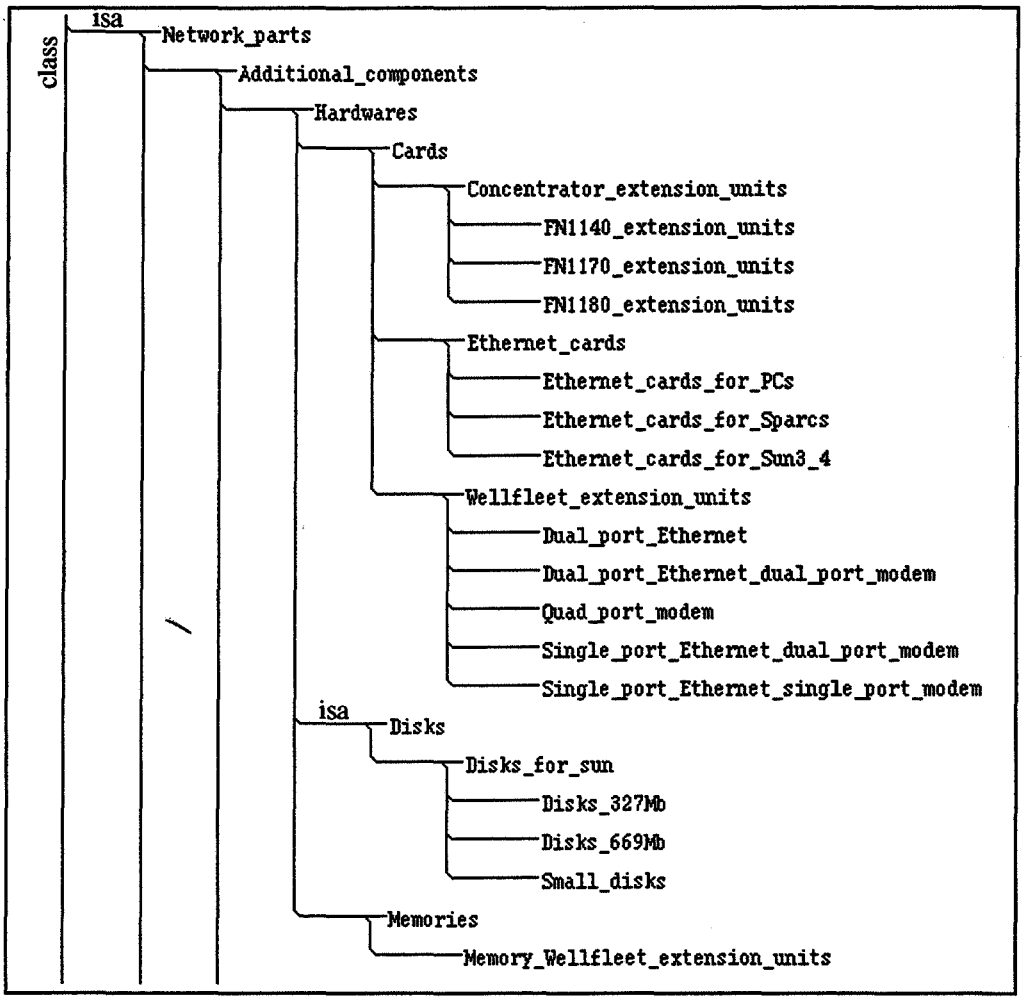


figure 29 : Les matériels additionnels.

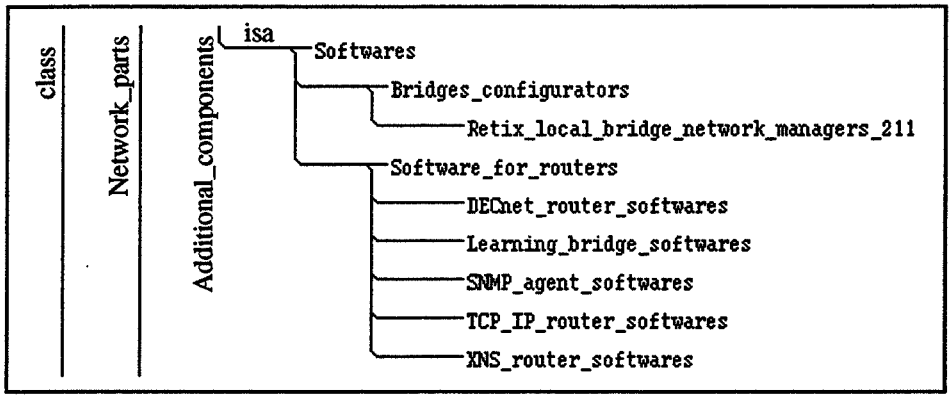


figure 30 : Les logiciels.

3.2.2.3 Les ports de sortie.

La classe Outputs contient tous les types de ports de sortie des équipements techniques. Lors de l'analyse de la bonne compatibilité au niveau de la connexion d'éléments techniques, il sera vérifié que les sorties connectées l'une à l'autre sont de même type mais de sexe opposé. C'est pourquoi dans cette hiérarchie, deux sous-sous-classes sont définies pour chaque sous-classe, une pour la partie femelle et l'autre pour la partie mâle. Par exemple, *Serial_outputs* est décomposé en *SerialF* et *SerialM*. La figure 31 ne représente qu'une partie de la hiérarchie des ports de sortie, la totalité n'offrant guère d'intérêt.

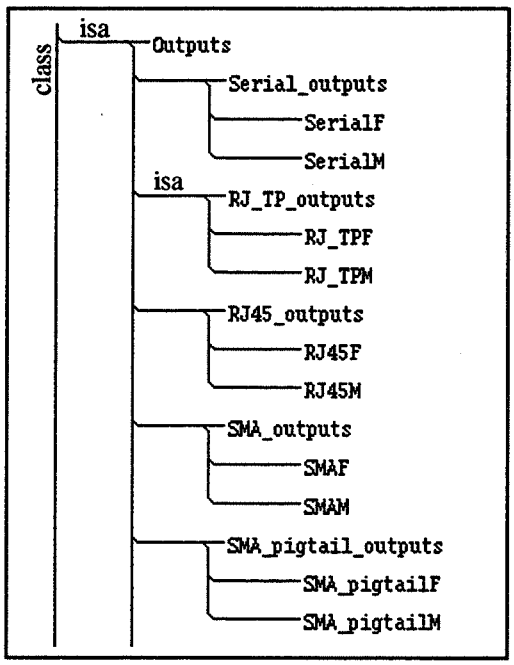


figure 31 : Une partie de la hiérarchie des ports de sortie.

3.2.2.4 Les structures architecturales.

Pour concevoir un réseau local, il est nécessaire de connaître l'environnement dans lequel il va être placé, c'est à dire les différents bâtiments à câbler et leurs caractéristiques : dimension, nombre d'étages, couloirs, gaines techniques où les câbles pourront passer, etc. La classe Building_structures assure la représentation de ces connaissances. Ainsi est-elle décomposée en plusieurs sous-classes pour la description des étages, des pièces, des murs, des gaines techniques et des unités organisationnelles telles que les départements et les bâtiments.

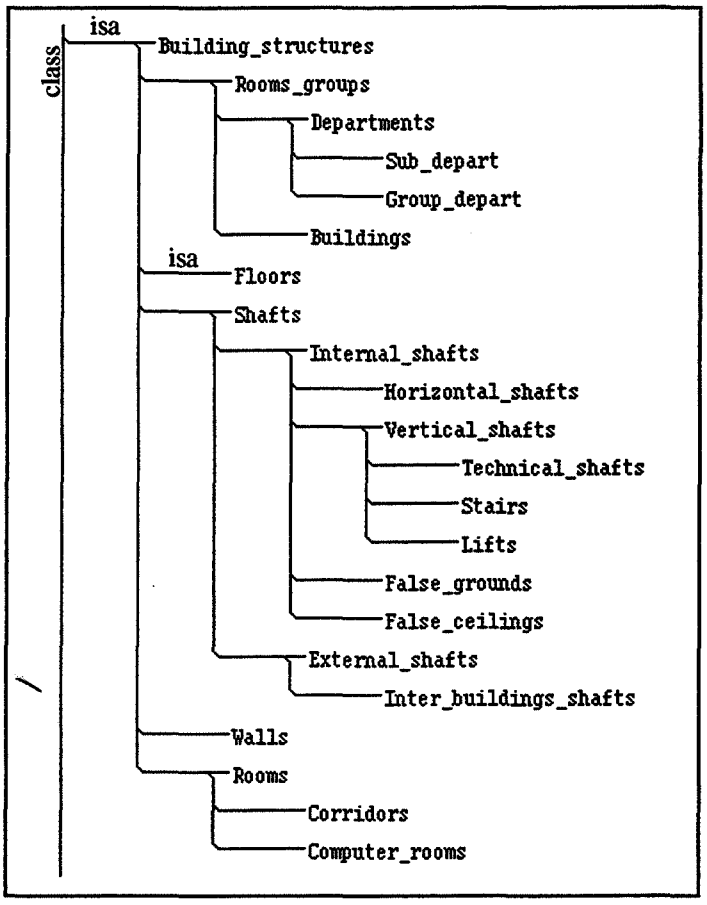


figure 32 : La structure des bâtiments.

Plusieurs raisons justifient cette description :

- Le passage des câbles est fait autant que possible à travers les gaines techniques horizontales ou verticales, les faux plafonds ou faux planchers, les couloirs, les montées d'escalier ou d'ascenseur. D'autre part, si un câble est amené à traverser un mur, celui-ci doit être perforable ; d'où la propriété booléenne *perforatable* pour la classe *Walls*.

- Un bâtiment peut être vieux, moderne ou en construction. Cette information apporte la connaissance sur l'existence de gaines techniques bien conçues ou non, ce qui influence la méthodolo-

gie de résolution de réseau. Les câbles passeront de préférence dans les gaines techniques dans un bâtiment moderne alors que dans un immeuble âgé les couloirs seront utilisés. En effet, dans les bâtiments modernes, des gaines techniques sont prévues avec une espace suffisant pour le passage des câbles alors que dans les anciens, les gaines sont inexistantes ou mal conçues. Pour les bâtiments en construction, les experts tentent d'influencer l'architecture des gaines. La propriété *building_type* de *Buildings* permet de spécifier cette caractéristique. Cette propriété est contrainte : sa valeur doit être un des termes prédéfinis représentatifs de chaque type de bâtiment possible, *[@to_built, @modern, @old].*

- Lors de la conception de réseau, il est important d'avoir une idée de l'extension future qui pourra être souhaitée sur ce réseau afin de s'assurer qu'elle sera possible. Pour permettre d'entrer cette donnée, *Buildings* comporte une propriété *extension* dans laquelle la liste des étages concernés par des extensions futures peut être spécifiée.

- Pour déterminer où faire passer les câbles, il est préférable de définir le chemin de longueur minimale permettant de connecter un ensemble donné de machines ou de pièces à câbler. Pour ce calcul, des informations sur la longueur des murs, sur le voisinage entre les pièces, etc, doivent être connues. La propriété *length* de *Walls* donne l'information sur la longueur d'un mur alors que le voisinage des pièces est obtenu par un prédicat spécifique déterminant la liste des pièces voisines. Pour se faire, il utilise les propriétés *walls_succession* de la classe *Rooms* donnant la succession des murs d'une pièce dans un ordre défini et *set_in_rooms* de *Walls*. Un mur peut appartenir à plusieurs pièces. Si un mur d'une pièce appartient à une autre salle, ces deux pièces sont voisines.

- Il est nécessaire de connaître les différentes unités professionnelles et leur implantation géographique dans les bâtiments pour appliquer le principe de la départementalisation. Ce principe consiste à grouper les machines d'un même département dans un même sous-réseau afin de minimiser le trafic sur le réseau global. Une propriété *set_in_departments* de la classe *Rooms* permet de spécifier la liste des départements auxquels cette pièce appartient. Le système peut aussi créer ses propres départements si par exemple aucun n'a été spécifié. Il regroupe parfois plusieurs départements si certains sont trop petits, créant ainsi un *Group_depart*. Et enfin, il peut avoir à diviser un département en plusieurs sous-groupes (*Sub_depart*) dans le cas où le nombre de machines de ce département dépasse le seuil acceptable.

- Si une pièce est dans un environnement perturbé pouvant provoquer des perturbations électriques, magnétiques, thermiques ou par rayons X sur les équipements que l'on souhaite y installer, il est important d'en être informé afin de choisir des équipements en conséquence ou de les placer dans un autre endroit. Ainsi la propriété *environmental_conditions* de la classe *Rooms* liste les perturbations rencontrées dans cette pièce. Comme pour *building_type* de *Buildings*, cette propriété est contrainte : elle doit être une sous-liste de termes prédéfinis représentatifs de chaque type de perturbation possible, *[@electrical, @X_ray, @thermical, @magnetical].*

- Afin de connaître le nombre de connexions souhaitées dans une pièce, une propriété *wished_connexions* de la classe *Rooms* assure cette fonctionnalité.

- Lors d'un premier appel d'offres, les experts n'ont pas toujours un plan très détaillé des bâtiments ; même s'ils en possèdent un, ils effectuent souvent une abstraction de ce plan, en le considérant comme constitué d'ensembles de pièces. Ceci est d'autant plus vrai que le site à câbler est grand. Par exemple, lors de l'étude du cas d'un hôpital, les experts ont retenu pour chaque étage des sous-ensembles de pièces appartenant à une même unité professionnelle et non chaque pièce indépendamment. Ainsi une pièce considérée par les experts peut exister réellement physiquement ou être un

ensemble de salles sans définir chacun des constituants de cet ensemble. La représentation adoptée permet également de traiter cette caractéristique.

- L'emplacement des salles informatiques (*Computer_rooms*) est intéressant. On entend par salle informatique : la pièce où sont regroupés les gros équipements informatiques tels que les mainframes, les serveurs, etc. Il nous permet de fixer le point de départ du réseau. C'est également ici que seront placés les utilitaires de gestion de réseau, les serveurs, les boîtes de filtrage et autres équipements.

Pour être en accord avec le module graphique de l'interface, plusieurs conventions géométriques ont été adoptées :

• L'origine d'un bâtiment ou d'une pièce est le point le plus bas et le plus à gauche de l'objet.

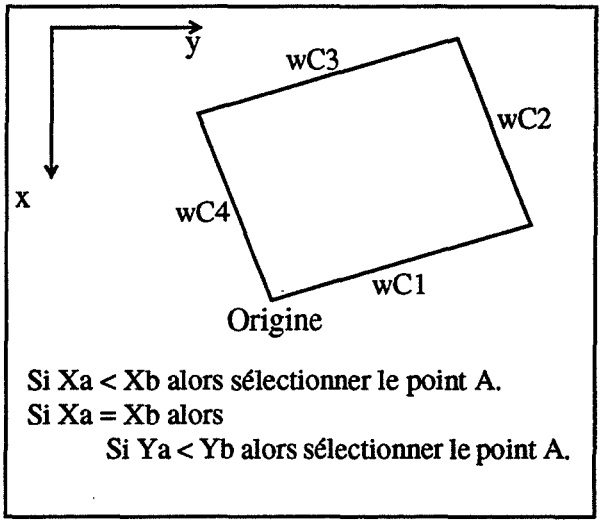


figure 33 : Convention pour l'origine d'un bâtiment ou d'une pièce.

Une pièce est considérée comme étant constituée d'une succession de murs. Il est nécessaire de fixer un ordre dans cette liste pour avoir une bonne utilisation de cette liste. L'ordre choisi est le sens trigonométrique commençant à l'origine de la pièce. Ainsi, dans la figure 35, la pièce C a pour liste de murs [wC1, wC2, wC3, wC4].

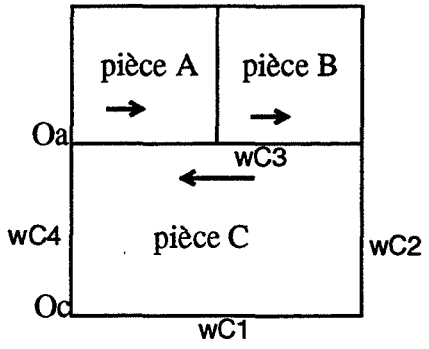


figure 34 : Convention sur la liste des murs d'une pièce.

• Représenter une pièce par une liste ordonnée de murs n'est pas suffisant. Un angle doit être introduit dans la description des murs. La convention adoptée est l'angle entre l'horizontale et la direction du mur. Ainsi *Walls* possède la propriété *angle* qui est contrainte puisque sa valeur doit être inférieure à 360 degrés.

3.3 L'outil d'analyse de réseaux.

A partir d'une description complète d'un réseau, différents types de contraintes sont vérifiés par cet outil et il rapporte les problèmes de configuration physique qu'il a rencontrés dans ce réseau. Cet outil peut accomplir plusieurs types d'analyses d'un réseau entièrement spécifié par l'utilisateur à partir d'un dessin sur l'interface graphique ou conçu par l'outil de conception. Ces différents types d'analyses [Balfroid 90] sont implémentés par des méthodes attachées à la classe Networks (voir page 120, la description de cette classe). Dans la description qui suit des différents types d'analyses des exemples effectués à partir de cet outil sont donnés pour chaque type d'analyse. La commande "send" est propre au langage BIM_Probe et permet de faire exécuter une méthode attachée à un objet (Network1 dans les exemples). Les résultats donnés sont ceux fournis par l'outil d'analyse et sont traités par un des modules de l'interface pour être présentés de façon plus conviviale à l'utilisateur du système.

• *L'analyse de validité* vérifie si un réseau donné est valide selon la technologie Ethernet et décrit les problèmes trouvés. Toutes les contraintes qui doivent être satisfaites pour s'assurer qu'un réseau est correct sont vérifiées lors de cette analyse. Ces contraintes concernent la technique des réseaux (par exemple, des composants non connectables entre eux, des segments de câble trop longs, ...) mais aussi la topologie du site câblé (par exemple, passage de câble coaxial épais dans des gaines horizontales).

```
   send (dt, connexion_problems (Network1, _messages), send (dt,
delayed_network_constraint_satisfaction (Network1, _messages2)).
   _messages = [not_1_connected (Sparcs_SLC2, nil),
forbidden_connexions (instance56, [instance106], [[AUIF2_2]]),
forbidden_connexions (instance101, [instance77], [[BNC_thinM1_1,
BNCM1_1]]), forbidden_connexions (instance105, [instance76],
[[BNC_thinM1_1, BNCM1_1]])]
   _messages2 = [too_long_segment ([Terminator1, Barrel1, BNC1, Thin1,
Tap1, Thin2, Tap2, Thin3, Tap3, Thin4, Tap4, Thin5, Tap5, Thin6, Tap6,
Thin7, Tap7, Thin8, Tap8, Thin9, Tap9, Thin10, Tap10, Thin11, Tap11,
Thin12, Tap12, Thin13, Tap13, Thin14, Tap14, Thin15, Tap15, Thin16,
Tap16, Thin17, Tap17, Thin18, Tap18, Thin19, BNC2, Barrel2, Terminator2],
185, 2.600000000000000e+03)]
```

exemple 5 : Résultats de l'analyse de validité d'un réseau.

• *L'analyse d'extensibilité* : Comme un des critères de qualité pour un réseau est sa capacité à pouvoir être facilement étendu pour répondre à des besoins futurs, une analyse de l'extensibilité d'un réseau est fournie par NEST. Pour ce faire, cette analyse cherche à répondre à des questions telles

que : est-il possible d'ajouter des segments de câble à ce réseau ? est-il possible de connecter d'autres machines sur chaque segment ? reste-t-il des connexions libres sur les composants de réseaux et est-il possible de leur ajouter des cartes ? etc.

```
   send (dt, analyze_extensibility (Network1, _x).
   _x = [possible_additional_cards (Machine1, Ethernet_cards, 1),
possible_additional_cards (Machine3, Ethernet_cards, 1),
possible_additional_cards (Machine8, Ethernet_cards, 1),
possible_additional_cards (Machine2, Ethernet_cards, 1),
possible_additional_cards (Machine5, Ethernet_cards, 1),
possible_additional_cards (Machine4, Ethernet_cards, 1),
possible_additional_cards (Machine6, Ethernet_cards, 1),
possible_additional_cards (Machine7, Ethernet_cards, 1), empty_outputs
(Tap3, [[ @ Twin_coax, @ male, 1]]), empty_outputs (Tap6, [[ @ Twin_coax,
@ male, 1]]), empty_outputs (Tap8, [[ @ Twin_coax, @ male, 1]]),
empty_outputs (Tap9, [[ @ Twin_coax, @ male, 1]]), empty_outputs (Tap11,
[[ @ Twin_coax, @ male, 1]]), empty_outputs (Tap12, [[ @ Twin_coax, @
male, 1]]), empty_outputs (Tap13, [[ @ Twin_coax, @ male, 1]]),
empty_outputs (Tap15, [[ @ Twin_coax, @ male, 1]]), empty_outputs (Tap17,
[[ @ Twin_coax, @ male, 1]]), empty_outputs (Tap18, [[ @ Twin_coax, @
male, 1]]), network (network_id1, [segment_id1], [available_segments (4),
available_not_links (2)]), segment (segment_id1, [Terminator1, Barrel1,
BNC1, Thin1, Tap1, Thin2, Tap2, Thin3, Tap3, Thin4, Tap4, Thin5, Tap5,
Thin6, Tap6, Thin7, Tap7, Thin8, Tap8, Thin9, Tap9, Thin10, Tap10, Thin11,
Tap11, Thin12, Tap12, Thin13, Tap13, Thin14, Tap14, Thin15, Tap15, Thin16,
Tap16, Thin17, Tap17, Thin18, Tap18, Thin19, BNC2, Barrel2, Terminator2],
[available_box_nbr (12)])]
```

exemple 6 : Un exemple de l'analyse de l'extensibilité faite par NEST.

• *L'analyse de la relation client-serveur* a pour but de vérifier si la relation client-serveur est correctement supportée par le réseau donné, c'est à dire si les machines sans disque et leur serveur sont sur le même sous-réseau, si chaque machine a un disque ou est liée à un serveur, si chaque serveur a un disque, si la répartition des machines sans disque sur les différents serveurs est bien faite.

```
   send (dt, analyze_client_server_relation (Network1, _x)).
   _x = [no_disk_server_for_diskless (Machine1),
no_disk_server_for_diskless (Machine8), no_disk_server_for_diskless
(Machine2), no_disk_server_for_diskless (Machine4),
no_disk_server_for_diskless (Machine7)]
```

exemple 7 : Résultats de la recherche d'une bonne relation client-serveur.

• *L'analyse de la départementalisation* regarde si les machines appartenant à un même département ne sont pas séparées par un bridge ou un routeur, ce qui pourrait provoquer des problèmes de performance pour la communication entre deux machines.

```
   send (dt, analyze_departmentalisation (Network1, _x)).
   _x = [not_on_same_network ([Terminals_X2, Terminals_X1, Sparc1_Plus,
instance183, Sparcs_SLC4, Sparcs_SLC3, Sparcs_SLC2], Department_1)]
```

exemple 8 : Une analyse de la bonne départemtalisation d'un réseau.

• *L'analyse du coût* calcule le coût du réseau.

```
send(dt, cost(Network1, _x)).
_x = 529220
```

exemple 9 : Calcul du coût.

Le résultat fourni par cet outil d'analyse est transmis à l'interface où un des modules est chargé de transcrire cette donnée en langage naturel. Suivant l'utilisateur, le résultat est donné avec plus ou moins d'explication. Le premier exemple ci-dessous utilise l'analyse de coût pour calculer la différence entre le coût du réseau et le budget. Le résultat est donné sous la forme d'une phrase en langage naturel.

```
  Utilisateur (Mode Anglais) : What is the difference between the network
cost and the budget?
    Systeme (Mode Anglais) : The cost is greater than the budget by
986 Sterling.
```

exemple 10 : Utilisation de l'analyse de coût d'un réseau.

Un deuxième exemple est donné sur la figure 35 dans le cas de l'analyse de l'extensibilité d'un réseau. C'est une copie d'écran de la fenêtre de dialogue en langage naturel du système développé. Le réseau étudié a été construit par NEST (et est présenté à la figure 48). Il peut donc être étendu puisqu'il est possible d'ajouter de nouvelles cartes sur différentes machines.

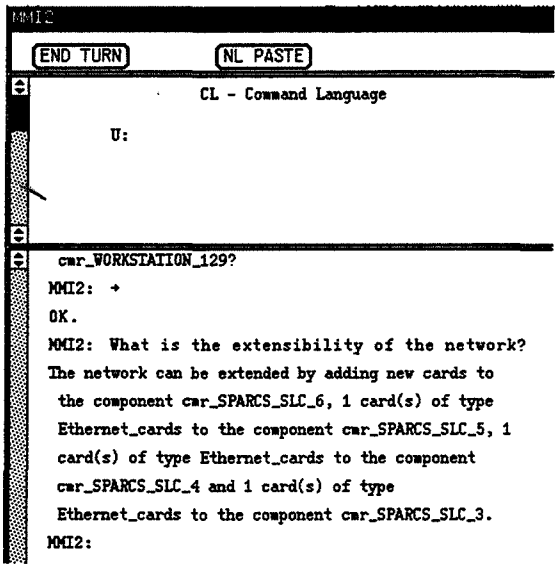


figure 35 : Réponse en langage naturel à la demande d'analyse de l'extensibilité d'un réseau.

3.4 L'outil de conception.

3.4.1 Architecture.

3.4.1.1 Planification et modularisation de l'activité de conception de réseaux.

Le problème de la conception de réseau peut être vu comme une décomposition en sous-réseaux avec des moyens assurant la liaison entre eux. Le besoin d'une répartition apparait à travers plusieurs principes : (1) la physionomie des bâtiments : par exemple, le câblage d'un bâtiment ayant plusieurs étages peut être réalisé par le câblage de chaque étage et par une épine dorsale verticale pour relier ces sous-réseaux ; (2) le principe de la départementatisation qui consiste à isoler les machines appartenant à une même unité professionnelle dans un sous-réseau ; (3) le regroupement d'ordinateurs homogènes car il est plus facile de câbler des machines ayant les mêmes caractéristiques puis de connecter ce sous-réseau au réseau global. De plus, la résolution d'un sous-réseau implique également la réalisation de plusieurs sous-tâches très couplées entre elles telles que la décision sur une structure de sous-réseau, le choix d'un type de câble, la sélection d'équipements de connexion, etc. Les nombreuses contraintes (décrites au paragraphe 2.3.2 page 107) attestent de la complexité des interactions entre les diverses tâches. D'autre part, la conception de réseau est un problème mal structuré. [Newell 69] explique qu'un problème mal structuré est caractérisé par des buts faiblement définis et par l'absence d'un chemin de décision prédéfini permettant de passer de l'état initial du processus de conception à la solution finale. Nous avons vérifié, lors de l'acquisition des connaissances, que les divers experts interrogés suivent un cheminement de résolution différent. De plus, suivant le problème qui lui est soumis, un expert donné ne suit pas forcément le même ordre au niveau de l'enchaînement des étapes qu'un autre expert. Cependant, un ordre pour cette succession de tâches va pouvoir être fixé comme nous le verrons plus tard lorsque sera expliqué comment l'outil de conception a été réalisé.

L'architecture choisie pour NEST reflète la manière suivant laquelle les experts humains conçoivent les réseaux. En effet, différentes tâches sont successivement réalisées par le concepteur de réseaux et celles-ci seront de même implantées dans le système sous forme de différents modules à activer. Ainsi l'architecture du système peut être vue d'un premier abord comme une suite de tâches à effectuer. Lorsqu'on rentre plus dans les détails, chaque tâche est composée d'un ensemble d'étapes qui pourront être activées dans un ordre non pré-déterminé du moment que l'ensemble des pré-requis de l'étape concernée est vérifié. L'architecture de NEST est donc très modulaire mais présente aussi une organisation hiérarchique. En effet, des plans de contrôle constituent la structuration des diverses étapes à accomplir pour parvenir à un réseau (voir figure 36). Un des plans est le sommet de la hiérarchie : le plan principal. Chacune de ses étapes est accomplie par l'activation de sous-plans, ceux-ci étant eux-mêmes réalisés par un ou plusieurs modules.

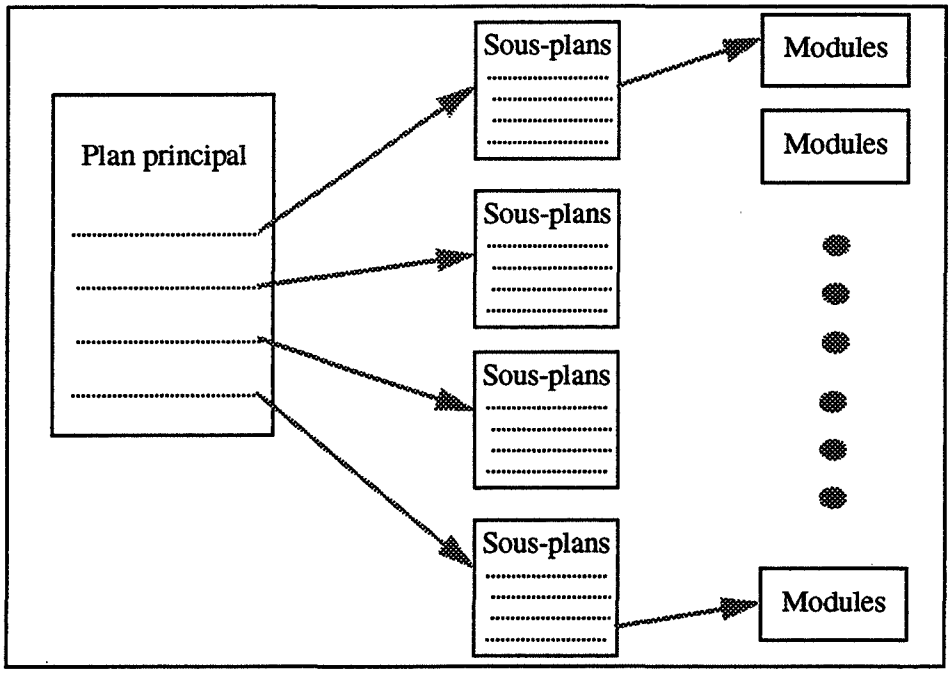


figure 36 : Organisation de l'outil de conception.

Après avoir décrit le plan de contrôle principal, les paragraphes suivants présenteront chacune des étapes le constituant. Comme nous l'avons déjà vu, chaque étape est effectuée par une succession de sous-étapes elles-mêmes réalisées par l'activation de modules. Chaque module est responsable d'une tâche particulière assurant la réalisation d'une partie de la conception. Il sélectionne parmi un ensemble de valeurs possibles la meilleure pour le problème en cours. Pour cela, il considère diverses contraintes de validité dans le but de rejeter les valeurs ne convenant pas au problème particulier puis il vérifie différentes contraintes de préférence pour sélectionner la solution la meilleure parmi les solutions possibles. Afin de présenter chacun de ces modules, un formalisme général a été défini. Il décrit, entre autres, les contraintes pour chaque module. L'annexe 2 décrit ce formalisme et présente ces tâches en utilisant le format défini.

L'espace de travail, les résultats et les limitations de cet outil sont également décrits dans les derniers paragraphes.

3.4.1.2 Le plan de contrôle principal.

S'appuyant sur le fait qu'un réseau peut être vu comme plusieurs sous-réseaux connectés entre eux, le plan de contrôle principal est composé des étapes suivantes :

- La *sélection d'un squelette* détermine la structure qu'aura le réseau à partir de la taille du site (bâtiment à étage unique ou multiple, un ou plusieurs bâtiments, dimensions des bâtiments) et de la répartition des départements dans les bâtiments. Cette étape sélectionne principalement les montées verticales qui seront utilisées et les sous-réseaux nécessaires.

• La *répartition en sous-réseaux* configure chaque sous-réseau, c'est à dire détermine sa structure, le type de câble, les boîtes de connexions nécessaires et s'assure de la bonne connexion des machines à ce sous-réseau.

• Le *mixage des sous-réseaux* fixe les épines dorsales verticales auxquelles sont reliés les différents sous-réseaux obtenus à l'étape précédente.

• L'*optimisation* : Selon le coût, le budget, les besoins technologiques, des optimisations peuvent être réalisées sur le réseau obtenu précédemment telles que l'amélioration des performances par l'ajout de nouveaux logiciels.

3.4.2 Description des différents modules.

3.4.2.1 Sélection d'un squelette de réseau. (CJ)

L'objectif de la sélection d'un squelette de réseau est non seulement de choisir une structure de réseau adaptée au problème à traiter mais aussi de prévoir le nombre de montées verticales qui seront nécessaires, de décider de leur emplacement et de donner la liste des étages à câbler. D'où le plan de contrôle suivant :

Sélection d'un squelette :

+ sélection d'un type de squelette,

  - sélection du nombre d'épines dorsales verticales,

+ sélection des étages à câbler,

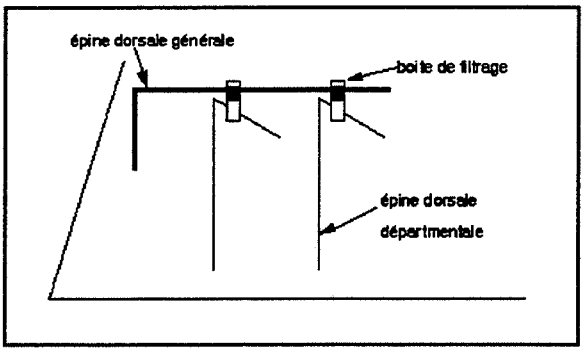
+ sélection de la localisation des épines dorsales verticales.

figure 37 : Plan de contrôle pour la sélection d'un squelette.

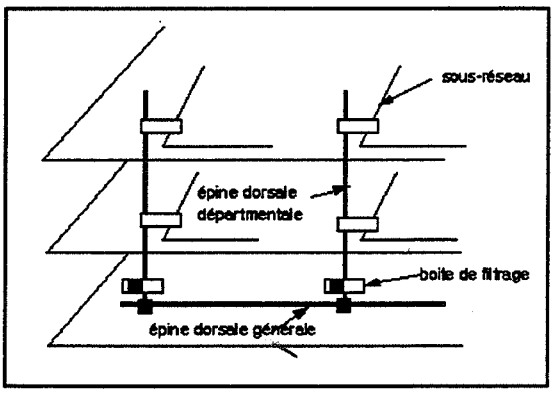
Ainsi définit-on, tout d'abord, un squelette pour le réseau à concevoir. Pour ce faire, suivant la physionomie du site à câbler, un type de squelette conviendra mieux qu'un autre. En fait, trois cas ont été identifiés pour le moment : le cas d'un bâtiment avec un seul étage ramenant le problème à la dimension deux, le cas d'un "petit" bâtiment ne nécessitant qu'une montée verticale et le cas plus général nécessitant une ou plusieurs épines dorsales verticales. C'est principalement pour ce dernier cas qu'il est nécessaire de prendre une décision sur le nombre d'épines dorsales verticales, de fixer, pour chacune d'eux, les étages auxquels elle sera connectée et enfin de sélectionner un emplacement pour le passage de ces épines dorsales.

sélecteur de type de squelette.

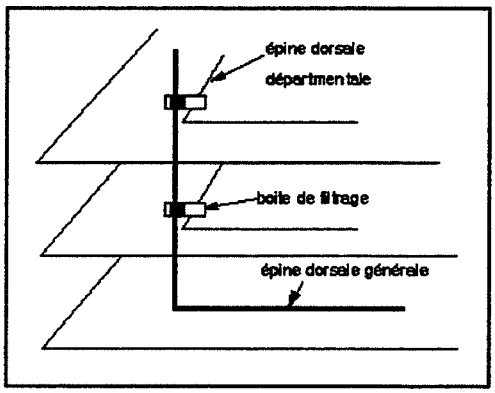
Un réseau peut également être vu comme constitué d'une épine dorsale générale sur laquelle viennent se greffer des épines dorsales départementales. S'appuyant sur cette structure, les trois types de sous-réseaux sont les suivants :



Squelette pour un étage



Squelette général



Squelette simple

figure 38 : Types de squelette.

calculateur du nombre d'épines dorsales verticales.

Dans le cas où plusieurs épines dorsales verticales sont nécessaires, cette tâche détermine leur nombre. Elle effectue aussi la décomposition en sous-réseaux en recherchant à isoler les départements les uns des autres, à ne pas dépasser le seuil maximum de machines par sous-réseau, à grouper un nombre minimum de machines dans chaque sous-réseau et à s'assurer qu'un serveur et ses machines associées sont dans le même sous-réseau.

sélecteur des étages à câbler.

Cette tâche est une sous-tâche de la tâche calculateur du nombre d'épines dorsales verticales. Elle permet d'aider à la décomposition en sous-réseaux. Elle détermine les étages qui doivent être câblés et

pour chacun d'eux fixe les sous-réseaux nécessaires. Pour cela, elle remplit la valeur de la propriété *floor_departments* pour chaque épine dorsale verticale. Cette propriété est une liste de listes du type : [étage, département]. Chaque élément de cette liste spécifie ainsi un sous-réseau. Il est à noter que les départements peuvent être des départements réels donnés par l'utilisateur ou des départements fictifs créés par NEST dans le but de regrouper plusieurs machines dans un même sous-réseau.

sélecteur de la localisation des épines dorsales verticales.

Cette tâche recherche une localisation pour chaque épine dorsale verticale parmi toutes les montées possibles dans ce bâtiment. Plusieurs critères de choix sont employés. Une gaine technique est préférée à un escalier ou à un ascenseur, dans le cas d'un bâtiment moderne où les gaines techniques ont été prévues avec suffisamment d'espace pour le passage de câbles. La montée la plus proche de la salle informatique ou la plus centrale est sélectionnée avant toutes les autres.

3.4.2.2 Conception des sous-réseaux pour chaque étage.

Ayant fait la répartition en différents sous-réseaux à l'étape précédente, l'objectif est maintenant de les concevoir. Pour cela, diverses opérations devront être effectuées telles que la sélection d'un type de câble, d'une structure de sous-réseau, de boîtes de connexion, de connecteurs et la localisation de tous ces éléments. Ces opérations peuvent être effectuées dans un ordre quelconque mais la réalisation de chacune d'elles a des conséquences sur les autres. Par exemple, le choix d'un type de câble contraint la sélection de la structure du sous-réseau et inversement. Les interactions entre ces opérations ont été présentées dans la figure 21 page 112. De plus, différentes contraintes doivent être vérifiées par l'outil de conception de NEST (celles listées au paragraphe 1.3.2 page 95). En fait, un ordre d'activation des différents modules permettant la conception de sous-réseaux a été fixé et est donné à la figure 39. Cet ordre a été choisi en fonction des informations dont un module a besoin et qui sont calculées par un autre module. Pour avoir une idée sur les besoins en calculs avant l'activation d'une tâche, le lecteur pourra consulter dans l'annexe 2 les paragraphes : données calculables requises et optionnelles pour chaque tâche de la conception de sous-réseaux.

Répartition en sous-réseaux :

+ calcul de la distance à câbler,
+ sélection du type de câble,
+ sélection de la structure du sous-réseau,
+ sélection des boîtes de connexion,
+ localisation des boîtes de connexion,
+ localisation des câbles,
+ sélection des connecteurs.

figure 39 : Plan de contrôle pour la réalisation des sous-réseaux.

calculateur de la distance à câbler. (CJ)

La distance à câbler est une valeur approximative donnant un ordre de grandeur de l'ampleur du problème. Elle va permettre de déduire un type de câble et une structure de sous-réseau utilisables à chaque étage. Elle représente la longueur d'une ficelle minimale passant par les couloirs et reliant toutes les pièces susceptibles d'être câblées (contenant au moins une machine ou/et pour laquelle un certain nombre de connexions est souhaité).

sélecteur du type de câble. (FB)

Ce module sélectionne parmi les types de câbles possibles le plus approprié pour un sous-réseau. Il procède en filtrant la liste des types de câbles possibles en considérant les exclusions technologiques (par exemple, la longueur maximale qui peut être câblée avec un type de câble donné), en utilisant les conflits d'environnement rencontrés et les spécifications de l'utilisateur (son budget, le nombre de connexions attendues, possibilité d'une extension future vers FDDI).

sélecteur de la structure du sous-réseau. (FB)

A partir de la distance à câbler et du choix fait pour le type de câble à utiliser pour un sous-réseau donné, ce module va déterminer sa structure. Celle-ci peut être soit linéaire (le sous-réseau est alors composé d'un seul câble), soit contenant un répéteur (deux câbles constitueront ce sous-réseau connectés à un répéteur), soit enfin une structure en étoile (le nombre de segments de câble sera alors supérieur à deux et une boîte de connexion adaptée devra être choisie).

sélecteur de boîte de connexion. (FB)

Si la structure choisie pour un sous-réseau est en étoile ou comporte un répéteur, une boîte de connexion particulière devra être sélectionnée pour assurer une bonne connexion des différents segments de câble. Ce module prend également en compte le budget du client et le besoin d'une départementalisation.

localisateur de boîte de connexion. (CJ)

Le type de boîte de connexion nécessaire étant déterminé, celle-ci peut être créée et placée dans une pièce de l'étage correspondant au sous-réseau. Cette localisation est choisie en fonction de la structure du sous-réseau. Si la structure est linéaire, aucune boîte n'est à localiser. Si on a affaire à une étoile, la boîte est installée dans la pièce considérée comme étant le point de départ du sous-réseau. Celui-ci est soit une salle informatique s'il en existe une à l'étage considéré, soit une pièce contenant des mainframes ou des serveurs, soit la pièce la plus proche du puits d'arrivée de l'épine dorsale verticale à laquelle sera relié le sous-réseau. Par contre, si la structure choisie est à deux branches, le répéteur nécessaire sera placé dans la pièce la plus proche de l'extrémité du premier segment.

créateur et localisateur de câbles. (CJ)

Jusqu'à cette étape, seule la décision sur le type de câble a été prise. Aucun câble n'a encore été créé. C'est l'objectif de ce module, de même que le placement de ceux-ci dans une gaine technique ou le long d'un couloir. Ce module va donc permettre de relier par des câbles les machines ou les points de connexion dans chaque pièce que le sous-réseau considéré doit connecter.

sélecteur de connecteurs. (FB)

Enfin, les petits éléments à interposer entre les boîtes de connexion et les câbles doivent être créés pour assurer une bonne connexion vérifiant la technologie Ethernet. De même, les terminateurs de câbles doivent être ajoutés.

3.4.2.3 Fusion des différents sous-réseaux.

Pour l'instant chaque sous-réseau a été développé. Mais ces sous-réseaux ne sont pas encore reliés aux épines dorsales verticales. C'est donc le but de cette phase. Des boîtes de filtrage vont être ajoutées pour s'assurer de l'isolement de chaque sous-réseau par rapport aux autres. De plus, les câbles constituant chaque épine dorsale verticale devront être créés et ces différentes épines dorsales verticales devront être reliées entre elles. Ainsi les différentes opérations constituant la phase de fusion des différents sous-réseaux ont été ordonnées par le plan de contrôle suivant :

Fusion des sous-réseau :

+ calcul de la distance verticale,
+ sélection d'un type de câble pour chaque épine dorsale verticale
+ ajout de boîtes de filtrage,
+ création des câbles verticaux,
+ lien entre les différentes épines dorsales verticales.

figure 40 : Plan de contrôle pour la fusion des différents sous-réseaux.

calcul de la distance verticale. (CJ)

Considérant les étages à câbler par une épine dorsale verticale donnée, ce module calcule la distance à câbler permettant de tous les relier.

sélection d'un type de câble pour chaque épine dorsale verticale. (FB)

Comme pour chaque sous-réseau, un type de câble est déterminé pour chaque épine dorsale verticale. En fait, ce module réutilise certains prédicats implémentés pour la sélection du type de câble de chaque sous-réseau.

ajout de boîtes de filtrage. (FB)

Des boîtes de filtrage sont nécessaires pour isoler chaque sous-réseau. Ce module choisit leur type, crée ces boîtes et les localise.

création des câbles verticaux. (CJ)

Ayant défini le type de câble, il est possible de créer les câbles.

lien entre les différentes épines dorsales verticales. (CJ)

Après une discussion avec les experts de la société BIM, il s'est avéré qu'il existait deux possibilités pour connecter les différentes épines dorsales verticales. La première consiste à créer une épine dorsale générale dans la salle informatique et à prolonger les épines dorsales verticales jusqu'à la salle informatique pour les connecter à l'épine dorsale générale. La figure 41 est un exemple d'une telle fusion dans le cas de deux épines dorsales verticales. Les éléments de filtrage (MAC-layer bridge dans la figure 41) permettant d'isoler les différentes unités professionnelles sont alors regroupés dans la salle informatique.

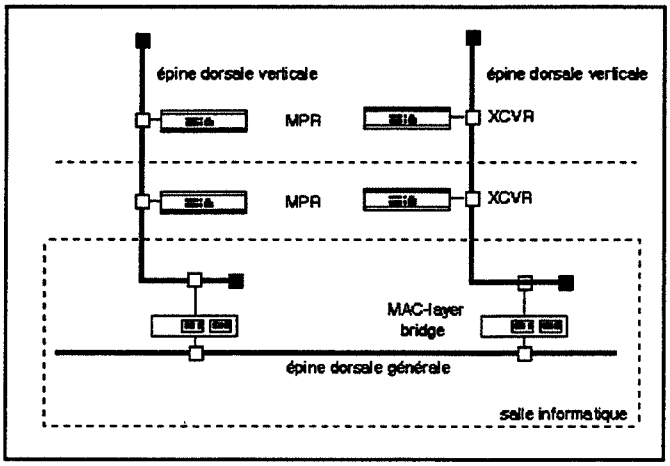


figure 41 : Première solution pour relier les épines dorsales verticales.

Cette solution est préférée si le réseau est important et si une salle informatique existe. De plus, elle a l'avantage de regrouper dans un même lieu les éléments de filtrage, les mainframes, les serveurs. Les experts ont en effet l'habitude de connecter les serveurs ou mainframes propres à une unité professionnelle sur l'épine dorsale départementale correspondante tout en les plaçant dans la salle informatique. Ceci est possible avec cette solution puisque toutes les épines dorsales arrivent jusqu'à cette pièce.

La deuxième méthode consiste à placer une boîte de connexion au premier étage sur laquelle les diverses épines dorsales verticales viendront se connecter, comme le montre la figure 42. Dans le cas où seulement deux épines dorsales verticales sont nécessaires, elles seront fusionnées pour ne faire qu'un seul segment de câble, sans emploi de boîte de connexion, si la longueur de ce segment vérifie la technologie ethernet. La boîte de connexion peut être un répéteur simple ou multi-portes, un bridge ou un routeur.

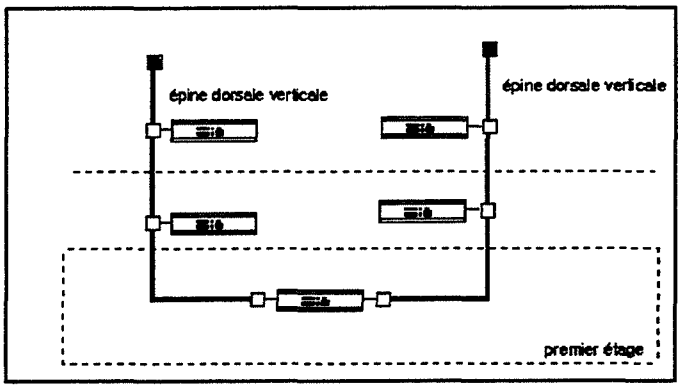


figure 42 : Deuxième solution pour relier les épines dorsales verticales.

C'est cette deuxième solution qui a été implantée du fait de la localisation des boîtes de filtrage à chaque étage. La première requiert à la fois de modifier ce module mais aussi celui effectuant l'ajout des boîtes de filtrage puisque ces dernières doivent être placées dans une même pièce et non à chaque étage comme c'est le cas actuellement.

3.4.2.4 Raffinement et optimisation de la solution obtenue précédement.

Généralement les experts procèdent de la manière suivante lorsqu'ils répondent à l'appel d'offre d'un client : un solution détaillée est fournie, suivie d'options possibles au réseau proposé. Ces options correspondent à des raffinements ou à des optimisations de la solution. Par exemple, les experts de BIM suggèrent les options suivantes pour l'amélioration d'une des solutions : câblage d'une épine dorsale générale en fibre optique, installation d'une station d'administration du réseau, ajout de bridges pour la redondance, etc. L'objectif de cette tâche était donc de répondre à cette caractéristique. Par manque de temps, cette étape n'a pu être implémentée.

3.4.3 L'espace de travail et les objets de conception.

Comme pour tout système à base de connaissances, l'espace de travail de NEST est constitué d'une part des données du problème à résoudre qui sont des instances des classes définies dans la base de connaissances et d'autre part de l'état courant de la résolution : l'espace des solutions. Pour modéliser ce dernier, des unités de travail, appelées des objets de conception, ont été définies par l'intermédiaire du mécanisme de classe. Les valeurs de leurs propriétés sont affectées au cours du processus de résolution. Les objets de conception sont des structures génériques qui ont uniquement une fonction opérative pendant le processus de conception parce qu'ils correspondent à des représentations de granules présentes dans la pensée des experts humains quand ils conçoivent un réseau. Ils sont donc explicitement utilisés par les concepteurs à diverses étapes du processus de conception : conception d'une épine dorsale verticale, conception d'un sous-réseau, etc. Dans NEST, trois types de tels objets ont été formalisés, un pour le squelette du réseau, un pour les diverses épines dorsales verticales et un pour les

différents sous-réseaux à chaque étage. La figure 43 présente le type d'objet de conception le plus pertinent : celui des sous-réseaux. Cette classe *Floor_networks* permet à NEST de mémoriser des informations déterminées lors de la réalisation d'un sous-réseau telles que les types de câble possibles pour ce sous-problème, le type de câble choisi parmi les types possibles, la structure, le nombre de branches qui est fortement dépendant de la structure choisie, la distance à câbler, le type de boîte de connexion, etc. Le lecteur pourra consulter l'annexe 1 : Base de connaissances de NEST pour consulter les autres objets de conception.

En plus des objets de conception qui correspondent à des solutions partielles, une instance de la classe *Networks* est créée au cours de la résolution, ainsi que des instances de câble, d'équipement technique, de connecteur, de terminateur, etc. Les valeurs de leurs propriétés sont également affectées au cours du processus de résolution de réseau.

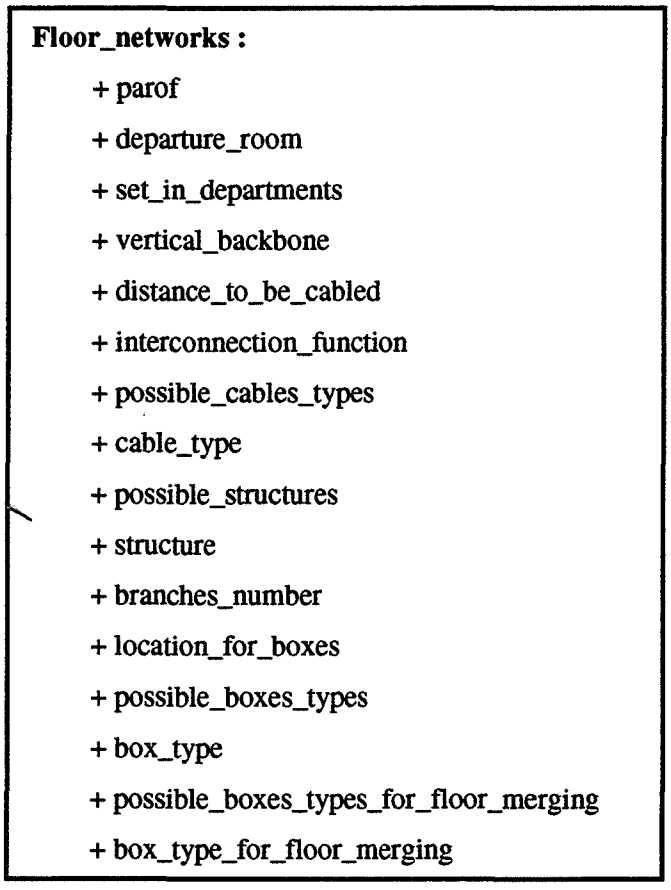


figure 43 : L'objet de conception pour les sous-réseaux.

3.4.4 Gestion des contraintes - prises de décision.

L'objectif ce paragraphe est double : montrer d'une part comment sont pris en compte les contraintes listées au paragraphe 2.3.2 page 107 et d'autre part comment l'outil de conception prend les décisions concernant des éléments de la solution finale. En fait, ces dernières sont très liées à la gestion des contraintes puisque la satisfaction de celles-ci vont permettre de limiter voir de supprimer les choix sur les différentes solutions partielles possibles.

Etre amener à gérer des contraintes implique la reconnaissance de plusieurs éléments :

- un ensemble de variables dont les valeurs devront être affectées au cours du processus de conception,
- un ensemble fini des valeurs possibles pour chaque variable,
- un ensemble de contraintes.

Le problème à résoudre est d'assigner une valeur à chaque variable choisie parmi les valeurs possibles de telle sorte que toutes les contraintes soient simultanément satisfaites.

Dans NEST, ces trois types d'information sont existants plus ou moins explicitement.

- Les variables correspondent, pour la plupart, aux propriétés des objets de conception. Des exemples de telles variables sont la propriété type d'un squelette de réseau, la localisation retenue d'un câble vertical (propriété *vertical* d'un *Vertical_backbones*), le *type* de câble pour un sous-réseau, sa structure, etc. A partir de la donnée des valeurs de ces propriétés, il est possible d'envisager la création des éléments du réseau puisque est alors connu le type de câble à employer, la structure d'un sous-réseau influençant le choix des boîtes de connexion, etc.

- Les valeurs possibles pour chaque variable ne sont pas définies explicitement. Dans certains cas, elles sont calculées. Par exemple, les localisations possibles des épines dorsales sont déterminées par NEST en recherchant la liste de toutes les montées verticales possibles (gaines techniques, ascenseurs, escaliers) existantes dans le bâtiment considéré. La liste des types de câbles possibles pour un sous-réseau est aussi calculés et sauvegardés dans la propriété *possible_cables_type* de la classe *Floor_networks*. Pour cela, les différentes sous-classes de câble sont recherchées dans la base de données. Sur cette liste résultante de cette recherche, un filtre est appliqué éliminant les éléments ne vérifiant pas les contraintes de validité. Enfin, les types de câbles sont ordonnés à l'aide de coefficients de priorité fixés arbitrairement en fonction d'observations de l'activité de conception des experts. Du câble coaxial fin a la priorité la plus forte (5) alors que du câble coaxial épais a une priorité de 4 et les paires torsadées et la fibre optiques 3. D'autres exemples de valeurs calculées possibles pour une variable pourraient être donnés mais passons au deuxième type : celles étant incluses dans les prédicats. C'est, par exemple, le cas du type d'un squelette dont les valeurs peuvent être : *only_one_floor*, *one_department_by_floor*, *mixed_departement*. Une règle pour chaque cas particulier de problème possible : un bâtiment avec un seul étage, une unité professionnelle à chaque étage, le cas général, a été écrite (le lecteur pourra consulter dans l'annexe 6, le prédicat *select_type/3* du fichier skeleton.pro).

- Les contraintes utilisées dans NEST sont de deux types : celles de validité et celles de préférence. Les contraintes de validité doivent être impérativement satisfaites. Elles permettent de restreindre le nombre de solutions possibles pour une ou plusieurs variables et parfois de déterminer totalement la valeur d'une variable. Les contraintes de préférence permettent de départager les solutions

restantes après la vérification des contraintes de validité. Par leur intermédiaire une seule solution sera retenue.

Les données du problème à résoudre, elles aussi, restreignent les possibilités au niveau des choix des valeurs de certaines variables. Typiquement, l'exemple précédent du type de squelette de réseau est contraint par l'architecture du bâtiment et la répartition des unités professionnelles. Ces informations suffisent à elles seules à déterminer le type de squelette adapté à un problème particulier.

L'annexe 2 récapitule ces informations pour chaque tâche : les variables dont les valeurs sont affecté au cours de la réalisation de cette tâche sont listés dans *post-action*, les valeurs possibles pour ces variables peuvent être trouvées dans *possible-answers*, les contraintes de validité et de préférence sont respectivement dans *validity constraints et preference constraints* et les données initiales contraignant le choix des valeurs des variables sont regroupées dans *used-information*.

3.4.5 Problèmes rencontrés.

Lors de l'implémentation de certains modules de l'outil de conception, j'ai été confrontée à plusieurs problèmes. Ceux-ci vont être relatés ci-après.

3.4.5.1 Vérification du principe de départementalisation.

Enoncé des problèmes :

Lors de l'étape "Sélection d'un squelette" pour le réseau, la phase suivante "répartition en sous-réseau" est préparée. Le regroupement des pièces devant être connectées à un même sous-réseau est effectué. Pour cela, on utilise le principe de la départementalisation qui incite à regrouper les pièces appartenant à un même département dans un même sous-réseau pour réduire le trafic du réseau global. Des seuils minimum et maximum de nombre de machines que peut supporter un sous-réseau doivent être respectés. Enfin, il faut veiller à ne pas séparer un serveur des machines qui en sont dépendantes toujours pour des problèmes de trafic. La combinaison de ces trois règles de validité rend la répartition difficile à implémenter parce qu'il est nécessaire de s'assurer qu'un choix effectué pour satisfaire l'une d'elles ne va pas contredire une autre règle. Par exemple, après avoir appliquer le principe de départementalisation, il faut vérifier que le nombre de machines pour chaque sous-réseau ainsi créé n'est pas supérieur au seuil maximum fixé par les experts humains ou inférieur au seuil minimum.

Résolution :

J'ai employé le procédé suivant pour résoudre ce problème (voir en annexe 6 dans le fichier skeleton.pro le prédicat *distribution/4*):

- Recherche des départements d'un étage ; Les pièces appartenant à aucun département et devant cependant être connectées au réseau sont regroupées dans un département fictif.
- Pour chaque département :

- Vérification de la relation client-serveur (prédicat *check_server/6*). Si un serveur sert une machine appartenant à un autre département, un département fictif est créé regroupant le département initial et celui auquel appartient cette machine.

- Vérification des seuils (prédicat *test_thresold/7*) par rapport au nombre (n) de connexions souhaitées pour le département considéré ou le département fictif créé précédemment. Si n est inférieur au seuil minimum, on effectue un regroupement de deux départements. Par contre, si n est supérieur au seuil maximum, le département est divisé en deux nouveaux départements fictifs.

En fin de traitement, on se retrouve avec une liste de départements créés par NEST ou non, consistant chacun un sous-réseau à l'étage considéré. Chacun d'eux est relié à une épine dorsale verticale. Ces résultats sont mémorisés dans la propriété *floor_departments* des objets de conception *Vertical_backbones* sous la forme de liste de liste. Les sous-listes ont pour premier élément un étage et pour deuxième argument une liste de départements.

Résultats :

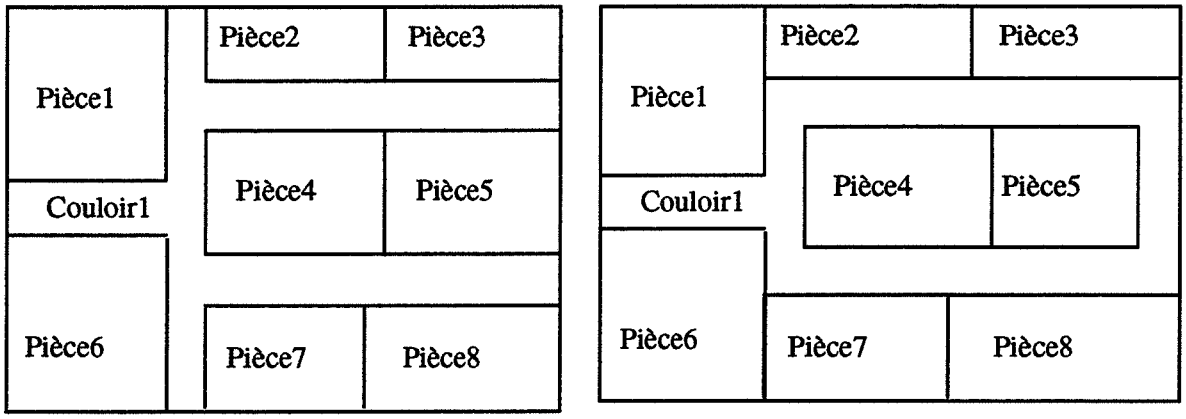
Le nombre d'épine dorsale verticale est fixée pendant cette phase de répartition en sous-réseaux ainsi que pour chacune d'elles la liste des montées verticales (gaines techniques, escaliers, ascenseurs) possibles. Il reste alors à trouver une localisation pour chacune des épines dorsales. C'est le rôle du prédicat *place_skeleton/1*. Pour effectuer ce choix, plusieurs règles sont à notre disposition. Il est préférable de sélectionner celui le plus prés du centre du bâtiment ou le plus prés de la salle informatique pour des raisons de longueur minimale de câble. Principalement pour des raisons d'esthétique, une gaine technique est préférable à un escalier ou à un ascenseur. Mais, il est aussi nécessaire de vérifier que la montée verticale choisie n'est pas une gaine technique dans le cas d'un bâtiment ancien car généralement ces gaines sont mal conçues. A l'opposé dans un bâtiment moderne, on sait qu'il y aura assez d'espace pour faire passer des câbles et particulièrement du câble coaxial épais qui requière une place suffisante. Enfin, cette montée verticale doit permettre de relier tous les étages dans lesquelles ont été prévu des sous-réseaux raccordés à cette épine dorsale.

3.4.5.2 Calcul de la distance à câbler.

Comme déjà mentionné, la distance à câbler est la longueur d'une "ficelle" minimale passant par les couloirs et reliant toutes les pièces susceptibles d'être câblées. Partant de la pièce choisie pour le départ du sous-réseau, les couloirs doivent être parcourus. L'information à notre disposition est la liste des murs de chaque couloir et les informations relatives à ses murs (longueur, pièces auxquelles il appartient permettant de connaître les pièces voisines au couloir considéré, etc). Le calcul de cette longueur s'effectue en fonction du type de couloir. Deux types sont définis : rectiligne et circulaire et sont présentés à la figure 44. On appelle couloir circulaire un couloir contenant une boucle alors qu'un couloir rectiligne est constitué de lignes droites.

Le type d'un couloir est facilement identifiable grâce à la convention choisie pour leur liste de murs. En fait, c'est une liste de listes de murs tel que chaque liste constitue un parcours fermé. Un couloir rectiligne est constitué d'une seule sous-liste alors qu'un couloir circulaire contient au moins deux listes. Le cas des couloirs circulaires m'a posé moins de problème car j'ai considéré que la distance à

câbler d'un tel couloir est la plus grande parmi les additions de longueur des murs de chaque liste. Ce qui est une bonne approximation car les experts humains ne sont pas précis au mètre prés. Pour l'autre type de couloir, j'ai rencontré les problèmes listés ci-après.



Couloir rectiligne Couloir circulaire

figure 44 : Types de couloir.

Enoncé des problèmes :

• Point d'arrêt de la "ficelle" :

Pour calculer cette longueur, on utilise la liste des murs composant le couloir. Le problème ici est de savoir quand s'arrêter dans le parcours de cette liste de murs. Le cas 2 de la figure 45 montre bien l'ampleur du problème. Il faut s'arrêter au mur w8 et ne pas prendre en compte w9, w10, w11 et w12. Ce test est effectué par recherche de la dernière extrémité dans la liste des murs à parcourir.

• minimisation de la longueur :

Suivant le chemin emprunté par la "ficelle", la longueur ne sera pas la même. Un schéma (figure 45) illustre mieux le problème. On s'aperçoit tout de suite que le parcours du câble est beaucoup plus long dans le cas 1 que dans le deuxième. Au niveau pratique, ce problème est pris en compte par calcul des deux longueurs et acceptation de la plus courte.

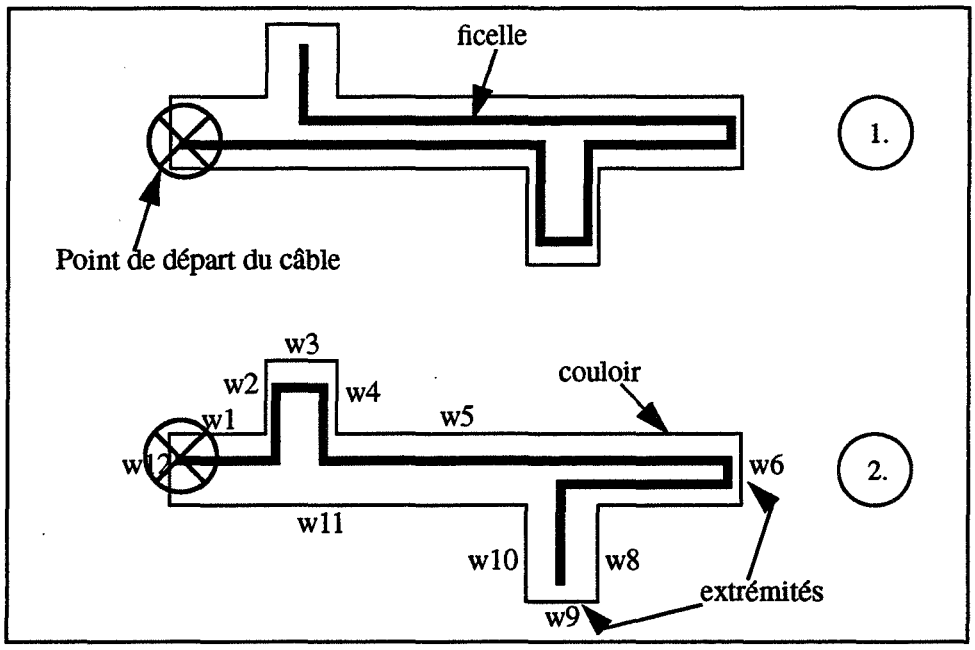


figure 45 : Minimisation de la longueur.

Résolution :

Pour résoudre ces deux problèmes, j'ai employé l'algorithme suivant (le lecteur pourra aussi consulter dans l'annexe 6 le fichier distance.pro) :

- Recherche de la liste des pièces à câbler pour l'étage considéré et pour la liste des départements concernés.
- Restriction de la liste des couloirs de l'étage à la liste des couloirs permettant de relier les pièces à câbler.
- Recherche du couloir proche de l'arrivée de l'épine dorsale verticale à l'étage dans le cas d'un bâtiment à plusieurs étages (sinon recherche du couloir proche de la salle où seront localisés les boîtes de connexion et où sont placés les "grosses" machines - serveurs, "mainframes").
- Calcul de la longueur à prendre en compte pour chaque couloir utile :
    - Recherche des murs (1 ou 2) voisins de l'arrivée de l'épine dorsale verticale ou voisins du couloir précédent et appartenant à un des couloirs non considérés.
    - Réorganisation de la liste des murs du couloir en quatre listes. En fait, le calcul s'effectue en considérant la partie à droite du point de départ du câble puis à gauche de celui-ci. Si on prend l'exemple de la figure 45, les listes à considérer sont : _Wall1_right = [w14, w13, w12, w11, w10, w9, w8, w7, w6, w5, w4, w3], _Wall2_right = [w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13], _Wall1_left = [w14, w15, w16, w17, w18, w19, w1], _Wall2_left = [w2, w1, w19, w18, w17, w16, w15].
    - Suppression des murs indésirables pour le calcul dans chacune des quatre listes.

- Comparaison des sommes des longueurs des murs des deux listes de droite pour ne garder que la liste de longueur la plus petite de même pour les listes de gauche.
- La longueur finale pour ce couloir est la somme des deux longueurs.

- La distance à câbler est la somme des longueurs pour chaque couloir.

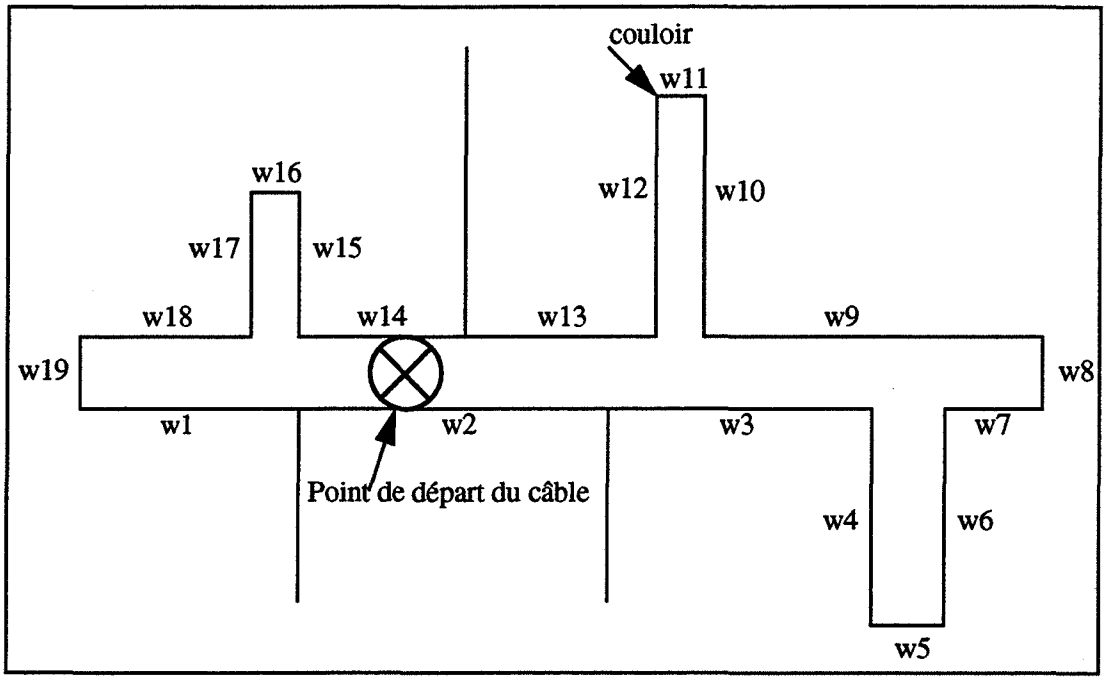


figure 46 : Calcul de la longueur à prendre en compte pour un couloir.

3.4.5.3 Localisation des câbles.

A partir des valeurs précédemment calculées par d'autres modules pour le type de câble et la structure du sous-réseau considéré, ce module fournit :

- le nombre de branches que doit comporter le sous-réseau. Ce calcul est utile pour une structure en étoile. Pour une structure linéaire, ce nombre est égal à 1 alors que pour une structure avec un répéteur simple, il est égal à 2.
- la longueur de chaque câble,
- leur localisation,
- pour chaque câble, sa propriété *connected_elements* spécifiant les éléments auxquels il est connecté.

Enoncé des problèmes :

Le premier problème est de trouver une localisation la plus adaptée en fonction du type de câble et du type de localisations existantes (faux-planchers, faux-plafonds, gaines horizontales, couloirs). Ayant choisi une localisation, il faut :

- considérer toutes les pièces qui peuvent être câblées en faisant passer le câble par cet endroit,

- prévoir des bouts de câble entre chaque machine et/ou chaque connexion souhaitée,
- trouver leur longueur en prenant en compte :
    - la longueur des murs communs à la pièce ou les pièces considérées et à la localisation choisie,
    - le nombre de bouts de câble à placer le long de ces murs, qui est fonction du nombre de connexions à prévoir pour câbler cette ou ces pièces,
    - le fait qu'on peut avoir des pièces à ne pas câbler,
    - que les câbles doivent respectés une certaine longueur minimale fixée par la technologie Ethernet en fonction du type de câble. Dans le cas où les bouts de câbles à mettre sur la longueur d'un ou plusieurs murs d'une pièce est trop courte, un "fan-out" doit être prévu.

Enfin, si toutes les pièces à câbler au sous-réseau en court n'ont pas été considérées, il faut choisir une nouvelle localisation proche de la précédente. La figure 45 est révélatrice de ces problèmes.

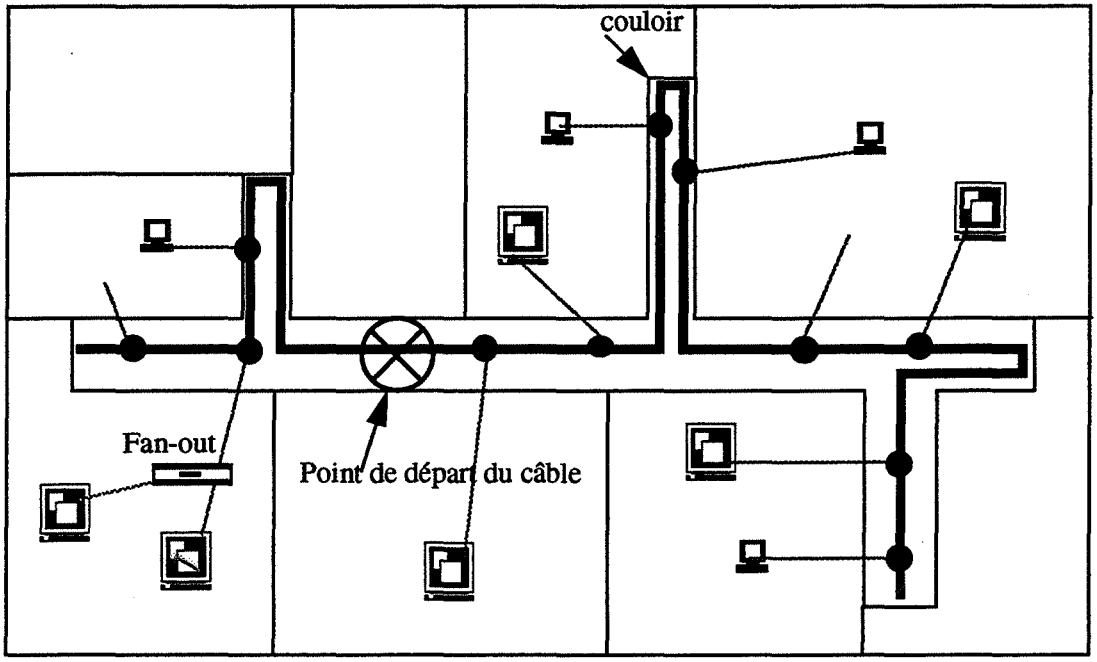


figure 47 : Exemple de câblage d'un étage.

Le problème à résoudre n'est pas tout à fait similaire suivant que l'on à affaire à une structure en étoile ou à une structure linéaire ou avec un répéteur simple. Il est nécessaire de mémoriser la longueur du segment de câble qui est en train d'être créé pour vérifier la contrainte Ethernet sur la longueur maximale d'un segment de câble en fonction du type de câble employé. Lorsque cette longueur est atteinte, avec une structure à répéteur simple, une pièce voisine du point auquel on est parvenu doit être mémorisée dans *location_for_boxes* de l'objet *Floor_networks* courant. C'est là que l'on devra plus tard placer le répéteur. Avec une structure en étoile, à chaque fois que cette longueur maximale est atteinte, il faudra revenir au point de départ où sera localisé le répéteur multi-porte ou une autre boîte de connexion assurant une structure en étoile. Les prédicats que j'ai réalisés prennent en compte ce problème et on retrouve dans le fichier cables_location.pro en annexe 6des prédicats dont le premier argu-

ment est la structure du réseau. C'est, par exemple, le cas de *set_cable_location/9* pour lequel deux règles ont été écrites, l'une pour la structure en étoile, la seconde pour les autres structures.

Résolution :

La méthode de résolution que j'ai employée pour résoudre ces problèmes est la suivante (consulter le fichier cables_location.pro en annexe 6):

- Recherche s'il existe un faux-plancher sur tout l'étage. Si c'est le cas, le câble passera à travers ce faux-plancher et permettra de couvrir toutes les pièces. Sinon :
  - Recherche d'un autre type de localisation (prédicat *select_cable_location/5*)en appliquant les priorités suivantes : 4 pour une gaine technique si le type de câble choisi n'est pas du coaxial épais et si le bâtiment n'est pas ancien, 3 pour un faux-plafond, 2 pour une gaine technique si le type de câble choisi n'est pas du coaxial épais et 1 pour un couloir.
  - Placer les câbles (prédicat *set_cable_location/9*) dans cette localisation :
    - en indiquant que le premier câble doit avoir un élément terminateur,
    - en parcourant les pièces le long de cette localisation,
    - pour chaque pièce à câbler (prédicat *find_room_to_connect/15*), en déterminant le nombre de bouts de câble nécessaires et si besoin en prévoyant des "fan-outs",
    - en mémorisant la longueur du segment de câble en court de création,
    - lorsque la longueur maximale du segment de câble est atteinte (prédicat *tes_length/3*)ou lorsque toutes les pièces à câbler ont été parcourue, en introduisant un autre câble devant avoir un élément terminateur et si besoin est, en débutant un nouveau segment par un câble devant avoir un élément terminateur (prédicat *terminate_cable/10*).

Résultats :

Durant cette étape, le choix des connecteurs (ou des éléments de connexion) entre les câbles et les machines ou prises ou "fan-outs" et celui des terminateurs de câble ne sont pas encore arrêtés. C'est seulement à l'étape suivante sélecteur des connecteurs que cette opération est effectuée. Une convention a dû être établie pour indiquer à ce module à quelle machine doit être connectée un câble, s'il est nécessaire d'avoir un terminateur de câble, l'autre câble auquel il doit être connecté, etc. Ces informations sont introduites dans la propriété *connected_elements* des instances de câbles créées. A la sortie du module localisateur de câbles, cette propriétés sera de l'une des forme suivantes :

```
* [@terminating_element, [@connecting_element, _liste_des_machines_ou_boîtes_ou_prises, _cable]]

* [[@connecting_element, _liste_des_machines_ou_boîtes_ou_prises, _cable], [@connecting_element, _liste_des_machines_ou_boîtes_ou_prises, _cable]]

* [[@connecting_element, _liste_des_machines_ou_boîtes_ou_prises, _cable], @terminating_element]
```

Les autres propriétés des objets câbles créés par ce module sont : partof (réseau), pass_in (gaines techniques), pass_along (murs) et length (naturel).

3.4.5.4 Lien entre les différentes épines dorsales verticales.

Ce module a la charge de relier les différentes épines dorsales et est effectué par le prédicat *merge_vertical_backbones/1* (annexe 6, fichier vertical_design.pro).

Un cas particulier a d'abord été défini : celui comportant deux épines dorsales passant dans une même localisation. Le problème est alors de retrouver leurs extrémités, de les supprimer et de faire un seul câble de ces deux câbles extrémités, bien entendu si ce nouveau câble n'est pas trop long selon la technologie Ethernet.

Dans le cas général, une boîte de connexion doit être sélectionnée (prédicat *choose_a_box/4*) pour faire la jonction entre toutes les épines dorsales. Les problèmes sont alors de déterminer les extrémités adéquates des épines dorsales (prédicat *find_terminating_cables_in_floor/3*), de prolonger les câbles extrémités jusqu'à la boîte de connexion (prédicat *link_vertical_backbone/4*).

3.5 Résultats.

3.5.1 Etat actuel.

L'état du système est tel que toutes les caractéristiques mentionnées ci-dessus ont été implémentées et intégrées à l'interface multi-modes, de telle sorte qu'il est possible d'analyser ou de faire une requête de conception de réseau. Une fois que l'utilisateur a dessiné un bâtiment, localisé les machines et spécifié ses exigences, NEST peut calculer un réseau.

D'autre part, NEST a été couplé à l'interface MMI^2, permettant ainsi d'avoir des échanges plus conviviaux avec ce système.

Comme pour les différentes analyses possibles sur un réseau, la méthode permettant la conception d'un réseau est attachée à l'objet Networks.

L'outil de conception est totalement automatique, c'est à dire qu'un réseau détaillé et complet est obtenu après avoir entré les informations nécessaires : topographie des bâtiments, exigences de l'utilisateur, budget, etc (voir Annexe 3 : Prérequis nécessaires avant une requête de conception.). Il permet donc la création d'un réseau avec sélection et localisation des câbles, boîtes de connexion, liens vers les machines, etc. La figure 48 est un exemple de réseau créé par NEST, par l'intermédiaire de l'interface MMI^2.

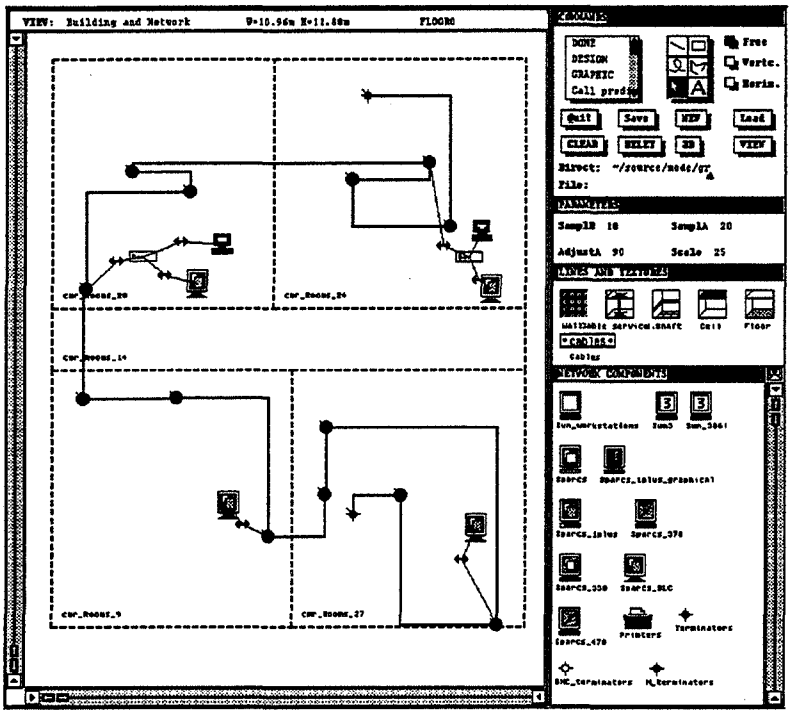


figure 48 : Un exemple de réseau obtenu par NEST.

Une fois que l'utilisateur a dessiné l'étage avec ses diverses pièces et placé les machines, une requête de conception est envoyée à NEST qui, en retour, propose une configuration de réseau. Ici du câble coaxial épais a été choisi, avec des transceivers (points en gras) plus des câbles AUI pour lier les machines au câble coaxial épais. Deux fan-outs sont placés dans les pièces pour lesquelles l'utilisateur a demandé dix connexions.

3.5.2 Liaisons avec l'interface MMI^2.

Quelques connaissances du domaine de la conception de réseaux ont été placées dans un des modules de l'interface appelé l'expert du domaine (voir paragraphe 1.9 page 88 sa description). Cet expert est, entre autres, chargé de répercuter les actions de l'utilisateur vers NEST quand nécessaire et également d'activer les méthodes implantées dans NEST afin de répondre à une requête de l'utilisateur. Ainsi il met à jour la base de connaissances quand l'utilisateur fournit une donnée relative à un problème particulier et il interroge NEST dans le cas d'une question concernant une tâche particulière que NEST est apte à résoudre. Pour ce faire, si cet expert reçoit un message correspondant à une simple action sur la base de connaissances, celle-ci sera consultée ou mise à jour si besoin est. Sinon, le message est associé à un plan de tâche. Un tel plan est constitué des deux éléments suivants : les prérequis (données qui doivent être connues pour pouvoir exécuter le plan proprement dit) et la liste des prédicats à activer pour réaliser le message correspondant au plan de tâche. Dans le cas où les prérequis du plan de tâche identifié ne sont pas complètement spécifiés, un sous-dialogue est initialisé pour

acquérir, via des questions à l'utilisateur, les informations manquantes. Par exemple, pour pouvoir activer le processus de conception de réseaux de NEST, une description détaillée du site à câbler, des machines à connecter au réseau, des exigences du client, etc, doivent être fournies. En annexe 3, la liste des prérequis nécessaires à la conception d'un réseau est détaillée. De même que pour la conception, les différentes analyses possibles d'un réseau correspondent à des plans de tâche particuliers. D'autres plans sont également définis et sont associés à des prédicats Prolog écrits dans NEST. Ces prédicats sont présentés en annexe 5.

- détruire un bâtiment pour supprimer dans la base de connaissances toutes les informations concernant un bâtiment (étages, pièces, murs, gaines techniques, etc).
- détruire un réseau et tous les éléments le constituants (machines, câbles, boîtes de connexion, connecteurs, etc).
- détruire un composant de réseau ; le composant est détruit mais aussi les connexions avec les éléments auxquels il était lié.
- détruire un composant additionnel (disques, mémoires, etc, logiciel).
- déplacer une machine dans un autre pièce ; les connexions de la machine sont d'abord détruites, puis la machine est déplacée dans la pièce désirée et elle est connectée à un connecteur vide dans cette pièce. L'opération échoue s'il n'y a pas de connecteur vide.
- connecter ou déconnecter une machine à un connecteur donné.
- vérifier si une machine a un disque.
- spécialiser ou généraliser un objet ; par exemple, cette opération permet de spécialiser un câble appartenant à la classe générale Cables comme étant un Thin_cables câble, coaxial fin.
- définir un serveur de réseau.
- insérer ou supprimer les exigences de l'utilisateur, par exemple le budget.

3.5.3 Apport de l'interface MMI^2 pour un système à base de connaissances.

Diverses fonctionnalités de cette interface font qu'elle est d'un grand intérêt dans l'utilisation de NEST et certainement pour bien d'autres systèmes à base de connaissances :

- L'expert graphique [Ben Amara & al 91] apporte des facilités particulièrement pour la description des bâtiments, des étages, des pièces ... mais aussi pour la sortie des résultats puisque des histogrammes, des graphiques, des camemberts peuvent présenter plus explicitement certains résultats. La figure 49 montre un exemple de camembert.
- Un expert gestuel permet des requêtes par dessin d'un geste sur la fenêtre graphique. Par exemple un point d'interrogation dessiné sur une machine indique que l'utilisateur désire avoir des renseignements sur cette machine, son type.
- Les langages naturels assurent la facilité du dialogue entre utilisateur et système, sans avoir à apprendre des commandes particulières.
- Le mixage des modes est très avantageux puisqu'il permet lors d'une même interaction l'emploi de plusieurs modes, particulièrement pour le graphique et le langage naturel. Par exemple, à la

question : "Où est la machine qui a un disque ?", le système "inverse vidéo" la pièce dans laquelle est la machine et répond par le nom de cette pièce.

- Une grande variété d'utilisateurs potentiels peut être intéressée par un tel système, que ce soient des experts du domaine : commerciaux voulant soumettre leur problème au système, experts désirant comparer plusieurs solutions, ..., ou des personnes moins compétentes dans le domaine : un client par exemple. L'interface gère cette variété d'utilisateurs par l'intermédiaire d'un modélisateur de l'utilisateur et le système adapte ainsi sa réponse suivant le type déterminé de l'utilisateur. Par exemple, si lors de sessions préalables, a été acquis l'information : l'utilisateur préfère la représentation des résultats de comparaison sous forme d'histogramme plutôt que par des camembert, un histogramme sera utilisé pour répondre à la question : quel est le coût des machines ?.

- Le module assurant la communication entre l'interface et l'application permet si toutes les informations nécessaires à l'exécution d'une tâche par NEST ne sont pas présentes, d'initialiser un sous-dialogue pour les acquérir auprès de l'utilisateur. De plus, cette acquisition est très conviviale puisque elle est faite en fonction du modèle de l'utilisateur et prend donc en compte son degré de compétence dans le domaine dans l'expression des questions.

3.5.4 Limitations de l'outil de conception.

Malgré le grand effort déployé au cours de ces trois premières années de projet Esprit pour implémenter l'outil de conception, NEST ne couvre pas tous les types de problèmes. En effet, il ne résout que des réseaux "simples", du fait de certaines restrictions qui doivent être imposées aux données :

- NEST ne peut pas configurer un réseau réparti sur plusieurs bâtiments.
- L'outil ne peut pas câbler des pièces non voisines d'une gaine technique ou d'un couloir.
- Les étages doivent avoir impérativement des couloirs sinon la distance à câbler est nulle. Or le choix d'un type de câble et celui d'une structure de sous-réseau dépendent de cette distance.
- L'outil de conception n'offre pas la possibilité de prendre un réseau pré-existant.

D'autre part, les résultats ne sont pas tout à fait optimaux :

- L'outil propose des solutions utilisant uniquement des câbles coaxiaux.
- NEST utilise pour chaque étage un seul type de câble.
- Quand un bâtiment à plusieurs étages doit être câblé, NEST place toujours des éléments de filtrage entre les différents étages. (Ces boîtes peuvent parfois être évitées en câblant deux étages dans un même sous-réseau.)

3.5.5 Extensions possibles.

Une première série d'extensions peut être proposée pour combler les limitations actuelles de l'outil de conception :

- le câblage de plusieurs bâtiments ⇒ être capable de câbler en fibre optique puisque le câble devra passer dans une gaine technique inter-bâtiment qui est considérée comme un environnement perturbé. Sinon ce type de problème se rapproche de ceux déjà traités.
- l'autorisation d'avoir un bâtiment ayant des pièces non voisines d'un couloir ou d'une gaine technique ⇒prendre en considération si un mur est perforable ou non.
- la possibilité d'étages sans couloir ⇒ calcul de la distance à câbler en ne la considérant pas uniquement comme étant la longueur d'une ficelle minimale passant par les couloirs et reliant toutes les pièces susceptibles d'être connectées au sous-réseau.
- extension aux autres types de câble ⇒ amélioration, principalement, du module : sélection d'un type de câble
- câblage d'un étage avec plusieurs types de câbles ⇒ modification de l'approche utilisée pour l'objet de conception Floor_networks et de certains modules de la conception des sous-réseaux.
- implémentation de la deuxième solution pour la fusion des épines dorsales verticales permettant de regrouper dans une même pièce les mainframes, les serveurs, les boîtes de filtrage et les stations d'administration du réseau.

Par ailleurs, l'objectif de ce travail a été de développer, dans un premier temps, un outil de conception totalement automatique, c'est à dire qu'un réseau détaillé et complet est obtenu après avoir entré les informations nécessaires. Dans un deuxième temps, il serait intéressant de rendre plus flexible cet outil afin d'autoriser l'utilisateur à intervenir pendant le processus de conception en modifiant les données qu'il a fournies précédement ou en contribuant au processus de conception. Cela implique de remplacer l'ordre fixé pour l'activation des différents modules permettant la conception d'un réseau par un ordre calculé dynamiquement dépendant de l'analyse faite sur les différentes nouvelles contraintes posées par l'utilisateur. C'est à dire avoir un outil ayant un comportement opportuniste. La migration de l'approche implémentée vers celle à tableau noir semble alors devenir utile puisque ce serait un moyen efficace pour apporter l'opportunisme souhaité. Or l'approche tableau noir semble tout à fait possible étant donné que l'architecture choisie est très modulaire. Ainsi ces modules pourraient être repris comme sources de connaissances d'une solution tableau noir.

NEST ne fournit pas d'explications pour le processus de conception de réseaux. Des connaissances dans le domaine d'application peuvent, cependant être acquises par l'intermédiaire de NEST en utilisant l'outil d'analyse. Celui-ci permet, entre autres, d'étudier les effets produits par une modification d'un réseau. La possibilité de réponses coopératives du système pourrait être améliorée en développant un module d'explications pour montrer le comportement de l'application. Un tel module pourrait ainsi répondre à des questions du type : comment et pourquoi ? Bien que généralement développé au sein d'un système à base de connaissances, l'idée est de construire un tel module en utilisant des méta-connaissances sur le contenu de NEST et plus spécifiquement en utilisant une mémorisation des contraintes appliquées pendant le processus de conception [Technical Annex 92].

3.5.6 Caractéristiques et intérêts de l'outil de conception.

Afin de dégager l'intérêt de la méthode de conception développée dans NEST, une étude comparative est menée, par rapport aux systèmes à base de connaissances existant soit en conception (voir chapitre 4 de la première partie) soit dans le domaine des télécommunications.

L'intérêt de NEST est dû, d'une part à la complexité de la tâche de conception de réseau, et d'autre part au fait qu'un seul autre système, ayant le même objectif, a été conçu. En outre, la méthode de conception adoptée pour l'implémentation de NEST est intéressante selon les aspects suivants :

• la décomposition en sous-problèmes :

Comme nous l'avons déjà noté (page 56), pour réduire la taille théoriquement large d'un espace de résolution, le concepteur décompose généralement le problème global en sous-problèmes. Dans NEST, la conception de réseau est ramenée au sous-problème du câblage de sous-réseaux suivi de la recherche d'une liaison entre les sous-réseaux ainsi créés. De même le problème de la conception de chaque sous-réseau est divisé en différentes étapes : calcul de la distance à câbler, sélection du type de câble, etc. Et ainsi de suite, chaque nouveau sous-but fixé est décomposé jusqu'à trouver un but soluble.

• la planification :

Les différentes étapes permettant à NEST de résoudre son problème de conception de réseau sont hiérarchiquement organisées. Deux niveaux d'abstraction sont définis, le premier étant constitué de quatre phases, chacune d'elles étant à son tour réalisée par l'exécution d'un plan de contrôle. Le système PRIDE (voir page 57) a également une résolution hiérarchiquement organisée, le sommet de la hiérarchie traitant des aspects plus généraux alors que les niveaux inférieurs ont affaire à des aspects plus spécifiques, comme c'est le cas d'ailleurs pour NEST.

• la modularisation :

Les sous-étapes sont réalisées par l'activation de modules spécialisés dans l'accomplissement d'une tâche bien spécifique, par exemple, choisir le type de câble. La plupart des systèmes destinés à la conception implémentent cette notion d'agent spécialisé, que ce soit avec une approche planifiée ou avec une architecture tableau noir (ils sont alors appelés : sources de connaissances). Par exemple, Brown et Chandrasekaran [Brown & al 86], tout en mettant en oeuvre la caractéristique hiérarchique de l'activité de conception, décomposent les connaissances en différents agents : spécialistes, plans, étapes, tâches et contraintes.

• Utilisations des contraintes.

Les contraintes de la conception de réseau présentées au paragraphe 2.3.2 page 107 sont prises en compte par l'outil de conception de réseau. Elles permettent soit d'éliminer les solutions partielles dont les contraintes ne sont pas vérifiées, soit de faire un choix parmi les solutions restantes pour n'en conserver qu'une. Ce sont alors des contraintes de préférence et non des contraintes de validité. Pour chaque tâche à implémenter, de telles contraintes ont été définies (voir annexe 2, formalisation des tâches).

• l'espace des solutions :

L'espace des solutions de NEST est organisé autour d'objets prédéfinis représentant des solutions partielles qui vont aider NEST à s'acheminer vers la solution finale. La conception peut également être interprétée dans une représentation des espaces d'états comme un état initial qui est transformé, en utilisant des connaissances, en une série d'états solutions. Par conséquent, la résolution de problèmes peut être vue comme un processus de recherche parmi des solutions alternatives qui satisfont à un certain but. C'est la méthode employée dans le système PREDICKT [Gero & al 87]. J.S. Gero et R. Oxman ont développé des concepts qui permettent à un système expert d'être utilisé à la fois pour le diagnostic et la synthèse de systèmes de conception. PREDICKT implémente ces concepts dans le domaine des cuisines domestiques.

NEST fournit une seule solution, ce qui ne veut pas dire que ce soit la seule possible mais celle satisfaisant les critères de sélection de solutions partielles établis dans NEST. En effet, pour éviter d'avoir à gérer un ensemble de solutions partielles et par la suite tenter de sélectionner la "meilleure", des critères de choix fixent la solution partielle à des points clés du mécanisme de résolution. Par exemple, même si du câble coaxial fin est choisi du fait que la distance à câbler est suffisamment petite, il n'en reste pas moins que du câble coaxial épais ou de la fibre optique aurait pu être possible mais à un coût plus élevé.

• Langage hybride : modèle centré objets et Prolog :

Les capacités du modèle centré objets ont permis d'implémenter très facilement tous les éléments de base relatifs à la conception de réseau que ce soit pour les composants de réseau ou pour les informations topologiques. D'autre part la possibilité d'attacher des méthodes écrites en Prolog à ces objets fut un grand avantage pour l'implémentation des connaissances opératoires.

• Deux outils complémentaires : un pour l'analyse et un pour la conception.

NEST apporte la possibilité de créer son propre réseau et de le tester suivant les cinq analyses décrites précédemment. Il permet aussi de modifier partiellement un réseau créé afin d'évaluer l'impact de ces changements et il répond à certaines questions sur le domaine. Par exemple, la question : est-il possible de mettre du câble coaxial fin dans cette gaine technique ? peut lui être posée.

Pour conclure ce chapitre, j'aimerais signaler que ces trois premières années de projet Esprit II MMI^2 ont abouti à un prototype [Wilson 91] au mois de Novembre 91. Il intègre les différents modes de communication et est interfacé avec l'application NEST. Ce prototype a été présenté lors de la semaine ESPRIT91 à Bruxelles. Des problèmes restent encore à résoudre tels que la possibilité d'interactions entre la langue naturelle et le langage de commande, l'enrichissement des générateurs des différents langages naturels, l'amélioration des performances des différents modules du système, l'interfaçage de la composante 3D de l'outil graphique avec NEST (tous deux étant capables indépendam-

ment de gérer cette dimension). Ces améliorations vont pouvoir être envisagées puisque le projet a été reconduit pour deux nouvelles années.

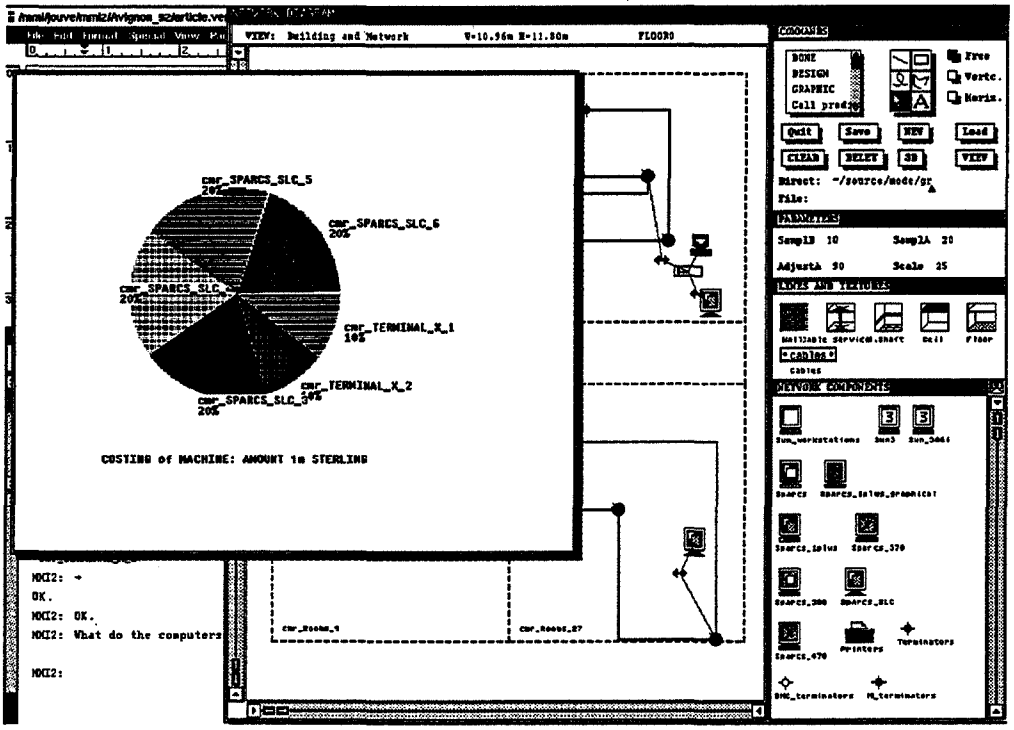


figure 49 : La représentation camembert du coût des machines.

La question "Quel est le coût des machines ?" a été posé au système en langage naturel. L'interface a choisi de répondre sous la forme d'un camembert à cette question. Six machines sont concernées. Les terminaux X représentent chacun d'eux 10% du coût total des machines alors qu'une sparc_SLC participe à 20%.

Conclusion.

Cette thèse a permis, tout d'abord de faire le point sur différents aspects de la représentation des connaissances. Après avoir introduit les problèmes auxquels on est confronté lorsqu'on essaie de modéliser des connaissances et avoir présenté les formalismes généraux de représentation des connaissances, je me suis intéressée à des points particuliers qu'il me semblait utile de développer avant de présenter la réalisation effectuée au cours de ces années de thèse. Ainsi les concepts propres à la programmation objet ont été énumérés puisque le langage que j'ai eu à utiliser est un langage prolog auquel, entre autres, une couche orientée objets a été adjointe. Le système ayant l'ambitieux objectif d'être un système pour la conception, les différents travaux dans ce domaine ont été présentés.

Ayant établi les bases théoriques relatives à mes recherches, la réalisation a pu être abordée. Celle-ci a consisté à implémenter un système à base de connaissances pour la conception de réseaux locaux satisfaisant les protocoles : Ethernet et TCP/IP. A travers le développement de NEST, deux principaux axes de recherche ont été entrepris et implantés : la modélisation d'une activité de conception pour laquelle l'intelligence artificielle a relativement peu de théories et la configuration de réseaux locaux pour laquelle peu de systèmes experts ont été conçus. L'originalité de l'approche est d'avoir employé plusieurs formalismes : la programmation orientée objets pour les concepts de base, prolog pour les différentes méthodes, un paradigme de plan de contrôle / tâche / sous-tâche / module pour l'architecture générale. En effet, les connaissances opératoires ont été organisées sous la forme de différents modules, chacun d'eux réalisant une tâche particulière. D'autre part, l'ordre d'activation de ces modules a été fixé par l'intermédiaire d'une hiérarchie de plans de contrôle. L'un d'eux : le plan de contrôle principal est au sommet de cet arbre et orchestre les autres plans. L'approche choisie assure aussi une bonne prise en compte des nombreuses contraintes de la conception de réseaux. Pour ce faire, deux types de contraintes ont été fixées : celles de validité que toute solution possible doit impérativement vérifier et celles de préférence permettant de départager les solutions possibles. Quant aux éléments de base, étant nombreux et facilement décomposables en familles ayant des caractéristiques communes, ils ont été représentés à l'aide de la programmation par objets et organisés sous la forme de différentes hiérarchies.

Ayant pour cadre un projet européen Esprit II dont le but est le développement d'une interface multi-modes pour un système à base de connaissances, l'outil réalisé (NEST) bénéficie d'un environnement appréciable pour l'utilisateur du système. En effet, cette interface apporte des facilités de sai-

sies des informations nécessaires à NEST pour résoudre un problème de conception mais aussi un dialogue convivial entre l'utilisateur et le système.

Le système à base de connaissances ainsi développé, permettant à la fois l'analyse ou la conception de réseaux locaux, a été couplé à l'interface MMI^2. Ces trois premières années de projet Esprit 2474 ont abouti à un prototype qui a été présenté lors de la semaine ESPRIT91 à Bruxelles au mois de Novembre 1991.

Enseignements à tirer de ce travail.

Pour terminer cette conclusion, j'aimerais retracer les enseignements que l'on peut tirer de ce travail :

- Fixer un ordre d'activation des tâches pour résoudre un problème de conception de réseaux apportent une plus grande efficacité qu'une méthode opportuniste (telles que dans les systèmes multi-agents). Dans NEST, ceci est dû au fait que l'approche suivie fixe à chaque étape un élément de la solution finale qui ne sera pas remis en cause plus tard. Les méthodes multi-agents développent des solutions en parallèle qui peuvent, à certains points du processus de résolution être supprimées parce que ne vérifiant pas certaines contraintes. Ce mécanisme requiert, de plus, des capacités de mémorisation beaucoup plus importantes.

- Concevoir un système implique la vérification de nombreuses contraintes. Il est important, lors de la phase d'acquisition, de mettre bien en valeur ces contraintes. En effet, c'est grâce à elles qu'il est possible pour un système à base de connaissances de prendre des décisions sur le choix d'un élément de solution. Dans NEST, ce choix s'effectue par satisfaction des contraintes et par l'intermédiaire de coefficients de priorité permettant de départager les solutions vérifiant toutes les contraintes courantes. Ces coefficients ont été établis en étudiant le comportement des experts face à un problème de conception de réseau.

- Le modèle centré objets peut être très profitable dans le cas où l'on a à représenter de nombreux éléments de base comme c'est le cas dans le domaine des réseaux locaux. Ceci est d'autant plus vrai que ces entités partagent des caractéristiques communes. Par contre, l'utilisation de prolog pour implémenter les règles nous a paru plus aisée.

Bibliographie.

[Albert 85] *Prolog et les objets.*
P. Albert. Actes des cinquièmes JISEA, p 331-350, Avignon, 1985.

[B.Alcantara & al 88] *The DECADE Catalyst Selection System.*
R. Banares-Alcantara, A.W. Westerberg, E.I. Ko, M.D. Rychener. In Expert Systems for Engineering Design, M.D. Rychener editor, Academic Press Inc., p53-91, 1988.

[Ben Amara & al 91] *Graphical Interaction in a Multimodal Interface.*
H. Ben Amara, B. Peroche, H. Chappel, M.D. Wilson. Esprit'91 Conference Proceedings, p 303-321, November 1991.

[Ben Amara 92] *Un outil graphique pour une interface Multi-Modes.*
H. Ben Amara. Thèse présentée à l'Ecole des Mines de Saint-Etienne, Mars 1992.

[Ben Amara & al 92] *Un outil graphique pour une interface multimodes.*
H. Ben Amara, A. Nahed, B. Peroche. Congrès INFORSID 92, Clermont-Ferrand, Mai 1992.

[Balfroid 89a] *A few expert or knowledge-based systems for network management.*
F. Balfroid. Literature Review. July 1989.

[Balfroid 89b] *Specification of the application.*
F. Balfroid. Deliverable d42, BIM/5, April 1989.

[Balfroid 90] *An Ethernet LAN analysis tool.*
F. Balfroid. MMI2 Progress report BIM/14, February 1990.

[Balfroid & al 90-1] *A Multi-Mode Interface for Man Machine Interaction with knowledge based systems. Knowledge acquisition.*
F. Balfroid, F. Darses, C. Jouve. Deliverable d43, Esprit Project 2474. May 90.

[Balfroid & al 90-2] *A Multi-Mode Interface for Man Machine Interaction with knowledge based systems. Knowledge representation.*
F. Balfroid, G. Doe, C. Jouve. Deliverable d44, Esprit Project 2474. June 90.

[Balfroid & al 91-1] *Knowledge Formalization.*
F. Balfroid, F. Darses, C. Jouve. Esprit Project 2474 internal report. January 91.

[Balfroid & al 91-2] *Proto 91/2, Description of the software of the application.*
F. Balfroid, C. Jouve. Esprit Project 2474 internal report. July 1991.

[Balfroid & al 91] *A Multi-Mode Interface for Man Machine Interaction with knowledge based systems. Description of NEST, a Network design Expert SysTem.*
F. Balfroid, C. Jouve. Deliverable d45, Esprit Project 2474. September 1991.

[Barr & al 81] *The Handbook of Artificial Intelligence.*
A. Barr, E.A. Feigenbaum (Editors). Volume 1, William Kaufmann Inc, California, 1981.

[Baudinet & al 90] *From an Object-Based Prototyping Tool to a complete Object Oriented Environment.*
M. Baudinet, R. Venken. Final report on the Daida Esprit project, 1990.

[BIM 90] *BIM_Probe V. 1.01 alpha Manual.*
BIM. June 1990.

[BIM Networking 88] *Principes de réseaux et Solutions de BIM.*
BIM Networking. December 1988.

[Binot & al 90] *Architecture of a Multimodal Dialogue Interface for Knowledge Based Systems.*
J.L. Binot, P. Falzon, R. Perez, B. Peroche, N. Sheehy, J. Rouault, M.D. Wilson.
Esprit'90 Conference Proceedings, p 412-434, 1990.

[Bittencourt 88] *Représentation des connaissances.*
G. Bittencourt. Rapport de recherche, IMAG, Grenoble, Septembre 1988.

[Bobrow & al 83] *The LOOPS Manual : A Data and Object Oriented Programming System for Interlisp.*
D.G. Bobrow and M. Stefik. Knowledge Based VLSI Design Group Memo KB-VLSI-81-13, Xerox PARC, Palo Alto, California, 1983.

[Brown & al 86] *Knowledge and Control for a Mechanical Design Expert System.*
D.C. Brown, B. Chandrasekaran. Computer IEEE, July 1986.

[Buchanan & al 85] *Rule based expert systems : the MYCIN experiments of Standsford Heuristic Program Project.*
B. Buchanan, E.H. Shortliffe. Readings, MA : Addison-Wesley Publishing Company, 1985.

[Bunt 89] *Information Dialogues as Communicative Action in Relation to Partner Modelling and Information Processing.*
H.C. Bunt. In M.M. Taylor, F. Neel and D.G. Bouwhuis (eds), The Structure of Multimodal Dialogue, North Holland : Elsevier Publishers B.V., p 47-73, 1989.

[Chappel & al 91] *User Modelling for Multi-Modal Co-operative Dialogue with KBS.*
H. Chappel, B. Cahour. Deliverable d3, Esprit Project 2474. January 1991.

[Clancey 83] *The epistemology of a rule based expert system, a frame work for explanation.*
J.W. Clancey. Artificial Intelligence 20. p 215-251. 1983.

[Colmerauer 83] *PROLOG, bases théoriques et développements actuels.*
A. Colmerauer. Techniques et Sciences Informatiques, 2(4), 1983.

[Cox 86] *Object-Oriented Programming : An Evolutionary Approach.*
B.J. Cox. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[CRISS 91] *The french syntactic analyser.*
CRISS. MMI2 Progress report, review, october 1991.

[Dahl & al 70] *SIMULA-67 Common Base Language.*
O. Dahl, B. Myhrhaug, K. Nygaard. Technical Report, Oslo. October 1970.

[Dahl 83] *Logic Programming as a Representation of Knowledge.*
V. Dahl. Computer, 16 (10), 1983.

[Darses & al 89] *The design activity in networking - General remarks and first observations.*
F. Darses, P. Falzon. INRIA, Mars 1989.

[Darses & al 90] *A Multi-Mode Interface for Man Machine Interaction with knowledge based systems. Knowledge acquisition.*
F. Darses, F. Balfroid, C. Jouve. Deliverable d43, Esprit Project 2474. May 90.

[Davis & al 77] *An overview of production systems.*
R. Davis, J.J. King. In E. Elcock and D. Michie (Eds), Machine intelligence 8, p 300-332, 1977.

[DeZegher 89] *Reinforcing Consistency in an Object Oriented System by use of a Constraint Language.*
I. De Zegher, J.M. Trinon, E. Meirlaen. Proceedings of TOOLS '89, Paris, November 1989.

[D2 89] *Common Meaning Representation.*
BIM/13, Deliverable d2, Esprit Project 2474. November 1989.

[Dicenet 88] *Le RNIS. Techniques et Atouts.*
G. Dicenet. Collection technique et scientifique des télécommunications, cnet, enst.
Ed. Masson. 1988.

[Dubois 88] *Les réseaux d'ordinateurs.*
R. Dubois, C. Bresillon. La recherche, No. 204, Novembre 1988.

[Dugerdil 88] *Contribution à l'étude de la représentation des connaissances fondée sur les objets. Le langage OBJLOG.*
P. Dugerdil. Thèse de l'Université d'Aix-Marseille II, 1988.

[Engelmore & al 88] *Blackboard Systems.*
R. Engelmore, T. Morgan. Addison-Wesley Publishing Company, 1988.

[Erman & al 80] *The HEARSAY-II speech understanding system : Integrating knowledge to resolve uncertainty.*
D.L. Erman, F. Hayes-Roth,V.R. Lesser, D.R. Reddy. ACM Computing Surveys 12, 213-253, 1980.

[Erman & al 81] *The design and an example of use of HEARSAY-III.*
D.L. Erman, P.E. London, S.F. Fickas. Proceedings of the seventh international joint conference on artificial intelligence, IJCAI-81, p 409-415, 1981

[Farreny 85] *Les systèmes experts - principes et exemples.*
H. Farreny. Cepadues Editions, 1985.

[Farreny & al 87] *Éléments d'Intelligence Artificielle.*
H. Farreny, M. Ghallab. Traité des Nouvelles Technologies, série Intelligence Artificielle. Ed. Hermès. 1987

[Ferber 87] *Des objets aux agents : une architecture stratifiée.*
J. Ferber. Actes du sixième CARFIA, p 275-286, Antibes, 1987.

[Ferber 89] *Objets et agents : une étude des structures de représentation et de communications en Intelligence Artificielle.*
J. Ferber. Thèse d'Etat. Université Paris VI. Cahiers du LA FORIA n° 74. Juin 89.

[Finin 89] *GUMS, A General User Modeling Shell.*
T. Finin. In A. Kobsa and W. Whalser (eds), User Models in Dialog Systems, Springer Verlag, Berlin, 1989.

[Francony 91] *The Context Expert.*
J.M. Francony. Esprit Project 2474 Literature Review. December 1991.

[Gero & al 87] *Using an expert system for design diagnosis and design synthesis.*
J.S. Gero, R. Oxman. Expert Systems, Vol. 4, No. 1, February 1987.

[Goldberg & al 83] *Smalltalk 80 : The language and its implementation.*
A. Goldberg, D. Robson. Reading, Massachusetts : Addison-Wesley, 1983.

[Gondran 84] *Introduction aux systèmes experts.*
M. Gondran. Editions Eyrolles, 1984.

[Grice 75] *Logic and Conversation.*
H.P. Grice. In Cole P. and Morgan J.L. (eds), Syntax and Semantics, Vol. 3, New York : Academic Press, 1975.

[Gross 88] *Applications of AI technology in communication networks.*
D. Gross. Expert Systems, Vol. 5, No. 3, August 1988.

[Harmon & al 90] *Object-Oriented Programming & Expert Systems.*
P. Harmon, J. Aikins. Notes de cours des dixièmes journées internationales sur les systèmes experts & leurs application, Avignon 90, Mai-Juin 1990.

[Hart & al 78] *PROSPECTOR - A Computer-Based Consultation System for Mineral Exploration.*
P.E. Hart, R.O. Duda, M.T. Einaudi.Mathematical Geology, Vol. 10, No. 5, 1978.

[Haton & al 91] *Le raisonnement en Intelligence Artificielle. Modèles, techniques et architectures pour les systèmes à bases de connaissances.*
J.P. Haton, N. Bouzid, F. Chapillet, M.C. Haton, B. Lâasri, H. Lâasri, P. Marquis, T. Mondot, A. Napoli. InterEditions, iia, 1991.

[Hayes-Roth 83] *The Blackboard Architecture : A General Framework for Problem Solving ?*
B. Hayes-Roth. Heuristic Programming Project, Report No HPP-83-30, Stanford University, May 1983.

[Hayes-Roth 85] *A Blackboard Architecture for Control.*
B. Hayes-Roth. Journal of Artificial Intelligence 26, p 251-321, 1985.

[Hewitt 72] *Description and Theoretical Analysis (Using Schemata) of PLANNER : A Language for Proving Theorems and Manipulating Models in a Robot.*
C.E. Hewitt. Report AI-TR-258 (Ph.D. dissertation), Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1972.

[Hewitt & al 73] *A Universal Modular ACTOR Formalism for Artificial Intelligence.*
C.E. Hewitt, P. Bishop, R. Steiger. Proceedings of the 3rd IJCAI, p235-245, Stanford, California, 1973.

[Israel 83] *The Role of Logic in Knowledge Representation.*
D.J. Israel. Computer, 16 (10), p 37-41, 1983.

[Jones & al 86] *A Blackboard Shell in PROLOG.*
J. Jones, M. Millington, P. Ross. Department of Artificial Intelligence Research Report 277, University of Edinburgh, 1986.

[Jouve 89] *A survey of approaches in knowledge based systems for design.*
C. Jouve. Esprit Project 2474 Literature Review. June 89.

[Jouve & al 90] *Knowledge Representation Framework for an Expert System in computer Network Design.*
C. Jouve, F. Balfroid. Proceedings of TOOL'S 90, Paris, June 1990.

[Jouve 90] *Why not use blackboard based system for implementation of NEST ?*
C. Jouve. Esprit Project 2474 internal report. February 1990.

[Jouve 92] *Un système à base de connaissances pour la conception de réseaux : NEST.*
C. Jouve, F. Balfroid. Rapport de recherche n° 92-1, Ecole des Mines de Saint-Etienne, Mars 92.

[Jouve & al 92] *Un système à base de connaissances pour la conception de réseaux : NEST.*
C. Jouve, F. Balfroid. Douzièmes journées internationales, Intelligence Artificielle, systèmes experts, langage naturel d'Avignon. Juin 92.

[Kass 87] *Implicit Acquisition of User Models in Cooperative Advisory Systems.*
Technical Report MS-CIS-87-05, Department of Computer and Information Science, University of Pennsylvania, 1987.

[Kayser 84] *Examen des diverses méthodes utilisées en représentation des connaissances.*
D. Kayser. Actes du quatrième CARFIA, p 115-144, Paris, 1984.

[Krickhahn & al 88] *Applying the KADS Methodology to Develop a Knowledge Based System - NetHandler.*
R. Krickhahn, R. Nobis, A. Mählmann, M.-J. Schachter-Radig. Proceedings of ECAI, 1988.

[Laasri & al 88] *ATOME : A Blackboard architecture with temporal and hypothetical reasoning.*
H. Lâasri, B. Maitre, T. Mondot, F. Charpillet, J.P. Haton. Rapport de recherche No 855 INRIA, June 1988.

[Laasri & al 89] *Coopération dans un univers multi-agents basée sur le modèle du Blackboard : étude et réalisation.*
H. Lâasri, B. Maitre. Thèse, CRIN, Nancy, Février 1989.

[Laurière 82] *Représentation et utilisation des connaissances. Première partie : Les systèmes experts. Deuxième partie : Représentation des connaissances.*
J.L. Laurière. Technique et Science Informatique, Vol. 1, No. 1 et 2, 1982.

[Laurière 84] *Un moteur d'inférences pour systèmes experts en logique du premier ordre : SNARK.*
J.L. Laurière. Bulletin de l'INRIA, No. 97, p 24-28, 1984.

[Lenat & al 83] *Building Expert Systems.*
D.B. Lenat, F. Hayes-Roth, D.A. Waterman. Addison-Wesley, 1983.

[Lesser & al 87] *An Update on the Distributed Vehicle Monitoring Testbed.*
V.R. Lesser, D.D. Corkill, E.H. Durfee. Technical Report 87-111, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, October 1987.

[Lieberman 87] *Concurrent Object-Oriented Programming in Act 1.*
H. Lieberman. In A. Yonezawa and M. Tokoro, editors, Object-Oriented Concurrent Programming, p 9-36, MIT Press, Cambridge, Massachusetts, 1987.

[Lütticke & al 89] *An Interactive graphical configurator for networked systems.*
B. Lütticke, D. McArthur, A. Neuhaus, S. Sachs, A. Swanson. Ninth International Workshop, Specialized Conference Artificial Intelligence, Telecommunication & computer systems, Avignon'89, May 29 - june 2,1989.

[Maher & al 85] *HI-RISE : An Expert System for the Preliminary Structural Design of High Rise Buildings.*
M.L. Maher, S.J. Fenves, in J. Gero (ed.), Knowledge Engineering in Computer-Aided Design, North Holland, Amsterdam, 1985.

[Masini & al 90] *Les langages à objets.*
G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre. InterEditions, Février 1990.

[McDermott 82] *R1 : A Rule-Based Configurer of Computer Systems.*
J. McDermott. Artificial Intelligence, Vol. 19, No. 1, p 39-88, September 1982.

[Metzler & al 88] *ISLAND : An Intelligent System for Local Area Network Design.*
D.P. Metzler, J. Williams. Methodologies for intelligent systems, Amsterdam, North Holland, Z.B. Ras & L. Saitta eds, p 122-131, 1988.

[Meyer 87] *Eiffel : A Language and Environment for Software Engineering.*
B. Meyer. rapport. 1987.

[Meyer 88] *Object-Oriented Software Construction.*
B. Meyer. Prentice Hall, New York, 1988.

[Middelton & al 85] *BLOBS : an object-oriented language for simulation and reasoning.*
S. Middelton, R. Zanconato. Proceedings of AI in Simulation, Ghent, 1985.

[Minsky 75] *A Framework for Representing Knowledge.*
M. Minsky. The psychology of Computer Vision, P. Winston editor, p 211-281, McGraw-Hill, New-York, 1975.

[Mitchell & al 85] *A Knowledge-Based Approach to Design.*
T.M. Mitchell, L.I. Steinberg, J.S. Shulman. IEEE Transactions on pattern analysis and machine intelligence, vol. PAMI-7, No. 5, September 1985.

[Mittal & al 86] *PRIDE : An Expert System for the Design of Paper Handing Systems.*
S. Mittal, C.L. Dyn, M. Majoria. Computer IEEE, 1986.

[MMI2 partners 88] *A Multi-Mode Interface for Man-Machine Interaction with knowledge based systems.*
Technical Annex, MMI2-ESPRIT2 P2474, November 1988.

[MMI2 partners 89-1] *Literature Review and General Architecture.*
Deliverable d1, July 1989.

[MMI2 partners 89-2] *Common Meaning Representation.*
Deliverable d2, BIM/13, November 1989.

[MMI2 partners 91] *First prototype of integrated dialogue management.*
Deliverable d5, May 1991.

[Mostow 85] *Towards Better Models of the Design Process.*
J. Mostow. AI Magazine 6, 1, p 44-57, 1985.

[Newell 69] *Heuristic programming : Ill-structured problems.*
A. Newell. Progress in operations research, New York : John Willey, III, p 360-414, 1969.

[Nii & al 82] *Signal-to-symbol transformation : HASP/SIAP case study.*
H.P. Nii, E.A. Feigenbaum, J.J. Anton, A.J. Rockmore. AI Magazine 3, p 23-35, 1982.

[Nii 86] *Blackboard Systems, Part one : The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures.*
H.P. Nii. The AI Magazine, July 1986.

[Nii 86] *Blackboard Systems, Part two : Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective.*
H.P. Nii. The AI Magazine, August 1986.

[Norton & al 88] *Learning Preference Rules for a VLSI Design Problem-Solver.*
S.W. Norton, K.M. Kelly. Proceedings of fourth conference on AI application, IEEE Computer society, Order No. 837, March 1988.

[Nurcan 91] *Représentation de connaissances et gestion de données évolutives dans le contexte de la programmation logique orientée objet.*
S. Nurcan. Thèse présentée à l'INSA de Lyon, No. 91-ISAL-0005, Janvier 1991.

[Ostler 89] *LOQUI : How Flexible can a formal prototype be ?*
N.D.M. Ostler. In M.M. Taylor, F. Neel & D.G. Bouwhuis (Eds), The Structure of Multimodal Dialogue, North-holland, 1989.

[Payne 90] *Developing Expert Systems : A Knowledge Engineer's Handbook for Rules and Objects.*
E.C. Payne, C. McArthur Robert. John Wiley, New York, 1990.

[Pérez & al 91] *A Multi-Mode Interface for Man Machine Interaction with knowledge based systems. Spanish Mode.*
Intelligent Software Solutions, S.A. Deliverable d24, Esprit Project 2474. December 91.

[Pinson 81] *Représentation des connaissances dans les systèmes experts.*
S. Pinson. RAIRO Informatique / Computer Science, Vol. 15, No. 4, p 343-367, 1981.

[Quillian 68] *Semantic memory.*
M.R. Quillian. In Minsky (editor), Semantic Information Processing, p 227-270, 1968.

[Robin 88] *Réseaux locaux. Normes et protocoles.*
P. Robin. Traité des nouvelles Technologies, Ed. Hermès, 1988.

[Rosch 75] *Journal of Experimental Psychology.*
E. Rosch. Cognitive Representations of Semantics Categories, 104(3), p 192-233, 1975.

[Rychener 88] *Expert Systems for Engineering Design.*
M.D. Rychener. Engineering Design Research Center, Carnegie-Mellon University, Pittsbourg, Academic Press Inc., 1988.

[Schank 72] *Conceptual Dependency: A Theory of Natural Language Understanding.*
R.C. Schank. Cognitive Psychology 3, p 552-631, 1972

[Shapiro 86] *The art of Prolog : Advanced Programming Techniques.*
E. Shapiro, L. Sterling. The MIT Press, 1986.

[Shortliffe 76] *Computer-Based Medical Consultation : MYCIN.*
E.H. Shortliffe. American Elsevier, New York, 1976.

[Simmons & al 84] *An Architecture for the Application of Artificial Intelligence to Design.*
M.K, Simmons, J.K. Dixon. Proc.ACM/IEEE 21st Ann. Design Automation Conf., Albuquerque, N.M., p 634-640, 1894.

[Simon 69] *The sciences of the artificial.*
H.A. Simon. Cambridge, Mass. : MIT Press. 1969.

[Simon 73] *The Structure of Ill-Structured Problems.*
H. Simon, Artificial Intelligence, vol. 4, p 181-201, 1973.

[Stefik 81] *Planning with Constraints (MOLGEN : Part 1 & Part 2).*
M. Stefick. Artificial Intelligence n. 16, p 111-170, 1981.

[Stefik & al 85] *Object-Oriented Programming : Themes and Variations.*
M. Stefik, D. G. Bobrow. The AI Magazine, Vol. 6, winter 1985.

[Stein 87] *Delegation IS Inheritance.*
L.A. Stein. In Proceedings of the 2nd OOPSLA, p 138-146, Orlando, Florida, 1987.

[Stroustrup 86] *The C++ Programming Language.*
B. Stroustrup. Addison-Wesley Series in Computer Science, Reading, Massachusetts. 1986.

[Tanenbaum 90] *Réseaux. Architectures, protocoles, applications.*
A. Tanenbaum. iia, InterEditions, 1990.

[Technical Annex 92] *Technical Annex 1992-1993.*
A Multi-Mode Interface for Man-Machine Interaction with knowledge based systems. ESPRIT Project 2474 MMI^2. October 1991.

[Terry 83] *The CRYSALIS Project : Hierarchical Control of Production Systems.*
A. Terry. Technical Report HPP-83-19, Standford University, 1983.

[Tesler 85] *Object Pascal Report.*
L. Tesler, Apple Computer Inc. 1985.

[Thayse & al 90] *Approche logique de l'intelligence artificielle.*
Vol. 1. De la logique classique à la programmation logique.
A. Thayse, A. Bruffaerts, P Dupont, E. Henin, Y. Kamp, J.P. Müller, J.L. Binot, D. Snyers, P. Delsarte. Ed. DUNOD informatique. 1990.

[Turing 50] *Computing Machinery and Intelligence.*
A. Turing. Mind, Vol. 59, No. 236, p 433-460, 1950.

[Vignard 86] *Représentation des connaissances, mécanismes d'exploitation et d'apprentissage.*
P. Vignard. Collection didactique, INRIA éditeur, 1986.

[Weiss & al 78] *A model-based method for computer-aided medical decision-making.*
S.M. Weiss, C.A. Kulikovski, S. Amarel, A. Safir. Artificial intelligence, Vol. 11, No. 1-2, p 145-172, August 1978.

[Williams & al 84] *TRICERO Design Description.*
M. Williams, H. Brown, T. Barnes. Technical Report ESL-NS539, ESL Inc, May 1984.

[Wilson 91] *The First MMI^2 Demonstrator : a Multi-modal Interface for Man Machine Interaction with Knowledge Based Systems.*
M.D. Wilson. Deliverable d7, RAL-91-093, United Kingdom, 1991.

[Zanconato 88] *BLOBS - An Object-Oriented Blackboard System Framework for Reasoning in Time.*
R. Zanconato. Blackboard Systems edited by R. Engelmore & T. Morgan, Addison Wesley Publishing Company, 1988.

Annexe 1 : Modélisation des éléments de base de la conception de réseaux.

Cette annexe est la base de connaissances de NEST. Avant d'introduire les différentes classes et leurs propriétés, quelques remarques seront émises pour expliquer les caractéristiques principales de BIM_Probe et ainsi permettre au lecteur une compréhension plus aisée de cette base de connaissances. Le lecteur pourra aussi consulter l'annexe 4 qui est une présentation de ce langage. La description des objets de conception fait également partie de la base de connaissances de NEST.

Pour des raisons de confidentialité, cette annexe ne peut être diffusée.

Annexe 2 : Formalisation des tâches.

Cette annexe a pour but de décrire certaines tâches développées pour permettre à NEST de concevoir des réseaux. Un formalisme de présentation de ces tâches a été défini pour en faciliter la lecture et ainsi ne pas avoir à consulter des programmes prolog pour comprendre les fondements de l'outil de conception. Dans cette annexe sera donc d'abord présenté le formalisme général puis pour chaque étape du plan principal (voir paragraphe 3.2.3.1 page 143) de l'outil de conception, les différentes tâches le constituant seront introduites en utilisant ce formalisme.

Cette annexe est extraite de [Balfroid & al 91-1]. De ce fait, la langue anglaise a été conservée pour la présentation de chaque tâche.

1. Formalisme.

ID : name of the task
GOAL : name of the function corresponding to the task
DESCRIPTION : text describing the task
USED-INFORMATION : (split in four lists)

- *required user data :* data that must be provided by the user before doing the task
- *optional user data :* data that could be asked to the user, depending on the particular case
- *required computable data :* data the value of which depends on triggering a correlated module and that must be computed for fulfilling the task
- *optional computable data :* data the value of which depends on triggering a correlated module and that could have to be computed for fulfilling the task

CONSTRAINTS : (split into two lists)

- *validity constraints :* relationships that the variables of the module must absolutely satisfy and that condition the validity of the solutions proposed by the task
- *preference constraints :* relationships that the variables of the module must satisfy according to the designers' choices

PREFERENCE-EVALUATION-FUNCTION : function dealing with preference constraints to range possible answers
POSSIBLE-ANSWERS : explicit list of possible answers or function to compute them
POST-ACTIONS : list of things that must be done when a task is fulfilled (for example, data storage so that stored data can be accessed later on by some other modules)

2. Sélection d'un squelette de réseau.

Sélecteur de type de squelette.

ID : **Selector_of_skeleton_type**
GOAL : select_type
DESCRIPTION : Four types of skeleton can be distinguished according to the building architecture and department distribution. This task creates Skeleton instances and computes their slots : type and building.
USED-INFORMATION :

- *required user data :*
  - Buildings instances,
  - values of set_in_building slot of Floors instances (for building#floors),
  - values of set_in_floor slot of Rooms instances (for floor#list_of_rooms),
- *optional user data :*
  - values of set_in_departments of Rooms instances. By default all rooms belong to the same department,
- *required computable data :* No
- *optional computable data :* No

CONSTRAINTS :

- *validity constraints :*
  - according to building architecture and departmentalization,

- *preference constraints :* No

PREFERENCE-EVALUATION-FUNCTION : No

POSSIBLE-ANSWERS :

• for the values of type slot of Skeletons instances : @only_one_floor, @one_department_by_floor or @mixed_depatment,

POST-ACTIONS : storage of :

• values of building slot of Skeletons instances,

• values of type of Skeletons instances.

Calculateur du nombre de backbones verticaux.

***ID :* Selector_of_vertical_backbone_number**

GOAL : nb_skeleton

DESCRIPTION : This task makes the decomposition of the network into subnetworks according to departmentalization principle, fixed thresholds on number of machines, presence of servers.

USED-INFORMATION :

- *required user data :*

• values of set_in_rooms slot of Machines instances,

• values of is_a_server slot of Machines instances,

• Vertical_shafts instances,

• values of walls_sucession slot of Rooms instances,

• values of set_in_floor slot of Rooms instances,

- *optional user data :*

• values of set_in_departments slot of Rooms instances (by default all rooms belong to the same department),

- *required computable data :* No

- *optional computable data :* No

CONSTRAINTS :

- *validity constraints :*

• on servers,

• departmentalization principle,

• on thresholds,

- *preference constraints :* No

PREFERENCE-EVALUATION-FUNCTION :

POSSIBLE-ANSWERS : Storage of

• values of depart slot of Vertical_backbones instances,

• values of possible_verticals slot of Vertical_backbones instances,

• values of group slot of Group_depart instances,

• values of set_in_depart slot of Sub_depart instances,

• values of rooms slot of Sub_depart instances,

• values of list_of_vertical_backbones of Skeletons instances.

Sélecteur des étages à câbler.

ID : **Selector_of_floors_to_be_cabled**
GOAL : floor_to_be_cabled
DESCRIPTION : it finds all floors which have to be cabled i.e. floors with machines or required floor extensions .
USED-INFORMATION :
- *required user data :*
  - values of set_in_building slot of Floors instances,
  - values of set_in_rooms slot of Machines instances,
  - values of set_in_floor slot of Rooms instances,
  - values of set_in_departments slot of Rooms instances (for department#elements),
- *optional user data :*
  - values of extension slot of Buildings instances,
- *required computable data :*
  - values of depart slot of Skeletons instances (from task2),
- *optional computable data :* No

CONSTRAINTS :
- *validity constraints :* No
- *preference constraints :* No

PREFERENCE-EVALUATION-FUNCTION : No
POSSIBLE-ANSWERS : No
POST-ACTIONS : storage of values of floors slot of Vertical_backbones instance.

Sélecteur de la localisation des backbones verticaux.

ID : **Selector_of_vertical_backbone_locations**
GOAL : place_skeleton
DESCRIPTION : it finds for each skeleton a vertical place (shaft, stair, ...).
- *required user data :*
  - values of centre slot of Buildings instances,
  - values of set_in_building slot of Floors instances,
  - values of centre slot of Computer_rooms instances,
  - values of jointed_floors slot of Vertical_shafts instances,
  - instances stairs, Lifts, Vertical_shafts,
- *optional user data :*
  - values of building_type slot of Buildings instances (by default : @modern),
- *required computable data :*
  - values of possible_verticals slot of Vertical_backbones instances (from task2),
  - values of floors slot of Vertical_backbones instances (from task 3),
- *optional computable data :* No

CONSTRAINTS :
- *validity constraints :* No
- *preference constraints :*
  - choice between the shaft the most central and the more near the computer_room

PREFERENCE-EVALUATION-FUNCTION : choose_between_central_and_near_info
POSSIBLE-ANSWERS :
POST-ACTIONS : storage of :

- values of vertical slot of Vertical_backbones instances,
- values of sub_vertical of Vertical_backbones instances.

3. Conception des sous-réseaux pour chaque étage.

Calculateur de la distance à câbler.

***ID :* Calculator_of_the_approximative_distance_to_be_cabled**
GOAL : compute_dist_to_be_cabled
DESCRIPTION : It computes the distance which has to be cabled on a floor i.e. the length of linear cable connecting all the connection points and going along corridors.
USED-INFORMATION :

- *required user data :*
  - values of set_in_floor slot of Rooms instances,
  - values of set_in_rooms slot of Machines instances,
  - values of set_in_rooms slot of Vertical_shafts instances,
  - values of walls_succession slot of Rooms instances,
  - values of angle slot of Walls instances,
  - values of length slot of Walls instances,
- *optional user data :* No
- *required computable data :*
  - values of vertical slot of Vertical_backbones (from task 4),
- *optional computable data :* No

CONSTRAINTS :

- *validity constraints :* No
- *preference constraints :* No

PREFERENCE-EVALUATION-FUNCTION :
POSSIBLE-ANSWERS : the length of corridor way which allows to connect all rooms to be cabled.
POST-ACTIONS : storage of distance_to_be_cabled in Floor_networks.

Sélecteur du type de câble.

***ID :* floor_cable_type_selector**
GOAL : select_cable_type_for_floor
DESCRIPTION : selects a particular cable type that should be used for the floor corresponding to the given floor_network
USED-INFORMATION :

- *required user data :* floor environmental conditions, building type, shafts description
- *optional user data :* budget, user rejections or requirements about cable type
- *required computable data :* distance to be cabled
- *optional computable data :* cost

CONSTRAINTS :

- *validity constraints :* compatible environment, no technological exclusion, user cable type requirements
- *preference constraints :* cost minimization, distance to be cabled consideration, expected cable location or support

PREFERENCE-EVALUATION-FUNCTION : range_cables_types_for_floor

POSSIBLE-ANSWERS : possible_cables_types_for_floor returning : [OF_cables, Twisted_cables, Thick_cables, Thin_cables]

POST-ACTIONS :

- to store the selected cable type in the cable_type slot of Floor_network design object
- to store the filtered and ranged list of possible answers in the possible_cables_types slot of the Floor_network design object

Sélecteur de la structure du sous-réseau.

***ID :* floor_cable_structure_selector**

GOAL : select_cable_structure_for_floor

DESCRIPTION : selects a particular cable structure that should be used for the floor corresponding to the given floor_network

USED-INFORMATION :

- *required user data :* number of machines to be connected on the considered floor
- *optional user data :* user filtering requirements, departments specifications, budget
- *required computable data :* cable type, distance to be cabled
- *optional computable data :* cost

CONSTRAINTS :

- *validity constraints :* limitations due to cable type and distance to be cabled, filtering
- *preference constraints :* extensibility (modularity or not), cost, resources administration (hierarchical or not)

PREFERENCE-EVALUATION-FUNCTION : range_cable_structures_for_floor

POSSIBLE-ANSWERS : [Linear, Rep, Star]

POST-ACTIONS :

- to store the selected cable structure in the structure slot of the Floor_network design object
- to store the filtered and ranged list of possible answers in the possible_structures slot of the Floor_network design object

Sélecteur de boîte de connexion.

***ID :* floor_box_type_selector**

GOAL : select_box_type_for_floor

DESCRIPTION : .selects a particular box type that should be used for the given floor_network

USED-INFORMATION :

- *required user data :* filtering requirement
- *optional user data :* budget
- *required computable data :* cable type, cable structure
- *optional computable data :* cost

CONSTRAINTS :

- *validity constraints :* cable structure compatibility, cable type compatibility, filtering requirement, cable_connecting property
- *preference constraints :* cost minimization, network management properties

PREFERENCE-EVALUATION-FUNCTION : range_boxes_types_for_floor

POSSIBLE-ANSWERS : possible_boxes which returns the different boxes types described in the network components hierarchy

POST-ACTIONS :

- to store the selected box type in the box_type slot of the Floor_network design object
- to store the filtered and ranged list of possible answers in the possible_boxes_types slot of Floor_network design object

Localiseur de boîte de connexion.

***ID :* Selector_of_boxe_location**

GOAL : box_location

DESCRIPTION : This module creates an object of type : box_type defined in every Floor_networks and it looks for a location (a room place) for these boxes. The location can be a computer_room if there is one at the considered floor or a room which contains a server or it is a room next to the arrival point of the vertical backbone.

USED-INFORMATION :

- *required user data :*
  - Computer_rooms instances,
  - Room architecture,
- *optional user data :* No
- *required computable data :*
  - box_type (from Floor_networks),
  - Vertical_backbones instances,
- *optional computable data :* No

CONSTRAINTS :

- *validity constraints :* No
- *preference constraints :*
  - preference of a computer_room if there is one at the considered floor else a room which contains a server else the room next to the arrival point of the vertical_backbone,

PREFERENCE-EVALUATION-FUNCTION : No

POSSIBLE-ANSWERS : a computer_room else a room,

POST-ACTIONS : creation of boxes and storage of its locations.

Créateur et localisateur de câbles.

***ID :* Selector_of_cable_location**

GOAL : cables_location

DESCRIPTION : This module analyses the building plan in order to choose where the cables can be located (shafts, false ceilings, ...).According to previously computed values on cable type, cable structure (linear, rep or star) and box type, it gives the number of needed branches, their length and finds their location. For each branch, its connected_elements property is filled in by one of the following forms which specify which elements have to be connected to this cable : (1) a terminating element, (2) a machine or a box or a plug via perhaps a connecting_element.

[@terminating_element, [@connecting_element, _list_of_machines_or_boxes_or_plugs, _cable]]

[[@connecting_element, _list_of_machines_or_boxes_or_plugs, _cable1], [@connecting_element, _list_of_machines_or_boxes_or_plugs, _cable3]]

[[@connecting_element, _list_of_machines_or_boxes_or_plugs, _cable], @terminating_element]

USED-INFORMATION :

- *required user data :*
  - Horizontal_shafts instances,
  - False_ceilings instances,
  - Corridors instances,
  - False_grounds instances,
  - Room architecture,
- *optional user data :*
- *required computable data :*
  - cable type, structure, box_type, (from Floor_networks object),
  - Vertical_backbones instances,
- *optional computable data :*

CONSTRAINTS :

- *validity constraints :* No
- *preference constraints :*
  - For the cable location, we choose first to locate it in shafts (false_groung, false_ceiling, horizontal shafts) and secondly if there is no suitable shaft we use the corridors.

PREFERENCE-EVALUATION-FUNCTION : No

POSSIBLE-ANSWERS :

POST-ACTIONS : the list of created cables,

- creation of cables and storage of their length, partof, connected_elements property and their location (pass_in or pass_trough or pass_along property),
- storage of branches_number in Floor_networks instance.

Sélecteur de connecteurs.

***ID :* connector_type_selector**

GOAL : select_connector_type_for_box

DESCRIPTION : selects a particular box connector that should be used to connect the given box to the given cable type in the given room

USED-INFORMATION :

- *required user data :* box description, room description
- *optional user data :* budget, user rejections or requirements (requirements about the easy connection of machines), number of machines
- *required computable data :* cable type, expected connector location
- *optional computable data :* cost

CONSTRAINTS :

- *validity constraints :* box compatibility, cable compatibility, location compatibility
- *preference constraints :* cost minimization

PREFERENCE-EVALUATION-FUNCTION : range_possible_connectors_for_box

POSSIBLE-ANSWERS : list of sub-classes leaves of Box_connectors

POST-ACTIONS :

- to create instances of the knowledge base corresponding to the network objects needed to connect the box to the cable type

4. Fusion des différents sous-réseaux.

Calculateur de la distance verticale.

***ID :* Calculator_of_distance_in_vertical_way**

GOAL : compute_vertical_distance

DESCRIPTION : This predicat computes the length to be cabled for a given vertical backbone.

USED-INFORMATION :

- *required user data :*
  - for Vertical_shafts, the joined_floors property,
  - for each Floors their height,
- *optional user data :*
- *required computable data :*
  - a Vertical_backbones,
  - and its floor_deparment and vertical properties,
- *optional computable data :*

CONSTRAINTS :

- *validity constraints :* No
- *preference constraints :* No

PREFERENCE-EVALUATION-FUNCTION : No

POSSIBLE-ANSWERS : The value of the vertical distance for the considered vertical backbone

POST-ACTIONS : No

Création de câbles verticaux.

***ID :* localization_of_vertical_cables**

GOAL : vertical_cables_location

USED-INFORMATION : This task creates vertical cables and puts their length, partof, pass_in, connected_elements properties. The form of connected_elements property is one of :

[@terminating_element, [@connecting_element, [], _cable]]

[[@connecting_element, [], _cable1], [@connecting_element, [], _cable3]]

[[@connecting_element, [], _cable], @terminating_element]

USED-INFORMATION :

- *required user data :*
  - for Vertical_shafts, the joined_floors property,
  - for each Floors their height,
- *optional user data :*
- *required computable data :*
  - selected cable type,

• a Vertical_backbones,
• and its floor_deparment and vertical properties,

- *optional computable data :*

CONSTRAINTS :

- *validity constraints :* No
- *preference constraints :* No

PREFERENCE-EVALUATION-FUNCTION : No

POSSIBLE-ANSWERS : the list of created cables,

POST-ACTIONS : creation of cables and storage of their length, pass_in, partof and connected_elements property.

Lien entre les différents backbones verticaux.

ID : **merging_of_vertical_backbones**

GOAL : merge_vertical_backbones

DESCRIPTION : This module computes the link between the different vertical backbones.

USED-INFORMATION :

- *required user data :*
  • for Vertical_shafts, the joined_floors property,
- *optional user data :*
  • the Vertical_backbones,
  • its cables,
  • its locations,
- *optional computable data :*

CONSTRAINTS :

- *validity constraints :*
- *preference constraints :*

PREFERENCE-EVALUATION-FUNCTION :

POSSIBLE-ANSWERS : the link between vertical backbones

POST-ACTIONS : creation of only one cable instead of two if two vertical backbones into the same shaft or creation of a new box and of the links of the vertical cables to this box.

Annexe 3 : Prérequis nécessaires avant une requête de conception.

Avant de pouvoir activer l'opération de conception de réseaux de NEST, plusieurs informations doivent être connues telles qu'une description détaillée du site à câbler, des machines à connecter au réseau, du nombre de connexions souhaité dans chaque pièce (si différent de zéro), des exigences du client (par exemple, son budget), etc. Ces données sont soit obligatoirement requises pour que NEST puisse travailler ou elles sont optionnelles. Si ces dernières sont données, l'outil de conception les prendra en compte. Ainsi deux tableaux constituent cette annexe : un pour les prérequis et l'autre pour les informations optionnelles. Dans ces tableaux, la deuxième colonne donne le mode (graphique, langage naturel, par exemple) qui nous semble le plus approprié pour entrer les informations décrites dans la première colonne. La troisième montre la représentation interne de ces données dans l'application. En gras est donnée la classe et entre parenthèses la ou les propriétés concernées.

Données prérequises	Format d' entrée	Représentation interne
Description du site	Graphique	**Buildings** (centre)
• description des étages	Graphique (préciser la hauteur - LN)	**Floors** (height, set_in_building)
-> echelle	Graphique (approximativement 10%)	
-> pièces	Graphique (localisation et taille)	**Rooms** (set_in_floor, origin_coordinates, walls_succession)
-> couloirs	Graphique (localisation et taille)	**Corridors** (set_in_floor, origin_coordinates, walls_succession)
-> salles informatiques	Graphique (localisation et taille)	**Computer_rooms** (set_in_floor, origin_coordinates, walls_succession, centre)
[murs	Graphique	**Walls** (angle, length)
• chemin de câbles	Graphique (localisation)	
-> service de câbles	Graphique (la super-classe)	**Vertical_shafts** (set_in_rooms, joined_floors, coord_x_y)
[gaines techniques	Graphique (sous-classe)	**Technical_shafts** (set_in_rooms, joined_floors, coord_x_y)
[escaliers	Graphique (sous-classe)	**Stairs** (set_in_rooms, joined_floors, coord_x_y)
[ascenseurs	Graphique (sous-classe)	**Lifts** (set_in_rooms, joined_floors, coord_x_y)
-> gaines	Graphique	**Horizontal_shafts** (set_in_rooms, length, set_on_walls)
-> faux plafonds	Graphique	**False_ceilings** (set_in_rooms)
-> faux planchers	Graphique	**False_grounds** (set_in_rooms)

Données prérequises	Format d' entrée	Représentation interne
Description du réseau souhaité	Langage Naturel	**Networks**
• coût d'installation des câbles	Langage Naturel	**Networks** (installation_cost_option) et **Buildings** (average_cable_installation_-cost) si installation_cost_option est @building ou **Departments** (average_cable_installation_cost) si installation_cost_option est @department ou **Networks** (average_cable_installation_-cost) si installation_cost_option est @network
• machines	Graphique ou Langage Naturel (spécifier les machines qui sont un serveur)	créer une machine selon son type (set_in_rooms, part_of, is_a_server)
• disques	Graphique ou Langage Naturel	**Disks** (installed_on)
• nombre de connexions souhaitées	Graphique	pour chaque **Rooms** (wished_connexions)

Données optionnelles	Format d'entrée	Représentation interne
Type de bâtiment	Langage Naturel	**Buildings** (building_type)
Spécification des mauvaises conditions d'environement dans les differentes pièces	Langage Naturel	pour chaque **Rooms**, si nécessaire, indiquer la liste des mauvaises conditions dans la propriété environmental_conditions qui peut être @electrical, @X_ray, @thermical ou @magnetical
Rejet absolu d'un type spécifique de câble	Langage Naturel	assert_fact_in_kb (**user_cables_rejection** (_network, _list_of_rejected_-cable_types)). où _list_of_rejected_cable_types est une sous-liste de [Thin_cables, Thick_cables, Twisted_cables, OF_cables] et _network est une instance de Networks
Exigence absolue d'un type spécifique de câble	Langage Naturel	assert_fact_in_kb (**user_cables_required** (_network, _list_of_required_-cable_types)). où _list_of_rejected_cable_types est une sous-liste de [Thin_cables, Thick_cables, Twisted_cables, OF_cables] et _network est une instance de Networks

Données optionnelles	Format d'entrée	Représentation interne
Filtrage requis dans un département spécifique	Langage Naturel	assert_fact_in_kb (**user_filtering_requirement** (_network, _department) où _network est une instance de Networks et _department une instance de Departments
Budget	Langage Naturel	assert_fact_in_kb (**budget** (_network, _budget)) où _network est une instance de Networks et _buget est le budget alloué pour ce réseau
Description des unités professionnelles	Graphique et/ou Langage Naturel (• L'utilisateur doit pouvoir nommer tous les départements, • puis il a à sélectionner un département et les pièces associées ou étages ou bâtiments).	**Departments** Pour chaque **Rooms** (set_in_departments)

Annexe 4 : Présentation du

langage BIM_Probe.

BIM_Probe est un outil développé par la société BIM (Belgique) en s'inspirant des travaux faits pour le langage TAXI et des projets Esprit DAIDA [Baudinet & al 90] et LOKI. Il est construit au dessus du langage Prolog_by_BIM et est couplé à un langage de programmation par contraintes : PCL. Ains combine-t-il à la fois la programmation orientée objet et la programmation par contraintes, tout en alliant les possibilités d'un langage prolog.

BIM_Probe en tant que langage de programmation orientée objet implémente les principes de base de ce mode de programmation. Ainsi, on retrouvera au sein de ce langage les notions de métaclasse, de classe, d'instance. Ces objets sont définis par des attributs, eux même décrits par des facettes. L'organisation en hiérarchie *isa*, l'héritage multiple, la spécialisation au niveau des attributs et l'attachement procédural sont des caractéristiques de BIM_Probe. A celles-ci s'ajoutent des mécanismes de vérification de la consistance lors de la création ou de la modification d'une base de connaissances.

Après avoir présenté le noyau implémentant les concepts orientés objet puis le langage de programmation par contraintes : PCL, un dernier paragraphe évoquera en détails les méthodes de vérification de la consistance de bases de connaissances développées au sein de BIM_Probe.

1. Le noyau orienté objet.

Dans le langage BIM_Probe, un objet peut être soit une métaclasse, soit une classe, soit une instance.

Comme dans tout langage orienté objet, un objet BIM_Probe combine à la fois les données par l'intermédiaire d'attributs appelés propriétés et les procédures relatives à l'objet dans des attributs : méthodes. Les propriétés d'un objet doivent vérifier les contraintes, si elles existent le concernant, définies à l'aide de l'attribut contrainte. Ainsi un objet est défini par la donnée d'une collection d'attributs.

1.1 Les différents types d'attribut d'un objet BIM_Probe.

Les attributs caractérisant un objet peuvent être de trois types :

- une *propriété* : Elles définissent l'état de l'objet et prennent des valeurs spécifiques.
- une *méthode* : est une procédure spécifiant un comportement de l'objet.
- une *contrainte* : définit un critère que l'objet doit vérifier. Une contrainte peut être définie entre des objets (inter-objets) ou peut porter sur des propriétés de l'objet (intra-objet).

Un attribut est défini par son nom et par une série de facettes. Pour chaque type d' attribut, des facettes prédéfinies existent (cf figure 50). Certaines doivent être obligatoirement affectées. C'est par exemple le cas du type d'une propriété.

nom	présence	valeurs permises	valeur par défaut
facettes de propriétés			
type	requise	Date, Time, List, String, Integer, Real, Boolean, toute_classe_définie_par_ l'utilisateur	
card	optionnel	non vérifié	0-U
category	optionnel	changing, unchanging	changing
comment	optionnel	non vérifié	
default	optionnel	non vérifié	
derivation	optionnel	non vérifié	
presence	optionnel	optional, mandatory	optional
pre_get	optionnel	non vérifié	
post_get	optionnel	non vérifié	
pre_mod	optionnel	non vérifié	
post_mod	optionnel	non vérifié	
pre_put	optionnel	non vérifié	
post_put	optionnel	non vérifié	
pre_del	optionnel	non vérifié	
post_del	optionnel	non vérifié	
facettes des contraintes			
definition	requise	non verifié	
category	optionnel	pre_cond, post_cond, invaraint	invariant
comment	optionnel	non vérifié	
facettes des méthodes			
definition	requise	non vérifié	
category	optionnel	PCL, Prolog, C	Prolog
comment	optionnel	non vérifié	

figure 50 : Les différentes facettes prédéfinis dans BIM_Probe.

L'exemple ci-dessous illustre les trois types d'attribut.

```
Class : Networks isa Class
  properties
    machines_and_devices : [type = List (Machines_and_devices), category = changing, card = 0-
U, derivation =
      (all (_mach where _mach is_in Machines_and_devices & _mach#parof = this))]
    cables : [ type = List (Cables), ...,derivation = ...]
    connexion_elements : [type = List (Connexion_elements), .., derivation = ...]
    additional_components : [type = List (Additional_components), .., derivation = ...]
    installation_cost_option : [type = String, category = changing, presence = mandatory]
    average_cable_installation_cost : [type = List, category = changing, card = 0-U, default = nil]
  methods
    cost : [definition = network_cost/2, category = Prolog]
    connexion_problems : [definition = connexion_problems/2, category = Prolog]
.../...
  constraints
    installation_cost_option_constraint : [ definition = (this#installation_cost_option = oneof
[@network,
      @building, @department])]
```

exemple 11 : Illustration des différents types d'attributs d'un objet BIM_Probe.

Cet exemple décrit la classe Networks. Un réseau est défini par la liste des machines qui le composent, la liste de ses câbles, la liste des éléments de connexion entre câbles et machines et la liste des composants additionnels logiciels ou matériels à ajouter à certains composants du réseau afin d'obtenir les caractéristiques de réseau demandées. Ces quatre listes sont des attributs du réseau et on les retrouve dans la classe Networks sous forme de propriétés. Les valeurs de ces propriétés sont obtenues par calcul (dérivation) car tous ces composants ont été définis avec une propriété partof indiquant le réseau auquel ils appartiennent.

Des exemples de facettes de propriétés sont ici : *type* qui permet de définir le type de l'objet, il peut être un type prédéfini, un objet construit par l'utilisateur ou une liste d'objets ; *category* indique si la valeur de la propriété peut être changée au cours de son cycle de vie; dans le cas d'une liste, *card* permet d'indiquer sa dimension : une liste de un ou deux éléments aura pour card 1-2 ; *dérivation* permet la recherche de la valeur de la propriété quand nécessaire par activation de la procédure PCL qu'elle contient ; *default* affecte une valeur par défaut à la propriété ; lorsque la valeur d'une propriété doit être obligatoirement donnée la facette *presence* est affectée à mandatory (optional sinon).

Deux exemples de méthodes sont donnés. Elles sont des appels à des prédicats prolog permettant pour la première de calculer le coût du réseau et pour la seconde d'analyser si des problèmes de connexion existent. Dans le langage BIM_Probe, la définition d'une méthode et son code peuvent être totalement séparés. Le code peut être écrit dans un fichier à part alors que sa définition est donnée dans

l'objet. Dans l'exemple, seule la définition des méthodes est exprimée (et son code est dans un fichier annexe). Les deux langages : prolog et PCL permettent d'écrire des méthodes. Cependant, l'utilisation de PCL apporte plus de déclarativité et d'expressivité que prolog mais demande plus de temps à l'évaluation.

La contrainte définie dans cet exemple est une contrainte intra-objet puisqu'elle s'applique à une propriété de l'objet Networks. *installation_cost_option_constraint* vérifie que la propriété *installation_cost_option* est bien une des chaîne de caractères suivantes : @network, @building, @department.

- L'attribut d'exception.

Cet attribut permet d'introduire une exception à un concept déjà défini dans la base de connaissance. Il est utile lorsque une différence exceptionnelle par rapport aux autres membres d'une même classe ou d'une même métaclasse survient. Dans ce cas, l'attribut d'exception permet d'exprimer cet différence facilement sans avoir à redéfinir une nouvelle classe ou métaclasse exclusivement pour cette objet.

La seconde utilisation de cet attribut est la redéfinition de l'attribut lorsque un objet appartient à deux classes ou métaclasses dans lesquelles deux définition différentes ont été écrites pour un même attribut.

1.2 Constitution d'un objet BIM_Probe.

Conceptuellement, un objet BIM_Probe est composé d'une ou plusieurs parties fictives suivantes, selon que l'on considère une métaclasse, une classe ou une instance :

- définition des attributs.
- valeurs des propriétés,
- définition des attributs d'exception,

Dans une métaclasse, seule la partie : définition des attributs sera présente. Pour les classes, de nouveaux attributs peuvent être définis, des propriétés définies dans une métaclasse à laquelle la classe est liée prennent une valeur au niveau de la classe et des exceptions peuvent être spécifiées. Ainsi dans une classe, les trois parties peuvent se trouver. Au niveau des instances, les valeurs des propriétés définies dans la classe seront instanciées et de nouvelles exceptions peuvent être définies.

1.3 Les différents liens - héritage multiple.

Comme dans tout langage orienté objet, BIM_Probe permet les deux types de lien : IN exprimant la notion de "membre de" et ISA apporte l'héritage et la spécialisation.

La figure 51 montre les différents liens possibles entre les objets de BIM_Probe. Un lien direct est un lien explicitement défini alors qu'un lien indirect est automatiquement créé par le système. Une instance peut avoir des liens IN avec plusieurs classes mais chaque classe ne peut être instance que d'une seule métaclasse. Les classes et métaclasses peuvent être organisées en hiérarchie ISA. Toute instance d'une classe appartient aussi aux super-classes de cette classe ; de même qu'une classe appartenant à une métaclasse appartient aux super-métaclasses. Le lien IN n'est pas transitif : une instance d'une classe n'est pas instance de la métaclasse ancêtre de la classe.

- Spécialisation.

BIM_Probe autorise la spécialisation. Une classe C, sous-classe de C', est spécialisée par rapport à C' si C contient ses propres attributs en addition de ceux de C' ou/et un ou plusieurs attributs de C' est modifié dans C.

- Héritage multiple.

Une classe peut être sous-classes de plusieurs classes (idem pour une métaclasse). BIM_Probe détecte les conflits possibles : attribut défini différement dans deux super-classes (ou super-métaclasses) ; l'utilisateur doit préciser sa préférence pour l'un d'eux ou donner une nouvelle définition de l'attribut concerné.

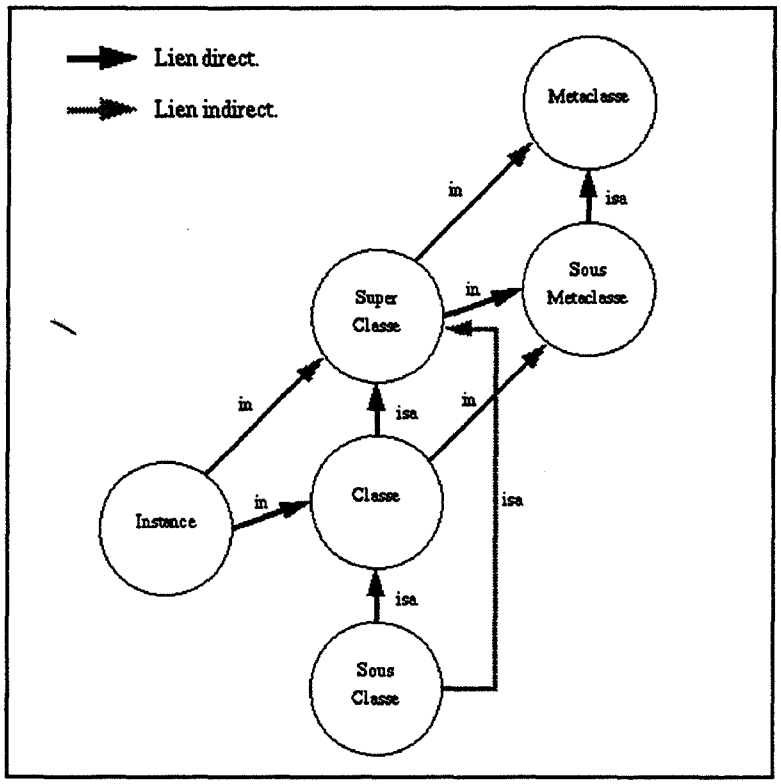


figure 51 : Les liens directs et indirects dans BIM_Probe.

1.4 Les objets prédéfinis.

BIM_Probe a trois principaux objets prédéfinis : Object, la racine, Metaclass et Class, qui sont liés entre eux comme le montre la figure 52. Toute nouvelle classe créée est liée à Class par un lien ISA direct ou indirect ; de même toute nouvelle métaclasse est liée à Metaclass. Class est liée à Metaclass par un lien IN ; ainsi toute classe créée est aussi membre de Metaclass. Plusieurs primitives de Class sont également définies : List, Natural, Integer, Real, String, Boolean, Date et Time.

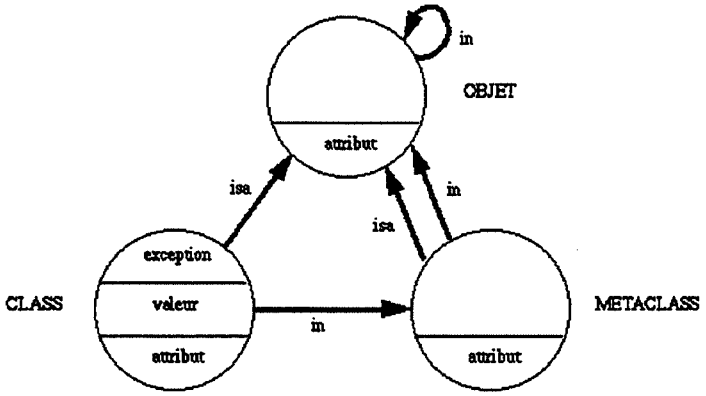


figure 52 : La hiérarchie mère de toute hiérarchie définie avec BIM_Probe.

2. Le langage de programmation par contrainte : PCL.

2.1 Attachement procédural.

L'attachement procédural traduit la capacité du langage à attacher des propriétés dynamiques aux objets. En programmation orientée objet, deux types d'attachement procédural sont généralement reconnus : les démons et les méthodes. Un démon est une procédure automatiquement évaluée et déclenchée seulement sous certaines conditions alors qu'une méthode est une procédure qui doit être explicitement appelée en envoyant un message.

- Démons.

En BIM_Probe, il existe deux types de démons :

- les valeurs actives appelées aussi dérivations : fonction automatiquement déclenchée quand la valeur d'une propriété est nécessaire, ("compute value when needed"). Dans l'exemple 11, la valeur de la propriété *machines_and_devices* d'un réseau particulier est obtenue en effectuant le calcul décrit par la facette derivation.
- les démons classiques : procédure activée quand il y a manipulation (ajout, modification, retrait, accès) d'une valeur d'une propriété. L'effet sur la valeur peut avoir lieu après ou

avant l'opération sur la valeur. Dans BIM_Probe, huit démons de ce type peuvent être utilisés :

- *pre_get, pre_put, pre_mod, pre_del* sont activés respectivement avant l'accès, l'insertion, la modification et la suppression de la valeur d'une propriété,
- *post_get, post_put, post_mod, post_del* sont activés après que l'opération ait réussi.

2.2 PCL.

PCL (Probe Constraint Language) est destiné à exprimer : les valeurs actives, les démons, les contraintes et les méthodes. Il est implementé en BIM_Prolog et est composé d'une série d'opérateurs et d'un méta-interprète.

Pour la réalisation de NEST, ce langage a principalement été utilisé pour l'implémentation des dérivations et des contraintes. Les méthodes ont été écrites au moyen du langage Prolog et pondérément en utilisant la commande *send* permettant d'exécuter une commande PCL au niveau de Prolog. En effet, il nous a souvent été utile de faire appel à PCL pour catégoriser une liste d'objets satisfaisant certaines contraintes. Malheureusement, ce langage reste d'une lenteur gênante pour l'efficacité de NEST.

- Les opérateurs.

PCL offre de nombreux opérateurs. Les plus utilisés dans NEST vont être présentés ci-après.

• *this* et # :

L'opérateur *this* représente toute instance de l'objet où il apparait. L'opérateur # réfère à la valeur de la propriété indiquée en deuxième argument de l'objet premier argument. Toujours dans l'exemple 11, la contrainte *installation_cost_option_constraint* utilise ce type d'opérateurs. *this#installation_-cost_option* réfère à la valeur de la propriété *installation_cost_option* de l'objet courant.

• *one_of* :

Cet opérateur retourne un des éléments de la liste argument. Pour la contrainte *installation_cost_option_contraint*, l'utilisation de *oneof* permet de vérifier que la valeur de la propriété *installation_-cost_option* de l'objet courant est une des valeurs suivantes : @network, @building, @department.

• @ :

L'@ est employée pour différencier les atomes BIM_Prolog qui représentent des noms d'atomes de ceux représentant réellement un atome. Toute instance de la classe Strings doit être préfixée par cet opérateur.

• *all* :

Il permet d'obtenir la liste de tous les objets vérifiant une condition donnée. Par exemple :

?? _L = all (_ser where _ser is_in Machines and _ser#is_a_server = true)

permet d'obtenir dans _L, la liste des serveurs. Pour être un serveur, l'objet doit appartenir à la classe des *Machines* et avoir sa propriété *is_a_server* affectée à vrai.

- *is_in* : vérifie qu'un objet appartient à la classe donnée en deuxième argument
- *.where* : retourne dans le premier argument une instance satisfaisant à la condition donnée en argument deux.
- *and, or,* etc: Des opérateurs sont définis pour permettre d'effectuer des opérations booléennes.
- *is_sublist* : vérifie qu'une liste est sous-liste d'une autre ou donne une liste sous-liste de celle donnée en argument deux.

3. Vérification de la consistance.

Un des principaux intérêts de BIM_Probe est la possibilité de s'assurer qu'une base de connaissances évolue en une séquence d'états consistants. Une base de connaissance est considérée consistante si toutes les contraintes sont satisfaites que ce soit système (nom unique d'objet, nom unique d'attribut dans une même classe que ce soit un attribut hérité ou un attribut défini localement, ...) ou définies par l'utilisateur.

Trois types de consistance sont définis :

- la consistance atomique.

Elle concerne une seule classe ou une seule métaclasse à la fois. Une classe est consistante atomiquement si tous les attributs, toutes les facettes ou toutes les autres informations concernant la classe sont consistantes. Ils ont alors tous un nom différent et les facettes sont syntaxiquement correctes. Cependant le type d'une propriété n'est pas vérifié à ce niveau car un type peut référer à une autre classe qui n'est peut-être pas encore définie. La consistance atomique est automatiquement vérifiée à la création d'une classe et de ses attributs.

- la consistance verticale .

Elle prend en compte la structure hiérarchique des objets et assure que :

- les classes sont organisées sous la forme d'un graphe ayant une racine unique,
- le même attribut hérité par différentes classes n'a qu'une occurrence au niveau de la sous-classe,
- le type d'une propriété défini dans une sous-classe est identique ou est une sous-classe du type défini au niveau de la super-classe,

- une classe doit hériter de tous les attributs de chacune de ses superclasses, même si le lien est explicitement établi ou si une violation des deux précédentes contraintes s'est avérée.

Cette consistance est également vérifiée à la création d'une classe. Les problèmes dus à l'héritage multiple sont aussi résolus à ce niveau.

- la consistance horizontale.

Elle assure que tous les types des attributs référant à une classe existent. Elle étudie également les différentes contraintes définies dans le modèle conceptuel. La définition des attributs est vérifiée en prenant en compte leur dépendance avec les autres objets. Cette consistance est vérifiée sur requête de l'utilisateur.

Au niveau des instances, la consistance atomique est restreinte. Elle consiste principalement à vérifier que les attributs spécifiés comme étant obligatoires (*mandatory*) ont une valeur. Pour une instance appartenant à une seule classe, la consistance verticale n'est pas pertinente.

Annexe 5: Prédicats Prolog assurant les fonctionnalités nécessaires à la liaison de NEST avec l'interface.

Ces prédicats ont été écrits par F. Balfroid et j'ai modifié certains de ces prédicats lors de la phase d'intégration avec l'interface à laquelle j'ai participé.

Détruire un bâtiment.

Permet de supprimer dans la base de connaissances un bâtiment et toutes les informations concernant ce bâtiment (étages, pièces, murs, gaines techniques, etc). Une méthode : "delete" est associée à la classe Buildings. Cette méthode active le prédicat prolog delete_building/1 listé ci-dessous. Pour activer cette méthode et donc détruire une instance de la classe Building et ses composants, l'appel est le suivant :

```
?- send (dt, delete (_building)).
```

```
delete_building(_building):-
   findall(_room,
     (getobject(dt,_room,in,Rooms),
      existspropvalue(dt,_room,set_in_building,_building)),
      _rooms_to_delete),
   findall(_wall,
     (getobject(dt,_wall,in,Walls),
      existspropvalue(dt,_wall,set_in_rooms,_wall_rooms),
      _wall_rooms = [_wall_room|_],
      existspropvalue(dt,_wall_room,set_in_building,_building)),
      _walls_to_delete),
   findall(_internal_shaft,
     (getobject(dt,_internal_shaft,in,Internal_shafts),
      existspropvalue(dt,_internal_shaft,set_in_building,
        _building)),
      _internal_shafts_to_delete),
   findall(_floor,
     (getobject(dt,_floor,in,Floors),
      existspropvalue(dt,_floor,set_in_building,_building)),
      _floors_to_delete),
   delete_instances(dt,_rooms_to_delete),
   delete_instances(dt,_walls_to_delete),
   delete_instances(dt,_internal_shafts_to_delete),
   delete_instances(dt,_floors_to_delete),
   delobject(dt,_building,in,Buildings).
```

prédicat 1 : delete_building/1

Détruire un réseau et ses éléments.

Comme pour un bâtiment, il est possible de détruire un réseau et les éléments le constituant (machines, câbles, boîtes de connexion, connecteurs, etc), en lançant la commande :

```
?- send (dt, delete (_network)).
```

```
delete_network(_network):-
   existspropvalue (dt, _network, machines_and_devices,
     _machines_and_devices),
   existspropvalue(dt,_network,cables,_cables),
   existspropvalue (dt, _network, connexion_elements,
     _connexion_elements),
   existspropvalue (dt, _network, additional_components,
     _additional_components),
   delete_instances(dt,_machines_and_devices),
   delete_instances(dt,_cables),
   delete_instances(dt,_connexion_elements),
   delete_instances(dt,_additional_components),
   delobject(dt,_network,in,Networks).
```

prédicat 2 : delete_network/1

Détruire un composant de réseau.

Le composant souhaité est détruit mais aussi les connexions avec les éléments auxquels il est lié.

?- send (dt, delete (_network_component)).

```
delete_network_component(_net_comp):-
   existspropvalue(dt,_net_comp,connected_elements,_connected),
   delete_connexion_from_to(_net_comp,_connected),
   existspropvalue(dt,_net_comp,additional_components,_add),
   delete_references_to(_net_comp),
   delete_instances(dt,[_net_comp|_add]).

delete_connexion_from_to(_,[]):-!.
delete_connexion_from_to(_net_comp,[_first|_others]):-
   existspropvalue(dt,_first,connected_elements,_connected),
   retract_element(_net_comp,_connected,_new_connected),
   modpropvalue(dt,_first,connected_elements,_new_connected),
   delete_connexion_from_to(_net_comp,_others).
```

../..

```
delete_references_to(_net_comp):-
   (getobject(dt,_net_comp,in,Machines),
    getobject(dt,_machine,in,Machines),
    (existspropvalue(dt,_machine,diskless_server,_net_comp),
    delpropvalue(dt,_machine,diskless_server)
    ;
    true
    ),
    (existspropvalue(dt,_machine,application_servers,_mach_list),
    retract_element(_net_comp,_mach_list,_new_mach_list),
    modpropvalue(dt,_machine,application_servers,_new_mach_list)
    ;
    true
    ),
    fail
   ;
    true
   ).
```

prédicat 3 : delete_network_component/1

Détruire un composant additionnel (disques, mémoires, etc, logiciel).

Pour détruire un élément additionnel : `?- send (dt, delete (_additional_component)).`

```
delete_additional_component(_add_comp):-
   getobject(dt,_add_comp,in,_class),
   delobject(dt,_add_comp,in,_class).
```

prédicat 4 : delete_additional_component/1

Déplacer une machine dans un autre pièce.

Les connexions de la machine sont d'abord détruites, puis la machine est déplacée dans la pièce désirée et elle est connectée à un connecteur vide dans cette pièce. L'opération échoue s'il n'y a pas de connecteur vide.

```
?- send (dt, move (Machine1, Room2, _x)).
```

```
move_machine_to_room(_machine,_room,_connector):-
   existspropvalue(dt,_machine,partof,_network),
   find_empty_connector(_network,_room,_connector),
   delete_connexion_to_box(_machine),
   setpropvalue(dt,_machine,set_in_rooms,[_room]),
   connect_connector_to_box (_connector,_machine).
```

prédicat 5 : move_machine_to_room/3

Connecter ou déconnecter une machine à un connecteur donné.

```
?- send (dt, add_link (Machine1, Connector)).
```

```
add_link_from_connector_to_box(_box,_connector):-
   connect_connector_to_box(_connector,_box).
```

prédicat 6 : add_link_from_connector_to_box/2

?- send (dt, delete_link (Machine1)).

```
delete_connexion_to_box(_element):-
   existspropvalue (dt, _element, connected_elements,
     _connected_elements),
   (_connected_elements == [],!
   ;
   _connected_elements = [_connected],
   modpropvalue(dt,_element,connected_elements,[]),
   delete_connexion_to_box(_element,_connected)
   ).
```

../..

```
delete_connexion_to_box(_element,_connected):-
    existspropvalue(dt,_connected,connected_elements,_connected2),
    retract_element(_element,_connected2,_new_connected2),
    ((getobject(dt,_connected,in,Box_connectors)
     ;
     getobject(dt,_connected,in,Fan_outs)
     ),!,
     modpropvalue(dt,_connected,connected_elements,_new_connected2)
    ;
     getobject(dt,_connected,in,_class),
     delobject(dt,_connected,in,_class),
     (_new_connected2 == [],!
     ;
     _new_connected2 = [_new_connected],
     delete_connexion_to_box(_connected,_new_connected)
     )
    ).
```

prédicat 7 : delete_connexion_to_box/1

Vérifier si une machine a un disque.

?- send (dt, has_a_disk (Machine1, _disk)).

```
has_a_disk(_machine,_disk):-
    getobject(dt,_disk,IN,Disks),
    existspropvalue(dt,_disk,installed_on,_machine).
```

prédicat 8 : has_a_disk/2

Spécialiser ou généraliser un objet.

Cette opération permet, par exemple, de spécialiser un câble appartenant à la classe générale Cables comme étant un Thin_cables câble, coaxial fin.

```
?- send (dt, secify (Cable1, dt, Cables, Thin_cables)).
```

```
specify(_object,_kb,_class,_subclass):-
    getvalues(_kb,_object, _propvalues),
    delobject(_kb,_object,in,_class),
    putobject(_kb,_object,in,_subclass),
    putdefinedvalues(_kb,_object, _subclass, _propvalues).
```

prédicat 9 : specify/4

```
?- send (dt, generalize (Cable1, dt, Thin_cables, Cables)).
```

```
generalize(_object,_kb,_class,_superclass):-
    getownvalues(_kb,_object, _class, _propvalues),
    delobject(_kb,_object,in,_class),
    putobject(_kb,_object,in,_superclass),
    putdefinedvalues(_kb,_object, _superclass, _propvalues).
```

prédicat 10 : generalize/4

Définir un serveur de réseau.

```
?- send (dt, define_network_sever (Machine1)).
```

```
define_network_server(_server):-
    setpropvalue(dt,_server,is_a_server,true),
    existspropvalue(dt,_server,partof,_network),
    (getobject(dt,_machine,in,Machines),
     _machine \== _server,
     existspropvalue(dt,_machine,partof,_network),
     setpropvalue(dt,_machine,diskless_server,_server),
     existspropvalue(dt,_machine,application_servers,_application_servers),
     union([_server],_application_servers,_new),
     modpropvalue(dt,_machine,application_servers,_new),
     fail
    ;
     true
    ).
```

prédicat 11 : define_network_server/1

Insérer ou supprimer les exigences de l'utilisateur.

Trois prédicats permettent respectivement d'insérer, de consulter ou de détruire un fait :

- assert_fact_in_kb (_kb, _fact),
- get_fact_in_kb (_kb, _fact),
- retract_fact_in_kb (_kb, _fact)

Par exemple, pour insérer le budget d'un client, l'appel suivant sera effectué :

```
?- send (dt, assert_fact_in_kb (dt, budget (Network1, 550 000)).
```

```
assert_fact_in_kb(_kb,_fact):-
    kbstatus(_kb,[_, _, _, modulename = _module | _]),
    _fact =.. [_pred|_args],
    module(_pred_mod,_module,_pred),
    _new_fact =.. [_pred_mod|_args],
    assert(_new_fact).

retract_fact_from_kb(_kb,_fact):-
    kbstatus(_kb,[_, _, _, modulename = _module | _]),
    _fact =.. [_pred|_args],
    module(_pred_mod,_module,_pred),
    _new_fact =.. [_pred_mod|_args],
    retract(_new_fact).

get_fact_from_kb(_kb,_fact):-
    kbstatus(_kb,[_, _, _, modulename = _module | _]),
    _fact =.. [_pred|_args],
    module(_pred_mod,_module,_pred),
    _new_fact =.. [_pred_mod|_args],
    _new_fact.
```

prédicat 12 : assert (retract et get)_fact_in_kb/2

Annexe 6 : Programmes

Prolog.

Cette annexe contient les programmes prolog que j'ai réalisés. Elle est composée

des fichiers:	*correspondant aux tâches :*
- *skeleton.pro*	*Sélection d'un squelette de réseau,*
- *distance.pro*	*Calculateur de la distance à câbler,*
- *cables_location.pro*	*Localisateur des câbles,*
- *vertical_design.pro*	*modules que j'ai crées pour la Fusion des différents sous-réseaux*
- *building_information.pro*	*prédicats utiles pour plusieurs modules.*

```
{------------------------------------------------------------------}
{       SKELETON SELECTION MODULE.                                 }
{------------------------------------------------------------------}

{==================================}
{AUTHOR : Christine JOUVE  EMSE     }
{CREATION DATE : June 90            }
{LAST UPDATE : 16 January 91        }
{==================================}

{-------------------------------------------------------------------------}
{The purpose of these modules is to suggest a skeleton of the network.    }
{They permit to choose a type of skeleton depending on the problem width, }
{to calculate the number of needed vertical backbones, to find a vertical }
{location (vertical shaft or stairs or lift) for these vertical backbones,}
{to give the list of floor to be cabled for each vertical backbone.       }
{-------------------------------------------------------------------------}

{-------------------------------------------------------------------------}
{resolve_skeleton is the global predicate which activates all the skeleton }
{modules and which makes the initialization.                               }
{-------------------------------------------------------------------------}

resolve_skeleton (_network, _List_of_Vertical_Bs) :-
        getlist (dt,_Bui, DIRECTIN, Buildings),
        select_type (_network, _Bui),
        getpropvalue (dt, Max_nb_of_machines_in_skeleton, value, _s),
        getlist (dt, _Sk, DIRECTIN, Skeletons),
        nb_skeleton (_network, _Sk, _s),
        getlist (dt,_List_of_Vertical_Bs, DIRECTIN, Vertical_backbones),
        place_skeleton (_List_of_Vertical_Bs).

{------------------------------------------------------------------}
{       SELECTOR OF SKELETON TYPE.                                 }
{This module puts Skeletons instances (one by building up to now) and }
{for each of these instances puts their type and building properties. }
{Three types of skeletons can be implemented depending on building    }
{architecture : for only one floor buidling, for buidling with one    }
{department by floor and the others (mixed departments).              }
{------------------------------------------------------------------}

{-----------------}
{Several buildings}
{-----------------}

select_type (_network, [_b|_B]) :-
        send (dt, _F = (all _fl where _fl is_in Floors and
                _fl#set_in_building = _b)),
        select_type (_network, [_b], _F),
        select_type (_network, _B).

select_type (_, []).

{--------------------------------}
{One building and only one floor }
{--------------------------------}

select_type (_network, [_b], [_one_floor]) :-
        !,
        putobject (dt, _up, IN, Skeletons),
        putpropvalue (dt, _up, type, @only_one_floor),
        putpropvalue (dt, _up, building, _b),
        putpropvalue (dt, _up, partof, _network).

{-------------------------------------------------------------------------}
{One building, one department by floor and number of connexions in each   }
{floor below 40 (little scale problem).                                   }
{-------------------------------------------------------------------------}

select_type (_network, [_b], [_fl|_F]) :-
        check_nb_of_machines_by_floor (_network, [_fl|_F]),
        check_one_dep_by_floor ([_fl|_F]), !,
        putobject (dt, _up, IN, Skeletons),
        putpropvalue (dt, _up, type, @one_department_by_floor),
        putpropvalue (dt, _up, building, _b),
        putpropvalue (dt, _up, partof, _network).

{-------------------------------------------------------------------}
{This predicat checks the number of connexions for each foor of the }
{given list is below 40.                                            }
{-------------------------------------------------------------------}

check_nb_of_machines_by_floor (_, []).
check_nb_of_machines_by_floor (_network, [_fl|_F]) :-
        getpropvalue (dt, _fl, list_of_rooms, _Lr),
        count_connexions_nb (_network, _Lr, 0, _n),
        (_n < 40),
        check_nb_of_machines_by_floor (_network, _F).

{-------------------------------------------------------------------------}
{count_connexions_nb (_network, _Lr, _accu, _n) gives the number (_n) of  }
{connexions which are in every room of the list Lr.                       }
{-------------------------------------------------------------------------}

count_connexions_nb (_network, [], _n, _n).
count_connexions_nb (_network, [_r|_Lr], _n1, _n) :-
        room_connexions_number (_network, _r, _c_n),
        room_wished_connexions_number (_network, _r, _w_c_n),
        (_n2 is _c_n + _w_c_n),
        (_n3 is _n1 + _n2),
        count_connexions_nb (_network, _Lr, _n3, _n).

{-------------------------------------------------------------}
{This predicat checks if all rooms of a same floor belong to a }
{same department.                                             }
{-------------------------------------------------------------}

check_one_dep_by_floor ([]).
check_one_dep_by_floor ([_fl|_F]) :-
        getpropvalue (dt, _fl, list_of_rooms, _R),
        one_dep_by_floor (_R, []),
        check_one_dep_by_floor (_F).

one_dep_by_floor ([], [_d]).
one_dep_by_floor ([], []).
one_dep_by_floor ([_r|_R], []) :-
        getpropvalue (dt, _r, set_in_departments, _D),
        one_dep_by_floor (_R, _D).

one_dep_by_floor ([_r|_R], [_d1]) :-
        getpropvalue (dt, _r, set_in_departments, _D2),
        send (dt, [_d1] = _D2  v _D2 = []),
        one_dep_by_floor (_R, [_d1]).

{---------------------------------}
{One building and mixed departments }
{---------------------------------}
```

```
select_type (_network, [_b], _) :-
        putobject (dt, _up, IN, Skeletons),
        putpropvalue (dt, _up, type, @mixed_department),
        putpropvalue (dt, _up, building, _b),
        putpropvalue (dt, _up, partof, _network).

{-------------------------------------------------------}
{validity_constraints for selector of skeleton type     }
{depend on building architecture and departmentalization. }
{-------------------------------------------------------}

validity_constraints (skeleton_type, [compatible_architecture_type]).

compatible_architecture_type (_skeleton, _type) :-
        existspropvalue (dt, _skeleton, building, _b),
        existspropvalue (dt, _skeleton, partof, _network),
        send (dt, _F = (all _fl where _fl is_in Floors and
                _fl#set_in_building = _b)),
        check_type_skeleton (_network, _type, _F).

check_type_skeleton (_network, @only_one_floor, [_one_floor]).
check_type_skeleton (_network, @one_department_by_floor, [_fl|_F]) :-
        check_nb_of_machines_by_floor (_network, [_fl|_F]),
        check_one_dep_by_floor ([_fl|_F]).
check_type_skeleton (_network, @mixed_department, [_fl|_F]) :-
        (not (check_nb_of_machines_by_floor (_network, [_fl|_F])) ;
         not (check_one_dep_by_floor ([_fl|_F]))).

{-------------------------------------------------------------}
{       SELECTOR OF VERTICAL BLACKBONE NUMBER.                  }
{It makes the decomposition of the network into subnetworks     }
{according to skeleton type, departmentalization principle, fixed }
{threshods onalowed number of machines on a subnetwork, presence  }
{of servers.                                                    }
{-------------------------------------------------------------}

nb_skeleton (_, [], _s).
nb_skeleton (_network, [_sk|_Sk], _s) :-
        getpropvalue (dt, _sk, type, _type),
        distribution (_network, _sk, _type, _s),
        nb_skeleton (_network, _Sk, _s).

distribution (_network, _sk, @only_one_floor, _s) :-
        !,
        putobject (dt, _vert, IN, Vertical_backbones),
        putpropvalue (dt, _vert, partof, _network),
        getpropvalue (dt, _sk, building, _b),
        putpropvalue (dt, _sk, list_of_vertical_backbones, [_vert]),
        putpropvalue (dt, _vert, vertical, []),
        send (dt, _Ld1 = (all _d where _d is_in Departments and
        _r = oneof _d#elements and _b = _r#set_in_building)),
        no_double (_Ld1, _Ld),
        putpropvalue (dt, _vert, depart, _Ld),
        send (dt, _floor = (_fl where _fl is_in Floors and
                _b = _fl#set_in_building)),
        compute_floor_departments_slot (_vert, [_floor], _floor_departments),
        putpropvalue (_vert, floor_departments, _floor_departments).

distribution (_network, _sk, @one_department_by_floor, _s) :-
        !,
        putobject (dt, _vert, IN, Vertical_backbones),
        putpropvalue (dt, _vert, partof, _network),
        getpropvalue (dt, _sk, building, _b),
        putpropvalue (dt, _sk, list_of_vertical_backbones, [_vert]),
        send (dt, _Lv = (all _v where _v is_in Vertical_shafts and
                [_fl|_Fl] = _v#joined_floors and
                _b = _fl#set_in_building)),
        putpropvalue (dt, _vert, all_verticals, _Lv),
        putpropvalue (dt, _vert, depart, []),
        floor_to_be_cabled (_network, [_vert], [[_vert, _L_floors]]),
        compute_floor_departments_slot (_vert, _L_floors, _floor_departments),
        putpropvalue (_vert, floor_departments, _floor_departments).

distribution (_network, _sk, @mixed_department, _s) :-
        getpropvalue (dt, _sk, building, _b),
        send (dt, _Lr2 = (all _r where _r is_in Rooms and
                _r#set_in_building = _b and
                _r#set_in_departments = [])),
        del_corridor_from_list (_Lr2, [_r|_Lr3]),
         !,
        putpropvalue (dt, _sk, list_of_vertical_backbones, []),
        putobject (dt, _d, IN, Group_depart),
        putpropvalue (dt, _d, additional_rooms, [_r|_Lr3]),
        getpropvalue (dt, _b, elements, _Lr),
        extract_departments (_Lr, _Ld),
        test_depart (_network, [_d|_Ld], _s, _sk),
        existspropvalue (dt, _sk, list_of_vertical_backbones, _Lvert),
        floor_to_be_cabled (_network, _Lvert, _Lvert_floors),
        compute_floor_departments_slot (_Lvert_floors).

distribution (_network, _sk, @mixed_department, _s) :-
        getpropvalue (dt, _sk, building, _b),
        getpropvalue (dt, _b, elements, _Lr),
        extract_departments (_Lr, _Ld),
        putpropvalue (dt, _sk, list_of_vertical_backbones, []),
        test_depart (_network, _Ld, _s, _sk),
        existspropvalue (dt, _sk, list_of_vertical_backbones, _Lvert),
        floor_to_be_cabled (_network, _Lvert, _Lvert_floors),
        compute_floor_departments_slot (_Lvert_floors).

{-------------------------------------------------------------}
{This predicat extracts from a given list of rooms the list of }
{their departments.                                             }
{-------------------------------------------------------------}

extract_departments ([_r|_Lr], _Ld) :-
        extract_departments (_Lr, _Ldi),
        getpropvalue (dt, _r, set_in_departments, _D),
        union (_D, _Ldi, _Ld).
extract_departments ([], []).

{-------------------------------------------------------------}
{For each department, test_depart calls check_server predicat. }
{-------------------------------------------------------------}

test_depart (_network, [], _, _).
test_depart (_network, [_depart|_Ldepart], _s, _sk) :-
        getpropvalue (dt, _depart, elements, _Lr),
        send (dt, _Lmach = (all _mach where
                _mach is_in Machines and
                is_sublist (_mach # set_in_rooms, _Lr))),
        send (dt, _Lser = (all _m where
                _m = oneof _Lmach and
                _m # is_a_server = true)),
        check_server (_network, _Lser, _Lmach, [_depart|_Ldepart], _s, _sk).

{-------------------------------------------------------------}
{check_server checks if a server and its relative machines are   }
```

```
{in a same department to link them through a same vertical backbone. }
{If it isn't the case, it regroups the different departments into a  }
{same one (a Group_depart instance).                                }
{-------------------------------------------------------------------}

check_server (_network, [], _Lmach, [_depart|_Ldepart], _s, _sk) :-
        getpropvalue (dt, Min_nb_of_machines_in_skeleton, value, _sinf),
        getpropvalue (dt, _depart, elements, _Lr),
        count_connexions_nb (_network, _Lr, 0, _n),
        test_thresold (_network, _sk, _n, _s, _sinf, [_depart|_Ldepart], _Ld),
        test_depart (_network, _Ld, _s, _sk).

check_server (_network, [_ser|_Lser], _Lmach, _Ldepart, _s, _sk) :-
        send (dt, _L = (all _m where _m is_in Machines and
                _ser = _m # diskless_server)),
        inclu (_L, _Lmach), !,
        check_server (_network, _Lser, _Lmach, _Ldepart, _s, _sk).

check_server (_network, [_ser|_Lser], _Lmach, _Ldepart, _s, _sk) :-
        send (dt, _L = (all _m where _m is_in Machines and
                _ser = _m # diskless_server)),
        group_dept_for_server (_sk, [_ser|_Lm], _Ldepart, _Ld),
        test_depart (_network, _Ld, _s, _sk).

{-------------------------------------------------------------------}
{This predicat regroups the different departments in which are the  }
{server and its relative machines into one Group_depart.            }
{-------------------------------------------------------------------}

group_dept_for_server (_sk, _Lmach, _Ldepart, [_d|_Ld]) :-
        get_dept (_Lmach, _L),
        putobject (dt, _gp, IN, Group_depart),
        put_set_in_group (_L, _gp),
        getpropvalue (dt, _sk, list_of_vertical_backbones, _Lv),
        check_skeleton (_sk, _Lv, _L),
        del_list (_Ldepart, _L, _Ld).

{-------------------------------------------------------------------}
{This predicat extracts from a given list of machines the list of   }
{their departments.                                                 }
{-------------------------------------------------------------------}

get_dept ([_mach|_Lmach], _Ld) :-
        getpropvalue (dt, _mach, set_in_rooms, [_room]),
        getpropvalue (dt, _room, set_in_departments, [_dept]),
        get_dept (_Lmach, _L),
        union ([_dept], _L, _Ld).
get_dept ([], []).

{-------------------------------------------------------------------}
{This predicat puts the value of set_in_group of all department which }
{belongs to the Group_depart (_gp).                                 }
{-------------------------------------------------------------------}

put_set_in_group ([], _).
put_set_in_group ([_d|_Ld], _gp) :-
        putpropvalue (dt, _d, set_in_group, _gp),
        put_set_in_group (_Ld, _gp).

{-------------------------------------------------------------------}
{check_skeleton checks if a vertical backbone hasn't been created for  }
{a department of the given list (one of the server and its relative    }
{machines department) yet. If it is the case, it deletes this vertical }
{backbone and it suppresses it from the list_of_vertical_backbones     }
{property of its relative skeleton.                                    }
{-------------------------------------------------------------------}

check_skeleton (_sk, [_v|_Lv], _Ld) :-
        getpropvalue (dt, _v, depart, [_d]),
        (not_member (_d, _Ld) ;
           (member (_d, _Ld), delobject (_v, IN, Vertical_backbones),
                subpropvalue (dt, _sk, list_of_vertical_backbones, _v))),
        check_skeleton (_sk, _Lv, _Ld).
check_skeleton (_, [], _).

{-------------------------------------------------------------------}
{It tests the number of machines (_n).                              }
{if _n=< _s_sup and _n >= _s_inf, it creates a Vertical backbone,   }
{if _n < _s_inf, it calls group_department which allows to regroup  }
{two departments in one Group_depart,                               }
{if _n > _s, it calls de_group which decomposes a department into   }
{two Sub_depart.                                                    }
{-------------------------------------------------------------------}

test_thresold (_network, _sk, _n, _s_sup, _s_inf, [_d|_Ld], _Ld) :-
        _n =< _s_sup,
        _n >= _s_inf,
        putobject (dt, _v, IN, Vertical_backbones),
        putpropvalue (dt, _v, partof, _network),
        addpropvalue (dt, _sk, list_of_vertical_backbones, _v),
        getpropvalue (dt, _sk, building, _b),
        send (dt, _Verticals = (all _vert where _vert is_in Vertical_shafts and
                [_fl|_Fl] = _vert#joined_floors and
                _b = _fl#set_in_building)),
        putpropvalue (dt, _v, all_verticals, _Verticals),
        putpropvalue (dt, _v, depart, [_d]).

test_thresold (_network, _sk, _n, _s_sup, _s_inf, [_d|_Ld], _Ld2) :-
        _n < _s_inf,
        group_department (_network, _sk, [_d|_Ld], _s_sup, _Ld2).

test_thresold (_network, _sk, _n, _s, _, [_d|_Ld], _Lsol) :-
        _n > _s,
        de_group (_network, [_d|_Ld], _s, _Lsol).

{-------------------------------------------------------------------}
{This predicat regroups two departments into one if their number    }
{of machines is below inferior thresold.                            }
{-------------------------------------------------------------------}

group_department (_network, _sk, [_d|_Ld], _s, [_newd|_Ld2]) :-
        get_new_depart (_network, _d, _s, _otherd),
        putobject (dt, _newd, IN, Group_depart),
        modpropvalue (dt, _d, set_in_group, _newd),
        modpropvalue (dt, _otherd, set_in_group, _newd),
        getpropvalue (dt, _sk, list_of_vertical_backbones, _Lv),
        check_skeleton (_sk, _Lv, [_otherd]),
        retract_element (_otherd , _Ld, _Ld2), !.
group_department (_network, _sk, [_d|_Ld], _, _Ld) :-
        putobject (dt, _v, IN, Vertical_backbones),
        putpropvalue (dt, _v, partof, _network),
        addpropvalue (dt, _sk, list_of_vertical_backbones, _v),
        getpropvalue (dt, _sk, building, _b),
        send (dt, _Verticals = (all _vert where _vert is_in Vertical_shafts and
                [_fl|_Fl] = _vert#joined_floors and
                _b = _fl#set_in_building)),
        putpropvalue (dt, _v, all_verticals, _Verticals),
        putpropvalue (dt, _v, depart, [_d]).
```

```
{------------------------------------------------------------------------}
{get_new_depart (_network, _d, _s, _newd) looks for a department _newd that }
{belongs to a room next to another room which is set in department _d. }
{------------------------------------------------------------------------}

get_new_depart (_network, _d, _s, _newd) :-
        getpropvalue (dt, _d, elements, _Lr),
        neighour_list_rooms (_Lr, _Lnr),
        send (dt, _newd = (_d3 where
                _room = oneof _Lnr and
                [_d2] = _room#set_in_departments and
                _gp = _d2#set_in_group and
                prolog (check (_network, _gp, _d2, _d, _s, _d3)) )),
        !.
get_new_depart (_network, _d, _s, _newd) :-
        getpropvalue (dt, _d, elements, _Lr),
        send (dt, _newd = (_d3 where
                _r1 = oneof _Lr and
                _floor = _r1#set_in_floor and
                _r2 is_in Rooms and
                _r2#set_in_floor = _floor and
                [_d2] = _r2#set_in_departments and
                _gp = _d2#set_in_group and
                prolog (check (_network, _gp, _d2, _d, _s, _d3)) )),
        !.
get_new_depart (_network, _d, _s, _newd) :-
        send (dt, _newd = (_d3 where
                _d2 is_in Departments and
                _gp = _d2#set_in_group and
                prolog (check (_network, _gp, _d2, _d, _s, _d3)) )),
        !.

{-------------------------------------------------------}
{This predicat checks if the number of machines of the }
{regrouped departments is below the thresold _s.       }
{-------------------------------------------------------}

check (_network, [], _d2, _d, _s, _d2) :-
        send (dt, _d2 <> _d),
        getpropvalue (dt, _d2, elements, _Lr2),
        getpropvalue (dt, _d, elements, _Lr),
        count_connexions_nb (_network, _Lr2, 0, _n2),
        count_connexions_nb (_network, _Lr, 0, _n),
        ((_n2 + _n) =< _s).
check (_network, [_gp],_d2, _d, _s, _gp) :-
        send (dt, _gp <> _d),
        getpropvalue (dt, _gp, elements, _Lr2),
        getpropvalue (dt, _d, elements, _Lr),
        count_connexions_nb (_network, _Lr2, 0, _n2),
        count_connexions_nb (_network, _Lr, 0, _n),
        ((_n2 + _n) =< _s).

{-------------------------------------------------------}
{de_group decomposes a department into two Sub_depart. }
{-------------------------------------------------------}

de_group (_network, [_d|_Ld], _s, _Lsol) :-
        getpropvalue (dt, _d, elements, _Lr),
        create_sub_depart (_network, _d, _s, _Lr, _Lsub_depart),
        append (_Lsub_depart, _Ld, _Lsol).

{-------------------------------------------------------}
{create_sub_depart creates the needed sub_departs. Frist }
{it checks if the department is not a Group_depart which }
{allows to regroup a server and its relative machines.   }
{-------------------------------------------------------}

create_sub_depart (_network, _d, _s, [_r|_Lr], [_sub_d|_Q]) :-
        send (dt, [_se|_Lser] = (all _ser where _ser is_in Machines and
                is_sublist (_ser#set_in_rooms, [_r|_Lr]) and
                _ser#is_a_server = true)), !,
        regroup_server_and_machines ([_se|_Lser], _d, [_r|_Lr], _new_Lr),
        create_sub_depart (_network, _d, _s, _new_Lr, [_sub_d|_Q]).
create_sub_depart (_network, _d, _s,[_r|_Lr] , [_sub_d|_Q]) :-
        putobject (dt, _sub_d, IN, Sub_depart),
        putpropvalue (dt, _sub_d, set_in_depart, _d),
        cut_list_of_rooms (_network, [_r|_Lr], _s, 0, _L),
        putpropvalue (dt, _sub_d, rooms, _L),
        del_list ([_r|_Lr], _L, _Lnr),
        create_sub_depart (_network, _d, _s, _Lnr, _Q).
create_sub_depart (_, _, _, [], []).

{-------------------------------------------------------}
{regroup_server_and_machines makes a sub_depart with rooms }
{which enclose a server or its relative machines.        }
{-------------------------------------------------------}

regroup_server_and_machines ([], _, _Lr, _Lr).
regroup_server_and_machines ([_ser|_Lser], _d, [_r|_Lr], _new_Lr) :-
        send (dt, _Lrooms_mach = (all _r where _m is_in Machines and
                _ser = _m#diskless_server and
                _r = oneof _m#set_in_rooms)),
        send (dt, _room_ser = oneof _ser#set_in_rooms),
        putobject (dt, _sub_d, IN, Sub_depart),
        putpropvalue (dt, _sub_d, set_in_depart, _d),
        putpropvalue (dt, _sub_d, rooms, [_room_ser|_Lrooms_mach]),
        del_list ([_r|_Lr], [_room_ser|_Lrooms_mach], _Lnr),
        regroup_server_and_machines (_Lser, _d, _Lnr, _new_Lr).

{-------------------------------------------------------}
cut_list_of_rooms (_network, [_r|_Lr], _s, _acc, [_r|_L]) :-
        count_connexions_nb (_network, [_r], 0, _n),
        (_acc2 is _acc + _n),
        (_acc2 =< _s), !,
        cut_list_of_rooms (_network, _Lr, _s, _acc2, _L).
cut_list_of_rooms (_, _, _, _, []).

{-------------------------------------------------------------}
{From the list of list of vertical backbones and its floors    }
{to be cabled, it computes and puts for each vertical backbone }
{its floor_departments property.                               }
{-------------------------------------------------------------}

compute_floor_departments_slot ([]).
compute_floor_departments_slot ([[_vert, _Lfloors]|_Lvert_floors]) :-
        getpropvalue (dt, _vert, depart, _Ld),
        transform (_Ld, _Lfloors, _floor_department),
        putpropvalue (dt, _vert, floor_departments, _floor_department),
        compute_floor_departments_slot (_Lvert_floors).

{-------------------------------------------------------}
{This predicat computes for the given vertical backbone its }
{floor_departments slot from the list of floors.           }
{-------------------------------------------------------}

compute_floor_departments_slot (_vert, [], []).
compute_floor_departments_slot (_vert, [_fl|_L_floors], [[_fl, _Ldepart]|_Q]) :-
```

```
        send (dt, _Lr = (all _r where _r is_in Rooms and
                _fl = _r# set_in_floor and
                [] = _r#set_in_departments)),
        del_corridor_from_list (_Lr, _Lr_without_corridors),
        test_depart_nil (_Lr_without_corridors, _fl, _Ldepart),
        compute_floor_departments_slot (_vert, _L_floors, _Q).

{-------------------------------------------------------------}
{This predicat transforms a list of floors into a list of list }
{of a floor and a list of departments.                         }
{-------------------------------------------------------------}

transform (_Ld, [_fl|_Lfloors], [[_fl, _Ld] |_floor_department]) :-
        transform (_Ld, _Lfloors, _floor_department).
transform (_, [], []).

{-------------------------------------------------------------}
{del_corridor_from_list (_Lr, _Lr_without_corridors) deletes }
{from the list of rooms _Lr the corridors and puts it into   }
{_Lr_without_corridors.                                      }
{-------------------------------------------------------------}

del_corridor_from_list ([_r|_Lr], _Lr_without_corridors) :-
        getobject (dt, _r, DIRECTIN, Corridors), !,
        del_corridor_from_list (_Lr, _Lr_without_corridors).
del_corridor_from_list ([_r|_Lr], [_r|_Lr_without_corridors]) :-
        del_corridor_from_list (_Lr, _Lr_without_corridors).
del_corridor_from_list ([], []).

{-------------------------------------------------------------}
{For @only_one_floor type of skeleton, if there are rooms    }
{which don't belong to a department, a Group_depart is       }
{created and these rooms are put into the additional_rooms   }
{property of this group.                                     }
{-------------------------------------------------------------}

test_depart_nil ([], _fl, [_d]) :-
        send (dt, [_d] = ([_dl] where [_r|_R] = _fl#list_of_rooms and
                        [_dl] = _r#set_in_departments)), !.
test_depart_nil ([_r|_Lr], _fl, [_gp]) :-
        send (dt, [_d] = ([_dl] where _room = oneof _fl#list_of_rooms and
                        [_dl] = _room#set_in_departments)),
        putobject (dt, _gp, IN, Group_depart),
        putpropvalue (dt, _gp, additional_rooms, [_r|_Lr]),
        putpropvalue (dt, _d, set_in_group, _gp), !.

test_depart_nil ([_r|_Lr], _fl, [_gp]) :-
        putobject (dt, _gp, IN, Group_depart),
        putpropvalue (dt, _gp, additional_rooms, [_r|_Lr]).

{-------------------------------------------------------}
{validity_constraints for number of vertical backbone  }
{depend on servers, departmentalization and thresholds }
{-------------------------------------------------------}

validity_constraints (vertical_backbone_number, [servers,
                                                  departmentalization,
                                                  thresholds]).
{-------------------------------------------------------------}
servers (_vertical_backbone) :-
        send (dt, [_depart] = _vertical_backbone # depart),
        getpropvalue (dt, _depart, elements, _Lr),
        send (dt, _Lmach = (all _mach where
                _mach is_in Machines and
                        is_sublist (_mach # set_in_rooms, _Lr))),
        send (dt, _L = (all _m where
                _m is_in Machines and
                _m # is_a_server = true)),
        intersection (_Lmach, _L, _Lser),
        check_diskless_server (_Lser, _Lmach).
servers (_vertical_backbone) :-
        send (dt, [] = _vertical_backbone#depart).

check_diskless_server ([], _).
check_diskless_server ([_ser|_Lser], _Lm) :-
        send (dt, _L = (all _m where _m is_in Machines and
                _ser = _m # diskless_server)),
        inclu (_L, _Lmach),
        check_diskless_server (_Lser, _Lm).
{-------------------------------------------------------------}
{departmentalization (_vertical_backbone) :- }
{cf departmentalization analyser             }
{-------------------------------------------------------------}
thresholds (_vertical_backbone) :-
        send (dt, [] = _vertical_backbone#depart).
thresholds (_vertical_backbone) :-
        send (dt, [_depart] = _vertical_backbone # depart),
        existspropvalue (dt, _depart, nb_machines, _n),
        existspropvalue(dt, Max_nb_of_machines_in_skeleton, value, _max),
        existspropvalue (dt, Min_nb_of_machines_in_skeleton, value, _min),
        (_n =< _max),
        (_n >= _min).
{-------------------------------------------------------------}

{-------------------------------------------------------------}
{       SELECTOR OF CONNECTION POINTS.                        }
{It finds all floors which have to be cabled. It take too in  }
{consideration the needed extension.                          }
{-------------------------------------------------------------}

floor_to_be_cabled (_, [], []).
floor_to_be_cabled (_network, [_v|_Q], [[_v,_Answer]|_R]) :-
        send (dt, _b = (_building where _sk is_in Skeletons and
                _v = oneof _sk#list_of_vertical_backbones and
                _building = _sk#building)),
        send (dt, _Floors = (all _floor where _floor is_in Floors and
                _floor#set_in_building = _b)),
        existspropvalue (dt, _v, depart, _Ldepart),
        to_be_cabled (_network, _Floors, _Ldepart, _Answer),
        floor_to_be_cabled(_network, _Q, _R).

{-------------------------------------------------------------}
{This predicat checks from the given list of floors if each one }
{contains rooms whit machines or connexions.                    }
{-------------------------------------------------------------}

{-----------------------------------------------------}
{From the given list of floors, this predicat looks for }
{which needs to be cabled.                              }
{-----------------------------------------------------}

to_be_cabled (_, [],  _, []).
to_be_cabled (_network, [_floor|_Q], [], _A) :-
        existspropvalue (_floor, list_of_rooms, _Lr),
        rooms_to_be_cabled (_network, _Lr, _Lrooms_to_be_cabled),
        cabled (_Lrooms_to_be_cabled, _floor, _F),
        to_be_cabled (_network, _Q, [], _Aq),
        append (_F, _Aq, _A).
```

```
to_be_cabled (_network, [_floor|_Q], [_depart|_D], _A) :-
        existspropvalue (_floor, list_of_rooms, _Lr_in_floor),
        restrict_L_rooms (_network, _Lr_in_floor,[_depart|_D], _Lr),
        rooms_to_be_cabled (_network, _Lr, _Lrooms_to_be_cabled),
        cabled (_Lrooms_to_be_cabled, _floor, _F),
        to_be_cabled (_network, _Q, [_depart], _Aq),
        append (_F, _Aq, _A).

{-------------------------------------------------------}
{If the list of rooms to be cable is nil, cabled predicat }
{considers the extension slot of building.              }
{-------------------------------------------------------}

cabled ([], _floor, _R) :-
        getpropvalue (dt, _floor, set_in_building, _b),
        getpropvalue (dt, _b, extension, _Lfloor),
        extension (_floor, _Lfloor, _R).

extension (_floor, _Lfloor, [_floor]) :-
        member (_floor, _Lfloor), !.
extension (_, _, []).

cabled ([_X|_Q], _floor, [_floor]).

{-------------------------------------------------------------------}
{       SELECTOR OF BACKBONE LOCATIONS.                             }
{It finds for each backbone a vertical place (shaft, stairs, lift).   }
{Between all possible vertical (shafts, stairs, lifts), the selection }
{of one of them is made as follows :                                }
{First, I search the following two kinds of vertical :              }
{               - the most central in the building - vert_c         }
{               - the nearest from the computer room - vert_i       }
{before the choose of one of both is made according to the next rules.}
{If the building is a modern one, there is enough space in shafts.    }
{So if there is a shaft I choose it otherwise I take the central      }
{vertical. For other types of building, I prefer the central vertical }
{and keep in mind the second posibility.                            }
{NB : A check is done to verify if the choosen vertical allows to     }
{joint all floors (ok_ predicats).                                  }
{-------------------------------------------------------------------}

place_skeleton ([]).

place_skeleton ([_vertical_backbone]) :-
        existspropvalue (dt, _vertical_backbone, vertical, []), !.

place_skeleton ([_v|_Q]) :-
        getpropvalue (dt, _v, all_verticals, _Verticals),
        choose_vertical (_v, _Verticals),
        place_skeleton (_Q).

{-------------------------------------------------------------}
{choose_vertical (_v, _Lv) chooses one vertical among the list }
{of verticals _Lv for the vertical backbone _v.               }
{-------------------------------------------------------------}

choose_vertical (_v, []) :-
        putpropvalue (dt, _v, vertical, []).

choose_vertical (_v, [_vertical]) :-
        check_joined_floors (_v, _vertical, _Vert),
        ok1 (_v, _Vert, _vertical).
choose_vertical (_v, [_vert1, _vert2|_Vert]) :-
        find_central (_v, [_vert1, _vert2|_Vert], _vert_c),
        send (dt, _sk = (_sk1 where _sk1 is_in Skeletons and
                _v = oneof _sk1#list_of_vertical_backbones)),
        send (dt, _Computer_rooms = (all _r where _r is_in Computer_rooms and
                _b = _sk#building and
                _b = _r # set_in_building)),
        find_near_info (_v, [_vert1, _vert2|_Vert], _Computer_rooms, _vert_i),
        choose_between_central_and_near_info (_v, _vert_c, _vert_i).

{-------------------------------------------------------------}

ok1 (_v, [], _) :-
        putpropvalue (dt, _v, vertical, []).
ok1 (_v, [_ve], _ve) :-
        putpropvalue (dt, _v, vertical, [_ve]).

{-------------------------------------------------------------}
{find_central (_v, _Vert, _vert_c) looks for the most central }
{vertical (_vert_c) through the list _Vert.                   }
{-------------------------------------------------------------}

find_central (_v, _Vert, _vert_c) :-
        send (dt, _b = (_building where _sk is_in Skeletons and
                _v = oneof _sk#list_of_vertical_backbones and
                _building = _sk#building)),
        central (_b, _Vert, _vert),
        check_joined_floors (_v, _vert, _R_check),
        ok2 (_v, _Vert, _vert, _R_check, _vert_c).
find_central (_, [], nil).

central (_b, _Vert, _vert) :-
        getpropvalue (dt, _b, centre, _Centre),
        minimal_distance (_Vert, _Centre, _vert).

ok2 (_, _, _, [_vert], _vert).
ok2 (_v, _Vert, _vert, [], _vert_c) :-
        retract_element (_vert, _Vert, _V),
        find_central (_v, _V, _vert_c).

{-------------------------------------------------------------}
{find_central (_v, _Vert, _vert_c) looks for the vertical the }
{most near the computer room vertical (_vert_i) through the   }
{list _Vert.                                                  }
{-------------------------------------------------------------}

find_near_info (_v, _Vert, _Crooms, _vert_i) :-
        near_info (_Vert, _Crooms, _vert),
        check_joined_floors (_v, _vert, _R_check),
        ok3 (_v, _Vert, _vert, _Crooms, _R_check, _vert_i).
find_near_info (_, [], _, nil).
find_near_info (_, _, [], nil).

near_info (_Vert, [_cr], _vert) :-
        getpropvalue(dt, _cr, centre, _Centre),
        minimal_distance (_Vert, _Centre, _vert).

near_info (_Vert, [_cr,_cr2|_Q], _vert) :-
        getpropvalue(dt, _cr, centre, _Centre),
        minimal_distance (_Vert, _Centre, _vert),
        near_info (_Vert, [_cr2|_Q], _vert).

ok3 (_, _, _, _, [_vert], _vert).
ok3 (_v, _Vert, _vert, _Crooms, [], _vert_i) :-
        retract_element (_vert, _Vert, _V),
```

```
        find_near_info (_v, _V, _Crooms, _vert_i).

{-------------------------------------------------------------------}
{minimal_distance (_Lv, [_xt,_yt], _v) looks for element _v of _Lv }
{from which the distance from it to [_xt,_yt] is the smaller.      }
{-------------------------------------------------------------------}

minimal_distance ([_v1, _v2 |_Q], [_xt,_yt], _v) :-
        getpropvalue (dt, _v1, coord_x_y, [_x1, _y1]),
        getpropvalue (dt, _v2, coord_x_y, [_x2, _y2]),
        distance ([_x1,_y1], [_xt,_yt], _d1),
        distance ([_x2,_y2], [_xt,_yt], _d2),
        ( (_d1=<_d2, minimal_distance ([_v1|_Q], [_xt,_yt], _v) );
          (_d1>_d2, minimal_distance ([_v2|_Q], [_xt,_yt], _v) )
        ).
minimal_distance ([_v], _L, _v).

{-------------------------------------------------------------}
{This predicat makes the choice between the vertical the most }
{central and the vertical the most near the computer room.    }
{-------------------------------------------------------------}

choose_between_central_and_near_info (_v, _vert_c, _vert_i) :-
        send (dt, [_fl|_] = _vert_c # joined_floors),
        send (dt, _b = _fl#set_in_building ),
        getpropvalue (dt, _b, building_type, @modern),
        getobject (dt, _vert_i, DIRECTIN, _ti),
        _ti = @shaft,
        getobject (dt, _vert_c, DIRECTIN, _tc),
        (_tc = @lift ; _tc = @stairs),
        putpropvalue (dt, _v, vertical, [_vert_i]),
        putpropvalue (dt, _v, vertical_aux, [_vert_c]), !.

choose_between_central_and_near_info (_v, _vert_c, _vert_i) :-
        putpropvalue (dt, _v, vertical, [_vert_c]),
        putpropvalue (dt, _v, vertical_aux, [_vert_i]).

choose_between_central_and_near_info (_v, nil, _vert_i) :-
        _vert_i \== nil,
        putpropvalue (dt, _v, vertical, [_vert_i]).

choose_between_central_and_near_info (_v, _vert_c, nil) :-
        vert_c \== nil,
        putpropvalue (dt, _v, vertical, [_vert_c]).

choose_between_central_and_near_info (_v, nil, nil) :-
        putpropvalue (dt, _v, vertical, []).

{-------------------------------------------------------------------}
{This predicat checks if the choosen vertical allows to join each   }
{floor of the floor_departments slot of the given vertical backbone. }
{-------------------------------------------------------------------}

check_joined_floors (_v, _vert, [_vert]) :-
        send (dt, _List = (all _fl where
                _r = oneof _vert#set_in_rooms and
                _fl = _r # set_in_floor)),
        getpropvalue (dt, _v, floor_departments, _floor_departments),
        extract_floors (_floor_departments, _L_floors),
        inclu (_L_floors, _List), !.
check_joined_floors (_v, _, []).

{-----------------------------------------------------}
{validity_constraints for selector of backbone location }
{-----------------------------------------------------}

validity_constraints (vertical_backbone_location, [check_joined_floors]).

check_joined_floors (_vertical_backbone) :-
        existspropvalue (dt, _vertical_backbone, vertical, [_vert]),
        send (dt, _List = (all _fl where
                _r = oneof _vert#set_in_rooms and
                _fl = _r # set_in_floor)),
        getpropvalue (dt, _v, floor_departments, _floor_departments ),
        extract_floors (_floor_departments, _L_floors),
        inclu (_L_floors, _List).

preference_constraint (vertical_backbone_location, [most_central,
                                        most_near_the_computer_room]).

most_central (_vertical_backbone) :-
        getpropvalue (dt, _vertical_backbone, vertical, [_vert]),
        send (dt, _b = (_building where _sk is_in Skeletons and
                _vertical_backbone = oneof _sk#list_of_vertical_backbones and
                _building = _sk#building)),
        send (dt, _Lve = (all _v where _v is_in Vertical_shafts and
                [_fl|_Fl] = _v#joined_floors and
                _b = _fl#set_in_building)),
        find_central (_vertical_backbone, _Lve, _vert).

most_near_the_computer_room (_vertical_backbone) :-
        getpropvalue (dt, _vertical_backbone, vertical, [_vert]),
        send (dt, _b = (_building where _sk is_in Skeletons and
                _vertical_backbone = oneof _sk#list_of_vertical_backbones and
                _building = _sk#building)),
        send (dt, _Lve = (all _v where _v is_in Vertical_shafts and
                [_fl|_Fl] = _v#joined_floors and
                _b = _fl#set_in_building)),
        send (dt, [_c|_Lcr] = (all _r where _r is_in Computer_rooms and
                _b = _r # set_in_building)),
        find_near_info (_vertical_backbone, _Lve, [_c], _vert).
```

```
{-----------------------------------------------------------------}
{     CALCULATOR OF THE APPROXIMATIVE DISTANCE TO BE CABLED       }
{-----------------------------------------------------------------}

{==================================}
{AUTHOR : Christine JOUVE  EMSE     }
{CREATION DATE : June 90            }
{LAST UPDATE : 16 January 91        }
{==================================}


{-------------------------------------------------------------------------}
{The aim of this module is to compute the length of a linear cable        }
{which allows to connect all machines or wished rooms on a floor_network}
{and which has for departure point the arrival point of the vertical/     }
{backbone in this floor                                                   }
{-------------------------------------------------------------------------}

compute_dist_to_be_cabled (_floor_network, _dist) :-
        existspropvalue (dt, _floor_network,distance_to_be_cabled,_dist),!.

compute_dist_to_be_cabled (_floor_network, _dist) :-
        existspropvalue (dt, _floor_network, vertical_backbone, _vertical_backbone),
        getpropvalue (dt, _vertical_backbone, vertical, []),!,
        existspropvalue (dt, _floor_network,set_in_floor,_floor),
        existspropvalue (dt, _floor_network,set_in_departments, _Ldepartments),
        existspropvalue (dt, _floor_network, partof, _network),
        list_of_rooms_to_be_cabled (_network, _floor,_Ldepartments,
                                            _Lrooms_to_be_cabled),
        existspropvalue (dt, _floor, list_of_corridors, _Lcorridors),
        useful_corri (_Lcorridors, _Lrooms_to_be_cabled, [_corr|_Luseful_corridors]),
        find_departure_room (_floor_network, _vertical_backbone, _departure_room),
        compute_length (_corr, _departure_room, [_corr|_Luseful_corridors], _dist),
        putpropvalue(dt,_floor_network,distance_to_be_cabled,_dist).


compute_dist_to_be_cabled (_floor_network, _dist) :-
        existspropvalue (dt, _floor_network,set_in_floor,_floor),
        existspropvalue (dt, _floor_network,set_in_departments, _Ldepartments),
        existspropvalue (dt, _floor_network, partof, _network),
        list_of_rooms_to_be_cabled (_network, _floor,_Ldepartments,
                                            _Lrooms_to_be_cabled),
        existspropvalue (dt, _floor, list_of_corridors, _Lcorridors),
        useful_corri (_Lcorridors, _Lrooms_to_be_cabled, _Luseful_corridors),
        existspropvalue (dt, _floor_network, vertical_backbone, _vertical_backbone),
        getpropvalue (dt, _vertical_backbone, vertical, [_vertical]),
        find_corri_for_vertical (_floor, _vertical, _corri),
        compute_length (_corri, _vertical, _Luseful_corridors, _dist),
        putpropvalue(dt,_floor_network,distance_to_be_cabled,_dist).

compute_dist_to_be_cabled (_floor_network, 0) :-
        putpropvalue(dt,_floor_network,distance_to_be_cabled,0).

{-------------------------------------------------------------}
{useful_corri gives all useful cooridors which allow to connect }
{every rooms to be cabled and which is neighour to a cooridor.  }
{-------------------------------------------------------------}

useful_corri ([], _, []).
useful_corri (_, [], []).
useful_corri ([_c|_Lc], _Lr, _Lcu) :-
        neighour_rooms (_c, _Lnc),
        intersection (_Lr, _Lnc, _LR),
        del_list (_Lr, _LR, _L),
        useful_corri (_Lc, _L, _Lcs),
        test_usefulness_of_corridor (_LR, _c, _Ltest),
        append (_Ltest, _Lcs, _Lcu).

{if the intersection between the initial list of rooms and the list of  }
{rooms which are next to the considered corridor is empty, this corridor}
{isn't useful ; else it is.                                             }

test_usefulness_of_corridor ([], _, []).
test_usefulness_of_corridor ([_r|_Lr], _c, [_c]).

{-------------------------------------------------------------------------}
{With floor and arrival point of vertical backbone (well) in this     }
{floor, it takes the cooridor in which _weel arrives or the corridor }
{next to the room in which _weel arrives at this floor.               }
{-------------------------------------------------------------------------}

find_corri_for_vertical (_floor, _well, _corr) :-
     getpropvalue (dt, _well, set_in_rooms, _Lr),
     send (dt, _corr = oneof _Lr and
             _corr#set_in_floor = _floor and
             _corr is_in Corridors), !.
find_corri_for_vertical (_floor, _well, _corr) :-
     getpropvalue (dt, _well, set_in_rooms, _Lr),
     send (dt, _room = oneof _Lr and
             _room#set_in_floor = _floor),
     neighour_corridors (_room, [_corr|_Q]).

{-------------------------------------------------------------}
{compute_length makes the calculus of length of corridors       }
{-------------------------------------------------------------}

compute_length (_, _, [], 0).
compute_length (_corri, _vertical, [_c|_Lcorridors], _l) :-
        existspropvalue (dt, _corri, walls_succession, _Lwalls_corri),
        length_according_to_corridor_type (_corri, _Lwalls_corri, _vertical, _l1),
        retract_element (_corri, [_c|_Lcorridors], _new_Lcorridors),
        compute_length (_c, _corri, _new_Lcorridors, _l2),
        (_l is _l1 + _l2).

{-------------------------------------------------------------}
{The calculus of the length to be considered for a corrifor depends}
{of the type of corridor.                                       }
{-------------------------------------------------------------}

length_according_to_corridor_type (_corri, [_La_walls_corri], _vertical, _length) :-
        cutting (_La_walls_corri, _Cutting),
        find_walls_next_to_vertical (_Cutting, _vertical, _corri,
                                            _Walls_next_to_vertical),
        rearrange (_Cutting, _Walls_next_to_vertical, _Wall1_right,
                                _Wall2_right, _Wall1_left, _Wall2_left),
        suppress_undesirable_walls (_Wall1_right, _new_Wall1_right),
        length_walls (_new_Wall1_right, _length1_right),
        suppress_undesirable_walls (_Wall2_right, _new_Wall2_right),
        length_walls (_new_Wall2_right, _length2_right),
        ( (_length1_right =< _length2_right, (_length_right = _length1_right));
          (_length1_right > _length2_right, (_length_right = _length2_right))),
        suppress_undesirable_walls (_Wall1_left, _new_Wall1_left),
        length_walls (_new_Wall1_left, _length1_left),
        suppress_undesirable_walls (_Wall2_left, _new_Wall2_left),
        length_walls (_new_Wall2_left, _length2_left),
        ( (_length1_left =< _length2_left, (_length_left = _length1_left));
          (_length1_left > _length2_left, (_length_left = _length2_left))),
```

```
        (_length is _length_right + _length_left).

length_according_to_corridor_type (_corri, [_Walls1, _Walls2|_Walls], _, _length) :-
        longer_length ([_Walls1, _Walls2|_Walls], _, _length).

{------------------------------------------------------------------}
{rearrange (_L1, _L2, _W1r, _W2r, _W1l, _W2l) reorders the list _L  }
{in new lists : _W1r, _W2r, _W1l, _W2l, by begining these new lists }
{with the elements given in _L2.                                   }
{------------------------------------------------------------------}

rearrange (_Cutting, [_A, _B], _W1r, _W2r, _W1l, _W2l) :-
        place_first (_A, _Cutting, _Cr),
        sub_list ([_A|_Qr], _Cr),
        suffixe ([_B], _Qr),
        retract_element (_B, [_A|_Qr], _W1r),
        reverse (_Qr, _W2r),
        place_first (_B, _Cutting, _Cl),
        sub_list ([_B|_Ql], _Cl),
        suffixe ([_A], _Ql),
        retract_element (_A, [_B|_Ql], _W1l),
        reverse (_Ql, _W2l).

rearrange (_Cutting, [_E], _W1r, _W2r, [], []) :-
        find_next_to (_Cutting, _E, _A, _B),
        place_first (_B, _Cutting, _Cr),
        retract_element (_E, _Cr, _W1r),
        reverse (_Cr, _Rev),
        retract_element (_E, _Rev, _W2r).

{------------------------------------------------------------------}
{place_first (_a, _L, _newL) reverses the given list _L until      }
{_a is the first element of the list.                              }
{------------------------------------------------------------------}

place_first (_a, [_x|_Q], _newL) :-
        _a \== _x,
        append (_Q, [_x], _Qx),
        place_first (_a, _Qx, _newL).
place_first (_a, [_a|_Q], [_a|_Q]).

{------------------------------------------------------------------}
{find_next_to (_L, _e, _a, _b) finds the two elements nest to _E}
{in the list _L.                                                   }
{------------------------------------------------------------------}

find_next_to ([_a, _e, _b|_Q], _e, _a, _b).

find_next_to ([_x, _y|_Q], _e, _a, _b) :-
        _y \== _e,
        append ([_y|_Q], [_x], _Qx),
        find_next_to (_Qx, _e, _a, _b).

{------------------------------------------------------------------}
{This predicate suppresses the undesirable walls in the computing  }
{of length. (generaly the latest walls of the corridor)            }
{------------------------------------------------------------------}

suppress_undesirable_walls (_Walls_initial, _Walls_result) :-
        reverse (_Walls_initial, _Walls),
        suppress (_Walls, _Walls_result), !.
suppress_undesirable_walls (_Walls, _Walls).

suppress ([[_w1|_W1], [_ex|_Ex], [_w2|_W2]|_Q], _Q) :-
        send (dt, _a1 = _w1#angle),
        send (dt, _ae = _ex#angle),
        send (dt, _a2 = _w2#angle),
        _test is 180 * sign (_a1 - _ae),
        _dif is (_a1 - _a2),
        ((_dif == 0); (_dif == _test)), !.
suppress ([_W1|_Q], _Result) :-
        suppress (_Q, _Result).
```

```
{-------------------------------------------------------------------}
{     LOCATION OF CABLES MODULE.                                    }
{-------------------------------------------------------------------}

{===================================}
{AUTHOR : Christine JOUVE  EMSE     }
{CREATION DATE : Febrary 91         }
{LAST UPDATE : 8 March 91           }
{===================================}


{-------------------------------------------------------------------------}
{According to previous computed values on cable type, cable structure,    }
{and box type, it gives the number of needed branches, their length and   }
{finds their location (a shaft or a corridor). For each branch, its       }
{connected_elements property is filled in by one of the following forms   }
{which specify which elemenbts have to be connected to this cable : (1) a}
{terminating element, (2) a machine or a box or a plug via perhaps a      }
{connecting element.                                                      }
{ [@terminating_element, [@connecting_element,                            }
{                      _list_of_machines_or_boxes_or_plugs, _cable] ]     }
{ [[@connecting_element, _list_of_machines_or_boxes_or_plugs, _cable],    }
{   [@connecting_element, _list_of_machines_or_boxes_or_plugs, _cable] ]  }
{ [[@connecting_element, _list_of_machines_or_boxes_or_plugs, _cable],    }
{                                          @terminating_element]          }
{-------------------------------------------------------------------------}


cables_location (_floor_network, _vertical_backbone, _cables) :-
        existspropvalue (dt, _floor_network, set_in_floor, _floor),
        send (dt, _Lfalse_grounds = (all _x where
                _x is_in False_grounds and
                [_r|_R] = _x#set_in_rooms and
                _floor = _r#set_in_floor)),
        rerecord (yet_used, []),
        rerecord(floor_cables,[]),
        rerecord (branches_number, 1),
        existspropvalue (dt, _vertical_backbone, partof, _network),
        select_box_type_for_floor (_floor_network, _box_type),
        find_box (_floor, _box_type, _L_box),
        consider_false_ground (_network, _floor_network, _vertical_backbone,
                                              _L_box, _Lfalse_grounds),
        recorded (branches_number, _n),
        putpropvalue (dt,_floor_network, branches_number, _n),
        ((_n = 2),
         recorded (box_location, _box_location),
         _L_box == [_box],
         modslotdef (dt, _box, set_in_rooms, _box_location)
        ; true),
        recorded(floor_cables,_cables),
        erase(floor_cables),
        erase (yet_used),
        erase (branches_number),
        erase (length),
        erase (box_location),
        check_set_in_rooms_for_graphics (_cables).

find_box (_, [], []) :- !.
find_box (_floor, _box_type, [_box]) :-
        send (dt, _box = (_b where _b is_in _box_type and
                _r = oneof _b#set_in_rooms and
                _floor = _r#set_in_floor)).

{-------------------------------------------------------------------}
{For the considered floor, if we have false_grounds, we use       }
{them for cable crossing else we use others shats or corridors. }
{-------------------------------------------------------------------}

consider_false_ground (_network, _floor_network, _vertical_backbone, _L_box, []) :-
        select_cable_type_for_floor (_floor_network, _cable_type),
        existspropvalue (dt, _floor_network, set_in_floor, _floor),
        existspropvalue (dt, _floor, set_in_building, _b),
        find_departure_room (_floor_network, _vertical_backbone,
                                                    _departure_room),

        existspropvalue (dt, _floor_network, set_in_departments, _Ldepartments),
        list_of_rooms_to_be_cabled (_network, _floor, _Ldepartments,
                                                    _Lrooms_to_be_cabled),
        select_cable_location (_b, _departure_room, _cable_type, [], _location),
        existspropvalue (dt, _cable_type, min_max_length, _min_max_length),
        reorder_list (_departure_room, _Lrooms_to_be_cabled,
                                                    _Lrooms_to_be_cabled2),
        select_cable_structure_for_floor(_floor_network, _structure),
        set_cable_location (_structure, _network, _b, _Lrooms_to_be_cabled2,
                _Lrooms_to_be_cabled,
                _cable_type, _location, _min_max_length, _L_box).

consider_false_ground (_network, _floor_network, _vertical_backbone,
                        _L_box, [_false_ground|_Lfalse_ground]) :-
        putpropvalue (dt, _floor_network, branches_number, 1),
        select_cable_type_for_floor (_floor_network, _cable_type),
        putobject (dt, _cable, IN, _cable_type),
        record_push (floor_cables, _cable),
        putpropvalue (dt, _cable, partof, _network),
        putpropvalue (dt, _cable, pass_in, [_false_ground|_Lfalse_ground]),
        send (dt, _set_in_rooms = (all _r where
                _fg = oneof [_false_ground|_Lfalse_ground] and
                _r = oneof _fg#set_in_rooms)),
        putpropvalue (dt, _cable, set_in_rooms, _set_in_rooms),
        existspropvalue (dt, _floor_network, distance_to_be_cabled, _length),
        putpropvalue (dt, _cable, length, _length).

{-------------------------------------------------------------------}
{select_cable_location allows to make the choose between the possible }
{locations of the cable according to preference factor based on       }
{cable type, building type, ...                                       }
{-------------------------------------------------------------------}

{-------------------------------------------------------------------}
{Preference factor is 4 for a shaft if the cable type isn't thick }
{and building type isn't old                                      }
{-------------------------------------------------------------------}
select_cable_location (_b, _room, _cable_type, _yet_used, _shaft) :-
        (_cable_type \== Thick_cables),
        existspropvalue (dt, _b, building_type, _building_type),
        (_building_type \== @old),
        send (dt, [_shaft|_Lshaft] = (all _shaft where
                _shaft is_in Horizontal_shafts and
                _room = oneof _shaft#set_in_rooms)),
        ([_shaft|_Lshaft] \== []),
        yet_used ([_shaft|_Lshaft], _yet_used, _shaft), !.
{-------------------------------------------}
{Preference factor is 3 for a false ceiling. }
{-------------------------------------------}
select_cable_location (_b, _room, _cable_type, _yet_used, _false_ceiling) :-
        send (dt, _Lfalse_ceiling = (all _fc where
                _fc is_in False_ceilings and
```

```
            _room = oneof _fc#set_in_rooms)),
        (_Lfalse_ceiling \== []),
        yet_used ([_false_ceiling|_Lfalse_ceiling], _yet_used, _false_ceiling), !.
{-------------------------------------------------------------------}
{Preference factor is 2 for a shaft if the cable type isn't thick. }
{-------------------------------------------------------------------}
select_cable_location (_b, _room, _cable_type, _yet_used, _shaft) :-
        (_cable_type \== Thick_cables),
        send (dt, [_shaft|_Lshaft] = (all _shaft where
                _shaft is_in Horizontal_shafts and
                _room = oneof _shaft#set_in_rooms)),
        ([_shaft|_Lshaft] \== []),
        yet_used ([_shaft|_Lshaft], _yet_used, _shaft), !.
{--------------------------------------}
{Preference factor is 1 for a corridor. }
{--------------------------------------}
select_cable_location (_b, _corridor, _cable_type, _yet_used, _corridor) :-
        getobject (dt, _corridor, IN, Corridors), !.
select_cable_location (_b, _room, _cable_type, _yet_used, _corridor) :-
        neighbour_rooms (_room, _Lnr),
        send (dt, [_corridor|_Lcorridors] = (all _c where _c = oneof _Lnr and
                _c is_in Corridors)),
        ([_corridor|_Lcorridors] \== []),
        yet_used ([_corridor|_Lcorridors], _yet_used, _corridor).

{-------------------------------------------------------------------------}
{set_cable_location, for a given location, creates the first cables and   }
{puts values of their partof, pass_in or pass_along depending on location }
{type (Horizontal_shaft, False_ceiling, Corridors), connected_elements    }
{properties.                                                              }
{-------------------------------------------------------------------------}
set_cable_location (Star, _network, _b, [_departure_room|_initial_Lr],
                _Lrooms_to_be_cabled,
                _cable_type, _way, _min_max_length, _L_box) :-
        !,
        rerecord (box_location, [_departure_room]),
        find_list_rooms (_way, _Lr_way),
        (getobject (dt, _way, in, Horizontal_shafts),!,
         existspropvalue (dt, _way, set_on_walls, _W);
         existspropvalue (dt, _way, walls_succession, _W)),
        putobject (dt, _cable1, IN, _cable_type),
        record_push (floor_cables, _cable1),
        rerecord (cable1, _cable1),
        putpropvalue (dt, _cable1, partof, _network),
        aplat (_W, _Wa),
        send (dt, [_wall|_L_walls] = (all _w where _w = oneof _Wa and
                _w = oneof _departure_room#walls_succession)),
        putpropvalue (dt, _cable1, pass_along, [_wall]),
        putlocationvalue (_cable1, _way),
        getfacetvalue (dt, _cable_type, min_inter_transceivers_length, default,
                                                                _min_l),
        putpropvalue (dt, _cable1, set_in_rooms, [_departure_room]),
        putpropvalue (dt, _cable1, length, _min_l),
        putobject (dt, _cable2, IN, _cable_type),
        putpropvalue (dt, _cable1, connected_elements, [@terminating_element,
                [@connecting_element, [], _cable2]]),
        record_push (floor_cables, _cable2),
        rerecord (cable2, _cable2),
        rerecord (L_mach, []),
        rerecord (last_wall, _wall),
        putpropvalue (dt, _cable2, partof, _network),
        putlocationvalue (_cable2, _way),
        rerecord (length, _min_l),
        put_cable (_network, _way, _cable_type, [_departure_room], _L_box,
                                                                _min_l, [_wall]),
        reorder_list (_departure_room, _Lr_way, _Lr),
        (member (_departure_room, _Lrooms_to_be_cabled), !,
         find_room_to_connect (_network, [_departure_room|_initial_Lr], _Lr,
                        [], _Lr_taken, 0, 0, _W, _cable_type,
                        _way, _departure_room, _last_room2, _last_length,
                        _min_max_length, []);
         find_room_to_connect (_network, _initial_Lr, _Lr,
                        [], _Lr_taken, 0, 0, _W, _cable_type,
                        _way, _departure_room, _last_room2, _last_length,
                        _min_max_length, []) ),
        del_list (_Lr, _Lr_taken, _Lr2),
        del_list ([_departure_room|_initial_Lr], _Lr_taken, _final_Lr),
        put_cable_location (Star, _network, _final_Lr, _Lr2, _cable_type,
           _way, _last_length, _last_room2, _min_max_length, _L_box).

set_cable_location (_structure, _network, _b, [_departure_room|_initial_Lr],
                _Lrooms_to_be_cabled,
                _cable_type, _way, _min_max_length, _L_box) :-
        find_list_rooms (_way, _Lr_way),
        (getobject (dt, _way, in, Horizontal_shafts),!,
         existspropvalue (dt, _way, set_on_walls, _W);
         existspropvalue (dt, _way, walls_succession, _W)),
        putobject (dt, _cable1, IN, _cable_type),
        record_push (floor_cables, _cable1),
        rerecord (cable1, _cable1),
        putpropvalue (dt, _cable1, partof, _network),
        aplat (_W, _Wa),
        send (dt, [_wall|_L_walls] = (all _w where _w = oneof _Wa and
                _w = oneof _departure_room#walls_succession)),
        putpropvalue (dt, _cable1, pass_along, [_wall]),
        putlocationvalue (_cable1, _way),
        getfacetvalue (dt, _cable_type, min_inter_transceivers_length, default,
                                                                _min_l),
        putpropvalue (dt, _cable1, set_in_rooms, [_departure_room]),
        putpropvalue (dt, _cable1, length, _min_l),
        rerecord (length, _min_l),
        putobject (dt, _cable2, IN, _cable_type),
        putpropvalue (dt, _cable1, connected_elements, [@terminating_element,
                [@connecting_element, [], _cable2]]),
        record_push (floor_cables, _cable2),
        rerecord (cable2, _cable2),
        rerecord (L_mach, []),
        rerecord (last_wall, _wall),

        putpropvalue (dt, _cable2, partof, _network),
        putlocationvalue (_cable2, _way),
        reorder_list (_departure_room, _Lr_way, _Lr),
        (member (_departure_room, _Lrooms_to_be_cabled), !,
         find_room_to_connect (_network, [_departure_room|_initial_Lr], _Lr,
                        [], _Lr_taken, 0, 0, _W, _cable_type,
                        _way, _departure_room, _last_room2, _last_length,
                        _min_max_length, []);
         find_room_to_connect (_network, _initial_Lr, _Lr,
                        [], _Lr_taken, 0, 0, _W, _cable_type,
                        _way, _departure_room, _last_room2, _last_length,
                        _min_max_length, []) ),
        del_list (_Lr, _Lr_taken, _Lr2),
        del_list ([_departure_room|_initial_Lr], _Lr_taken, _final_Lr),
        put_cable_location (_structure, _network, _final_Lr, _Lr2, _cable_type,
           _way, _last_length, _last_room2, _min_max_length, _L_box).

{-------------------------------------------------------------------------}
{put_cable_location, for a given location, creates the others cables and  }
```

```
{puts values of their partof, pass_in or pass_along depending on location }
{type (Horizontal_shaft, False_ceiling, Corridors), connected_elements    }
{properties.                                                              }
{-------------------------------------------------------------------------}

put_cable_location (_structure, _network, [_r|_initial_Lr], [_rway| _Lr_way],
                _cable_type,  _way, _last_length, _last_room,
                                        _min_max_length, _L_box) :-
        (getobject (dt, _way, in, Horizontal_shafts),!,
         existspropvalue (dt, _way, set_on_walls, _W);
         existspropvalue (dt, _way, walls_succession, _W)),
         find_room_to_connect (_network, [_r|_initial_Lr], [_rway| _Lr_way],
                [], _Lr_taken, 0,
                _last_length, _W, _cable_type, _way, _last_room, _last_room2,
                _last_length2, _min_max_length, []),
        !,
        del_list ([_rway| _Lr_way], _Lr_taken, _Lr_way2),
        del_list ([_r|_initial_Lr], _Lr_taken, _final_Lr),
        put_cable_location (_structure, _network, _final_Lr, _Lr_way2,
                _cable_type, _way, _last_length2, _last_room2,
                                        _min_max_length, _L_box).

put_cable_location (_structure, _network, [_r|_initial_Lr], [_rway| _Lr_way],
                _cable_type,  _way, _last_length, _last_room,
                                        _min_max_length, _L_box) :-
        terminate_cable (_structure, _network, _min_max_length, [_r|_initial_Lr],
                [_rway| _Lr_way], _cable_type, _way, _last_length,
                _last_room, _L_box).

{when all rooms along _way were used, it finds another way.}
put_cable_location (_structure, _network, [_r|_Q], [], _cable_type, _way,
           _last_length, _last_room, _min_max_length, _L_box) :-
        record_push (yet_used, _way),
        send  (dt, _b = (_b1 where _floor = _r#set_in_floor and
                _b1 = _floor#set_in_building)),
        recorded (yet_used, _yet_used),
        select_cable_location (_b, _last_room, _cable_type, _yet_used, _answer),
        find_list_rooms (_answer, _Lr_answer),
        reorder_list (_last_room, _Lr_answer, _Lr),
        put_cable_location (_structure, _network, [_r|_Q], _Lr, _cable_type,
                _answer, _last_length, _last_room, _min_max_length, _L_box).

{end test : no more room to be cabled.}
put_cable_location (_structure, _network, [], _Lr_way, _cable_type, _way,
           _last_length, _last_room, _, _L_box) :-
        recorded (branches_number, _n),
        check_structure_branche_number (_structure, _n, _network, _way,
                _cable_type, _last_room, _L_box),
        recorded (cable1, _cable1),
        recorded (cable2, _cable2),
        recorded (L_mach, _L_mach),
        recorded (last_wall, _wall),
        putpropvalue (dt, _cable2, set_in_rooms, [_last_room]),
        putpropvalue (dt, _cable2, length, _last_length),
        putpropvalue (dt, _cable2, connected_elements, [
                [@connecting_element, _L_mach, _cable1],
                @terminating_element]),
        putpropvalue (dt, _cable2, pass_along, [_wall]).

{-------------------------------------------------------------}
terminate_cable (Rep, _network, _min_max_length, _final_Lr, _Lr_way,
          _cable_type, _way, _last_length, _last_room, _L_box) :-
        !,
        rerecord (length, 0),
        recorded (branches_number, 1),
        rerecord (box_location, [_last_room]),
        put_terminating_cable (Rep, _network, _way, _cable_type, [_last_room],
                        _L_box, [_last_room]),
        rerecord (branches_number, 2),
        put_cable_location (Rep, _network, _final_Lr, _Lr_way,
                _cable_type, _way, _last_length, _last_room,
                                        _min_max_length, _L_box).

terminate_cable (Star, _network, _min_max_length, _final_Lr, _Lr_way,
                _cable_type, _way, _last_length, _last_room, _L_box) :-
        rerecord (length, 0),
        recorded (branches_number, _n),
        recorded (box_location, [_departure_room]),
        put_terminating_cable (Star, _network, _way, _cable_type, [_last_room],
                                        _L_box, [_departure_room]),
        (_n1 is _n +1),
        rerecord (branches_number, _n1),
        put_cable_location (Star, _network, _final_Lr, _Lr_way,
                _cable_type, _way, 0, _departure_room,
                                        _min_max_length, _L_box).

{-------------------------------------------------------------}
{yet_used returns an element of _L which has not been used yet. }
{-------------------------------------------------------------}

yet_used (_L, _yet_used, _answer) :-
        del_list (_L, _yet_used, [_answer|_Q]).

{-----------------------------------------------------------------------------}
find_room_to_connect (_network, _L_rooms_to_be_cabled, [_r|_Lr], _Acc,
                _Lr_taken, _add_length,
                _last_length, _L_walls_shaft, _cable_type, _way, _last_room,
                _last_room2, _last_length2, _min_max_length, _L_walls) :-
        connecting_elements (_network, _r, _L_mach_in_room_r,
                                _nb_whised_connection),
        find_room_to_connect (_network, _L_rooms_to_be_cabled, [_r|_Lr], _Acc,
           _Lr_taken, _add_length,
           _last_length, _L_walls_shaft, _cable_type, _way,
           _last_room, _last_room2, _last_length2, _L_mach_in_room_r,
           _nb_whised_connection, _min_max_length, _L_walls).
{-----------------------------------------------------------------------------}
find_room_to_connect (_network, _, [], _Acc, _Acc, _add_length, _last_length,
        _, _, _, _last_room, _last_room, _last_length,  _, _, _, _).

{room which do not need to be connected}

find_room_to_connect (_network, _L_rooms_to_be_cabled, [_r|_Lr], _Acc,
        _Lr_taken, _add_length, _last_length, _L_walls_shaft, _cable_type, _way,
        _last_room, _last_room2, _last_length3, _, _, _min_max_length, _L_walls) :-
        not_member (_r, _L_rooms_to_be_cabled),!,
        aplat (_L_walls_shaft, _Washaft),
        send (dt, _walls = (all _w where _w = oneof _Washaft and
                _w = oneof _r#walls_succession)),
        length_walls (_walls, _length_walls),
        (_last_length2 is _length_walls / 2),
        (_add_length2 is _add_length + _last_length + _last_length2),
        test_length (_add_length2, _cable_type, _min_max_length),
        append (_L_walls, _walls, _L_walls2),
        find_room_to_connect (_network, _L_rooms_to_be_cabled, _Lr, [_r|_Acc],
          _Lr_taken, _add_length2,
          _last_length2, _L_walls_shaft, _cable_type, _way, _last_room,
          _last_room2, _last_length3, _min_max_length, _L_walls2).
```

```
{room with no machine and no whished connection}

find_room_to_connect (_network, _L_rooms_to_be_cabled, [_r|_Lr], _Acc,
        _Lr_taken, _add_length, _last_length, _L_walls_shaft, _cable_type, _way,
        _last_room, _last_room2, _last_length3, [], 0, _min_max_length,
                                                        _L_walls) :-
        aplat (_L_walls_shaft, _Washaft),
        send (dt, _walls = (all _w where _w = oneof _Washaft and
                _w = oneof _r#walls_succession)),
        length_walls (_walls, _length_walls),
        (_last_length2 is _length_walls / 2),
        (_add_length2 is _add_length + _last_length + _last_length2),
        test_length (_add_length2, _cable_type, _min_max_length),
        append (_L_walls, _walls, _L_walls2),
        find_room_to_connect (_network, _L_rooms_to_be_cabled, _Lr, [_r|_Acc],
          _Lr_taken, _add_length2,
          _last_length2, _L_walls_shaft, _cable_type, _way, _last_room,
          _last_room2, _last_length3, _min_max_length, _L_walls2).

{room with one machine or one whised connection}

find_room_to_connect (_network, _L_rooms_to_be_cabled, [_r|_Lr], _Acc,
                [_r|_Acc], _add_length,
                _last_length, _L_walls_shaft, _cable_type, _way, _last_room,
                _r, _last_length2, _L_machines, _n, _min_max_length, _L_walls) :-
        ((_L_machines = [_machine], _n = 0) ; (_L_machines = [], _n = 1)), !,
        aplat (_L_walls_shaft, _Washaft),
        send (dt, _walls = (all _w where _w = oneof _Washaft and
                _w = oneof _r#walls_succession)),
        length_walls (_walls, _length_walls),
        (_last_length2 is _length_walls / 2),
        (_length_cable is _add_length + _last_length + _last_length2),
        reverse ([_r|_Acc], _set_in_rooms),
        test_length (_length_cable, _cable_type, _min_max_length),
        append (_L_walls, _walls, _pass_along),
        put_cable (_network, _way, _cable_type,
                [_last_room|_set_in_rooms], _L_machines, _length_cable,
                                                        _pass_along).

{room with machines}

find_room_to_connect (_network, _L_rooms_to_be_cabled, [_r|_Lr], _Acc,
                [_r|_Acc], _add_length,
                _last_length, _L_walls_shaft, _cable_type, _way, _last_room,
                _r, _length, [_machine1 |_L_machines],
                _nb_whised_connection, _min_max_length, _L_walls) :-
        aplat (_L_walls_shaft, _Washaft),
        send (dt, _walls = (all _w where _w = oneof _Washaft and
                _w = oneof _r#walls_succession)),
        nb_cable_in_room (_walls, [_machine1|_L_machines],
                _nb_whised_connection, _cable_type, _length), !,
        (_length_cable1 is _add_length + _last_length + _length),
        test_length (_length_cable1, _cable_type, _min_max_length),
        reverse ([_r|_Acc], _set_in_rooms),
        append (_L_walls, _walls, _pass_along),
        put_cable (_network, _way, _cable_type, [_last_room|_set_in_rooms],
                [_machine1], _length_cable1, _pass_along),
        create_cable_in_room (_network, _way, _r, _cable_type,
                _L_machines, _nb_whised_connection, _length, _walls).

{no machine and number of whised connection > 1}

find_room_to_connect (_network, _L_rooms_to_be_cabled, [_r|_Lr], _Acc,
                [_r|_Acc], _add_length, _last_length, _L_walls_shaft, _cable_type,
                _way, _last_room, _r, _length , [], _nb_whised_connection,
                _min_max_length, _L_walls) :-
        aplat (_L_walls_shaft, _Washaft),
        send (dt, _walls = (all _w where _w = oneof _Washaft and
                _w = oneof _r#walls_succession)),
        nb_cable_in_room (_walls, [],
                _nb_whised_connection, _cable_type, _length), !,
        (_length_cable1 is _add_length + _last_length + _length),
        reverse ([_r|_Acc], _set_in_rooms),
        test_length (_length_cable1, _cable_type, _min_max_length),
        append (_L_walls, _walls, _pass_along),
        put_cable (_network, _way, _cable_type,
                [_last_room|_set_in_rooms], [], _length_cable1, _pass_along),
        (_nb_whised_connection2 is _nb_whised_connection - 1),
        create_cable_in_room (_network, _way, _r, _cable_type, [],
          _nb_whised_connection2, _length, _walls).

{too small inter_transceivers_length}

find_room_to_connect (_network, _L_rooms_to_be_cabled, [_r|_Lr], _Acc,
                [_r|_Acc], _add_length, _last_length, _L_walls_shaft,
                _cable_type, _way, _last_room, _r, _min_l, _L_machines,
                _nb_connections, _min_max_length, _L_walls) :-
        aplat (_L_walls_shaft, _Washaft),
        send (dt, _walls = (all _w where _w = oneof _Washaft and
                _w = oneof _r#walls_succession)),
        length_walls (_walls, _length_walls),
        list_length (_L_machines, _n_mach),
        reverse ([_r|_Acc], _set_in_rooms),
        append (_L_walls, _walls, _pass_along),
        getfacetvalue (dt, _cable_type, min_inter_transceivers_length, default,
                                                        _min_l),
        (_nb_possible_connections is (_length_walls / _min_l) - 1),
        put_possible_connections (_nb_possible_connections, _length_walls,
                _add_length, _last_length, _cable_type, _min_max_length, _network,
                _way, _last_room, _set_in_rooms, _L_machines, _pass_along, _min_l,
                _n_mach, _r, _walls).

put_possible_connections (_nb_possible_connections, _length_walls,
                _add_length, _last_length, _cable_type, _min_max_length, _network,
                _way, _last_room, _set_in_rooms, _L_machines, _pass_along, _min_l,
                _n_mach, _r, _walls) :-
        (_nb_possible_connections < 1), !,
        (_last_length2 is _length_walls / 2),
        (_length_cable is _add_length + _last_length + _last_length2),
        test_length (_length_cable, _cable_type, _min_max_length),
        put_cable (_network, _way, _cable_type, [_last_room|_set_in_rooms],
                _L_machines, _length_cable, _pass_along).

put_possible_connections (_nb_possible_connections, _length_walls,
                _add_length, _last_length, _cable_type, _min_max_length, _network,
                _way, _last_room, _set_in_rooms, _L_machines, _pass_along, _min_l,
                _n_mach, _r, _walls) :-
         (_length_cable1 is _add_length + _last_length + _min_l),
         test_length (_length_cable1, _cable_type, _min_max_length),
         put_cable (_network, _way, _cable_type, [_last_room|_set_in_rooms],
                        _L_machines, _length_cable1, _pass_along),
         (_nb_empty_connector is _nb_possible_connections - _n_mach),
         create_cable_in_room (_network, _way, _r, _cable_type,
                [], _nb_empty_connector, _min_l, _walls).

{-------------------------------------------------------}
{connecting_elements returns the list of machines which are }
```

```
{in the given room or returns [] if the number of connexions}
{isn't 0.                                                    }
{------------------------------------------------------------}

connecting_elements (_network, _room, _Lmach, _w_c_nb) :-
        send (dt, _Lmach = ( all _mach where _mach is_in Machines and
                _room = oneof _mach#set_in_rooms and
                _mach#partof = _network)),
        room_wished_connexions_number (_network, _room, _w_c_nb).

{------------------------------------------------------------}

nb_cable_in_room (_L_walls, _L_machines, _nb_connections, _cable_type,
                                                        _length) :-
        list_length (_L_machines, _n_mach),
        (_nb_cable is _n_mach + _nb_connections + 1),
        length_walls (_L_walls, _l),
        getfacetvalue (dt, _cable_type, min_inter_transceivers_length, default,
                                                            _min_l),
        (_length is _l / _nb_cable),
        (_length > _min_l).

{------------------------------------------------------------}
create_cable_in_room (_network, _way, _r, _cable_type, [_machine3|_L_machines],
_n, _length, _pass_along) :-
        put_cable (_network, _way, _cable_type, [_r], [_machine3], _length,
                                                        _pass_along),
        create_cable_in_room (_network, _way, _r, _cable_type, _L_machines, _n,
                _length, _pass_along).

create_cable_in_room (_network, _way, _r, _cable_type, [], _n, _length,
                                                        _pass_along) :-
        _n > 0, !,
        put_cable (_network, _way, _cable_type, [_r], [], _length, _pass_along),
        (_n2 is _n - 1),
        create_cable_in_room (_network,_way, _r, _cable_type, [], _n2, _length,
                                                        _pass_along).

create_cable_in_room (_, _, _, _, [], _, _, _).

{------------------------------------------------------------------}
{find_list_rooms gives the list of rooms which are linked by       }
{_answer (an horizontal_shaft or a false_ceiling or a corridor.    }
{------------------------------------------------------------------}

find_list_rooms (_answer, _Lr_answer) :-
        getobject (dt, _answer, IN, Horizontal_shafts), !,
        existspropvalue (dt, _answer, set_in_rooms, _Lr_answer).
find_list_rooms (_answer, _Lr_answer) :-
        getobject (dt, _answer, IN, False_ceilings),
        existspropvalue (dt, _answer, set_in_rooms, [_corridor]),
        getobject (dt, _corridor, Corridors), !,
        neighour_rooms (_corridor,_Lr_answer).
find_list_rooms (_answer, _Lr_answer) :-
        getobject (dt, _answer, IN, False_ceilings), !,
        existspropvalue (dt, _answer, set_in_rooms, _Lr_answer).
find_list_rooms (_answer, _Lr_answer) :-
        getobject (dt, _answer, IN, Corridors),
        neighour_rooms (_answer, _Lr_answer).

{---------------------------------------------}
{put_cable puts all value required for a cable. }
{---------------------------------------------}

put_cable (_network, _way, _cable_type, _Lrooms, _L_mach2, _length, _pass_along) :-
        recorded (cable1, _cable1),
        recorded (cable2, _cable2),
        recorded (L_mach, _L_mach1),
        putobject (dt, _cable3, IN, _cable_type),
        rerecord (cable1, _cable2),
        rerecord (cable2, _cable3),
        rerecord (L_mach, _L_mach2),
        record_push (floor_cables, _cable3),
        putpropvalue (dt, _cable3, partof, _network),
        putlocationvalue (_cable3, _way),
        no_double (_Lrooms, _Lrooms2),
        putpropvalue (dt, _cable2, set_in_rooms, _Lrooms2),
        recorded (last_wall, _last_wall),
        (not_member (_last_wall, _pass_along),!,
         putpropvalue (dt, _cable2, pass_along, [_last_wall|_pass_along]) ;
         putpropvalue (dt, _cable2, pass_along, _pass_along)),
        (last (_last, _pass_along),
        rerecord (last_wall, _last) ; true),
        putpropvalue (dt, _cable2, length, _length),
        recorded (length, _l),
        (_cable_type == Thin_cables, !,
         default_cable_length_on_plug (@Thin, _connex_length),
         (_l2 is _l + _length + _connex_length)
        ;
         (_l2 is _l + _length)
        ),
        rerecord (length, _l2),
        putpropvalue (dt, _cable2, connected_elements, [
                [@connecting_element, _L_mach1, _cable1],
                [@connecting_element, _L_mach2, _cable3]]).

{------------------------------------------------------------}
{test if the length of the segment of cables isn't too long. }
{------------------------------------------------------------}

test_length (_length_cable, _cable_type, _min_max_length) :-
        recorded (length, _l),
        (_cable_type == Thin_cables, !,
         default_cable_length_on_plug (@Thin, _connex_length),
         (_l2 is _l + _length_cable + _connex_length)
        ;
         (_l2 is _l + _length_cable)
        ),
        (_l2 < _min_max_length).

{------------------------------------------------------------}
{put_terminating_cable puts all value required for a cable. }
{------------------------------------------------------------}
put_terminating_cable (Star, _network, _way, _cable_type, _Lrooms1, _L_box,
                                                        _Lrooms2) :-
        getfacetvalue (dt, _cable_type, min_inter_transceivers_length, default,
                                                            _min_l),
        recorded (cable1, _cable1),
        recorded (cable2, _cable2),
        recorded (L_mach, _L_mach),
        no_double (_Lrooms1, _Lrooms12),
        recorded (last_wall , _wall),
        putpropvalue (dt, _cable2, set_in_rooms, _Lrooms12),
        putpropvalue (dt, _cable2, length, _min_l),
        putpropvalue (dt, _cable2, pass_along, [_wall]),
        putpropvalue (dt, _cable2, connected_elements, [
                [@connecting_element, _L_mach, _cable1],
                @terminating_element]),
```

```
        putobject (dt, _cable3, IN, _cable_type),
        putpropvalue (dt, _cable3, partof, _network),
        putlocationvalue (_cable3, _way),
        putobject (dt, _cable4, IN, _cable_type),
        putpropvalue (dt, _cable4, partof, _network),
        putlocationvalue (_cable4, _way),
        rerecord (cable1, _cable3),
        rerecord (cable2, _cable4),
        rerecord (L_mach, _L_box),
        record_push (floor_cables, _cable3),
        record_push (floor_cables, _cable4),
        putpropvalue (dt, _cable3, set_in_rooms, _Lrooms2),
        putpropvalue (dt, _cable3, length, _min_l),
        putpropvalue (dt, _cable3,  pass_along, [_wall]),
        putpropvalue (dt, _cable3, connected_elements,
                [@terminating_element,
                [@connecting_element, _L_box, _cable4]]),
        (_cable_type == Thin_cables, !,
         default_cable_length_on_plug (@Thin, _connex_length),
         (_l2 is _min_l + _connex_length)
        ;
         (_l2 is _min_l)
        ),
        rerecord (length , _l2).

put_terminating_cable (_, _network, _way, _cable_type, _Lrooms1, _L_box,
                                                        _Lrooms2) :-
        getfacetvalue (dt, _cable_type, min_inter_transceivers_length, default,
                                                                _min_l),
        put_cable (_network, _way, _cable_type, _Lrooms1, _L_box, _min_l, []),
        recorded (cable1, _cable1),
        recorded (cable2, _cable2),
        no_double (_Lrooms1, _Lrooms12),
        putpropvalue (dt, _cable2, set_in_rooms, _Lrooms12),
        putpropvalue (dt, _cable2, length, _min_l),
        putpropvalue (dt, _cable2, connected_elements, [
                [@connecting_element, _L_box, _cable1],
                @terminating_element]),
        recorded (last_wall , _wall),
        putpropvalue (dt, _cable2, pass_along, [_wall]),
        putobject (dt, _cable3, IN, _cable_type),
        putpropvalue (dt, _cable3, partof, _network),
        putlocationvalue (_cable3, _way),
        putobject (dt, _cable4, IN, _cable_type),
        putpropvalue (dt, _cable4, partof, _network),
        putlocationvalue (_cable4, _way),
        rerecord (cable1, _cable3),
        rerecord (cable2, _cable4),
        rerecord (L_mach, _L_box),
        record_push (floor_cables, _cable3),
        record_push (floor_cables, _cable4),
        putpropvalue (dt, _cable3, set_in_rooms, _Lrooms2),
        putpropvalue (dt, _cable3, length, _min_l),
        putpropvalue (dt, _cable3, connected_elements,
                [@terminating_element,
                [@connecting_element, _L_box, _cable4]]),
        putpropvalue (dt, _cable3, pass_along, [_wall]),
        (_cable_type == Thin_cables, !,
         default_cable_length_on_plug (@Thin, _connex_length),
         (_l2 is _min_l + _connex_length)
        ;
         (_l2 is _min_l)
        ),
        rerecord (length , _l2).
```

```
{------------------------------------------------------------}
{putlocationvalue puts for the cable the value of its location }
{in the appropriate slot.                                      }
{------------------------------------------------------------}

putlocationvalue (_cable, _loc) :-
        getobject (dt, _loc, IN, Corridors), !,
        existspropvalue (dt, _cable, set_in_rooms_for_graphics, _graphics),
        modpropvalue (dt, _cable, set_in_rooms_for_graphics, [_loc|_graphics]).

putlocationvalue (_cable, _loc) :-
        existspropvalue (dt, _cable, pass_in, _pass_in),
        modpropvalue (dt, _cable, pass_in, [_loc|_pass_in]).

{------------------------------------------------------------}

check_structure_branche_number (Linear, 1, _, _, _, _, _) :- !.
check_structure_branche_number (Rep, 2, _, _, _, _, _) :- !.
check_structure_branche_number (Rep, 1, _network, _way,
                _cable_type, _last_room, _L_box) :-
        rerecord (box_location, [_last_room]),
        getfacetvalue (dt, _cable_type, min_inter_transceivers_length, default,
                                                                _min_l),
        put_terminating_cable (Rep, _network, _way, _cable_type, [_last_room],
                        _L_box, [_last_room]),
        rerecord (branches_number, 2).
check_structure_branche_number (Star, _n, _, _, _, _, _) :-
        (_n > 2), !.

check_structure_branche_number (Star, _n, _network, _way,
                _cable_type, _last_room, _L_box) :-
        getfacetvalue (dt, _cable_type, min_inter_transceivers_length, default,
                                                                _min_l),
        put_terminating_cable (Star, _network, _way, _cable_type, [_last_room], _L_box,
                                                [_last_room]),
        rerecord (branches_number, 3).

{------------------------------------------------------------}

check_set_in_rooms_for_graphics([]).
check_set_in_rooms_for_graphics ([_cable|_cables]) :-
        existspropvalue (dt, _cable, pass_along, _Walls),
        existspropvalue (dt, _cable, pass_in, _pass_in),
        existspropvalue (dt, _cable, set_in_rooms_for_graphics, _graphics),
        reorder_set_in_rooms_for_graphics (_cable, _Walls, _graphics, _pass_in, [], _list
),
        modpropvalue (dt, _cable, set_in_rooms_for_graphics, _list),
        check_set_in_rooms_for_graphics (_cables).

reorder_set_in_rooms_for_graphics (_, [], _, _, _acc, _list) :-
        reverse (_acc, _list).

reorder_set_in_rooms_for_graphics (_c, [_w|_Walls], _graphics, _pass_in, _acc,
                                                                _list) :-
        send (dt, _room = (_r where _shaft = oneof _pass_in and
                                _r = oneof _shaft#set_in_rooms and
                                _w = oneof _r#walls_succession)), !,
        reorder_set_in_rooms_for_graphics (_c, _Walls, _graphics, _pass_in,
                                                [_room|_acc], _list).

reorder_set_in_rooms_for_graphics (_c, [_w|_Walls], _graphics, _pass_in, _acc,
                                                                _list) :-
        send (dt, _room = (_r where _r = oneof _graphics and
                                _w  = oneof (oneof _r#walls_succession))),
```

```
        reorder_set_in_rooms_for_graphics (_c, _Walls, _graphics, _pass_in,
                                                [_room|_acc], _list).

reorder_set_in_rooms_for_graphics (_c, [_w|_Walls], _graphics, _pass_in, _acc,
                                                        _list) :-
        existspropvalue (dt, _c, set_in_rooms, _set_in_rooms),
        send (dt, _room = (_r where _r = oneof _set_in_rooms and
                          _w  = oneof_r#walls_succession)),
        reorder_set_in_rooms_for_graphics (_c, _Walls, _graphics, _pass_in,
                                                [_room|_acc], _list).
```

```
{-------------------------------------------------------------}
{     LOCATION OF BOXES MODULE.                               }
{-------------------------------------------------------------}

{=================================}
{AUTHOR : Christine JOUVE  EMSE   }
{CREATION DATE :  Febrary 91      }
{LAST UPDATE : 8 April 91         }
{=================================}

{-------------------------------------------------------------}
{This module creates an object of type : box_type defined in  }
{every floor_network and it looks for a location (a room place)}
{for these boxes. If the structure is a REP one then the location}
{is calculated during activation of box_location and put in the}
{location_for_boxes property of Floor_networks else we put the boxe}
{in the departure_room of the subnetwork at this floor.       }
{-------------------------------------------------------------}

box_location (_floor_network, _vertical_backbone, _box) :-
        select_box_type_for_floor (_floor_network, _box_type),
        (_box_type \== nil), !,
        putobject (dt, _box, IN, _box_type),
        existspropvalue (dt, _floor_network, set_in_floor, _floor),
        existspropvalue (dt, _floor_network, location_for_boxes, _Loc),
        find_a_location (_Loc, _floor_network, _vertical_backbone, _location),
        putpropvalue (dt, _box, set_in_rooms, [_location]),
        existspropvalue (dt, _vertical_backbone, partof, _network),
        putpropvalue (dt, _box, partof, _network).

box_location (_floor_network, _vertical_backbone, []).

find_a_location ([_loc], _, _, _loc).
find_a_location ([], _floor_network, _vertical_backbone, _loc) :-
        find_departure_room (_floor_network, _vertical_backbone, _loc).
```

```
{------------------------------------------------------------}
{     VERTICAL DESIGN MODULES.                               }
{------------------------------------------------------------}

{==================================}
{AUTHOR : Christine JOUVE  EMSE     }
{CREATION DATE :  April 91          }
{LAST UPDATE : 31 April 91          }
{==================================}

{-----------------------------------------}
{  CALCULATOR OF DISTANCE IN VERTICAL WAY. }
{-----------------------------------------}

compute_vertical_distance (_vertical_backbone, 0) :-
        existspropvalue (dt, _vertical_backbone, vertical, []), !.

compute_vertical_distance (_vertical_backbone, _distance) :-
        existspropvalue (dt, _vertical_backbone, floor_departments,
                                                 _floor_departments),
        extract_floors (_floor_departments, _Lfloors),
        existspropvalue (dt, _vertical_backbone, vertical, [_vert]),
        existspropvalue (dt, _vert, joined_floors, _joined_floors),
        restrict_list_floors (_Lfloors, _joined_floors, _Lfls),
        compute_height (_Lfls, _distance).

restrict_list_floors (_Lfloors, _joined_floors, [_fl1|_Lfls]) :-
        sub_list ([_fl1|_Lfls], _joined_floors),
        inclu (_Lfloors, [_fl1|_Lfls]),
        reverse ([_fl1|_Lfls], [_last_el|_]),
        member (_fl1, _Lfloors),
        member (_last_el, _Lfloors).

compute_height ([], 0).
compute_height ([_floor|_Lfloors], _height) :-
        existspropvalue (dt, _floor, height, _height1),
        compute_height (_Lfloors, _height2),
        (_height is _height1 + _height2).

{-----------------------------------------}
{   LOCALIZATION OF VERTICAL CABLES.      }
{-----------------------------------------}

vertical_cables_location (_vertical_backbone, _, _) :-
        existspropvalue (dt, _vertical_backbone, vertical, []), !.

vertical_cables_location (_vertical_backbone, _cables, _boxes) :-
        select_cable_type_for_vertical (_vertical_backbone, _cable_type),
        existspropvalue (dt, _vertical_backbone, floor_departments,
                                                 _floor_departments),
        extract_floors (_floor_departments, _Lfloors),
        existspropvalue (dt, _vertical_backbone, vertical, [_vert]),
        existspropvalue (dt, _vert, joined_floors, _joined_floors),
        existspropvalue (dt, _vert, set_in_rooms, _Lrooms),
        restrict_list_floors (_Lfloors, _joined_floors, [_fl1|_Lfls]),
        rerecord(floor_cables,[]),
        existspropvalue (dt, _vertical_backbone, partof, _network),
        put_first_cable (_network, [_fl1|_Lfls], _Lrooms, _cable_type, _vert,
                                                                  _boxes),
        recorded(floor_cables,_cables),
        putpropvalue (dt, _vertical_backbone, list_of_cables, _cables),
        erase (floor_cables).

put_first_cable (_, [_fl], _, _, _) :- !.

put_first_cable (_network, [_fl1|_Lfls], _Lrooms, _cable_type, _vert, _boxes) :-
        putobject (dt, _cable1, IN, _cable_type),
        record_push (floor_cables, _cable1),
        putpropvalue (dt, _cable1, partof, _network),
        putpropvalue (dt, _cable1, pass_in, [_vert]),
        existspropvalue (dt, _fl1, height, _height1),
        (_length is _height1/2),
        putpropvalue (dt, _cable1, length, _length),
        putobject (dt, _cable2, IN, _cable_type),
        record_push (floor_cables, _cable2),
        putpropvalue (dt, _cable2, partof, _network),
        putpropvalue (dt, _cable2, pass_in, [_vert]),
        send (dt, _box = (_bo where _bo = oneof _boxes and
                _bo#set_in_rooms = [_r1|_] and _r1#set_in_floor = _fl1)),
        send (dt, _room = (_r2 where _r2 = oneof _Lrooms and
                _r2#set_in_floor = _fl1)),
        putpropvalue (dt, _cable1, set_in_rooms, [_room]),
        putpropvalue (dt, _cable2, set_in_rooms, [_room]),
        putpropvalue (dt, _cable1, connected_elements, [@terminating_element,
                [@connecting_element, [_box], _cable2]]),
        put_others_cables (_Lrooms, _cable1, _cable2, _box, _length, _network,
                        _cable_type, _vert, _Lfls, _boxes),
        recorded(floor_cables,_cables).

put_others_cables (_, _cable1, _cable2, _box, _length, _, _, _, [], _) :-
        putpropvalue (dt, _cable2, connected_elements, [
                [@connecting_element, [_box], _cable1],
                @terminating_element]),
        putpropvalue (dt, _cable2, length, _length).

put_others_cables (_Lrooms, _cable1, _cable2, _box1, _last_length, _network,
                _cable_type, _vert, [_fl2|_Lfl], _boxes) :-
        putobject (dt, _cable3, IN, _cable_type),
        record_push (floor_cables, _cable3),
        putpropvalue (dt, _cable3, partof, _network),
        putpropvalue (dt, _cable3, pass_in, [_vert]),
        existspropvalue (dt, _fl2, height, _height2),
        (_last_length2 is _height2/2),
        (_length is _last_length + _last_length2),
        putpropvalue (dt, _cable2, length, _length),
        send (dt, _box2 = (_bo where _bo = oneof _boxes and
                _bo#set_in_rooms = [_r1|_] and _r1#set_in_floor = _fl2)),
        send (dt, _room = (_r2 where _r2 = oneof _Lrooms and
                _r2#set_in_floor = _fl2)),
        addpropvalue (dt, _cable2, set_in_rooms, _room),
        putpropvalue (dt, _cable3, set_in_rooms, [_room]),
        putpropvalue (dt, _cable2, connected_elements, [
                [@connecting_element, [_box1], _cable1],
                [@connecting_element, [_box2], _cable3]]),
        put_others_cables (_Lrooms, _cable2, _cable3, _box2, _length,
                _network, _cable_type, _vert, _Lfl, _boxes).

{-----------------------------------------}
{  MERGING OF VERTICAL BACKBONES.         }
{-----------------------------------------}
{This module computes the link between the different vertical backbones}

{--------------------------------------}
{no merge for only one vertical backbone}
{--------------------------------------}
merge_vertical_backbones ([_vertical_backbone1]) :- !.
{------------------------------------------------------------------------}
{merge for two vertical backbones which goes througth a same way (shaft }
```

```
{or stairs or lift)                                                    }
{----------------------------------------------------------------------}
merge_vertical_backbones ([_vertical_backbone1, _vertical_backbone2]) :-
        existspropvalue (dt, _vertical_backbone1,  vertical, [_vertical]),
        existspropvalue (dt, _vertical_backbone2, vertical, [_vertical]), !,
        existspropvalue (dt, _vertical_backbone1, list_of_cables, _L_cable1),
        existspropvalue (dt, _vertical_backbone2, list_of_cables, _L_cable2),
        existspropvalue (dt, _vertical, joined_floors, [_floor|_]),
        find_terminating_cables_in_floor (_L_cable1, _floor, [_cable1]),
        find_terminating_cables_in_floor (_L_cable2, _floor, [_cable2]),
        getobject (dt, _cable1, DIRECTIN, _cable_type),
        putobject (dt, _cable, IN, _cable_type),
        existspropvalue (dt, _cable1, partof, _network),
        putpropvalue (dt, _cable, partof, _network),
        putpropvalue (dt, _cable, pass_in, [_vertical]),
        existspropvalue (dt, _cable1, length, _length1),
        existspropvalue (dt, _cable2, length, _length2),
        (_length is _length1 + _length2),
        putpropvalue (dt, _cable, length, _length),
        existspropvalue (dt, _cable1, set_in_rooms, _Rooms1),
        existspropvalue (dt, _cable2, set_in_rooms, _Rooms2),
        append (_Rooms1, _Rooms2, _Rooms),
        no_double (_Rooms, _R),
        putpropvalue (dt, _cable, set_in_rooms, _R),
        existspropvalue (dt, _cable1, connected_elements, _connected_elements1),
        find_not_terminator (_cable1, _connected_elements1, _el1),
        existspropvalue (dt, _cable2, connected_elements, _connected_elements2),
        find_not_terminator (_cable2, _connected_elements2, _el2),
        putpropvalue (dt, _cable, connected_elements, [_el1, _el2]),
        subpropvalue (dt, _el1, connected_elements, _cable1),
        addpropvalue (dt, _el1, connected_elements, _cable),
        subpropvalue (dt, _el2, connected_elements, _cable2),
        addpropvalue (dt, _el2, connected_elements, _cable),
        retract_element (_el1, _connected_elements1, [_terminating_el1]),
        delete_terminating_elements (_terminating_el1, [_terminating_el1]),
        retract_element (_el2, _connected_elements2, [_terminating_el2]),
        delete_terminating_elements (_terminating_el2, [_terminating_el2]),
        delobject (dt, _cable1, IN, _cable_type),
        delobject (dt, _cable2, IN, _cable_type).
{------------}
{other cases }
{------------}
merge_vertical_backbones ([_vertical_backbone1|_L_vertical_backabones]) :-
        existspropvalue (dt, _vertical_backbone1,  vertical, [_vertical]),
        existspropvalue (dt, _vertical, joined_floors, [_floor|_]),
        send (dt, [_floor_network|_] = (all _f_n where _f_n is_in Floor_networks
                and _floor = _f_n#set_in_floor)),
        find_departure_room (_floor_network, _vertical_backbone1, _room),
        list_length ([_vertical_backbone1|_L_vertical_backabones], _n),
        existspropvalue (dt, _vertical_backbone1, list_of_cables,
                                        [_cable1|_L_cable1]),
        existspropvalue (dt, _cable1, partof, _network),
        choose_a_box (_n, _room, _network, _box),
        link_vertical_backbone ([_vertical_backbone1|_L_vertical_backabones],
                                        _floor, _room, _box).

{----------------------------------------------------------------------}
{it finds througth the list of cables (_L_cable) the cable which is ended }
{by a terminator and which is put in the floor _floor.                 }
{----------------------------------------------------------------------}

find_terminating_cables_in_floor (_L_cable, _floor, _L_terminating_cables_in_floor) :-
        find_terminating_cables (_L_cable, _L_terminating_cables),
        send (dt, _L_terminating_cables_in_floor = (all _cable where
                _cable = oneof _L_terminating_cables and
                _r = oneof _cable#set_in_rooms and
                _floor = _r#set_in_floor)).

{----------------------------------------------------------------------}
{it finds througth the list of cables (_L_cable) the cable which is ended }
{by a terminator.                                                      }
{----------------------------------------------------------------------}

find_terminating_cables ([_cable|_L_cable], [_cable|_L_terminating_cables]) :-
        existspropvalue (dt, _cable, connected_elements, _connected_elements),
        has_a_terminator (_cable, _connected_elements),
        find_terminating_cables (_L_cable, _L_terminating_cables).

find_terminating_cables ([_cable|_L_cable], _L_terminating_cables) :-
        find_terminating_cables (_L_cable, _L_terminating_cables).
find_terminating_cables ([], []).

{----------------------------------------------------------------------}
{has_a_terminator checks if _el has in its list of connected_elements }
{a element which is eneded by a terminator.                            }
{----------------------------------------------------------------------}

has_a_terminator (_el, [_el1|_Q]) :-
        getobject (dt, _el1, IN, Terminators), !.

has_a_terminator (_el, [_el1|_Q]) :-
        getobject (dt, _el1, IN, Cables), !, fail.

has_a_terminator (_el, [_el1|_Q]) :-
        existspropvalue (dt, _el1, connected_elements, _connected_elements),
        retract_element (_el, _connected_elements, _L),
        (has_a_terminator (_el1, _L), !;
          has_a_terminator(_el, _Q)).

{------------------------------------------------------------}
{find_not_terminator finds in the list of connected_elements of }
{_cable the element which is not ended by a terminator.        }
{------------------------------------------------------------}

find_not_terminator (_cable, [_el|_Q], _not_terminator) :-
        has_a_terminator (_cable, [_el]), !,

        find_not_terminator (_cable, _Q, _not_terminator).
find_not_terminator (_, [_not_terminator|_Q], _not_terminator).

{------------------------------------------------------------}
{it deletes all elements which are connected to _el_d and which are }
{ended by a terminator.                                        }
{------------------------------------------------------------}

delete_terminating_elements (_, []).
delete_terminating_elements (_, [_el|_Q]) :-
        getobject (dt, _el, IN, Terminators), !,
        delobject (dt, _el, IN, _P).

delete_terminating_elements (_el_d, [_el|_Q]) :-
        getobject (dt, _el, IN, Cables), !,
        delete_terminating_elements (_el_d, _Q).

delete_terminating_elements (_el_d, [_el|_Q]) :-
        existspropvalue (dt, _el, connected_elements, _connected_elements),
        retract_element (_el_d, _connected_elements, _new_L),
        delete_terminating_elements (_el, _new_L),
```

```
        delete_terminating_elements (_el_d, _Q),
        delobject (dt, _el, IN, _c).

{------------------------------------------------------------------------}
{it creates a repeater which allows to link the different vertical       }
{backbones. If there are only two vertical backbones it slects a REP     }
{else a MPR.                                                             }
{------------------------------------------------------------------------}

choose_a_box (2, _room, _network, _box) :-
        putobject (dt, _box, IN, REP),
        putpropvalue (dt, _box, partof, _network),
        putpropvalue (dt, _box, set_in_rooms, [_room]).
choose_a_box (_n, _room, _network, _box) :-
        (_n > 2),
        putobject (dt, _box, IN, MPR),
        putpropvalue (dt, _box, partof, _network),
        putpropvalue (dt, _box, set_in_rooms, [_room]).

{------------------------------------------------------------------------}
{it makes the links between the vertical backbones and the choosen box. }
{------------------------------------------------------------------------}

link_vertical_backbone ([], _, _, _).
link_vertical_backbone ([_vertical_backbone1|_L_vertical_backabones], _floor,
                                                        _room, _box) :-
        existspropvalue (dt, _vertical_backbone1, list_of_cables, _L_cable1),
        find_terminating_cables_in_floor (_L_cable1, _floor, [_cable1]),
        existspropvalue (dt, _vertical_backbone1,  vertical, [_vertical]),
        existspropvalue (dt, _cable1, cable_type, _cable_type),
        send (dt, _vertical_room = (_r where _r = oneof _vertical#set_in_rooms and
                _floor = _r#set_in_floor)),
        find_way_vertical_room (_vertical_room, _cable_type, _room, _L_ways),
        existspropvalue (dt, _cable1, set_in_rooms, _set_in_rooms),
        append (_set_in_rooms, _L_ways, _new_set_in_rooms),
        no_double (_new_set_in_rooms, _no_double_set_in_rooms),
        modpropvalue (dt, _cable1, set_in_rooms, _no_double_set_in_rooms),
        compute_length (_vertical_room, _room, _L_ways, _l2),
        existspropvalue (dt, _cable1, length, _l1),
        (_length is _l1 + _l2),
        modpropvalue (dt, _cable1, length, _length),
        existspropvalue (dt, _cable1, connected_elements, _connected_elements1),
        find_not_terminator (_cable1, _connected_elements1, _el1),
        retract_element (_el1, _connected_elements1, [_terminating_el1]),
        subpropvalue (dt, _cable1, connected_elements, _terminating_el1),
        putobject (_dt, _little_cable, IN, _cable_type),
        putlocationvalue (_cable1, _L_ways),
        existspropvalue (dt, _cable1, partof, _network),
        putpropvalue (dt, _little_cable, partof, _network),
        putpropvalue (dt, _little_cable, set_in_rooms, [_room]),
        getfacetvalue (dt, _cable_type, min_inter_transceivers_length, default,
                                                        _min_l),
putpropvalue (dt, _little_cable, length, _min_l),
        putpropvalue (dt, _little_cable, connected_elements, [_terminating_el1]),
        connect_cable_to_boxes (_cable1, _cable_type, [_box], _little_cable,_room),
        link_vertical_backbone (_L_vertical_backabones, _floor, _room, _box).

{------------------------------------------------------------------------}
{it finds the way (shaft, corridors) which the cable linking _vertical_room }
{and _room will pass throught.                                           }
{------------------------------------------------------------------------}

find_way_vertical_room (_vertical_room, _cable_type, _room, [_way|_L_ways]) :-
        existspropvalue (dt, _room, set_in_building, _b),
        select_cable_location (_b, _vertical_room, _cable_type,
                        [], _way),
        find_list_rooms (_way, _Lr_way),
        (member (_room, _Lr_way), !, (_L_ways = []) ;
         last (_Lr_way, _last_room),
         find_way_vertical_room (_b, _cable_type, _last_room, _room, [_way],
                                                        _L_ways)).
find_way_vertical_room (_b, _cable_type, _last_room, _room, _yet_used,
                                                        [_way|_L_ways]) :-
        select_cable_location (_b, _last_room, _cable_type, _yet_used, _way),
        find_list_rooms (_way, _Lr_way),
        (member (_room, _Lr_way), !, (_L_ways = []) ;
         last (_Lr_way, _last_room),
         find_way_vertical_room (_b, _cable_type, _last_room, _room,
                                        [_way|_yet_used], _L_ways)).

{------------------------------------------------------------------------}
{it computes the length of the _way which allows to link _vertical_room  }
{and _room.                                                              }
{------------------------------------------------------------------------}

compute_length (_, _, [], 0).
compute_length (_vertical_room, _room, [_way|_L_ways], _length) :-
        find_list_rooms (_way, _L_r_way),
        reorder_list (_vertical_room, _L_r_way, _ordred_L_r),
        existspropvalue (dt, _way, walls_succession, _W),
        aplat (_W, _Wa),
        compute_length (_ordred_L_r, _room, _Wa, _l1),
        compute_length (_vertical_room, _room, _L_ways, _l2),
        (_length is _l1 + _l2).

compute_length ([], _, _, 0).
compute_length ([_room|_ordred_L_r], _room, _W, _l) :-
        !, send (dt, _walls = (all _w where _w = oneof _room#walls_succession and
                _w = oneof _W)),
        length_walls (_walls, _l1),
        (_l is _l1 / 2).
compute_length ([_r|_ordred_L_r], _room, _W, _l) :-
        send (dt, _walls = (all _w where _w = oneof _room#walls_succession and
                _w = oneof _W)),
        length_walls (_walls, _l1),
        compute_length (_ordred_L_r, _room, _W, _l2),
        (_l is _l1 + _l2).
```

```
{************************************************}
{File with general functions on building components}
{************************************************}

{==================================}
{AUTHORS: Fabienne Balfroid  BIM R&D}
{         Christine Jouve EMSE      }
{CREATION DATE : January 91         }
{LAST UPDATE : 15 February 91       }
{==================================}

{-------------------------------------}
{finds the neighours of a specified room}
{-------------------------------------}

neighour_rooms (_room, _L) :-
        getpropvalue (dt, _room, walls_succession, _Lw),
        aplat (_Lw, _Law),
        getpropvalue (dt, _room, set_in_floor, _floor),
        getpropvalue (dt, _floor, list_of_rooms, _Rooms),
        retract_element (_room,_Rooms, _R),
        record (h, []),
        neighour (_R, _Law),!,
        recorded (h, _L),
        erase (h).
neighour ([], _).
neighour ([_r|_Q], _Lw) :-
        getpropvalue (dt, _r, walls_succession, _Lw2),
        aplat (_Lw2, _Law2),
        intersection (_Lw, _Law2, _Inter),
        test (_Inter, _r),
        neighour (_Q, _Lw).

test ([], _).
test ([_w|_Q], _r) :-
        record_push (h, _r).

{-----------------------------------------}
{finds the neighours of a specified corridor}
{-----------------------------------------}

neighour_corridors (_corridor, _L) :-
        getpropvalue (dt, _corridor, walls_succession, _Lw),
        aplat (_Lw, _Law),
        getpropvalue (dt, _corridor, set_in_floor, _floor),
        getpropvalue (dt, _floor, list_of_corridors, _Lcorridors),
        retract_element (_corridor,_Lcorridors, _Lc),
        record (h, []),
        neighour (_Lc, _Law),
        recorded (h, _L),
        erase (h).

{-----------------------------------------------}
{finds the neighours of a specified list of rooms}
{-----------------------------------------------}

neighour_list_rooms ([_r|_Lr], _L) :-
        neighour_rooms (_r, _Lnr),
        neighour_list_rooms (_Lr, _Lallnr),
        union (_Lnr, _Lallnr, _L).
neighour_list_rooms ([], []).

{-------------------------------------------------------------}
{list_of_rooms_to_be_cabled looks for all rooms which have to be}
{cabled for a given floor and which their set_in_departments    }
{property belongs to the list of departments.                   }
{-------------------------------------------------------------}

list_of_rooms_to_be_cabled (_network, _floor, _L_departments,
                                                  _Lrooms_to_be_cabled) :-
        existspropvalue (dt, _floor, list_of_rooms, _L_all_rooms_in_floor),
        restrict_L_rooms (_network, _L_all_rooms_in_floor, _L_departments,
                                                  _L_rooms),
        rooms_to_be_cabled (_network, _L_rooms, _Lrooms_to_be_cabled).

{-------------------------------------------------------------}
{list_of_rooms_to_be_cabled looks for all rooms which have to be}
{cabled for a given floor.                                      }
{-------------------------------------------------------------}

{list_of_rooms_to_be_cabled  (_network, _floor, _Lrooms_to_be_cabled) :-
        existspropvalue (dt, _floor, list_of_rooms, _L_all_rooms_in_floor),
        rooms_to_be_cabled (_network, _L_all_rooms_in_floor, _Lrooms_to_be_cabled).}

{-------------------------------------------------------------------------}
{restrict_L_rooms (_network, _L_rooms, _L_departments, _restricted_List) : }
{_restricted_List is a sub_list of _L_rooms where all rooms belongs to a   }
{department of _L_departments.                                             }
{-------------------------------------------------------------------------}

restrict_L_rooms (_, _, [], []).
restrict_L_rooms (_network, _L_rooms, [_depart|_L_departments],
                                                  _restricted_List) :-
        existspropvalue (_depart, elements, _Lr_depart),
        intersection (_L_rooms, _Lr_depart, _L1),
        del_list (_L_rooms, _L1, _Lr),
        restrict_L_rooms (_network, _Lr, _L_departments, _L2),
        append (_L1, _L2, _restricted_List).

{-------------------------------------------------------------------}
{a room has to be cabled if the number of connections isn't nil or if  }
{whished connections number isn't nil or some machines are in this room. }
{-------------------------------------------------------------------}

rooms_to_be_cabled (_, [], []).
rooms_to_be_cabled (_network, [_r|_Lr], [_r|_Lrf]) :-
        room_connexions_number(_network,_r,_c_nb),
        (_c_nb >0), !,
        rooms_to_be_cabled (_network, _Lr, _Lrf).
rooms_to_be_cabled (_network, [_r|_Lr], [_r|_Lrf]) :-
        room_wished_connexions_number (_network, _r, _w_c_nb),
        (_w_c_nb >0), !,
        rooms_to_be_cabled (_network, _Lr, _Lrf).
rooms_to_be_cabled (_network, [_r|_Lr], _Lrf) :-
        rooms_to_be_cabled (_network, _Lr, _Lrf).

{-------------------------------------------------------------}
{cutting changes the list of walls of a corridor in a list of  }
{sublists of walls. Every sublist contains walls with same angle.}
{-------------------------------------------------------------}

cutting ([_w1, _w2|_Qw], [[_w1|_Q1]|_Q2]) :-
        getpropvalue (dt, _w1, angle, _a1),
        getpropvalue (dt, _w2, angle, _a2),
        (_a1 == _a2), !,
        cutting ([_w2|_Qw], [_Q1|_Q2]).

cutting ([_w1, _w2|_Qw], [[_w1]|_Q]) :-
```

```
        cutting ([_w2|_Qw],_Q).

cutting ([_w1], [[_w1]]).

{------------------------------------------------------------------}
{longer_length finds the list of walls with the longer length  }
{through a list obtained by the cutting predicate.             }
{------------------------------------------------------------------}

longer_length ([_W|_Q], _Walls, _length) :-
        length_walls (_W, _l),
        longer_length (_Q, _W, _l, _Walls, _length).

longer_length ([_Walls2|_QWalls], _Walls1, _l1, _Walls, _length) :-
        length_walls (_Walls2, _l2),
        ( (_l1 =< _l2, longer_length (_QWalls, _Walls2, _l2 ,_Walls, _length));
          (_l1 > _l2 , longer_length (_QWalls, _Walls1, _l1, _Walls, _length))).
longer_length ([], _W, _l, _W, _l).

{------------------------------------------------------------------}
{smaller_length finds the list of walls with the smaller length }
{through a list obtained by the cutting predicate.              }
{------------------------------------------------------------------}

smaller_length ([_W|_Q], _Walls, _length) :-
        length_walls (_W, _l),
        smaller_length (_Q, _W, _l, _Walls, _length).

smaller_length ([_Walls2|_QWalls], _Walls1, _l1, _Walls, _length) :-
        length_walls (_Walls2, _l2),
        ( (_l1 > _l2, smaller_length (_QWalls, _Walls2, _l2 ,_Walls, _length));
          (_l1 =< _l2 , smaller_length (_QWalls, _Walls1, _l1, _Walls, _length))).
smaller_length ([], _W, _l, _W, _l).

{------------------------------------------------------------------}
{length_walls computes the length of a list of walls.           }
{------------------------------------------------------------------}

length_walls ([], 0).

length_walls ([[_w|_Q] | _W],_length) :-
        !,
        length_walls ([_w|_Q], _l1),
        length_walls (_W, _l2),
        (_length is _l1 +_l2).

length_walls ([_w|_Qw], _l3) :-
        existspropvalue (dt, _w, length, _l1),
        length_walls (_Qw, _l2),
        (_l3 is _l1 + _l2).

{-------------------------------------------------------------------------}
{distance computes the distances between two points of coordinates (x,y) }
{-------------------------------------------------------------------------}

distance ([_x,_y], [_xt,_yt], _d) :-
        _d is sqrt ((_x-_xt)**2 + (_y-_yt)**2).

{------------------------------------------------------------------}
{find_walls_next_to_vertical finds walls which belong to the    }
{given corridor and which are next to the chosen vertical.      }
{------------------------------------------------------------------}

find_walls_next_to_vertical (_Cutting, _vertical, _corri, _Walls) :-
        getobject (dt, _vertical, IN, Vertical_shafts),
        send (dt, _coori = oneof _vertical # set_in_rooms), !,
        existspropvalue (dt, _vertical, coord_x_y, _coord_v),
        existspropvalue (dt, _corri, origin_coordinates, [_coord|_Q]),
        walls_next_to_vertical (_Cutting, _coord, _coord_v, 100000, [], _Walls).

walls_next_to_vertical ([[_w|_Walls]|_Q], [_x1, _y1], _coord_v, _d1, _R1, _R) :-
        length_walls ([_w|_Walls], _l),
        existspropvalue (dt, _w, angle, _a),
        _x2 is _x1 + _l * cos(_a),
        _y2 is _y1 + _l * sin(_a),
        projection (_coord_v, [_x1, _y1], [_x2, _y2], _coord_proj),
        distance (_coord_v, _coord_proj, _d2),
        ( (_d1 > _d2,
              walls_next_to_vertical (_Q, [_x2, _y2], _coord_v, _d2, [[_w|_Walls]], _R)
          );
          (_d1 < _d2),
              walls_next_to_vertical (_Q, [_x2, _y2], _coord_v, _d1, _R1, _R
          );
          (_d1 = _d2,
              walls_next_to_vertical (_Q, [_x2, _y2], _coord_v, _d1, [[_w|_Walls]|_R1],
 _R)
        ) ).
walls_next_to_vertical ([], _, _, _, _R, _R).

projection ([_xV, _yV], [_x1, _y1], [_x2, _y2], [_xP, _yP]) :-
        (_a is _x2 - _x1),
        (_b is _y2 - _y1),
        (_c is _x2*_y1 + _x1*_y2),
        (_d is _xV*_a + _yV*_b),
        (_e is _a**2 - _b**2),
        (_e \==0), !,
        (_xP is (_d*_a + _b*_c) / _e),
        (_yP is  (_b*_d + _a*_c) / (-1)*_e).

projection ([_xV, _yV], [_x1, _y1], [_x2, _y2], [100000, 100000]).

find_walls_next_to_vertical (_Cutting, _vertical, _corri, _Walls) :-
        getobject (dt, _vertical, IN, Vertical_shafts), !,
        existspropvalue (dt, _vertical, set_in_rooms, _Lr),
        send (dt, _room = oneof _Lr and
              _room#set_in_floor = _corri # set_in_floor),
        existspropvalue (dt, _room, walls_succession, _Wroom),
        aplat (_Wroom, _Wa_rooms),
        existspropvalue (dt, _corri, walls_succession, _Wcorri),
        aplat (_Wcorri, _Wa_corri),
        intersection (_Wa_room, _Wa_corri, [_Wall1|_]),
        find_in_cutting (_Wall1, _Cutting, [_w1|_Walls1]),
        send (dt, _room_w1 = oneof _w1#set_in_rooms),
        (_room_w1 \== _corri),
        existspropvalue(dt, _room_w1, walls_succession, _walls_succession1),
        existspropvalue(dt, _room_w1, origin_coordinates, _origin_coordinates1),
        find_begining_coord_wall (_walls_succession1, _origin_coordinates1, _w1,
                                                        _coord_begining),
        last (_last, [_w1|_Walls1]),
        send (dt, _room_l = oneof _last#set_in_rooms),
        (_room_l \== _corri),
        existspropvalue(dt, _room_l, walls_succession, _walls_succession2),
        existspropvalue(dt, _room_l, origin_coordinates, _origin_coordinates2),
        find_end_coord_wall (_walls_succession2, _origin_coordinates2, _last,
                                                          _coord_end),
        reorder_list ([_w1|_Walls1], _Cutting, [[_w1|_Walls1] | _Q]),
        walls_next_to_vertical (_Q, _coord_end, _coord_begining, 100000, [], _Walls2),
        append ([[_w1|_Walls1]], [_Walls2], _Walls).
```

```
find_walls_next_to_vertical (_Cutting, _room, _corri, _Walls) :-
        existspropvalue (dt, _corri, walls_succession, _Wcorri),
        aplat (_Wcorri, _Wa_corri),
        existspropvalue (dt, _room, walls_succession, _Wroom),
        aplat (_Wroom, _Wa_rooms),
        intersection (_Wa_rooms, _Wa_corri, [_Wall1|_]),
        find_in_cutting (_Wall1, _Cutting, [_w1|_Walls1]),
        send (dt, _room_w1 = oneof _w1#set_in_rooms),
        (_room_w1 \== _corri),
        existspropvalue(dt, _room_w1, walls_succession, _walls_succession1),
        existspropvalue(dt, _room_w1, origin_coordinates, _origin_coordinates1),
        find_begining_coord_wall (_walls_succession1, _origin_coordinates1, _w1,
                                                        _coord_begining),
        last (_last, [_w1|_Walls1]),
        send (dt, _room_1 = oneof _last#set_in_rooms),
        (_room_1 \== _corri),
        existspropvalue(dt, _room_1, walls_succession, _walls_succession2),
        existspropvalue(dt, _room_1, origin_coordinates, _origin_coordinates2),
        find_end_coord_wall (_walls_succession2, _origin_coordinates2, _last,
                                                        _coord_end),
        reorder_list ([_w1|_Walls1], _Cutting, [[_w1|_Walls1] | _Q]),
        walls_next_to_vertical (_Q, _coord_end, _coord_begining, 100000, [], _Walls2),
        append ([[_w1|_Walls1]], _Walls2, _Walls).

{-------------------------------------------------}
find_in_cutting (_wall, [_List | _Q], _List) :-
        member (_wall, _List), !.
find_in_cutting (_wall, [_List | _Q], _List2) :-
        find_in_cutting (_wall, _Q, _List2).

{-------------------------------------------------}
find_begining_coord_wall ([_w | _walls_succession], _origin_coordinates, _w,
                                        _origin_coordinates) :- !.
find_begining_coord_wall ([_w1 | _walls_succession], [_x1, _y1], _w,
                                        _begining_coordinates) :-
        existspropvalue (dt, _w1, length, _l),
        existspropvalue (dt, _w1, angle, _a),
        _x2 is _x1 + _l * cos(_a),
        _y2 is _y1 + _l * sin(_a),
        find_begining_coord_wall (_walls_succession, [_x2, _y2], _w,
                                        _begining_coordinates).
{-------------------------------------------------}
find_end_coord_wall ([_w | _walls_succession], [_x1, _y1], _w,
                                        [_x2, _y2]) :-
        existspropvalue (dt, _w, length, _l),
        existspropvalue (dt, _w, angle, _a),
        _x2 is _x1 + _l * cos(_a),
        _y2 is _y1 + _l * sin(_a), !.
find_end_coord_wall ([_w1 | _walls_succession], [_x1, _y1], _w,
                                        _end_coordinates) :-
        existspropvalue (dt, _w1, length, _l),
        existspropvalue (dt, _w1, angle, _a),
        _x2 is _x1 + _l * cos(_a),
        _y2 is _y1 + _l * sin(_a),
        find_end_coord_wall (_walls_succession, [_x2, _y2], _w,
                                        _end_coordinates).

{-------------------------------------------------}
{computes the half of the maximum size of a given room}
{-------------------------------------------------}

half_room_size(_room,_half_room_size):-
        max_room_size(_room,_max_room_size),
        (_half_room_size is _max_room_size / 2.0).

{-------------------------------------------------}
{gives an estimation of the maximum size of a given room}
{-------------------------------------------------}

max_room_size(_room,_max_room_size):-
        existspropvalue(dt,_room,walls_succession,_walls),
        biggest_wall(_walls,_max_room_size). {first estimation} {see CJ}

biggest_wall(_walls,_max_length):-
        biggest_wall(_walls,_max_length,0).

biggest_wall([],_max_length,_max_length):-!.
biggest_wall([_wall|_walls],_max_length,_current_max):-
        existspropvalue(dt,_wall,length,_length),
        (_length > _current_max,!,
         biggest_wall(_walls,_max_length,_length)
        ;
         biggest_wall(_walls,_max_length,_current_max)
        ).

{-------------------------------------------------------------}
{This predicat looks for the intersection of the floor and the }
{vertical backbone. If this room isn't a corridor, it gives it }
{as departure room else it gives one of the neighour of the    }
{corridor.                                                     }
{-------------------------------------------------------------}
find_departure_room (_floor_network, _vertical_backbone, _departure_room) :-
        existspropvalue (dt, _floor_network, departure_room, _departure_room ),
        !.

find_departure_room (_floor_network, _vertical_backbone, _computer_room) :-
        find_computer_room (_computer_room),
        existspropvalue (dt, _floor_network, set_in_floor, _floor),
        getpropvalue (dt, _computer_room, set_in_floor, _floor), !,
        putpropvalue (dt, _floor_network, departure_room, _computer_room).

find_departure_room (_floor_network, _vertical_backbone, _location) :-
        existspropvalue (dt, _vertical_backbone, vertical, []), !,
        existspropvalue (dt, _floor_network, set_in_floor, _floor),
        existspropvalue (dt, _floor, list_of_rooms, _L_rooms),
        choose_one_departure_room (_L_rooms, _location),
        putpropvalue (dt, _floor_network, departure_room, _location).

find_departure_room (_floor_network, _vertical_backbone, _location) :-
        existspropvalue (dt, _floor_network, set_in_floor, _floor),
        existspropvalue (dt, _vertical_backbone, vertical, [_vert]),
        send (dt, _vert_room = (_r where _r = oneof _vert#set_in_rooms and
                _floor = _r#set_in_floor)),
        is_or_not_a_corridor (_floor_network, _vert_room, _vert, _location),
        putpropvalue (dt, _floor_network, departure_room, _location).

is_or_not_a_corridor (_floor_network, _vert_room, _vert, _location) :-
        getobject (dt, _vert_room, DIRECTIN, Corridors), !,
        existspropvalue (dt, _vert_room,  walls_succession, _walls_succession),
        aplat (_walls_succession, _a_walls_succession),
        cutting (_a_walls_succession, _Cutting),
        find_walls_next_to_vertical (_Cutting, _vert, _vert_room, [[_w|_W]|_Walls]),
        send (dt, _possible_locations = (all _r where _r is_in Rooms and
        _w = oneof _r#walls_succession)),
        existspropvalue (dt, _floor_network, set_in_departments, _L_depart),
        existspropvalue (dt, _floor_network, partof, _network),
        restrict_L_rooms (_network, _possible_locations, _L_depart, _Lrooms),
```

```
        (_Lrooms == [],
                choose_one_departure_room (_possible_locations, _location), !;
        choose_one_departure_room (_Lrooms, _location)).
is_or_not_a_corridor (_, _location, _, _location).

{-------------------------------------------------------------}
{choose_one_departure_roomcompute_dist_to_be_cabled(instance3, _M).
 selects one room through the given }
{list by use of following criterions : it prefers first a      }
{computer_room, secondly a room with a mainframe, third a room }
{with a server else it takes the first element of the list.    }
{-------------------------------------------------------------}

choose_one_departure_room ([_x|_Q], _location) :-
        send (dt, _location = (_r where _r = oneof [_x|_Q] and _r is_in Computer_rooms))
, !.
choose_one_departure_room ([_x|_Q], _location) :-
        send (dt, _location = (_r where _r = oneof [_x|_Q] and
                                _m is_in Mainframes and
                                _r = oneof _m#set_in_rooms)), !.
choose_one_departure_room ([_x|_Q], _location) :-
        send (dt,_location = (_r where _server is_in Machines and
                _server#is_a_server = true and
                _r = oneof _server#set_in_room)), !.
choose_one_departure_room ([_location|_Q], _location).

{-------------------------------------------------------------------}
{This predicat looks for the computer room (only one) of the site. }
{-------------------------------------------------------------------}

find_computer_room (_computer_room) :-
        getlist (dt, [_r|_L], DIRECTIN, Computer_rooms), !,
        choose_one_computer_room ([_r|_L], _computer_room).

find_computer_room (_computer_room) :-
        send (dt, _Lr_with_mainframe = (all _r1 where
                _mainframe is_in Mainframes and
                _r1 = oneof _mainframe#set_in_rooms)),
        send (dt, _Lr_with_server = (all _r2 where _server is_in Machines and
                _server#is_a_server = true and
                _r2 = oneof _server#set_in_rooms)),
        append (_Lr_with_mainframe, _Lr_with_server, _Lr),
        no_double (_Lr, _Lr2),
        choose_one_computer_room (_Lr2, _computer_room).

{-------------------------------------------------------------------------}
{It selects the apppropriate candidate for been the computer room among   }
{the list of possible computer_room depending on the numbers of Mainframe }
{and of servers.                                                          }
{-------------------------------------------------------------------------}

choose_one_computer_room ([_computer_room], _computer_room).

choose_one_computer_room ([_r, _r2|_Lr], _computer_room) :-
        send (dt, _L_mainframe_in_r = (all _m where _m is_in Mainframes and
                                _r = oneof _m#set_in_rooms)),
        list_length (_L_mainframe_in_r, _n),
        test_nb_mainframe ([_r2|_Lr], [_r], _n, _computer_room), !.

choose_one_computer_room ([_r, _r2|_Lr], _computer_room) :-
        send (dt, _L_servers_in_r = (all _m where _m is_in Machines and
                        _m#is_a_server = true and
                                _r = oneof _m#set_in_rooms)),
        list_length (_L_servers_in_r, _nb_server),
        test_nb_servers ([_r2|_Lr], _r, _nb_server, _computer_room).

test_nb_mainframe ([], [_r], 0, _) :-
        !, fail.
test_nb_mainframe ([], [_computer_room], _n, _computer_room) :- !.
test_nb_mainframe ([], [_r|_Lr], _n, _computer_room) :-
        send (dt, _L_servers_in_r = (all _m where _m is_in Machines and
                        _m#is_a_server = true and
                                _r = oneof _m#set_in_rooms)),
        list_length (_L_servers_in_r, _nb_server),
        test_nb_servers (_Lr, _r, _nb_server, _computer_room).

test_nb_mainframe ([_r2|_Lr], _Lr2, _n, _computer_room) :-
        send (dt, _L_mainframe_in_r2 = (all _m where _m is_in Mainframes and
                                _r2 = oneof _m#set_in_rooms)),
        list_length (_L_mainframe_in_r2, _n2),
        ( (_n > _n2, test_nb_mainframe (_Lr, _Lr2, _n, _computer_room) ) ;
          (_n = _n2, test_nb_mainframe (_Lr, [_r2|_Lr2], _n, _computer_room) ) ;
          (_n < _n2, test_nb_mainframe (_Lr, [_r2], _n2, _computer_room) )
        ).

test_nb_servers ([], _computer_room, _, _computer_room).
test_nb_servers ([_r2|_Lr], _r, _n, _computer_room) :-
        send (dt, _L_servers_in_r2 = (all _m where _m is_in Machines and
                        _m#is_a_server = true and
                                _r2 = oneof _m#set_in_rooms)),
        list_length (_L_servers_in_r2, _n2),
        ( (_n >= _n2, test_nb_servers (_Lr, _r, _n, _computer_room) ) ;
          (_n < _n2, test_nb_servers (_Lr, _r2, _n2, _computer_room) )
        ).

{-------------------------------------------------------------------------}
{From the list of floor_departments slot of a Floor_networks composed     }
{by lists with the form [floor, List of departments], this predicat       }
{extracts all the floors.                                                 }
{-------------------------------------------------------------------------}

extract_floors ([], []).
extract_floors ([[_fl|_D]|_floor_departments], [_fl|_L_floors]) :-
        extract_floors (_floor_departments, _L_floors).

{-----------------------------------------------------------}
{deletes all the instances belonging to the given building}
{and so the building itself                                 }
{-----------------------------------------------------------}

delete_building(_building):-
        findall(_room,
                (getobject(dt,_room,in,Rooms),
                 existspropvalue(dt,_room,set_in_building,_building)),
                _rooms_to_delete),
        findall(_wall,
                (getobject(dt,_wall,in,Walls),
                 existspropvalue(dt,_wall,set_in_rooms,_wall_rooms),
                 _wall_rooms = [_wall_room|_],
                 existspropvalue(dt,_wall_room,set_in_building,_building)),
                _walls_to_delete),
        findall(_internal_shaft,
                (getobject(dt,_internal_shaft,in,Internal_shafts),
                 existspropvalue(dt,_internal_shaft,set_in_building,_building)),
                _internal_shafts_to_delete),
        findall(_floor,
                (getobject(dt,_floor,in,Floors),
                 existspropvalue(dt,_floor,set_in_building,_building)),
```

```
            _floors_to_delete),
        delete_instances(dt,_rooms_to_delete),
        delete_instances(dt,_walls_to_delete),
        delete_instances(dt,_internal_shafts_to_delete),
        delete_instances(dt,_floors_to_delete),
        delobject(dt,_building,in,Buildings).

{What about departments and Inter_buildings_shafts ???}
```

Résumé.

Réalisée dans le cadre d'un projet Esprit II, cette thèse a eu pour objectif principal la réalisation d'un système à base de connaissances pour la conception de réseaux. A travers le développement de ce système appelé NEST (pour Network design Expert SysTem), deux principaux axes de recherche ont été entrepris : la modélisation d'une activité de conception pour laquelle l'intelligence artificielle a relativement peu de théories et la configuration de réseaux locaux pour laquelle peu de systèmes experts ont été conçus. L'originalité de l'approche est d'avoir employé plusieurs formalismes : un modèle centré objets pour les concepts de base, prolog pour l'implémentation des connaissances opératoires et une architecture constituée de plans de contrôle / tâches / sous-tâches / modules. En outre, implémenter un système à base de connaissances nécessite de représenter les connaissances propres au domaine d'application choisi. Une étude a été menée afin d'établir l'état de l'art en ce qui concerne la représentation des connaissances. Certains points particuliers ont été développés tels que la programmation orientée objets et les méthodes de représentation des connaissances employées pour résoudre des problèmes de conception. Le développement de NEST fait partie prenante du projet MMI^2 dont le but est la réalisation d'une interface multi-modes pour des systèmes à base de connaissances. Les différents modes de communication permis par cette interface sont les langues naturelles (anglais, français, espagnol), le graphique, le gestuel et un langage de commande. Un premier démonstrateur intégrant ces modes et le système NEST a été présenté en Octobre 1991 et lors de la conférence Esprit en Novembre 1991 à Bruxelles.

Mots clés.

Représentation des connaissances, programmation orientée objets, architecture tableau noir, méthodes de conception pour des système à base de connaissances, réseaux locaux, interface multi-modes.

Abstract.

Achieved as part of an Esprit II project, the main purpose of this thesis was the realisation of a knowledge based system for network design. Through the development of this system called NEST (for Network design Expert SysTem), two mains research axes have been undertaken : the modelling of a design activity for which the artificial intelligence has relatively weak theories and the local area network design for which few experts systems have been computed. The originality of the used approach is to have made use of several formalisms : object centred models for the basic concepts, prolog to implement operative knowledge and an architecture composed of control plans / tasks / sub-tasks / modules. Furthermore, to implement a knowledge based system, it is necessary to represent domain knowledge. Some particular points have been emphasised such as object oriented programming and design methods used in knowledge based system.The development of NEST is part of MMI^2 project whose main objective is to realize a multi-modal interface for knowledge based systems. The various modes allowed by this interface are the natural languages (english, french, spanish), the graphics, the gesture and a command language. A first demonstrator has been presented in October 91 and during the Esprit conference in November 1991 in Bruxelles.

Keywords.

Knowledge representation, object oriented programming, blackboard architecture, design methods for knowledge based systems, local area networks, multi-modal interface.