Implémentation d'un modèle d'acteur, application au traitement de données partielles en audit thermique de bâtiment

Xiaohua Le

Thèse de doctorat de l'Ecole Nationale des Ponts et Chaussées

Spécialité

Science et Technique de Bâtiment

Présentée par

LE Xiaohua

pour obtenir le titre de

Docteur de l'Ecole Nationale des Ponts et Chaussées

sujet:

Implémentation d'un modèle d'acteur, Application au traitement de données partielles en audit thermique de bâtiment

soutenue le 30 Mars 1992
devant le jury composé de:

M. Jacques Rilling	:	Président
M. Paul Reboux	:	Rapporteur
M. François Xavier Testard-Vaillant	:	Rapporteur
M. Jérôme Adnot	:	Examinateur
M. Paul Brejon	:	Examinateur
M. René Lalement	:	Examinateur
M. Jacques Ferber	:	Examinateur

Remerciements

Ce travail est le fruit d'une coopération entre l'Ecole Nationale des Ponts et Chaussées et l'Ecole des Mines de Paris.

Je voudrais exprimer ma gratitude et mes remerciments à Monsieur J. Rilling de m'avoir faire l'honneur de présider le jury.

Je voudrais remercier Monsieur F.X. Testard-Vaillant et Monsieur P. Reboux de rapporter cette thèse. Je leur suis d'autant plus reconnaissant que la thèse n'est concernée que partiellement par l'informatique et l'énergétique.

Je voudrais remercier Monsieur J. Ferber d'avoir accepté de faire partie du jury.

Je voudrais remercier Monsieur R. Lalement et Monsieur P. Brejon pour les conseils et les aides qu'ils m'ont apportés tout au long de ce travail. Tous deux m'ont appris la rigueur et la précision. Travailler avec eux est un plaisir pour moi.

Mes remerciements vont également à Monsieur N. Bouleau et Monsieur J. Adnot pour m'avoir accueilli dans leurs laboratoires, et fourni les conditions idéales de travail.

Je voudrais remercier Monsieur B. Delcombre, Monsieur M. Raoust et Monsieur M. Clerjo pour les conseils et critiques d'experts qu'ils ont apportés à ce travail.

Une partie de ce travail est en effet réalisée en collaboration avec Anne Rialhe du Centre d'Energétique, et a bénéficié de nombreux conseils et résulats de sa part, un grand merci.

Sans les aides et les encouragements de tous ceux qui travaillent au CERMA, CERMICS, GISE, et du Centre d'Energétique, ce travail ne serait pas ce qu'il est aujourd'hui, merci à tous.

Tous mes amis qui m'ont supporté pendant des années, merci à vous.

Enfin, je voudrais remercier He Ying pour tout ce qu'elle m'a apporté durant ces années : les encouragements, les aides et de nombreuses discussions sur les problèmes d'apprentissage et cognitifs.

RESUME

L'audit thermique des bâtiments existants, par la nature de l'opération, pose un certain nombre de problèmes liés à la réalisation d'outils informatiques. Un des aspects importants, concernant les informations patielles (incertaines, imprécises et lacunaires) et leur traitement dans un programme informatique, n'est jusqu'alors pas pris en compte par des outils "classiques". On propose donc, dans cette thèse, d'établir les bases d'une nouvelle génération d'outils d'aide au diagnostic, en utilisant les techniques de programmation avancée. Après une analyse détaillée de la problèmatique, nous avons construit une couche d'un modèle d'acteur à partir de Scheme, et réalisé une maquette informatique. Certains concepts importants, notamment celui de la "continuation" permettant la mise en œuvre de structures de contrôle souples, puissantes et élégantes, ont été largement exploités. La réalisation de la maquette permet de montrer les intérêts d'une telle approche.

ABSTRACT

The nature of energy audit in existing buildings put down some difficulties for computer software realisation. One of the most important aspects, related to the uncertainty of data and their processing in computer programs, has not been taken in account by "classical" systems. The purpose of this thesis is to establish the basis of a new generation of tools for energy audit, using advanced programming techniques. After a detailed analysis of the problem, we give an implementation of an actor model in Scheme, and an experimental realisation. The use of certain important concepts, such as the "continuation", allows us to implement powerful and elegant control structures. The experimental realisation shows clearly the advantage of this new approach to energy audit in existing buildings.

Table des Matières

Introduction

Le Zen, c'est comme un homme qui s'accroche à un arbre par les dents au-dessus d'un précipice. Ses mains ne s'agrippent à aucun rameau, ses pieds ne reposent sur aucune branche, et sous l'arbre, une autre personne lui demande : "Pourquoi Bodhidharma est-il venu d'Inde en Chine ?" S'il ne répond pas, il échoue et s'il répond, il tombe et perd sa vie. Alors, que doit-il faire ?

— *Exposé par le Maître Kyogen*

L'objectif de notre recherche est d'établir les bases d'une nouvelle génération d'outils d'audit thermique de bâtiments existants, en utilisant de nouveaux paradigmes de programmation, introduits par les recherches menées en intelligence artificielle et en génie logiciel.

Les crises énergétiques survenues dans les années 70 ont amené les pouvoirs publics et les différents acteurs de l'économie à découvrir une des sources les plus importantes de l'économie d'énergie : l'utilisation énergétique dans les bâtiments existants qui constituent un important parc immobilier. A cette exigence économique, il faut ajouter celle sans cesse croissante en terme de confort thermique. Le diagnostic thermique des bâtiments existants s'inscrit ainsi dans le cadre de cette double exigence, et s'est rapidement développé.

Avec ce besoin de diagnostic, de nombreux outils spécialisés ont vu le jour, dont la plupart sous forme de logiciel informatique.

L'analyse de ces logiciels d'aide au diagnostic thermique [4], développés au cours des 5 ou 6 dernières années, conduit au constat d'une certaine rigidité de ces outils vis à vis des problèmes spécifiques du diagnostic. Ceci tient essentiellement au caractère particulier d'une opération de diagnostic par rapport à celle de conception de bâtiment neuf, alors que les outils mentionnés font généralement usage des méthodologies développées en conception, et ne tiennent pas compte la spécificité d'un diagnostic donné, ni de la nature des données recueillies. Cette rigidité est donc indépendante de l'ergonomie qui s'améliore d'ailleurs rapidement (utilisation de souris, de menu etc...). Il s'agit bien d'un *fossé* entre :

- d'une part, les informations *partielles*, qui expriment la façon dont un bâtiment réel est perçu par le diagnostiqueur (univers du réel, incertain),
- et d'autre part, l'ensemble des données exigées, qui correspondent à des modèles de représentation *figés*, souvent sur-informés et qui ne font aucune place à l'incertitude (univers du modèle, certain).

On sait par ailleurs que les données recueillies sont très variables selon :

- le type de bâtiment considéré (maison individuelle, petit, moyen ou grand collectif, différents types de locaux tertiaires ou industriels),
- le type de mission confiée au diagnostiqueur (pré-diagnostic ou conseil, diagnostic type "AFME", diagnostic approfondi),
- le cas particulier traité qui impose toujours, de façon plus ou moins marquée, sa spécificité.

Il est envisageable de concevoir plusieurs outils de diagnostic, dédiés aux principaux couples "type de bâtiment/type de mission", — par exemple, logiciel de diagnostic intermédiaire pour surfaces commerciales, ou bien diagnostic simplifié pour maison individuelle. En revanche il est indispensable de s'affranchir de la spécificité de chaque audit, ce qui signifie qu'un outil donné doit être à même de prendre en considération la diversité des situations d'accès aux données que peut rencontrer le diagnostiqueur. En effet, chaque audit met en œuvre des moyens variés d'investigation [24] :

- analyse visuelle,
- entretien avec les usagers, avec le gestionnaire,
- *auscultation* des différentes parois,
- relevé de la géométrie des bâtiments,
- relevé des caractéristiques des équipements (corps de chauffe, réseau de distribution, description de la chaufferie...),
- mesure physique (combustion, thermographie infra-rouge de l' enveloppe, températures des fluides caloporteurs, des ambiances chauffées...),
- analyse des factures et des consommations passées...

Ainsi dans la grande majorité des cas, et avec une acuité particulière pour les diagnostics "rapides", le diagnostiqueur se trouve en présence de données :

- *imprécises*. Ces données peuvent être sous forme numérique approximatives (par exemple "température intérieure d'un local comprise entre 19 et 21°C"), ou bien littérale (par exemple "état médiocre du calorifuge des réseaux", il s'agit bien d'une *imprécision* qui portera sur la valeur du coefficient de pertes thermiques de ce réseau, déduite de la donnée initiale) ;

- *incertaines*. Par exemple "je pense que cette paroi n'est pas isolée thermiquement". il s'agit cette fois d'une assertion qui a de bonnes chances d'être vraie, mais qui peut être fausse :

- *inconnues*. Par exemple "le scénario d'ouverture des fenêtres est inconnu".

A ces données de nature très différente, doivent correspondre des modes de traitement appropriés.

L'amélioration que nous pouvons apporter à ces outils existants tient donc à la possibilité de traiter d'une façon pertinente les données disponibles à la seule issue de l'analyse du bâtiment considéré.

Le développement récent de l'informatique nous offre des outils de programmation et de développement particulièrement intéressants, et la recherche en I.A., ouvre de nouvelle forme d'approche pour la représentation des connaissances. Parmi ces outils et ces approches, les systèmes experts, les langages orientés objets et les modèles d'acteurs etc. semblent adaptés au problème que pose le diagnostic thermique des bâtiments, ceci par la nature des objets et des modèles (et connaissances) manipulés.

L'approche des *systèmes experts*, avec sa puissance de déduction et sa facilité de représentation des connaissances, paraît en premier abord d'un grand intérêt pour le diagnostic thermique de bâtiment où les connaissances sont a priori considérées comme pouvant s'exprimer sous forme de *règles expertes*. Comme on le verra, ce point de vue n'est valable que pour certains types de diagnostic, le recours aux modèles physiques sont les plus souvent fréquents dans le traitement des informations.

Les langages à objets ont plusieurs origines différentes : celle du besoin de la représentation des connaissances en intelligence artificielle qui a donné les réseaux sémantiques, les frames etc, celle du besoin de génie logiciel pour construire de gros systèmes informatiques et celle de la simulation. Ce paradigme de programmation connaît un rapide développement, et de nombreux langages et applications ont été développés.

La programmation par objet convient naturellement à la structuration des objets physiques que l'on manipule (un bâtiment chauffé et occupé). La notion d'encapsulation des objet permet de définir les objets manipulés en audit d'une façon simple et modulaire. Le partage de connaissances entre les différents objets par le mécanisme de l'héritage ou de la délégation permet de mieux utiliser les objets définis et facilite la mise en point des programmes.

Le modèle d'acteur a pour origine également la représentation des connaissances. Par ses caractéristiques particulièrement intéressantes, notamment l'utilisation du mécanisme de *continuation*, les acteurs sont plus souvent utilisés pour la construction de modèle de calcul en parallèle dans des architectures distribuées.

Dans notre problème, par la présence des informations que l'on qualifie de *partielles*, l'approche des modèles d'*acteurs* offre des structures de contrôle plus souples et mieux adaptées. Un acteur est un objet actif. L'une des caractéristiques des acteurs, la continuation permet une gestion plus souple des envois de messages. Il est plus facile et plus élégant de construire des structures de contrôle telles que le retour-arrière pour pouvoir prendre en compte la multiplicité des modèles physiques en jeu.

Il faut aussi noter que le domaine des recherches sur les modèles de traitement des informations partielles est particulièrement actif, le grand nombre de modèles déjà proposés en témoigne (théorie des probabilités, des possibilités, de l'évidence...) [83]. Ces modèles de raisonnement permettent donc de mieux prendre en compte les aspects *incertains* et *imprécis* des informations que l'on manipule quotidiennement.

Nos objectifs concernant cette recherche sont doubles :

- d'une part établir les bases d'une nouvelle génération d'outils d'audit thermique de bâtiments existants ;
- et d'autre part chercher les nouveaux modèles de programmation afin de mieux prendre en compte les divers aspects des informations à traiter.

Ces objectifs peuvent se traduire en effet par les points résumés ci-dessous :

- expérimenter un style de programmation combinant les approches objets et acteurs ;
- prendre en compte et traiter les données imprécises (numériques ou littérales) et incertaines, et accepter a priori de données inconnues, chaque fois que ce manque d'information est contournable, donc co-existence de plusieurs modèles thermiques mis en oeuvre selon les données disponibles, et gestion de la cohérence données-modèles,
- construire une interface utilisateur *intelligente* pour exploiter au mieux les données disponibles.

La satisfaction de ces objectifs nécessite une meilleure compréhension des problèmes posés, et des outils disponibles. Nous allons donc dans cette thèse analyser les outils existants pour le diagnostic thermique de bâtiment, les différents modèles de représentation et de traitement des connaissances, et à partir de ces éléments proposer un modèle de programmation alliant les concepts des objets et des acteurs afin de réaliser une maquette informatique destinée à l'audit thermique des bâtiments existants. Ainsi la suite du texte est structurée de la façon suivante :

- Le chapitre I présente d'une manière générale les problèmes liés au diagnostic, notamment le problème lié à la disponibilité et l'incomplétude des informations. Et après une analyse des méthodes et des outils existants, nous dégageons quelques points concernant la spécificité de diagnostic, ainsi que les exigences concernant les outils à développer ;
- L'univers physique de l'audit, à savoir les objets et les modèles physiques en bâtiment sera décrit au chapitre II. On montre par cette description la complexité des objets et les différents types de modèles manipulés lors d'un diagnostic ;
- Le chapitre III parcourt les différents modèles de traitement des informations partielles existants. L'analyse de ces modèles nous permet d'une part de mieux saisir les aspects incertains et imprécis des informations et d'autre part de savoir ce que l'on peut utiliser comme modèle de traitement dans le cas de l'audit. La nature des informations manipulées lors d'un diagnostic thermique est explicitée ;

- Nous présentons ensuite dans le chapitre IV les modèles d'objets et d'acteurs après une discussion générale sur les différentes approches de la représentation des connaissances. Ainsi, nous justifions notre choix du modèle de représentation concernant la maquette réalisée ;

- Dans le chapitre V, nous donnons les éléments essentiels du langage de programmation SCHEME. Quelques concepts intéressants de SCHEME seront mis en évidence tels que les objets de première classe, les liaisons statiques et les continuations. On montrera l'intérêt d'un langage tel que SCHEME comme outil de construction de nouveau modèle de programmation ;

- Le chapitre suivant concerne plus particulièrement la réalisation pratique d'une maquette informatique. Nous allons proposer une implémentation relativement simple d'un modèle de programmation, et à partir de ce modèle et d'une structuration des objets et de la prise en compte de certains modèles thermiques utilisés, la réalisation pratique est décrite.

Chapitre I

Audit thermique de bâtiment

L'intelligence apparaît, au total, comme une structuration imprimant certaines formes aux échanges entre le ou les sujets et les objets environnants, auprès ou au loin. Son originalité tient essentiellement à la nature des formes qu'elle construit à cet effet.

— *Jean Piaget*

Le parc immobilier existant en France, avec ses 22 millions de bâtiments résidentiels, dont plus de la moitié a été construit avant la première crise pétrolière, constitue une des sources les plus importantes pour l'économie d'énergie.

Le calcul thermique en conception des bâtiments neufs, fort récent d'ailleurs, avait à l'origine pour but principal d'améliorer le confort thermique des occupants : une température intérieure raisonnablement élevée en hiver par exemple. Il faut maintenant ajouter, à cet objectif de confort, un deuxième objectif qui est celui de l'économie d'énergie, c'est-à-dire obtenir ce même confort en dépensant le moins d'énergie possible, ce qui se traduit en terme économique par le moindre coût d'utilisation. En ce qui concerne les bâtiments existants, nombreux sont ceux qui ne répondent pas à cette double exigence de confort-économie. Le diagnostic thermique de bâtiments existants doit donc répondre à plusieurs attentes en terme d'économie, de confort et de revalorisation du cadre de vie. Il doit apporter généralement un accroissement de la valeur d'usage et de la valeur patrimoniale des bâtiments.

Par l'importance de l'enjeu, le diagnostic thermique des bâtiments existants s'est considérablement développé depuis deux décennies, dans les pays industrialisés qui possèdent un parc immobilier existant important. Et de nombreux d'outils d'aide au diagnostic ont été développés. Ces outils sont pour la plupart basés sur les méthodologies utilisées pour les outils de conception de bâtiments neufs. Dans ce chapitre, nous allons essayer de dégager quelques points importants sur la méthodologie de diagnostic, et de faire une synthèse des

méthodes et des outils existants en diagnostic thermique de bâtiment, et à partir de ces éléments de définir les besoins et les spécificités des outils de diagnostic.

I.1 Généralités sur le diagnostic

Un système évolue dans le temps, et ses constituants et ses paramètres se trouvent souvent modifiés au cours de son évolution. Cette évolution conduit souvent le système à un dysfonctionnement (la détérioration de la surface d'une paroi, la fatigue des pièces d'une machine...). En plus, l'environnement dans lequel évolue le système se trouve constamment changé d'une manière incontrôlable par le système (apport d'une nouvelle technologie pour les machines par exemple, une crise énergétique...). Pour réparer le système ou le faire adapter à son nouvel environnement, un diagnostic s'impose.

Par la nature différente des systèmes, la méthodologie de diagnostic peut être différente, il est évident qu'on ne diagnostique pas une maladie d'une personne humaine de la même façon qu'un dysfonctionnement d'une machine. On constatera dans la suite que le diagnostic thermique se différencie nettement d'un diagnostic classique d'une panne par exemple. Néanmoins, on peut constater des étapes indispensables et des problèmes rencontrés dans tous les diagnostics. En effet, pour avoir un diagnostic correct et fiable, une bonne connaissance des différents paramètres et composants du système et aussi de leur relations sont nécessaires. On peut d'une manière générale distinguer les trois étapes suivantes dans une opération de diagnostic :

- étude du système existant. Dans la majorité des cas, cette étape consiste à relever des données du système existant par des moyens appropriés. Parfois cette étape peut être très difficile selon la nature du système, c'est notamment le cas pour un bâtiment existant où l'accès des données n'est pas toujours aisé comme on le verra plus tard ;
- analyse des données recueillies afin de détecter les anomalies du système par l'utilisation des modèles ou méthodes appropriés ;
- proposition d'un ensemble de solutions cohérentes, en tenant compte de l'environnement extérieur du système, afin de l'améliorer.

D'un domaine à l'autre, l'importance de ces trois étapes peut varier sensiblement. Dans notre travail, nous employons dans la mesure du possible le terme *audit* pour désigner les deux premières étapes, et *diagnostic* pour l'ensemble de ces trois étapes.

La réalisation d'un diagnostic demande souvent des compétences particulières au diagnostiqueur du fait de la complexité du système. Par exemple, la difficulté liée à l'accès de données fait que le diagnostiqueur peut se trouver dans une situation où il ne peut pas appliquer un modèle d'analyse directement ; il est dans ce cas soit obligé d'utiliser un autre modèle si cela est possible, soit tenté d'émettre certaines hypothèses afin de poursuivre son analyse. Ou bien un diagnostiqueur peut, à partir de la donnée de certains paramètres, faire l'hypothèse d'une panne et écarter ainsi d'autres hypothèses pour se concentrer sur une partie du système. Dans ces deux cas, il fait intervenir dans son analyse ce qu'on appelle des *expertises*.

On constate par ces facteurs qu'il est souvent nécessaire de conduire les deux premières étapes d'une manière *parallèle*, c'est-à-dire qu'après une analyse de premières données recueillies, il est nécessaire de refaire le relevé des autres paramètres ou de préciser les informations mal connues.

Le diagnostic est ainsi apparu comme un domaine d'application par excellence des systèmes experts [47]. L'exemple des systèmes destinés au diagnostic des maladies en médecine en est une illustration ; le premier système expert MYCIN [90] est un système de diagnostic des infections sanguines. De nombreux systèmes ont été développés dans différents domaines appliqués au diagnostic. Le domaine du bâtiment et de la thermique ne font pas exception ; on peut citer la tentative de réalisation d'un système expert pour le diagnostic thermique des maisons individuelles [62].

I.1.1 Disponibilité des informations

Comme on l'a vu dans la section précédente, dans de nombreux cas, une des difficultés essentielles de diagnostic réside dans la première étape, c'est-à-dire le recueil des informations du système existant. Cette étape est importante, car les propositions d'améliorations dépendent directement de la qualité de l'analyse de ces informations recueillies.

Le diagnostiqueur met en œuvre un certain nombre de moyens qu'il considère appropriés afin d'obtenir d'une façon fiable un plus grand nombre d'informations nécessaires à son analyse. Dans la plupart des cas, il est difficile d'obtenir toutes les informations souhaitées, à cause des facteurs divers qui sont :

- les moyens mis en œuvre sont dans la plupart des cas limités, ceci en temps ou en moyens matériels. Une opération de diagnostic est soumise comme d'autres activités à des contraintes économiques et temporaires. Le cas du diagnostic thermique montre bien cette contrainte, il n'est pas toujours possible à un diagnostiqueur de revenir sur site, une fois qu'il a effectué son analyse ;

- la compétence et le savoir-faire du diagnostiqueur interviennent d'une façon importante sur la façon de procéder, et influencent directement la fiabilité des informations obtenues.

Le deuxième facteur peut influencer évidement le premier, les moyens utilisés peuvent être différents lorsqu'il s'agit de deux diagnostiqueurs pour obtenir les mêmes informations.

Il nous paraît important d'étudier ce premier facteur dans la fiabilité d'un diagnostic. En effet, dans certains types de diagnostic, le but est de proposer un ensemble d'améliorations possibles, dans le cas du diagnostic énergétique de bâtiment, le résultat escompté est le rendement des opérations d'amélioration. Ceci se traduit souvent par un facteur économique qu'on appelle le *temps de retour*. Selon la fiabilité des informations obtenues au départ, ce facteur peut avoir une fiabilité plus ou moins grande. En générale, on fixe une fiabilité sur le résultat à obtenir, ce qui nous conduit également à fixer une fiabilité au départ.

Du fait du coût important dans la réalisation de l'opération de recueil des informations, il est souvent bon de limiter les moyens à mettre en œuvre pour ainsi trouver une cohérence de fiabilité entre le résultat et les données.

Nous pouvons ainsi schématiser cette relation de fiabilité entre le résultat et les données par une courbe de la forme de la figure I.1 que l'on appelle courbe de relation coût-précision.

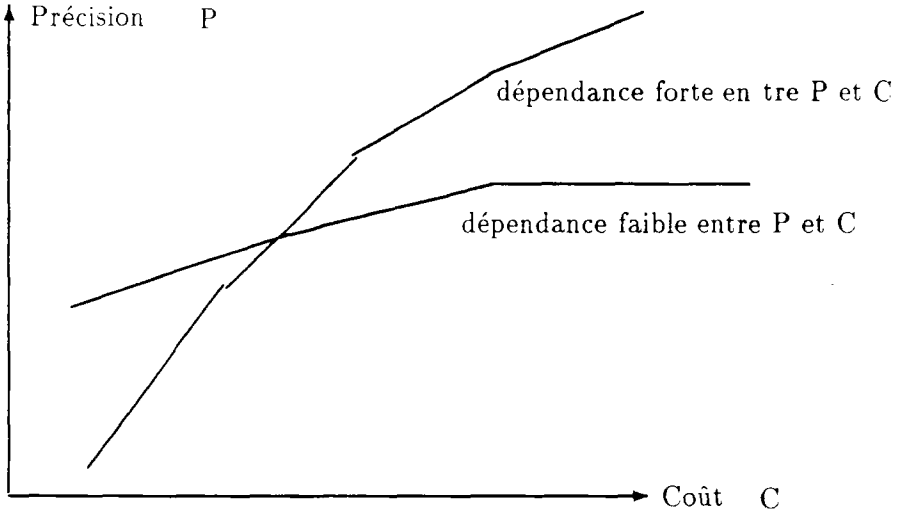


Figure I.1: Relation entre coût et précision

L'existence de cette relation a une grande importance, car elle permet dans de nombreux cas de définir certaines stratégies dans la recherche des informations afin de répartir les moyens sur les différents postes du système pour avoir une cohérence sur l'ensemble. Par ceci, on pourrait limiter l'investigation pour l'obtention d'un paramètre précis avec des moyens importants, alors qu'on sait que les autres paramètres du système seront obtenus avec une fiabilité nettement inférieure. Dans le cas de diagnostic thermique, nous reviendrons sur cette question qui se trouve souvent négligée dans la pratique professionnelle.

I.1.2 Incomplétude des informations

Les informations que l'on recueille dépendent des moyens mis en œuvre et du savoir-faire du diagnostiqueur. Par conséquent, les informations obtenues, peuvent donc être partielles, c'est-à-dire imprécises et incertaines, et dans certains cas elles peuvent être manquantes [1]. Ceci est dû au fait que les moyens sont limités (le coût de mise en œuvre, la difficulté d'accès, etc) et que le diagnostiqueur peut "oublier" certaine informations ,

L'incomplétude des informations est donc inhérente à la nature du diagnostic ; on est amené à raisonner avec ces informations de nature différente. Les outils d'aide au diagnostic essaient en général de prendre en compte cet aspect, mais souvent ne parviennent pas à leur but. Un grand nombre de modèles de raisonnement ont été proposés. Nous reviendrons sur ce problème au chapitre III.

[1] Nous employons le terme *information partielle* pour désigner l'ensemble de trois types d'information qui sont *imprécision*, *incertitude* et *information lacunaire*.

I.2 Diagnostic énergétique

Nous employons le terme de *diagnostic* dans le sens usuel, et d'*audit* pour désigner les deux premières étapes du diagnostic. En "diagnostic thermique" de bâtiment, le terme "diagnostic" est souvent utilisé pour désigner un type de diagnostic particulier qui est celui défini par l'Agence Française pour la Maîtrise de l'Energie (AFME) dans le guide de diagnostic thermique [2], dans ce cas, nous préciserons par "diagnostic de type AFME".

Dans le domaine de bâtiment existant, on peut distinguer deux sortes de diagnostics : diagnostic global prenant en compte l'ensemble des aspects en question dans un bâtiment (amélioration de cadre de vie, valorisation du patrimoine, amélioration technique...) et le diagnostic technique partiel (acoustique, structurel et thermique etc). Nous nous intéressons essentiellement au diagnostic thermique et dans certaines mesures à l'aspect économique.

Il ne faut cependant pas oublier que pour un diagnostic technique partiel (thermique, acoustique...), l'amélioration apportée est souvent plus importante (ou dans certains cas, elle peut détériorer la performance d'autres caractéristiques) que la seule considération technique. Les améliorations proposées issues d'un diagnostic technique doivent impérativement intégrer d'autres facteurs techniques, esthétiques, humains et sociaux.

Comme on a indiqué plus haut, un diagnostic thermique de bâtiments existants peut être divisé en trois étapes. Cette division est également préconisée par [2] :

- recueillir les données du bâtiment à diagnostiquer avec les différents moyens disponibles, visite sur site (analyse visuelle, entretient avec les usagers), relevé sur plan, les mesures physiques par les instruments appropriés (combustion, les températures...), analyse des factures de consommation etc... ;

- à partir des informations recueillies dans la première étape, construire un modèle physique explicatif du bâtiment, et analyser la consommation théorique calculée à partir de ce modèle, et la consommation facturée si la donnée de cette consommation est disponible, et ensuite comparer ces deux consommations afin de valider le modèle physique construit. Dans le cas de non concordance entre ces deux consommations, il est nécessaire de revenir sur le modèle physique. Cette étape est souvent appelée le *bouclage de bilan* ;

- proposer un ensemble cohérent des améliorations en tenant compte d'autres paramètres en jeux tels que les paramètres sociaux, économiques.... La proposition des améliorations se fait en général en deux étapes

  - une liste de toutes les propositions possibles classées selon le temps de retour croissant ;
  - une liste des ensembles de propositions *cohérentes* permettant de réaliser les travaux de différents types (avec un temps de retour long, court ou moyen) selon la politique de réhabilitation.

  Il est à noter que certains travaux sont toujours souhaitables du fait de leur faible coût.

La première étape est en effet primordiale, car le résultat des deux étapes suivantes dépendent de l'exactitude et de la précision des données recueillies. La deuxième étape est souvent plus facile à réaliser, mais du fait des données disponibles qui peuvent être de nature très différente, le traitement peut être souvent assez délicat. Quant à la troisième étape, une considération purement thermique n'est pas suffisante, car d'autres facteurs peuvent exercer des influences sur le choix technique. Ces facteurs sont souvent économique, social. Dans beaucoup de cas, une opération de diagnostic thermique n'est pas réalisée seule, mais dans le cadre de réhabilitation plus générale du bâtiment (acoustique, mécanique etc...).

Pour certains types de diagnostic comme on le verra plus loin, cette division en trois étapes n'est pas appropriée. C'est notamment le cas pour le diagnostic du type "rapide" dont le but n'est pas de proposer des solutions d'amélioration, mais de localiser les éventuelles "sources" d'économie.

I.2.1 Les différents types de diagnostic thermique

Comme on a signalé plus haut, il existe une relation coût-précision dans les diagnostics, ceci est le cas pour un diagnostic thermique de bâtiment.

L'existence de cette relation nous conduit à classer trois types de diagnostic (cette classification peut être arbitraire, mais correspond à une certaine réalité dans la pratique des diagnostics effectués en thermique de bâtiment, et dans un sens plus large dans le domaine de gestion technique des patrimoines) :

- diagnostic *simplifié*, qui correspond à un niveau "faut-il faire ?". Ce type de diagnostic est souvent appelé un pré-diagnostic ;
- diagnostic *intermédiaire* qui correspond à "que faire ?". Ce type de diagnostic correspond au type de diagnostic défini par AFME ;
- enfin diagnostic *complet* qui répond à la question "comment-faire ?".

Cette classification aura des conséquences importantes dans la suite pour les outils que l'on propose.

On constate que les diagnostiqueurs sont souvent des spécialistes dans un domaine (un diagnostiqueur-installateur se focalisera sur les équipements, tandis qu'un diagnostiqueur architecte analysera plus en détail l'enveloppe). Dans son investigation, un spécialiste en équipement recueille des données sur les équipements d'une façon précise et fiable, alors qu'il peut négliger facilement les données sur l'enveloppe du bâtiment. Le fait de définir ces différents niveaux de diagnostic, et d'introduire une cohérence entre les différents postes du bâtiment aura certainement pour effet de mieux répartir les moyens mis en œuvre, et d'obtenir des résultats plus cohérents à la précision voulue.

Ainsi, on peut résumer par la figure I.2 les différents niveaux d'audit.

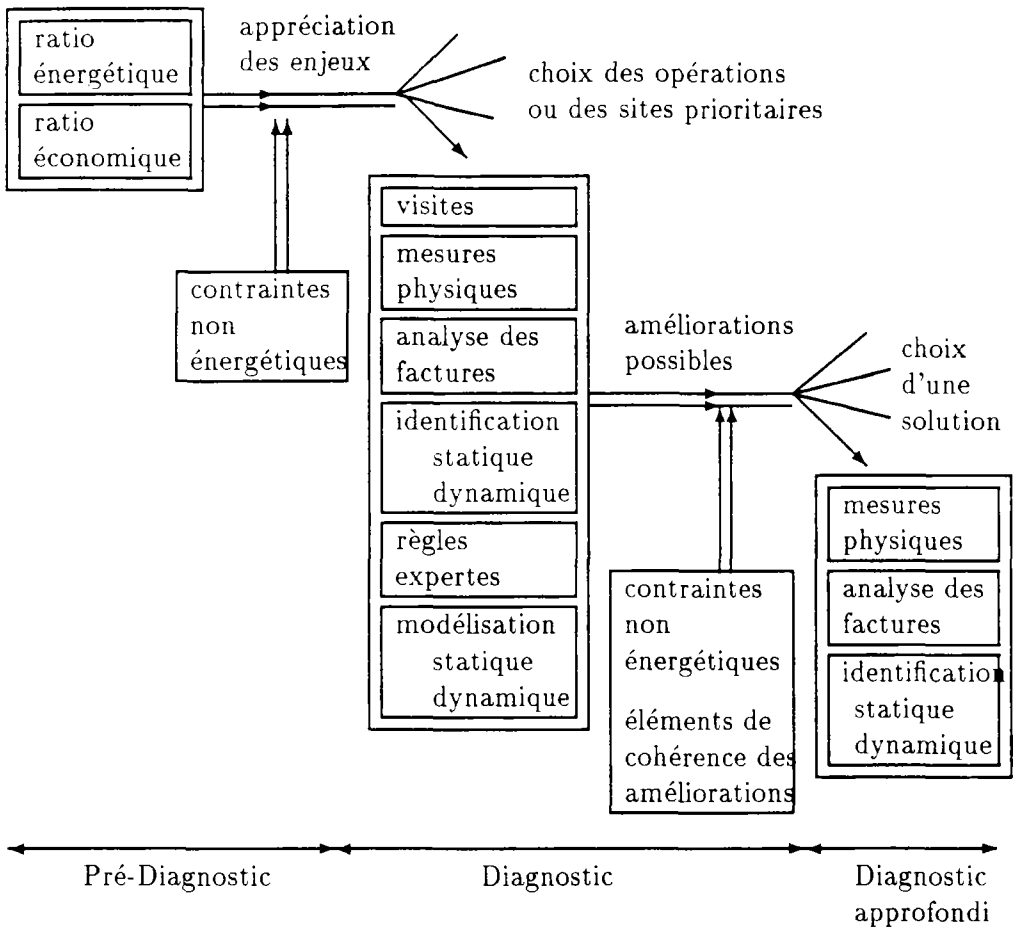


Figure I.2: Les différents niveaux d'audit

I.2.2 Analyse de différentes approches et outils existants en diagnostic thermique

Nous allons analyser les différents modèles utilisés dans les différents types de diagnostic. En dehors de ces trois types de diagnostic cités plus haut, on analysera également certains diagnostics pratiqués dans des bâtiments spéciaux, tels que les bâtiments de tertiaire, qui ont souvent des préoccupations et des objectifs différents de ceux couramment réalisés dans le secteur résidentiel [4, 13].

Les outils, développés depuis les crises énergétiques et souvent sous forme de logiciels informatiques, sont pour la plupart basés sur des modèles thermiques utilisés en conception des bâtiments neufs, c'est-à-dire les réglementations, les DTU et les autres règles techniques [34, 36, 37],. On peut classer ces outils selon les catégories suivantes qui correspondent plus ou moins à notre classification de trois niveaux de diagnostic :

- outils simplifiés qui sont d'une grande simplicité d'usage. En général, ces outils

utilisent des méthodes simples tels que les ratios. Ils donnent dans des cas bien précis des résultats assez satisfaisants, par exemple dans les cas où le gestionnaire veut connaître s'il est nécessaire d'effectuer un diagnostic thermique. Il faut signaler que ce genre d'outils sont à utiliser avec beaucoup de précautions, car ils demandent à l'utilisateur d'avoir une bonne connaissance de l'existant [7] ;

- outils utilisant les modèles thermiques statiques traditionnels, par exemple calcul des coefficient des déperditions de paroi K et GV, des pertes de distribution... [36, 34] ;
- enfin des outils utilisant les modèles de simulation dynamique. Ces outils peuvent évidement donner des résultats très précis, mais demandent une mise en œuvre de moyens sophistiqués [52, 39].

Méthodes utilisées dans le pré-diagnostic

L'approche de pré-diagnostic consiste à localiser les "sources" d'économie d'énergie possibles pour un bâtiment ou un ensemble de bâtiments. Cette approche est souvent utile, et dans certains cas fondamentale.

Les méthodes utilisées dans ce type de diagnostic sont en général d'utilisation simple et basées sur des expériences de professionnels. Parmi ces méthodes, celles utilisant des *ratios* sont les plus courantes, dont on peut citer la méthode de ratios utilisée par le cabinet Bernard [7].

A partir de certains paramètres essentiels, cette méthodes utilise un certain nombre de rapport entre ces paramètres afin de déterminer les sources d'économie possibles et les améliorations éventuelles. La méthode nécessite en effet une base de données relativement importante sur les bâtiments déjà analysés.

Ce genre d'approche s'adapte sans doute bien à des réalisations de systèmes experts qui exploiterait d'une manière efficace la base de données en question.

Méthodes utilisées dans le diagnostic type AFME

Les méthodes sont des méthodes analytiques. Un diagnostiqueur construit un modèle explicatif du bâtiment (occupé et chauffé). A partir de ce modèle le diagnostiqueur peut reconstituer la consommation annuelle de l'énergie et calculer l'économie d'énergie après la proposition de certaines améliorations. La plupart des méthodes utilisent un modèle statique de calcul, mais les modèles dynamiques sont également utilisés dans des cas encore très rares.

Modèles statiques

Les modèles statiques sont souvent proches de ceux utilisés en conception de bâtiment neuf.

On peut distinguer deux catégories d'outils qui sont :

- outils destinés aux professionnels ;
- outils destinés aux grands publics

Pour les outils destinés aux professionnels, la structure des outils sous forme de logiciel est en général la suivante :

- Saisie des données :
  - le bâti ;
  - le site et le climat ;
  - les équipements ;
  - l'occupation ;
  - les aspects économiques ;
- calcul d'une consommation théorique...
- proposition d'améliorations et classement des solutions par le temps de retour ;
- proposition de plusieurs programmes de travaux techniques cohérents, choisi selon leur rentabilité :
  - rentabilité élevée, coût limité, réalisation rapide ;
  - coût et rentabilité intermédiaires ;
  - coût élevé, rentabilité moins importante
- description de chaque intervention envisagée.

Ce type de diagnostic se réalise en général par des relevés simples et quelques mesures sur site rapides (par exemple : le taux de CO_2, la température de fumée de la chaudière etc). Dans des cas spéciaux, ces relevés et ces mesures peuvent être également complétés par des mesures beaucoup plus approfondies avec des instruments spécialisés. Certains équipements de mesure sont spécifiquement conçus à cet effet qui demande une mise en œuvre relativement simple sur le site, on peut citer les "boites blanches" de CoSTIC (référence ? ? ?).

En france, le "Guide de diagnostic thermique" de AFME, préconise l'évaluation des différentes pertes par des bilans thermiques qui donne une meilleure définition que le rendement classiquement utilisé dans d'autres méthodes. Ce guide propose également des méthodes originales pour évaluer le taux de renouvellement d'air et les apports gratuits.

En ce qui concerne les outils destinés au grand public, l'accent est surtout mis sur la simplicité d'utilisation. Le principe de ces outils repose sur l'examen des consommations énergétiques, poste par poste dans l'optique soit d'une substitution d'énergie soit d'une rénovation. Ce type d'outils sont donc plutôt des outils d'aide à la décision.

Modèles dynamiques

Les modèles dynamiques sont utilisés pour simuler le comportement du bâtiment et des équipements surtout en ce qui concerne les différentes inerties. Ils demandent souvent des mises en œuvre complexes et leur usage est limité à quelques professionnels expérimentés. On trouve l'utilisation de ces modèles dans les bâtiments et les installations où présente un enjeu financier important.

Le principe de ces outils reposent donc sur des modèles thermiques dynamiques utilisant des méthodes de différences finies et des éléments finis, on peut également trouver les outils issus de la synthèse modale.

Les méthodes d'identification

Les méthodes d'identification estiment globalement les caractéristiques thermiques d'un bâtiment et de son installation de chauffage à partir des mesures. Dans ces méthodes, on distingue également les méthodes statiques et les méthodes dynamiques. Les méthodes statiques (connues sous l'appellation de *signature énergétique*) nécessitent souvent de longue période d'observation (plusieurs mois), mais peuvent se dispenser d'instrumentation, alors que les méthodes dynamiques demandent un temps plus faible (de l'ordre d'une semaine), mais nécessitent une instrumentation.

Les méthodes d'identification statiques

Il existe un grand nombre de méthodes sous l'appellation de la signature énergétique. La méthode ANAGRAM [33] et PRISME [55] en sont deux exemples. La première, qui consiste à tracer des courbes de consommation en fonction de degrés-jours pour les périodes normales, présente des caractéristiques intéressantes en ce qui concerne les équipements et a déjà été utilisée pour de nombreux diagnostics. La seconde est destinée à traiter des informations pour la constitution d'une base de données BECA qui est un outil de synthèse puissant.

Les méthodes d'identification dynamiques

L'identification en régime dynamique des caractéristiques physiques des bâtiments consiste à présupposer un type de modèle d'évolution thermique de ceux-ci, et à identifier les paramètres. Les méthodes, notamment l'algorithme de calcul dérivent directement de celles utilisées en automatique.

Diagnostic en tertiaire

Les bâtiments et leur installation dans le cas des tertiaires sont souvent complexes, et demandent de méthodes de diagnostic spécifiques. Le tertiaire recouvre un grand nombre de bâtiments tels que les bâtiments scolaires, les hôpitaux, les bureaux, les commerces etc....

Du fait de la complexité du système, il est nécessaire de pouvoir localiser d'une façon fiable le *gisement* d'économie possible avant de procéder au diagnostic. Selon le type de bâtiment, le besoin de consommation peut être très différent, par exemple la consommation de l'eau chaude sanitaire est un poste important dans le cas d'un hôpital, alors qu'elle n'en est pas du tout la même dans le cas d'un bureau.

Pour parvenir à localiser les gisements, les méthodes de type *ratio* sont très souvent utilisées, dans un premier temps. A partir de cette première localisation de "sources" d'énergie, d'autres outils plus sophistiqués sont utilisés (les méthodes statiques, dynamiques ou d'identification citées plus haut).

Diagnostic spécialisé

A partir d'un pré-diagnostic ou d'un diagnostic de type classique, il peut y avoir des situations où un certain nombre de paramètres doivent être précisés. Dans ces cas, il y a souvent besoin d'un diagnostic spécialisé où font intervenir des mesures spécifiques de certains équipements.

L'approche par système expert

Le diagnostic est a priori un domaine qui se prête assez bien à la réalisation de systèmes dits *experts*. Ceci est dû à la nature de l'opération qui demande au diagnostiqueur une bonne expérience du domaine pour déceler certaines pannes d'un système. Dans ce domaine, on peut citer les recherches menés pour la réalisation d'un système expert dans les maisons individuelles [61, 10, 62].

Cette réalisation a pour but de résoudre le problème de cohérence de données lors de saisi, et de réduire le coût de diagnostic dans les maisons individuelles. [10] donne les éléments d'expertise que l'on peut rencontrer tout au long de diagnostic :

- Dans la première phase de diagnostic. Il s'agit souvent des règles permettant une estimation des données non relevées sur site : constitution des parois, taux de renouvellement d'air etc.... La formalisation de ces règles semble difficile ;
- Dans la deuxième étape de diagnostic. C'est la démarche de l'expert pour identifier les anomalies et pour proposer des améliorations adéquates de ces anomalies ;
- Dans la dernière étape. L'expertise permet de proposer un ensemble de travaux d'amélioration cohérent à partir de ceux proposés dans la deuxième phase.

Deux maquettes de systèmes experts ont été réalisées sur deux langages de représentation différents : l'un est couche objet sur PROLOG et l'autre une couche objet sur LISP.

[10] montre également que le domaine de diagnostic est un domaine très complexe, et que les connaissances dites *expertes* dans ce domaine ne sont pas formalisées et restent difficile à formaliser.

En conclusion, on peut dire qu'un système expert sera sans doute bien adapté pour savoir s'il y a une *panne*, par exemple, savoir les causes de la condensation sur les mûrs et les améliorations. Mais le but des diagnostics thermiques de bâtiment est plutôt d'établir un *bilan de santé* du bâtiment, ce qui nécessite l'utilisation des modèles physiques précis qui existent déjà.

I.2.3 Conclusion sur les outils existants

On peut résumer les caractéristiques des outils existants les plus couramment utilisés par un schéma de calcul qui peut être du type suivant :

- reconstitution des déperditions moyennes pour un degré d'écart entre ambiance intérieure et extérieure (calcul d'un coefficient GV), à partir des règles Th-K et Th-G (parfois utilisées d'une manière simplifiée) ;

- détermination des besoins de chauffage à partir des déperditions, des degrés-jours du site, de la température moyenne intérieure et des apports internes (calculs du type Th-B) ;
- estimation des différentes pertes thermiques liées à la génération de chaleur, à la distribution et à l'émission ;
- reconstitution d'une consommation moyenne annuelle, comparaison avec la ou les consommations constatées (relevés ou factures) et modification éventuelle de certaines données de façon à obtenir une concordance satisfaisante entre les deux valeurs de la consommation énergétique annuelle (cette procédure est souvent appelée "bouclage de bilan") ;
- calcul des gains énergétiques et économiques liés aux améliorations envisagées de l'enveloppe et de l'équipement ;
- proposition d'un ou de plusieurs scénarios de travaux d'amélioration, présentant une cohérence technique, assorties des résultats correspondant en terme énergétiques et économiques.

Les limites de ces outils sont principalement de deux ordres : les méthodes de calculs utilisées et l'utilisation des données telles quelles.

Les méthodes utilisées font souvent appel aux méthodes réglementaires, qui sont bien adaptées à la conception de bâtiment neufs, sans une réelle adaptation.

La prise en compte de l'intermittence et le calcul des gains liés à cette intermittence sont très limités.

L'un des points faibles des méthodes utilisées est celui du "bouclage du bilan". On constate que cette opération relève souvent de l'arbitraire du diagnostiqueur sans une véritable analyse physique.

Les auteurs des outils existants semblent ne pas être préoccupés par le problème de la nature et de l'accessibilité des données. Ils font comme si l'utilisateur possède toutes les informations qu'ils ont besoin pour effectuer les calculs. Ils ne se posent pas en général de question quant à la fiabilité des données ni celle du résultat obtenu. Tout cela est laissé à l'appréciation de l'utilisateur.

I.3 Spécificités des outils de diagnostic thermique

I.3.1 Les données dans le bâtiment

La complexité de l'objet bâtiment augmente les problèmes évoqués liés à la disponibilité et à l'incomplétude des données lors d'un diagnostic thermique. Cette complexité est réelle quel que soit le point de vue technique : structure de bâtiment, métré, diagnostic, thermique etc., et elle est liée

- à la configuration géométrique tri-dimensionnelle (la séparation entre les espaces, les contiguïtés entre les espaces...) ;

- aux liens physiques et morphologiques entre les constituants de l'enveloppe ;
- au descriptif du mode constructif, des composants constitutifs de l'enveloppe, des matériaux et des équipements mis en œuvre.

Cette complexité est augmentée par celle de son environnement : climatique, social, économique, urbain...

Le diagnostic thermique de bâtiment n'échappe pas à cette complexité.

Devant la diversité de ces éléments et les multiples relations existantes entre eux, nous pouvons établir une classification générale des données qui peuvent intervenir dans un audit. Dans le but d'ordonner la saisie des données, de connaître leur utilité dans les modèles thermiques, il semble intéressant de classer ces données selon les catégories suivantes [16] :

- *l'origine*. Selon l'origine d'une donnée, on peut déterminer souvent d'une manière assez précise, les imprécisions et les incertitudes qui lui sont attachées. Une donnée peut être obtenue par l'un des moyens :
  - observation sur le terrain,
    - * mesure in situ,
    - * estimation au son ou à l'aspect, (pour la nature des matériaux par exemple).
  - relevé à partir d'un plan ou d'un descriptif technique, (pour un détail de construction),
  - déduction ou une implication,
  - réglementation en vigueur à l'époque de la construction.
- *la nature*. Une donnée peut être :
  - numérique ou alpha numérique,
  - quantitative ou qualitative.
- *le niveau d'utilité*. Une donnée peut être : impérative, optionnelle ou facultative en fonction du modèle et du degré de précision de celui-ci.
- *les incertitudes et les imprécisions*. Certaines informations recueillies peuvent être certaines et précises en toute circonstance. D'autres sont en général soit incertaines, soit imprécises ou les deux à la fois.

I.3.2 Les besoins des outils

Les besoins actuels des outils de diagnostic peuvent être classés par deux types, ceux des thermiciens et ceux des gestionnaires [14].

Le gestionnaire assure le suivi technique et économique des bâtiments, et a la tâche d'améliorer le confort de ses occupants au moindre coût, par conséquent il est naturellement bien placé pour détecter et prévenir les désordres et le dysfonctionnement qui puissent apparaître et également pour envisager les améliorations. Pour parvenir à son objectif, le

gestionnaire a besoin des outils qui lui permettent d'analyser la situation des bâtiments et de savoir s'il est nécessaire d'engager un diagnostic. Les outils qu'il attend sont donc de type "outils d'aide à la décision légers".

Les besoins des thermiciens en outils de diagnostic peuvent être de deux catégories :

- des outils d'analyse globale des consommations de chauffage reposant classiquement sur des modèles explicatifs à une variable climatique ;
- des outils de reconstitution des consommations énergétiques, permettant de chiffrer les économies associées aux différents types d'améliorations envisageables.

I.3.3 Les outils à développer

L'analyse précédentes des méthodes et des outils de diagnostic thermique de bâtiment nous montre clairement qu'au point de vue méthodologique, il existe un grand nombre de méthodes adaptées à des situations particulières. Les outils logiciels reprennent ces méthodes. Cependant, ces outils présentent encore des lacunes dues aux difficultés de prendre en compte les données réelles du bâtiment, et servent plus comme outils de calcul que comme des véritables outils d'aide au diagnostiqueur.

Les outils à réaliser pour le diagnostic thermique doivent également s'inscrire dans un cadre plus général de gestion technique de patrimoine. De ce fait, ils doivent satisfaire d'autres conditions telles ouvertures vers d'autres outils de gestion ou de diagnostic non thermique.

Il nous semble qu'un outil de diagnostic doit répondre principalement aux exigences suivantes :

- prendre en compte la relation coût-précision telle que nous l'avons définie, donc pouvoir tester la cohérence entre les données et les résultats escomptés, c'est-à-dire adapter le type de modèle de reconstitution des consommations à la nature et à la précision des données disponibles ;
- exploiter les données communiquées directement par le diagnostiqueur sous leur forme la plus naturelle ;
- pouvoir *raisonner* avec les données disponibles, donc avec les données partielles, et avoir une relative souplesse ;
- permettre au diagnostiquer de mieux maîtriser la procédure de "bouclage de bilan", afin d'améliorer la fiabilité dans le résultat final ;
- aider d'une manière intelligente le diagnostiqueur dans les saisis des données.

Ces exigences supposent donc l'intégration de savoir-faire des experts de diagnostic, et l'utilisation des modèles de traitement appropriés et les outils informatiques puissants pour construire des interfaces *intelligentes*.

Dans l'esprit de ces exigences, un certain nombre de modules informatiques peuvent et doivent être réalisés et peuvent intégrés comme l'a indiqué [14], par la figure I.3. Nous

nous intéressons dans le stade actuel de la recherche au deux modules qui sont le module de saisi spécialisé et le module de calcul et de bouclage de bilan.

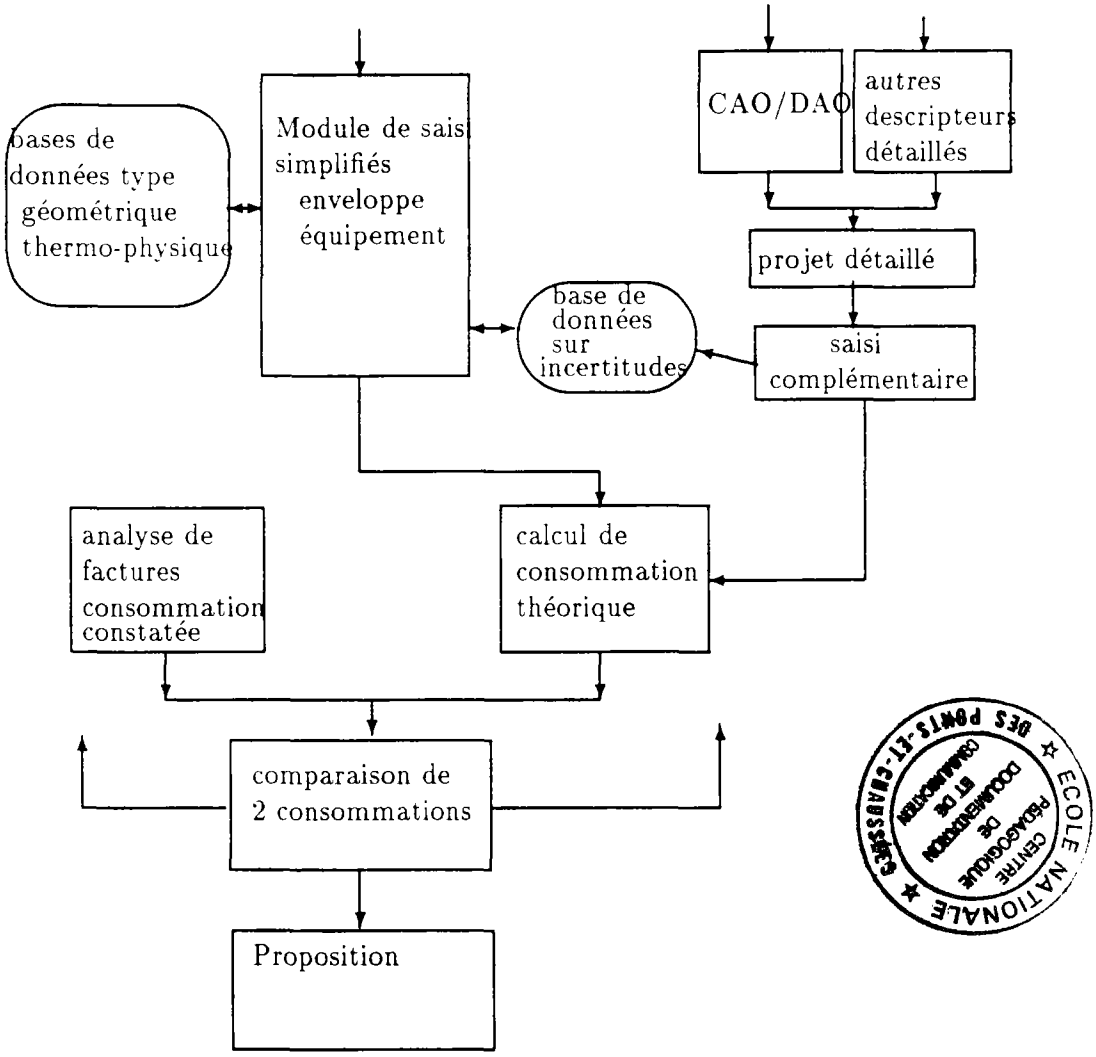


Figure I.3: Différents composants d'un outil de diagnostic thermique

Les modules spécialisés de saisi

Les outils existants présentent dans l'ensemble une ergonomie fort appréciable pour les utilisateurs. Mais une réelle amélioration de l'interface utilisateur doit se faire par une meilleure compréhension de mécanisme d'interaction des objets mis en jeu. Dans le cas de diagnostic thermique de bâtiments existants, il apparaît clairement certaines relations existantes entre les différents composant du bâtiment tant au point de vue de la géométrie

qu'au point de vue de propriétés thermo-physiques.

Une des difficultés de saisi réside dans celui des caractéristiques géométriques. Il est souvent difficile de décrire toutes les contiguïtés de différentes zones d'un bâtiment. Un module spécialisé dans le saisi des métrés est sans conteste d'une grande utilité, et ceci est tout à fait faisable, car les bâtiments existants présentent de typographie que l'on peut classer d'une façon simple (on peut parler par exemple de bâtiment en forme de U, L etc...). Cette constatation est également valable pour les caractéristiques thermo-physiques des matériaux. Nous allons dans la suite de cette partie décrire une façon possible de réalisation d'un module de saisi géométrique, une réalisation concrète est décrite dans [16].

Module de saisi de métré

Un module de saisi de métré spécialisé (ou plus simplement *métreur*) en audit doit satisfaire un certain nombre de conditions qui sont :

- les entités manipulées doivent être à fois des entités géométriques et thermiques. Elles comprennent toutes les données géométriques nécessaires à un audit ;
- le métreur doit être utilisable pour un grand nombre de type de bâtiments, sans toutefois prendre en compte tous les types de bâtiments, car ceci sera très difficile par le grand nombre de formes existantes, notamment les bâtiments *modernes*.

Cet outil doit également satisfaire un certain compris entre la simplicité et la souplesse d'utilisation, et son coût de mise en œuvre et d'utilisation.

La solution graphique pour la réalisation d'un tel outil est bien adaptée en ce qui concerne l'exigence sur la simplicité et la souplesse, mais demande une mise en œuvre importante et un matériel informatique sophistiqué. Une description syntaxique, avec les graphiques servant à contrôler la saisie, permet de satisfaire dans une large mesure les exigences de départ. C'est cette dernière solution qui est adoptée.

La notion de *zone thermique* est couramment utilisée et familière en thermique de bâtiment. L'entité de base, à la fois géométrique et thermique sera donc des zones thermiques dans le métreur.

Dans le saisi des zones thermiques, deux solutions peuvent être envisagées : par forme type ou par une synthèse des différents *blocs* géométriques. Dans [16], cinq formes type sont définies :

- parallélépipède à base rectangulaire ;
- parallélépipède à base quelconque ;
- forme en L ;
- forme en U ;
- bâtiment forme une cours fermée.

Dans la saisi par la synthèse, la notion de *blocs* est introduite. Un bloc est une entité géométrique régulière et simple à décrire. Une zone est formée d'un ou plusieurs blocs. Les blocs et les zones sont définis par leurs attributs de définition (forme, longueur...) et de liaison (contiguïtés avec les autres blocs ou zones).

Certaines formes géométriques complexes peuvent être saisies en faisant des approximations (certains angles non droits approximés en angles droits n'affectent que très peu le résultat). Et il existe également un certain nombre de règles qui peuvent être utilisées pour déduire d'une façon rapide certaines paramètres ou vérifier la cohérence. Par exemple, le pourcentage et la répartition des surfaces vitrées sont souvent fonction de l'époque de construction, du site et de l'orientation, les imprécisions de mesure peuvent être déduites à partir de la nature de relevé (sur plan, sur site etc...).

La réalisation du module montre clairement l'intérêt de cette approche de saisie spécialisée. Le nombre d'informations à saisir diminue sensiblement par rapport aux approches classiques. Ainsi le diagnostiqueur peut consacrer plus de temps aux autres informations du bâtiment qui sont souvent plus difficiles à obtenir.

Les modules de calcul

Les modules de calculs doivent exploiter les informations obtenues par les différents modules de saisi spécialisés et fournissent les résultats sur les déperditions, les besoins en chauffages, les consommations, effectuent une analyse fine des factures et la comparaison de deux consommations. Les calculs sont menés en prenant évidement en compte les incertitudes et les imprécisions sur les données, et préservent dans la mesure où les données le permettent, la cohérence entre les données et le résultat. Le module de bouclage de bilan doit également indiquer au diagnostiqueur s'il faut poursuivre la suite du diagnostic (dans le cas de la validation de l'audit, donc de différents paramètres et de modèles) ou s'il faut approfondir certaines recherches de l'audit (dans le cas contraire).

I.4 Conclusion sur les outils de diagnostic

Nous avons vu que les problèmes posés par l'audit sont essentiellement les suivants :

- le nombre important des objets et des données manipulés nécessite une bonne structuration dans la réalisation d'un outil ;
- le traitement des données partielles est indispensable, si on veut donner une meilleure fiabilité de résultat et une meilleure cohérence des données ;
- la cohérence des données et l'utilisation des informations telles quelles imposent le choix de modèle approprié à chaque phase de calcul. C'est-à-dire que l'outil doit choisir en fonction des données disponibles et de la précision voulue le modèle de traitement, et ceci nécessite de la part de cet outil, une structure de contrôle adaptée et souple permettant ce choix.

On constate également que la plupart de modèles de traitement en audit sont des modèles mathématiques bien explicités, sauf les modèles utilisés dans le cas de pré-diagnostic qui sont le plus souvent sous la forme que l'on peut qualifier de *règles expertes*.

La recherche en intelligence artificielle et en génie logiciel, comme on le verra au chapitre IV ouvre de nouvelles perspectives pour la représentation des connaissances et la programmation.

Nous pouvons à partir de ces éléments d'analyse fixer l'objectif concernant la réalisation d'un outil de diagnostic qui peut être résumé par les éléments suivants :

- expérimentation d'un style de programmation combinant les approches objets et acteurs ;
- prise en compte et traitement des données numériques ou littérales imprécises,
- prise en compte et traitement des données ou assertions incertaines,
- acceptation a priori de données inconnues, chaque fois que ce manque d'information est contournable,
- co-existence de plusieurs modèles thermiques mis en oeuvre selon les données disponibles, et gestion de la cohérence données-modèles,
- inter-activité du dialogue entre l'"auditeur" et le système de traitement de l'information.

Ainsi une maquette informatique est réalisée dans le but de montrer l'intérêt de notre démarche. Cette réalisation est décrite au chapitre VI.

Chapitre II

Univers physique de l'audit : objets et modèles

Penser un objet et connaître un objet, ce n'est donc pas une seule et même chose. La connaissance, en effet, suppose deux éléments : d'abord le concept, par lequel en général, un objet est pensé et ensuite l'intuition par laquelle il est donné.

— *E. Kant*

Dans le chapitre I, nous avons eu une vue assez générale du diagnostic thermique de bâtiments existants. Nous allons dans ce chapitre parcourir l'ensemble des objets physiques que l'on peut rencontrer lors d'une opération de diagnostic et proposer une structuration possible.

Notre structuration des objets est destinée à la réalisation d'une maquette informatique d'un outil de diagnostic, et présente donc certaines particularités liées au type d'audit et au type de bâtiments, et a donc une préoccupation différente des structurations plus générales telles que [98].

Dans ce chapitre, nous allons aussi présenter un exemple de modèles physiques régissant les divers objets. Ce sont des modèles d'évaluation physico-mathématiques à distinguer de modèles de représentation. Ces modèles seront utilisés dans la réalisation de la maquette décrite au chapitre VI [1].

II.1 Objets physiques de l'audit

Nous nous intéressons à un bâtiment existant occupé et chauffé, donc dans ce qui suit, nous entendons toujours par le terme *bâtiment* un bâtiment existant occupé et chauffé dans un

[1] Le terme modèle est utilisé ici dans un sens général. Les modèles mathématiques, les règles expertes etc sont tous regroupés sous ce terme.

environnement climatique donné.

Ce bâtiment est composé d'un grand nombre d'objets :

- l'enveloppe du bâtiment qui est partitionnée en différentes zones thermiques si le bâtiment n'est pas chauffé uniformément ou si le bâtiment est occupé différemment ;
- l'environnement du bâtiment, c'est-à-dire le site sur lequel le bâtiment est construit, le climat du site etc... ;
- les équipements de chauffage comprenant celui de production (différents types de chaudières), de distribution et d'émission.

A part ces principaux objets, nous devons également tenir compte d'un certain nombre de paramètres liés globalement au bâtiment étudié qui sont principalement de deux catégories :

- les paramètres nécessaires à la définition du bâtiment tels que le niveau d'audit, date de construction... ;
- les paramètres à calculer, la consommation annuelle d'énergie, par exemple.

Nota : Dans notre description, nous nous intéressons uniquement à la consommation d'énergie liée au chauffage, la consommation par l'eau chaude sanitaire n'est pas considérée dans notre stade de recherche, bien que dans de nombreux cas, l'eau chaude sanitaire constitue une consommation d'énergie importante et en même temps puisse présenter des difficultés tant au plan méthodologique qu'au plan d'observation.

La structuration des objets dépend fortement des besoins des outils à réaliser, pour cela, on est amené à préciser les choix suivants :

- les bâtiments que l'on considère sont des bâtiments à caractère dominant résidentiel qui peuvent comporter éventuellement des parties tertiaires. Par exemple, un immeuble résidentiel comportant un rez-de-chaussée commerçant ;
- trois niveaux d'audits sont pris en compte, l'audit rapide, intermédiaire, et détaillé qui sont définis au chapitre I. Les deux derniers niveaux (intermédiaire et détaillé) sont très souvent confondus au stade actuel.

Précisons également que dans notre description, nous nous limitons aussi aux choix suivants :

- seules les chaudières fuel sont étudiées, les autres types de chaudières (électriques, à gaz...) sont moins courantes dans les bâtiments anciens, seront étudiées plus tard ;
- en ce qui concerne la ventilation, seule la ventilation naturelle est considérée ventilation mécanique n'est pas considérée ;
- les équipements sont pris en compte au niveau global du bâtiment, i.e, on considère que toutes les zones thermiques chauffées possèdent les mêmes équipements de chauffage.

Remarque : La description des objets et des modèles thermiques est donc également limitée par ces choix, mais ceci est sans influence sur la structuration des données.

Dans notre description, nous utiliserons les termes *composant* et *paramètre* pour désigner deux types différents d'attributs d'un objet. Un composant est un attribut particulier qui sera défini comme un objet à part, alors qu'un paramètre est un attribut qui aura une valeur (dans l'état actuel, on ne distingue pas ce qui est calculé et ce qui est donné). Cette distinction facilitera la définition de différents objets. Bien qu'en terme de programmation par objet, un paramètre soit aussi un objet comme les autres objets.

Objet :	bâtiment	
	Composants :	
		zones thermiques
		site climat
		équipement
	Paramètres :	
		nom
		niveau audit
		type bâtiment
		date construction
		consommation annuelle facturée
		consommation annuelle théorique

Figure II.1: L'objet bâtiment chauffé et occupé

L'objet **bâtiment** comportera principalement deux méthodes : comparaison de deux consommations et le calcul de consommation théorique (cf. section II.2.2).

II.1.1 Zones thermiques

Une zone thermique est un volume géométrique thermiquement uniforme comme défini au I.3.3. Un bâtiment peut être constitué par plusieurs zones. On distingue deux sortes de zones :

- les zones thermiques normales, c'est-à-dire, chauffées et occupées ;
- les zones particulières. Ce sont les zones qui ne nécessitent pas une étude détaillée de leur comportement thermique. Il y a essentiellement les combles, les cages d'escaliers, les autres bâtiment contigus. On peut également considérer le sol et l'ambiance extérieure comme des zones particulières qui ont une température constante. La température extérieure est en effet considérée comme un attribut du climat.

La structuration des objets **zones thermiques** peut donc être la suivante : On définit un objet **zone-générale**, et deux objets **zone-particulière** et **zone-normale** (Figure II.2, les flèches

indiquent les liens de parentés entre les objets, ici on peut dire que **zone-normale** est une spécialisation de **zone-générale**).

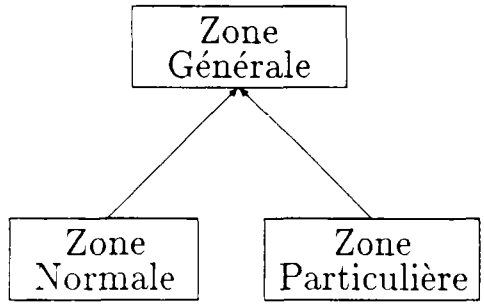


Figure II.2: Relation entre les différentes zones

Une zone particulière est définie (si on ne considère pas l'extérieur et le sol) par :

- son taux de ventilation (le plus souvent donné sous forme qualitative du type *forte*, *faible*, *moyenne*) ;
- son isolation (parois isolées ou non) ;
- son volume ;
- sa température.

On s'intéresse dans la suite plus particulièrement à la zone thermique normale, notée **zone thermique.** Une zone thermique est composée par les éléments suivants :

- enveloppe ou parois, une zone est délimitée par les parois donnant sur une autre zone (l'extérieur, les autres zones, le sol, etc...) ;
- occupation qui décrit le fonctionnement et les occupants de la zone ;
- ventilation.

En effet, la géométrie d'une zone peut être très complexe, nous supposons qu'une zone thermique a une hauteur constante. Cette restriction introduit une rigidité, mais simplifiera l'implémentation des objets au stade de maquette. Et si on tient compte de l'utilisation du métreur géométrique, ce problème ne sera plus posé comme étant essentiel.

Les principaux paramètres à calculer d'une zone thermique sont ses déperditions (ou bien ses besoins bruts en chauffage), et les apports gratuits internes.

Parois

Les parois considérées sont uniquement les parois délimitant les zones thermiques (extérieure, zone non chauffée, autre zone chauffée, autre bâtiment...). Les parois sont soit des parois verticales (murs), soit des parois horizontales (planchers, plafonds).

Objet :	zone thermique	
	Composants :	
		parois
		occupation
		ventilation
	Paramètres :	
		nom
		volume
		déperditions volumiques
		apports gratuits
		date construction
		hauteur de la zone
		surface plancher

Figure II.3: L'objet zone thermique

Une paroi peut comporter deux parties : opaque et vitrée. Les paramètres essentiels d'une paroi sont les dimensions géométriques, la composition, la conductance...

Pour définir les parois d'une façon claire, on est amené à définir les objets suivants :

- l'objet **paroi-générale** dans lequel sont regroupées les caractéristiques communes de toutes les parois ;
- l'objet **paroi** qui comporte une partie opaque et une partie vitrée ;
- l'objet **paroi-verticale** et l'objet **plancher** ;
- et l'objet **paroi-vitrée**.

En résumé, voici les relations entre ces différents objets ainsi définis (figure II.4, les flèches désignent les liens de parenté entre les objets) :

En terme des langages orientés objets, on peut dire que **paroi opaque** et **paroi vitrée** sont deux sous-classes de **paroi générale**, alors que **paroi verticale** et **planchers** sont sous-classes de **paroi**. Et **paroi** comporte deux composants qui sont **paroi opaque** et **paroi vitrée**.

Occupation

Par occupation, on désigne les occupants et leur comportement : est donc également intégrée l'intermittence de chauffage, c'est-à-dire, la programmation du chauffage.

Cet objet comporte les éléments suivants :

- type d'occupation ;
- nombre de jours de chauffage ;

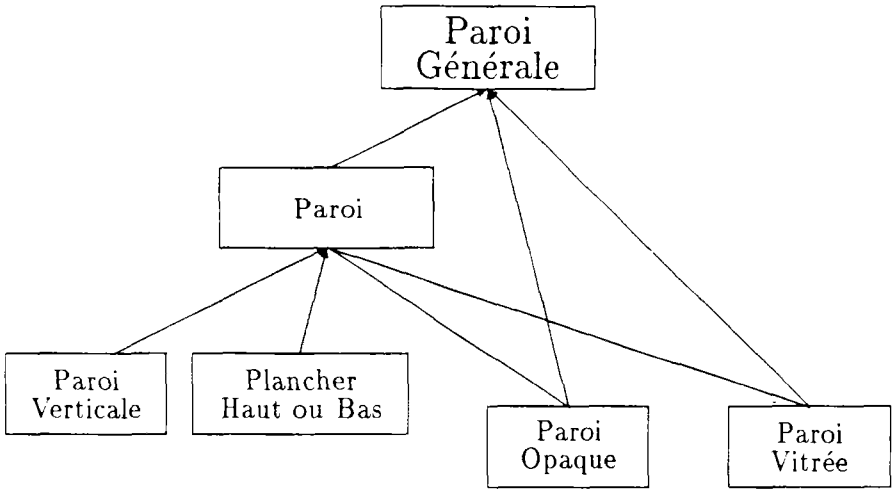


Figure II.4: Les parois

- température intérieure de base ;
- nombre moyens d'occupants permanents ;
- programmation du chauffage etc....

La température intérieure est un paramètre difficile à appréhender. Une attention particulière est portée sur ce problème dans les modèles que l'on utilise.

Ventilation

La détermination des déperditions par la ventilation est l'une des opérations les plus difficiles dans un audit.

Il existe deux types de ventilation : ventilation mécanique et ventilation naturelle. Ces deux types peuvent aussi co-exister dans un même bâtiment. Pour l'instant, seule la ventilation naturelle est programmée.

L'objet **ventilation naturelle** est défini par les principaux éléments suivants :

- taux de ventilation,
- nature et surface des orifices de ventilation.

II.1.2 Equipements de chauffage

Les équipement de chauffage comportent quatre objets : production (chaudière), distribution, émission et régulation.

Production

En ce qui concerne le système de production de chaleur, nous nous intéressons uniquement aux chaudières de type fuel.

L'objet **chaudière fuel** est caractérisé par les éléments suivants :

- caractéristiques générales (type, marque et âge),
- isolation,
- puissance,
- surface,
- nature-métal,
- température fumée, température ambiante, température apparente,
- état entretien,
- taux CO, taux CO_2...

Distribution et émission

Le réseau de distribution de chaleur et l'émission n'ont pas été étudiés d'une manière détaillée dans l'état actuel.

L'objet **distribution** est défini (d'une manière générale) par le type de réseau, les caractéristiques de réseau etc...

L'objet **émission** est défini par le type de radiateur.

Régulation

L'objet **régulation** est défini par son type :

- centrale sur température extérieure avec ou sans correctif d'ensoleillement par façade ;
- centrale sur température intérieure ;
- locale ;
- centrale et locale ;
- manuelle.

II.1.3 Environnement

L'environnement du bâtiment désigne le site et le climat.

Climat

L'objet climat est défini par les éléments suivants :

- sa situation géographique ;
- zone climatique ;
- température extérieure moyenne (mensuelle, annuelle), donc les degrés-jours à base $18°C$;
- son ensoleillement.

L'objet climat peut aussi être défini comme une zone thermique particulière.

Site

Le site indique l'emplacement du bâtiment. Il existe plusieurs sortes de sites qui sont :

- île,
- zone plateau dégagé,
- zone urbaine,
- zone côtière.

II.1.4 Récapitulatif

La figure II.5 résume les objets que l'on manipule. Certains détails ont été simplifiés sur cette figure.

Les paramètres de ces différents objets sont souvent liés entre eux. Leurs relations sont détaillées par les modèles thermiques dans la section suivante.

Un certain nombre d'objets n'ont pas été étudiés dans cette partie. Ces objets devront à l'avenir être ajoutés. L'ajout de ces objets ne devrait pas poser de problème d'organisation générale.

Précisons que le choix sur l'emplacement du système de chauffage au niveau du bâtiment ne poserait pas de problèmes, car s'il est nécessaire de prendre en compte les équipements dans certaines zones thermiques, on peut facilement définir un objet équipement dans une zone. Cette situation peut sans doute se produire dans d'autres cas.

II.2 Modèles thermiques

Dans I.2.2, nous avons parcouru les différentes approches existantes en diagnostic thermique de bâtiment. Nous allons dans cette partie présenter quelques modèles thermiques, dont les principales sources sont [2, 25, 28, 32, 38]. Le but de cette présentation est double :

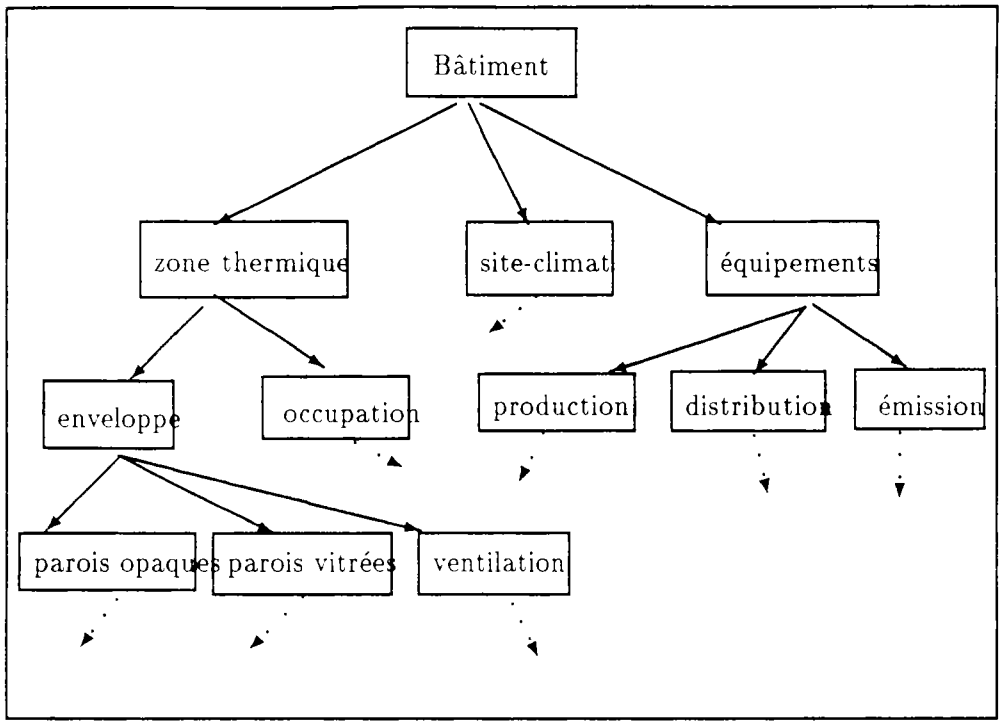


Figure II.5: Organisation des objets

- constituer la base pour la réalisation pratique d'une maquette informatique pour l'audit thermique ;
- montrer clairement les différents types de relation existants entre les paramètres des objets en question.

L'exhaustivité des modèles n'est donc pas le souci premier dans notre présentation. Un certain nombre de modèles ne seront pas étudiés [2].

II.2.1 Type de Modèles

Nous avons vu qu'il existait deux principaux types de modèles pour le calcul thermique : dynamique ou statique. Le diagnostiqueur utilise, sauf cas exceptionnel, les modèles statiques, et ceci pour les raisons suivantes :

1. Les modèles statiques présentent une plus grande facilité de mise en œuvre ;
2. Les modèles dynamiques s'adaptent mal à des calculs effectués à partir de données ayant souvent 10% d'erreur ou davantage ;

[2] Cette partie s'inspire largement du travail effectué par A. Rialhe. Pour plus de détail sur les modèles thermiques, on peut se rapporter à [15].

3. Les modèles statiques ont un temps de calcul plus court. La méthode de validation utilisée jusqu'à présent en est ainsi facilitée : le bouclage du bilan par le recalage consommation calculée-consommation estimée se fait par essais successifs sur les facteurs les plus douteux ;

4. Les modèles statiques utilisent des fonctions numériques dérivables, de forme analytique, qui sont indispensables pour le mode de propagation des imprécisions retenu. Les modèles dynamiques utilisent des fonctions numériques discrétisées dans le temps ou dans l'espace, plus lourdes à manipuler.

Ces modèles statiques travaillent en régime permanent et avec des lois empiriques ou de corrélation qui forfaitisent les régimes dynamiques et les effets de l'inertie.

Pour notre part, nous allons donc décrire un modèle statique de calcul de consommation de chauffage.

II.2.2 Consommation de chauffage du bâtiment

On présente d'abord un modèle statique de calcul de consommation de chauffage du bâtiment, ensuite la comparaison de la consommation théorique et celle de facture.

Calcul de consommation

Le calcul de consommation de chauffage C se fait en tenant compte du besoin brut du bâtiment (c'est-à-dire la somme des besoins bruts de chaque zone), les pertes dues à la génération, la distribution et l'émission et la récupération des apports gratuits des zones. Ceci s'exprime donc par

$$C = B_{brut} + p_{equip} - \tau Ag + C_{aux} \quad (kWh)$$

Et si on exprime les pertes en terme de rendement (avec η_{global} rendement moyen annuel de génération, de distribution et d'émission), on a la formule plus classique :

$$C = \frac{B_{brut} - \tau Ag}{\eta_{global}} + C_{aux}$$

C_{aux} est la consommation des auxiliaires de chauffage (pompes, moteurs...) qui ne sera pas prise en compte dans la suite, et sa réintégration ne pose pas de problème. Le calcul des pertes se trouve en II.2.4. Les apports gratuits seront calculé dans II.2.3. Et le besoin brut s'exprime de la façon suivante :

$$B_{brut} = 0.024 \sum_{zone} GV Dj(T_{int}) \quad (kWh)$$

où GV est le coefficient de déperditions (par paroi et par ventilation en W/K), $Dj(T_{int})$ degrés-jours en base T_{int}, et T_{int} température intérieure moyenne de zone (prise hors période de surchauffe),

On voit également que la consommation C de chauffage s'écrit souvent par

$$C = \frac{0.024 \sum GVDj(T_{int}) - \tau Ag - \beta I}{\eta_{global}} \quad (kWh)$$

avec βI exprimant la récupération de l'intermittence, dans ce cas la température intérieure ne doit pas tenir compte de la variation due à l'intermittence.

Comparaison de consommation calculée et facturée

Si on dispose d'une ou de plusieurs consommations antérieures : $C_i(Dj_i)$ (avec i l'indice de l'année où la consommation est disponible), on peut les utiliser soit telles qu'elles, soit en les corrigeant des variations annuelles des degrés-jours. On peut moyenner les consommations pour éliminer les variations annuelles des degrés-jours. On réalise une correction climatique entre une année particulière et les degrés-jours de cette année.

On peut chercher à valider le modèle de reconstitution par la situation de la valeur de la consommation dans l'intervalle de confiance. Dans le recalage de ces deux consommations, trois cas peuvent se produire (cf figure II.6).

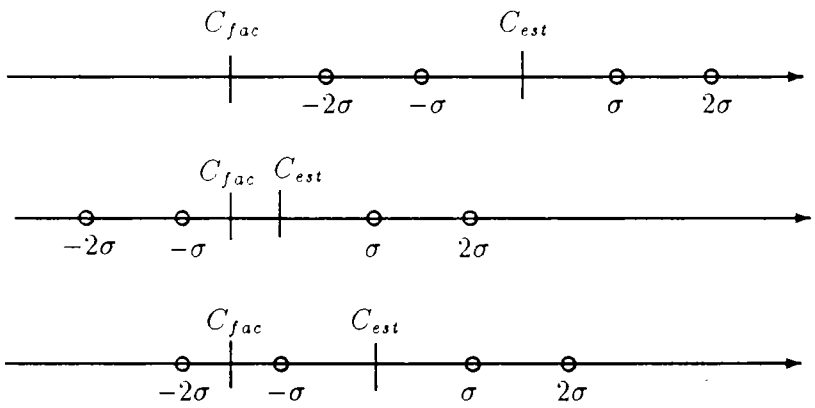


Figure II.6: Différents cas de bouclage de bilan

1. Dans le premier cas, on a $C_{est} - C_{fact} > 2\sigma$ (σ écart-type, cf. III). On est amener à suspecter :

   - une sous-estimation de l'incertitude sur une ou plusieurs données du problème ;
   - une erreur d'analyse physique.

   Un approfondissement de l'audit est alors nécessaire.

2. Dans le deuxième cas où $C_{est} - C_{fact} < \sigma$, on peut estimer que la concordance entre deux consommations est satisfaisante, et que le modèle physique explicatif peut servir de base pour calculer le gain apporté par différentes solutions d'amélioration.

Toutefois cette concordance n'exclut pas certaine compensation entre les différentes erreurs :

3. Le troisième cas est bien entendu difficile pour émettre une conclusion. Il est toujours bon de vérifier certaines données et de l'analyse effectuée.

La suite donne quelques méthodes de calcul pour les déperditions (par enveloppe et par ventilation), les gains des apports gratuits et les pertes par les équipements.

II.2.3 Gains dus aux apports gratuits

Les apports gratuits sont constitués de deux termes distincts, apports solaires (Ag_s) et apports internes (Ag_i). Le calcul des apports gratuits s'effectuent pas zone thermique (c'est un attribut de l'objet **zone thermique**). Leur taux (τ) de récupération est fonction de l'inertie et des déperditions de la zone.

Les apports gratuits internes

La méthode la plus courante ne réalise ni un calcul, ni un inventaire de toutes les contributions aux apports gratuits internes, mais les forfaitise en fonction du nombre d'occupants du logement et d'une valeur moyenne pour les apports dus aux appareils électro-ménagers, aux lampes, etc. Dans le cas de l'habitat, voici la méthode proposée par [2] :

$$Ag_i = 3,5 + 2,5 \times Nb_{occ} \times Nb_{sdc} \quad (kWh/an)$$

Avec Nb_{occ} nombre moyen d'occupants permanents (de **occupation**), et Nb_{sdc} durée de la saison de chauffe (de **zone**).

La durée de la saison de chauffe est une fonction de la zone climatique, et du comportement de l'occupation. En absence d'information précise, on peut souvent adopter les valeurs par défaut suivantes : 230 jours en zone H1, 216 en zone H2 et 181 en zone H3. Un calcul précis des Ag_i peut également être réalisé.

Les apports gratuits solaires

La méthode que nous utilisons pour le calcul des apports solaires est une adaptation directe des règles Th-BV du CSTB [36]. Cette méthode s'applique essentiellement à l'audit du type intermédiaire :

$$Ag_s = Sse \times E = \sum_{parois} [A \times Fts \times Fe \times C_1]E$$

Avec
Sse : surface équivalente sud (m^2),
A : surface de paroi (m^2),
E : ensoleillement moyen annuel sur un plan horizontal (kWh/m^2),
Fts : facteur de transmission solaire,
Fe : facteur d'ensoleillement lié à la présence de masque,
C_1 : coefficient d'orientation et d'inclinaison de la paroi.

Le facteur de transmission solaire Fts

Il est calculé différemment pour une paroi vitrée ou une paroi opaque.

Pour les parois vitrées : on a

$$Fts = k_1 \times S,$$

k_1 est une fonction des rideaux et des fermetures de volets, nous proposons :

k_1 = 0,85 par défaut
= 0,80 s'il y a des volets et des rideaux,
= 0,90 s'il y a des rideaux et pas de volets,
= 1,00 s'il n'y a ni rideaux ni volets.

S est le facteur solaire de la paroi. Dans le cas des bâtiments existants, on peut supposer des fenêtres sans double vitrage, en bois, au nu intérieur, la valeur de S est dans ce cas fixée à 0,55.

Pour les parois opaques :

$$Fts = 0,3 \times \frac{K}{h_e},$$

$\frac{K}{h_e}$ prend dans les cas courants les valeurs suivantes : $K \times 0,018$ pour les murs ; et $K \times 0,015$ pour les toitures.

Le facteur d'ensoleillement Fe

Fe est lié aux masques et a été forfaitisé de la façon suivante :

Fe = 1,0 pas de masque
= 0,9 paroi très peu masquée
= 0,6 paroi assez masquée
(rideau d'arbres à plus de deux mètres des murs)
= 0,3 paroi très masquée
(construction en ville, sur rue, avec immeubles mitoyens)

coefficient d'orientation C_1

C_1 est fonction de l'orientation et de l'inclinaison. On peut prendre les valeurs données par le tableau suivant, obtenus par simplification du tableau analogue des règles Th-BV [36] :

	Orientation		
	SSE-ESE SSO-OSO	ESE-ENE	ENE-NNE ONO-NNO
Inclinaison			
horiz.(0–10°)	0,80	0,80	0,80
faible pente (10–40°)	1,00	0,77	0,70
forte pente (40–80°)	1,10	0,65	0,35
vertical(80–90°)	1,00	0,55	0,25

Ensoleillement E

L'ensoleillement moyen annuel (kWh/m^2) E est un attribut de la zone climatique et prend les valeurs suivantes : 410 en zone H1, 440 en zone H2 et 460 en zone H3.

La récupération des apports gratuits

Deux méthodes peuvent être utilisées pour le calcul du taux (τ) de récupération des apports gratuits :

1. méthode rapide. On considère que, dans le cas de bâtiments existants, le rapport $X = Ag_s/Dep$ est toujours petit devant 1 et le taux de récupération τ, égal à F/X, proche de l'unité. F est la couverture d'apports gratuits. Aussi τ est pris égal à 1.
2. méthode intermédiaire. On distingue les bâtiments à inertie forte et moyenne et calcule le taux de récupération par :

$$\text{inertie forte} \quad F = \frac{X - X^{3,6}}{1 - X^{3,6}}$$

$$\text{inertie moyenne} \quad F = \frac{X - X^{2,9}}{1 - X^{2,9}}$$

avec X défini comme précédemment $X = Ag/Dep$ et $\tau = F/X$. La classe d'inertie "forte" ou "moyenne" peut être déduite à partir d'autres éléments dont les relations restent à expliciter.

Nota : Dans le cas où l'installation de chauffage ne comprend qu'une régulation centrale sur la température extérieure et où aucun dispositif ne permet de tenir compte des apports gratuits (pas de régulation locale d'ambiance), le modèle de calcul des besoins de chauffage retenu aura la forme classique :

$$B = 24GVDj(T_{cons})$$

au lieu de

$$B = 24GVDj(T_{int}) - \tau Ag$$

En effet, l'apport de chaleur gratuit n'est alors pas pris en compte et la température intérieure n'est pas le reflet de la régulation et des apports extérieurs mais seulement d'une régulation aveugle aux apports gratuits ; c'est donc principalement la température de consigne T_{cons} qui détermine les besoins de chauffage.

II.2.4 Les pertes par l'équipement de chauffage

Nous allons étudier dans la suite

- le bilan thermique des chaudières,
- les pertes de distribution en réseau,
- les pertes à l'émission,

Nous nous limitons aux chaudières à fuel domestique, pour le seul usage "chauffage de locaux", et avec émission par radiateurs.

Le bilan thermique des chaudières

On peut calculer pour chaque chaudière de l'installation le *rendement instantané de génération*, noté η_{chaud}, comprenant les pertes par les fumées (P_f), les pertes par rayonnement (P_r) et éventuellement les pertes par imbrulés gazeux (P_g).

$$\eta_{chaud} = 1 - \frac{P_f + P_r + P_g}{P_{nom}}$$

avec P_{nom} = puissance nominale de la chaudière en kW.

On peut utiliser deux méthodes selon le type de diagnostic et les données disponibles :

- les mesures de température et de relevés de taux de CO_2 des fumées sont réalisées, on utilise une méthode par mesure ;
- sinon, on utilise une méthode par valeurs forfaitaires, à partir des informations recueillies au cours d'une simple visite.

Par valeurs forfaitaires

On utilise dans ce cas non pas des valeurs forfaitaires des pertes mais des valeurs forfaitaires de rendement pour une plus grande simplicité.

L'expression de η_{chaud} devient :

$$\begin{aligned} \eta_{chaud} &= 1 - \frac{P_{pertes}}{P_{nom}} \\ &= (\eta_{comb} - C_{cor})\eta_{cal} \end{aligned}$$

η_{comb} = rendement de combustion intégrant les pertes par rayonnement et par imbrulés,

C_{cor} = coefficient de correction intégrant les pertes par balayage et dépôts,

η_{cal} = rendement de calorifugeage intégrant les pertes liées à l'isolation de la chaudière.

D'après [2] nous avons retenu les valeurs suivantes :

$C_{cor} = C'_{cor} + 0,04$ si la chaudière est mal entretenue,

$C_{cor} = C'_{cor}$ si la chaudière est bien entretenue,

les valeurs de C'_{cor} et de η_{comb} étant données dans le tableau suivant.

chaudières	en fonte		en acier	
	η_{comb}	C'_{cor}	η_{comb}	C'_{cor}
brûleur ancien modèle (1400tr/mn)	0,82	0,04	0,82	0,04
brûleur nouveau modèle (2400tr/mn)	0,86	0,04	0,86	0,04
modèle à brûleur intégré	0,90	0,00	0,90	0,00

$\eta_{cal} = 0,98$ si la chaudière est calorifugée
$\eta_{cal} = 0,94$ sinon

Par mesures

On a

$$P_f = \frac{100bf(T_f - T_c)}{[CO_2]\%}$$

bf = coefficient de Siegert. C'est une fonction du combustible,
= 0,59 pour les combustibles liquides, dont le fuel,
$T_f - T_c$ = écart de température entre les fumées (T_f) et l'air comburant (T_c),
$[CO_2]$ = teneur en CO_2 des fumées.

$$P_r = 12S(T_p - T_a)$$

S = surface extérieure de la chaudière en m^2,
$T_p - T_a$ = écart de température entre la paroi extérieure de la chaudière (T_p) et l'ambiance (T_a).

$$Py = \frac{b_i[CO]\%}{[CO] + [CO_2]\%}$$

avec $b_i = 52$ pour le fuel.

Le rendement moyen annuel de génération η_{global} est calculé par :

$$\eta_{global} = \frac{\eta_{nom}}{1 + \eta_{nom}\frac{PA}{P_{nom}}(\frac{1}{\tau_m} - 1)}$$

avec

η_{nom} : rendement nominal, caractéristiques de la chaudière fournis par le constructeur. On utilise pour le rendement η_{nom} la valeur du rendement de chaudière η_{chaud} défini précédemment.

P_{nom} : puissance nominale de la chaudière fournie par le constructeur.

τ_m : est le taux de marche du brûleur, défini par

$$\tau_m = \frac{nbH_m}{nbH_{tot}} = \frac{B_{net}}{P_{br}\ nbH_{tot}}$$

nbH_m = nombre d'heures de marche du brûleur,
nbH_{tot} = nombre d'heures total de la saison de chauffe,
B_{net} = besoins nets de chauffage en kWh sur l'année,
P_{br} = puissance du brûleur en kW.

PA : représentent les pertes à l'arrêt (arrêt du brûleur et émission de la chaudière). Pour des chaudières classiques elles sont estimées à 0,03 fois la puissance nominale (mais peuvent varier de 0,01 à 0,08 fois la puissance nominale).

Les pertes de distribution en réseau

Les pertes de distribution pour une zone traversée et un type de tuyau s'expriment par

$$P_{dis} = kl\ L(T_d - T_a) \quad (W)$$

avec

kl : $1/Re_{th} = \frac{\log(d_e/d_i)}{2\Pi\lambda}$, coefficient de déperditions linéiques,

λ : conductivité thermique du tuyau (W/mK),

L : longueur de tuyau dans la zone (m),

d_e, d_i : diamètre extérieur et intérieur (m) du tuyau (ou du calorifuge si le tube est calorifugé),

T_d : $(T_{dép} + T_{ret})/2$ = température moyenne du fluide au départ et au retour,

T_a : température ambiante moyenne caractéristique de la zone traversée,

P_{dis} est utilisé soit directement dans le calcul de la consommation, soit par l'intermédiaire d'un rendement si on connaît la puissance émise en sortie de chaudière, $\eta_{dis} = (P_{émi} - P_{dis})/P_{émi}$.

Remarque : Dans un audit, les sur-consommations liées à la non idéalité de l'émission et de la régulation, ne sont pas calculées mais on peut considérer qu'elles sont prises en compte de façon implicite par les températures intérieures de zone (hors surchauffe), par les éventuelles hétérogénéités de température entre les locaux et des températures différentes de celles attendues. Les pertes à l'émission pourraient être estimées dans un audit approfondi chaque fois que les corps de chauffe à haute ou moyenne température sont situés sur des parois déperditives non isolées.

Quelques exemples pour l'exploitation des données de nature qualitative

Dans le cas des équipements (tout comme dans celui de la ventilation), un diagnostiqueur recueille souvent des données qualitatives qui ne sont pas directement exploitées par les deux méthodes de calcul citées ci-dessus. Néanmoins on peut les exploiter par d'autres moyens.

Nous donnons ici quelques états descriptifs des équipements et leur incidence éventuelle sur un diagnostic.

Pour la chaufferie

- marque et âge de la chaudière et du brûleur : si > à 15 et 10 ans ⟶ vérification de leur état et remplacement éventuel.

- la régulation : aquastat sur chaudière, thermostat intérieur, régulation centrale sur température intérieure ou extérieure ⟶ pour estimer l'adéquation de la fourniture de chaleur aux besoins.

- stockage du combustible : situation, volume, âge et température ⟶ le rendement est croissant avec la température du combustible.

- buses en sortie des fumées sur les chaudières

	P_u nominale	diamètre
(chaudières d'avant 70)	500kW	0,4 m
	1500kW	0,55 m
	3000kW	0,8 m

  nature du raccordement, dispositif de limitation du tirage ⟶ pour améliorer le rendement

- isolation des chaudières à l'arrêt

  hydraulique = non effectué
  effectué manuellement
  effectué automatiquement

  aéraulique = non réalisé
  par clapet motorisé sur l'admission d'air
  par clapet motorisé sur le départ des fumées

  ⟶ limiter les pertes à l'arrêt

Distribution et émission

Mode de distribution

- boucle monotube, série, dérivation ;

- raccordement bitube sur colonnes montantes ;

- régulation sur les émetteurs : robinets sur corps de chauffe 2 ou 4 voies, robinets thermostatiques, volets sur convecteurs, absence de régulation ;
- équilibrage : té sur le retour des radiateurs, des panneaux, robinets à double réglage, en gaine technique, organe de réglage en pied de colonne, plainte des utilisateurs — équilibrer l'installation.

II.2.5 Déperditions par les parois

Le coefficient GV_z est le coefficient de déperditions de chaque zone du bâtiment. Il fait appel à des données d'origines diverses : géométriques, thermiques ou aérauliques.

Les méthodes de calcul de GVz

Les déperditions GV_z comprennent deux parties : déperditions par enveloppe (parois opaques, vitrées donnant sur extérieur, ou sur les autres locaux chauffés ou non) GV_p et déperditions par le renouvellement d'air (voir II.2.7 :

$$GV_z = (GV_p)_z + 0,34NV_z$$

Pour chaque zone on peut décomposer GV_p comme suit :

$$GV_p = GV_{opa} + GV_{ouv}$$

avec GV_{opa} pertes par les parois opaques, et GV_{ouv} pertes par les ouvrants.

Déperditions par parois opaques

$$GV_{opa} = \sum KS\tau + \sum kl\tau$$

avec

S : surface des parois opaques en m^2,
K : coefficient de déperdition surfacique en W/m^2K,
k : coefficient de déperdition linéique en W/mK,
l : longueur des ponts thermiques en m,
τ : coefficient de pondération, sans dimension.

K est calculé suivant deux méthodes, par valeurs forfaitaires ou par calcul.

- Par valeurs forfaitaires

  On estime l'isolation de la paroi, en distinguant 3 cas. Selon le type d'isolation estimée de la paroi ("non isolée", "faiblement isolée", "isolée"), on attribue un coefficient K au mur, auquel est affecté un facteur d'imprécision. On a donc trois classes de coefficient K, assorties de leur imprécision. Le terme "isolée" n'est pas employée dans son acceptation courante.

On entend par paroi non isolée les murs constitués de matériaux massifs pleins jusqu'à 70 cm d'épaisseur et les parois constituées de blocs creux jusqu'à 30 cm d'épaisseur finale. Les parois à lame d'air sont exclues. La valeur de K est prise égale à $2,7 W/m^2K \pm 1,2$ (soit de 1,5 à $3,9 W/m^2K$).

Les parois faiblement isolées comprennent les parois à lame d'air, les parois construites à partir de blocs creux mesurant plus de 30 cm d'épaisseur finie et les parois pleines en matériaux massifs de plus de 70 cm d'épaisseur. La valeur de K est prise égale à $1,4 W/m^2K \pm 0,6$ (soit de 0,8 à $2,0 W/m^2K$).

Enfin les parois isolées désignent les parois dans lesquelles il y a un isolant léger d'épaisseur au moins égale à 2 cm. La valeur de K est prise égale à $0,65 W/m^2K \pm 0,23$ (soit de 0,42 à $0,88 W/m^2K$) [42, 35].

- Par calcul classique

$$K = \frac{1}{\frac{1}{h_e} + \frac{1}{h_i} + \sum \frac{e_m}{\lambda_m}}$$

avec

$\frac{1}{h_e}$, $\frac{1}{h_i}$	résistances superficielles extérieures et intérieures (m^2K/W),
$\frac{e}{\lambda}$	résistance thermique de la couche,
e_m	épaisseur d'une couche de matériau m,
λ_m	conductivité thermique du matériau m (W/mK).

Ceci peut être simplifié en adoptant :

$$K = \frac{1}{\frac{1}{h_e} + \frac{1}{h_i} + \frac{e}{\lambda}}$$

avec

λ : conductivité thermique du matériau dominant,

e : épaisseur totale de la paroi.

$\sum kl$: somme des coefficients de déperditions linéiques (ponts thermiques), elle est forfaitisée à 10% des déperditions surfaciques.

Les déperditions d'une zone par une paroi i varient en fonction de l'espace contigu à cette paroi, ce qui est pris en compte dans le coefficient τ.

- Pour une paroi donnant sur l'extérieur τ est égal à 1.

- Pour une paroi donnant sur un espace non chauffé (garage...) τ est compris entre 0 et 1.

$$\tau = \frac{T_{int} - T_{inc}}{T_{int} - T_{ext}}$$

avec

T_{int} = température intérieure,

T_{inc} = température moyenne du local non chauffé,

T_{ext} = température extérieure moyenne

- Pour une paroi donnant sur une zone chauffée à une température intérieure différente, T'_{int}

$$\tau = \frac{T_{int} - T'_{int}}{T_{int} - T_{ext}}$$

Déperditions par les ouvrants

$$GV_{ouv} = \sum KS$$

avec K et S définis comme précédemment et appliqués aux ouvrants.

On entend par ouvrants les parois vitrées, fenêtres et portes-fenêtres et aussi les portes, qui ont un K spécifique. Les K des fenêtres et portes-fenêtres sont fonction de :

- la nature de la menuiserie, bois ou métal,
- la nature du vitrage, simple ou double et dans ce cas l'épaisseur de la lame d'air,
- le type de fenêtre, fenêtres battantes ou coulissantes, portes-fenêtres battantes ou coulissantes, avec ou sans soubassement ou vérandas,
- la perméabilité de la fermeture, forte ou moyenne.

En audit rapide, on considère que la menuiserie est en bois, la perméabilité forte et la fenêtre battante. Le cas de la véranda n'est pas considéré.

Pour les portes on retrouve les mêmes critères que pour les fenêtres : nature de la menuiserie et type de portes (opaque, avec vitrage), orientation de la porte (extérieur ou local non chauffé). On ne considère en audit rapide que les menuiseries en bois.

II.2.6 Détermination de la température intérieure et de la base des degrés-jours

La détermination de température intérieure d'une zone thermique constitue l'un des points délicats dans le diagnostic thermique. La température intérieure a une influence importante sur la consommation finale, un écart de 1°C sur la température intérieure amène une variation de la consommation d'énergie de chauffage de 5 à 10% selon les régions. La difficulté d'appréhension de ce facteur est due à plusieurs causes :

1. La température intérieure intègre implicitement les imperfections de la régulation, de l'émission et de l'équilibrage thermo-hydraulique de l'installation et résulte également du comportement des habitants.
2. La température intérieure doit être appréciée durant des périodes où les apports gratuits sont faibles devant les déperditions, c'est-à-dire où l'on est sûr de ne pas être en présence d'une surchauffe temporaire. Elle doit être représentative de la saison de chauffage.

Son appréciation reste une question délicate. Il faut estimer aussi précisément que possible la température intérieure, et certaines précautions doivent être prises.

Estimation de la température intérieure

Pour estimer la température intérieure, deux méthodes peuvent être utilisées selon les précisions et les disponibilités de certaines données.

Méthode par défaut

Si aucune mesure n'a été réalisée sérieusement, on se contente d'une appréciation du niveau de chauffage. On peut considérer 3 niveaux : si le bâtiment est *surchauffé* alors la température intérieure T_{int} est choisie à 22°C ; s'il est *chauffé normalement*, la température intérieure T_{int} est choisie à 19°C ; s'il est *sous-chauffé*, la température intérieure T_{int} est choisie à 16°C. Chaque de ces températures sera associée d'un intervalle de confiance à 95% de $\pm 1°C$.

Méthodes par mesure et calcul

Les données à recueillir se limitent à la température intérieure T_{int} de chaque zone thermique du bâtiment. Pour une même zone, plusieurs schémas sont possibles, en fonction des variations temporelles de T_{int}.

1. La température T_{int} est constante sur les 24 heures d'un jour. C'est alors la température de base des degrés-jours.

2. Il existe un scénario d'intermittence jour-nuit, ce qui amène des températures différentes sur les 24 heures d'un jour. Le calcul de la base des degrés-jours va tenir compte des deux températures, pondérées par leur importance horaire. Ainsi supposons 14 heures à $19°C$ et 10 heures à $16°C$, la température de base est de $(14 \times 19 + 10 \times 16)/24°C$, soit 17,75°C. (Nota : les valeurs de 19 et 16°C de cet exemple ne sont pas les valeurs de consigne mais les températures moyennes observées en période jour et nuit pour la zone considérée.)

Calcul des degrés-jours en base T_{int}

Nous présentons trois méthodes pour calculer $Dj(T_{int})$. Deux méthodes utilisent des données mensuelles, la troisième est une méthode plus simple globale qui nécessite seulement la valeur des degrés-jours en base 18 sur l'ensemble de la saison de chauffe.

- En audit intermédiaire et détaillé, on utilise la corrélation de P. Diaz Pédregal ou de B. Bourges [12]. La corrélation de P. Diaz Pédregal permet, à partir des degrés-jours en base 18 d'un mois, de calculer les degrés-jours en base quelconque du même mois par la formule :

$$Dj_X = C_1 + C_2Y + C_3Y^2 + C_4X + C_5X^2 + C_6XY + C_7Y^2X + C_8YX^2 + C_9X^2Y^2$$

Avec $X = T_{int}$, la température intérieure de base des degrés-jours, Y les degrés-jours à base 18 pour le mois et le lieu considéré (on se réfère à la station météo la plus proche), et C_i des constantes.

La méthode de B. Bourges est plus longue mais plus précise. Elle utilise des données météo : les quintiles supérieur Q_s et inférieur Q_i (à 80 et 20%) des températures moyennes mensuelles et la moyenne des températures mensuelles T_{ext}.

$$\begin{aligned} S &= \frac{(Q_s - Q_i)}{1,683} \\ h &= \frac{(T_b - T_{ext})}{(S \times N^{0,5})} \\ Dj(T_b) &= S \times N^{1,5} \times [\frac{h}{2} + \frac{\ln(\exp(-2,1h) + \exp(2,1h))}{4,2}] \end{aligned}$$

avec N = nombre de jours du mois T_b = base des degrés-jours = T_{int}. Les valeurs de T_{ext} sont prises pour les stations météo, de même que celles de Q_s et Q_i.

- En audit simplifié on utilise une loi approchée valable entre 16 et 24°C du type :

$$Dj_{sdc}(T_{int}) = Dj_{sdc}(18) + (T_{int} - 18)Nb_{sdc} \times \alpha$$

avec α coefficient "de lieu", dépendant de la zone thermique, on adopte les valeurs suivantes

H1 : $\alpha \approx 1$, H2 : $\alpha \approx 1$, H3 : $\alpha \approx 0,95$.

sdc : saison de chauffe,
Nb_{sdc} : nombre de jours de la saison de chauffe,
T_{int} : la valeur de la température intérieure, comprise entre 16 et 24°C.

Cette loi approchée a été établie à partir des degrés-jours à base de température quelconque établis pour la période hivernale, soit 232 jours, quelque soit la zone thermique [46].

L'imprécision sur la température intérieure est en général estimée à $\pm 1°C$.

II.2.7 Déperditions par ventilation

Remarque : Nous n'étudions dans cette partie que la ventilation naturelle, les systèmes de ventilation mécanique restent donc à étudier d'une façon approfondie.

Le calcul des déperditions par ventilation reste une question délicate à résoudre. On dispose de trois approches principales. Une approche métrologique, qui nécessite du temps et des appareils, sans pour autant fournir des résultats parfaitement fiables. Nous ne la présentons pas. Une approche par calcul, utilisant des données simples, approche que nous présentons. Enfin une approche par base de connaissances, qui s'efforce de combiner des règles et des situations types pour estimer le taux de renouvellement d'air. Quelques unes de ces règles sont présentées à titre indicatif.

Le modèle physique

Les déperditions par ventilation et infiltrations est égale à $0,34NV$ en W/K, où 0,34 est la chaleur volumique de l'air en Wh/m^3K, N est taux moyen de renouvellement d'air du volume considéré, pour une heure, et V le volume de la zone.

Dans tous les cas rencontrés, N ne sera jamais inférieur à 0,5 vol/h (ou même 0,7). Le produit NV est de la forme [2] :

$$NV = C_p \sum p_e$$

avec C_p le coefficient de pression, de la forme $\alpha + \beta \times$ hauteur du bâtiment, fonction du site et de la zone climatique, α et β coefficients donnés par le tableau suivant

	zone 1 îles, sauf Corse, sommets en altitudes	zone 2 sites côtiers, plateaux dégagés, pentes exposées	zone 3 zones urbaines et suburbaines
α	2,02	1.23	0,92
β	0,008	0,003	0,005

et

$$\begin{aligned} \sum p_e &= \text{perméabilité globale du bâtiment} \\ &= 3000 \times S_{orifice}(cm^2) \\ &\quad +\delta \times S_{fenêtres\ ouvrantes}\ (m^2) \\ &\quad +\lambda \times S_{portes\ ouvrantes}\ (m^2), \end{aligned}$$

On peut prendre les valeurs suivantes pour δ et λ

λ = 8 pour des portes mal calfeutrées,
= 2 pour des portes calfeutrées,

δ = $f(zoneclimatique,\ catégorie\ de\ fenêtre)$, selon le tableau ci-après,

fenêtre d'étanchéité	zone1	zone2	zone3
mauvaise	7,5	6,4	5,9
normale	3,4	2,9	2,7
améliorée	1,5	1,3	1,2
renforcée	0,6	0,5	0,45

En ce qui concerne les imprécisions, on trouve généralement des taux de ventilation compris entre 0,8 et 2,4 vol/h. On prend habituellement une imprécision globale de $\pm 0,4$ vol/h.

La résolution par base de connaissances

Il s'agit ici de proposer quelques règles, qui en fonction de la présence de certains éléments, peuvent aider à estimer la ventilation d'une zone. Quelques exemples se trouvent dans les documents existants. Le "guide pour l'amélioration des logements existants" [77] nous

fournit un exemple des règles que nous souhaitons établir. Celle-ci concerne le bâtiment entier :

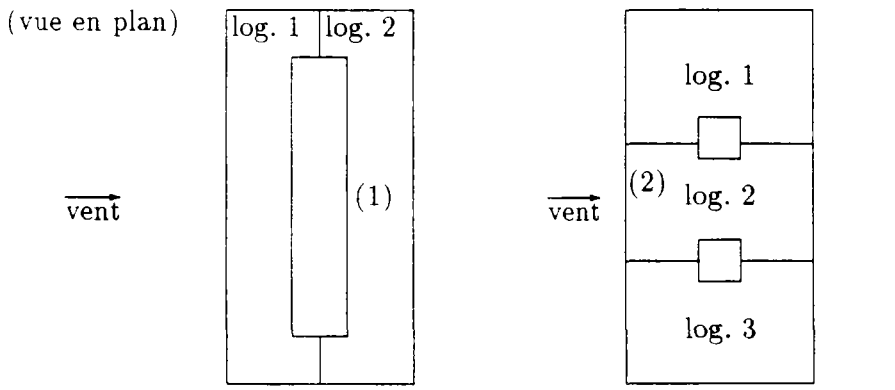
si le bâtiment a été construit avant 1950,
son volume est important,
l'étanchéité des ouvrants est faible,
il existe une ou plusieurs cheminées raccordées à des foyers ouverts,
alors on adopte $N = 1\ vol/h \pm 0,3$.

Les valeurs suivantes sont d'un usage courant pour les pièces non chauffées, cage d'escalier, caves et vides sanitaires, combles. Suivant que l'espace est ventilé

faiblement : $N = 1\ vol/h \pm 0,2$
moyennement : $N = 1,5\ vol/h \pm 0,4$
fortement : $N = 2\ vol/h \pm 0,4$

Les données qualitatives prédominent dans cette approche. Nous énumérons ci-dessous celles qui nous semblent les plus importantes. Elles sont à recueillir pendant les conditions hivernales. Ces règles donnent les tendances de variation du taux de renouvellement d'air.

- La qualité des ouvrants : le taux de ventilation diminue quand l'étanchéité des ouvrants augmente.
- Les habitudes d'ouverture des occupants.
- L'exposition du site : site venté (île, plateau, colline exposée au vent), site abrité (zones urbaines). La position de la zone par rapport aux vents dominants localement et une protection éventuelle de leurs effets (zone donnant sur une cour, encaissée, etc...).
- L'exposition des façades, simple ou double et la configuration de la zone : la configu-



cloison ou cage d'escalier séparant nettement les deux côtés de la zone

pas de cloison entre les deux côtés du bâtiment

ration de type (1) tend à diminuer le taux de renouvellement d'air, la configuration de type (2) facilite le tirage et la ventilation.

- L'état des bouches de ventilation : leur encrassement diminue d'autant leur efficacité.
- La présence de condensation (avec une température intérieure normale) : elle permet de supposer un taux de renouvellement d'air (trop) faible dans les pièces considérées.
- L'existence de foyer ouvert raccordé facilite des taux de renouvellement d'air élevés.
- La taille importante d'un bâtiment accentue le taux de renouvellement d'air.

On adopte les valeurs suivantes, valeurs issues de la pratique, pour les trois niveaux de ventilation :

ventilation faible	$N = 0,75 \pm 0,25\ vol/h$
ventilation moyenne	$N = 1,1 \pm 0,3\ vol/h$
ventilation forte	$N = 1,5 \pm 0,3\ vol/h$

A titre d'exemple, on montre quelques combinaisons de règles.

S'il y a présence de condensation, une configuration de la zone de type (1), peu de bouches d'aération,
si le site est abrité du vent,
si les ouvrants sont de qualité moyenne,
alors le taux de ventilation est faible.

S'il y a une mauvaise qualité des ouvrants, des foyers ouverts, un site venté et aucune trace de condensation,
alors le taux de ventilation est fort.

S'il y a présence de condensation et une mauvaise qualité des ouvrants,
alors les données sont incomplètes et contradictoires, on ne peut conclure ; il faut de plus raisonner alors pièce par pièce, et une nouvelle visite sur site s'impose.

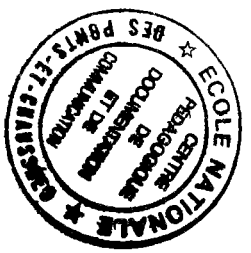
II.3 Conclusion sur les objets et les modèles

On constate qu'il y a un grand nombre d'objets à manipuler. Il est important d'avoir une bonne organisation pour pouvoir créer de nouveaux objets si nécessaire. Les relations physiques existantes entre les objets sont d'une grande complexité, ceci est dû à plusieurs facteurs : complexité géométrique, mais également complexité du fait des relations entre les paramètres de différents objets. Il est souvent difficile de décider si un objet doit être l'attribut d'un objet ou d'un autre, par exemple la régulation doit-elle être considérée comme un attribut de l'équipement ou de la zone thermique ?

A cette complexité des objets, il faut ajouter une multiplicité de modèles pour obtenir un seul paramètre, selon le type de diagnostic (donc les précisions voulues) ou selon les données disponibles. Les méthodes de calcul doivent tenir compte de cet aspect des modèles.

Il faut également distinguer différents types de modèles que l'on est amené à utiliser au cours d'un diagnostic thermique : modèles mathématiques bien explicités, méthodes par défaut et règles (elles peuvent être expertes ou réglementaires).

Par ces considérations, les objectifs d'un outil destiné à l'audit définis au chapitre précédents sont tout à fait justifiés.



Chapitre III

Quelques réflexions sur les informations partielles

A mesure que la complexité d'un système s'accroît, notre aptitude à formuler des affirmations précises, mais significatives sur son comportement diminue jusqu'à un seuil au de-delà duquel la précision et le sens deviennent mutuellement exclusifs.

— *Zadeh*

Dans le chapitre consacré au problème de diagnostic thermique de bâtiment, on a montré que l'un des problèmes fondamentaux est celui de traitement des informations partielles, c'est-à-dire, les informations incertaines, imprécises et manquantes. A partir des modèles existants et du cadre de diagnostic, nous nous proposons dans ce chapitre de donner quelques éléments de réflexion dans le but d'avoir une solution tangible au problème posé.

Nous allons d'abord présenter d'une façon assez complète les différents modèles de traitement existants, ensuite nous essayons de montrer la nature des données manipulées lors d'un audit thermique, et de proposer une solution de traitement pour la réalisation d'une maquette informatique.

III.1 Les informations partielles

Avec le développement des techniques nouvelles en informatique, les problèmes qu'on cherche à résoudre font de plus en plus appel à des modèles complexes faisant intervenir un nombre important de données sous formes très diverses. Ces données sont souvent de nature imprécise, incertaine ou incomplète, du fait

- des imprécisions de mesure ;

- des incertitudes liées aux appréciations subjectives humaines ;
- de l'impossibilité d'accéder à certaines informations ;
- et des problèmes de mise à jour des informations etc...

Nous utilisons les termes *incertitude* et *imprécision*. Afin d'éviter toute ambiguïté, nous les précisons tout de suite :

- L'*incertitude* concerne la *vérité* d'une proposition. Nous disons qu'une proposition est incertaine, si on ne sait pas d'une façon sûre si elle est vraie ou fausse.
- L'*imprécision* affecte le *contenu* des règles ou des faits exprimés. Une proposition est imprécise, si la valeur de certains paramètres n'y est que partiellement spécifiée.

Pour bien illustrer la différence entre ces deux types de données, nous donnons ci-dessous quelques exemples de propositions comportant incertitudes et/ou imprécisions.

- D'abord deux propositions incertaines (mais précises) :
  - Il est *possible* qu'il pleuve demain. L'assertion "Il pleut demain" est *possible*, c'est donc une incertitude sur cette assertion.
  - Il y a une *probabilité .8* que le mûr soit en béton.
- Les informations imprécises (mais certaines) :
  - Il est *très grand*. Le mot *très grand* est un concept vague, ici la taille n'est pas déterminée d'une façon précise.
  - L'épaisseur de la paroi est *approximativement* de 20cm. On peut seulement dire que l'épaisseur est comprise (par exemple) entre [15,25] cm.
- Une information peut évidement être à la fois incertaine et imprécise, en voici un exemple :
  - Il est *vraisemblable* que le renouvellement d'air est *important*. Ici, *important* est une grandeur imprécise par rapport à une certaine échelle de mesure, par exemple le taux de renouvellement d'air est supérieur à 2 volume/heure, mais inférieur à 4 volume/heure. Et cette assertion n'est pas certaine.

On peut en effet souvent constater l'antagonisme entre l'incertitude et l'imprécision, car si on veut rendre plus précis le contenu d'une proposition, on tend à accroître son incertitude, et d'une façon générale la certitude conduit à une imprécision sur les résultats obtenus.

III.2 Modèles de traitement

Du fait de développement de plus en plus important des systèmes de traitement d'information (surtout en I.A.), les problèmes concernant la nature incertaine et imprécise des informations manipulées se posent de plus en plus, les recherches sur les modèles de traitement sont développées. Ainsi un grand nombre de modèles théoriques et expérimentaux ont été proposés par différents auteurs [83].

Les moyens classiques pour traiter les informations partielles sont d'une part le calcul des erreurs par intervalle et le calcul des probabilités. Mais ces moyens ne sont plus suffisants pour les besoins actuels. Il apparaît très clairement aujourd'hui deux approches différentes pour aborder ce problème : l'une cherche à proposer une théorie solidement fondée, mais se soucie peu de son utilisation pratique ; l'autre plus pragmatique propose des méthodes simples et efficaces dont les résultats sont parfois difficilement vérifiables et interprétables.

Dans la suite, on fera un bref rappel des deux méthodes classiques : calcul des erreurs et probabilités. Ensuite on introduira les concepts de base de la théorie des possibilités et de la théorie de l'évidence de Dempster-Shafer. Signalons simplement qu'il existe beaucoup d'autres méthodes que nous ne citons pas dans ce document.

III.2.1 Calcul des erreurs

Cette méthode est très liée aux imprécisions des instruments de mesure en physique. Les imprécisions sont exprimées sous forme d'intervalles. Le calcul des erreurs est non nuancé. On ne connaît pas la valeur exacte d'un paramètre, mais on sait les limites de son domaine de variation.

Dans le cas de diagnostic thermique, certains auteurs [20] ont trouvé des résultats aberrants par ce calcul simpliste.

III.2.2 Probabilités : réseaux bayésiens

La théorie des probabilités est maintenant considérée comme une théorie classique, fondée sur une base axiomatique solide. On ne s'étend pas sur la théorie de probabilités, mais on va simplement indiquer comment est faite l'inférence déductive par probabilité.

L'inférence est en effet basée sur la formule de Bayes. Dans un système à base de règles, une règle peut être exprimée en terme de probabilité, sous la forme suivante :

Si A;
alors B avec une probabilité p.

Une telle règle définit une probabilité conditionnelle $P(B \mid A)$.

Si on note $B_1, \cdots, B_n$ les conditions qui sont remplies simultanément, et D_j la conclusion, on peut calculer $P(D_j \mid B_1 \cup \cdots \cup B_n)$ par la formule de Bayes, (dans la suite on note $E = B_1 \cup \cdots \cup B_i \cup \cdots \cup B_n$)

$$P(D_j \mid E) = \frac{P(E \mid D_j) \cdot P(D_j)}{P(E)} \qquad (III.1)$$

Le dénominateur peut être décomposé par :

$$P(E) = \sum_{j=1}^{n} P(E \mid D_j) P(D_j)$$

$P(D_j)$ est la probabilité à priori de D_j en l'absence de toutes hypothèses B_i.

La relation (III.1) est dans la pratique difficilement utilisable, car il est nécessaire de connaître toutes les $P(E \mid D_j)$, ce que les experts ne peuvent fournir dans la plupart des cas. Sous l'hypothèse d'indépendance entre les B_i et entre les D_j on peut simplifier (III.1) sous la forme suivante :

$$P(D_j \mid E) = \frac{P(B_1 \mid D_j) \times \cdots \times P(B_n \mid D_j) \times P(D_j)}{P(B_1) \times \cdots \times p(B_n)}$$

Les objections contre l'utilisation directe de ces deux formules sont nombreuses, beaucoup d'auteurs (voir par exemple [65]) proposent des versions améliorées, notamment en utilisant les intervalles...

III.2.3 Théorie des possibilités

La théorie des possibilités a été introduite par Zadeh à partir des sous-ensembles flous vers 1977. Elle a pour objectif d'offrir un cadre général de quantification de jugement permettant aussi le calcul des erreurs. Dans ce modèle, l'imprécision est représentée sous forme de sous-ensembles flous, alors que l'incertitude est quantifiée par un couple de valeurs possibilité/nécessité. Les deux notions essentielles dans cette théorie sont les mesures de possibilité et de nécessité [43].

Sous-ensembles flous

La théorie de sous-ensembles flous a pour but de construire une structure mathématique avec laquelle on peut manipuler les concepts mal définis, mais dont l'appartenance à des sous-ensembles a pu être hiérarchisée [64]. La définition la plus directe peut être donnée par :

Définition 1 *soit Ω un ensemble (référentiel), un sous-ensemble flou A de E est la donnée d'un ensemble de couple :*

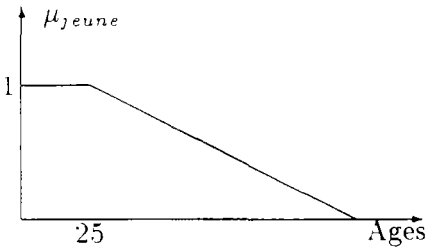
$$\{(x \mid \mu_A(x)\}, \forall x \in \Omega;$$

où $\mu_A(x)$ est le "degré d'appartenance" de x dans A. Cette fonction prend ses valeurs dans un "ensemble d'appartenance" M.

L'ensemble M est en général l'intervalle unité $[0, 1]$. Mais il peut être une structure plus générale telle qu'un treillis. A devient "un sous-ensemble non flou" quand M est l'ensemble $\{0,1\}$.

Exemple : Voici quelques exemples pour illustrer cette définition.

- Soit Ω l'ensemble des entiers $\{1..5\}$, un sous-ensemble flou A est par exemple : $A = \{1 \mid 0.4,\ 2 \mid 0,\ 3 \mid 1,\ 4 \mid 0.8,\ 5 \mid 0.5\}$;
- Soit Ω l'ensemble des âges, le concept *jeune* est un sous-ensemble dont la fonction d'appartenance peut être définie par la courbe suivante :



Remarquons que l'on travaille essentiellement sur la fonction d'appartenance en théorie des sous-ensembles flous.

Rapidement voici les définitions de quelques opérations ensemblistes floues élémentaires :

Définition 2 (Inclusion)

$$F \subseteq G \quad \forall \omega \in \Omega, \quad \mu_F(\omega) \leq \mu_G(\omega)$$

Définition 3 (Egalité)

$$F = G \quad \forall \omega \in \Omega, \quad \mu_F(\omega) = \mu_G(\omega)$$

Définition 4 (Complément) *Le sous-ensemble flou $\overline{F}$, complémentaire d'un sous-ensemble flou F est défini par :*

$$\forall \omega \in \Omega, \quad \mu_{\overline{F}}(\omega) = 1 - \mu_F(\omega)$$

Définition 5 (Intersection)

$$\forall \omega \in \Omega, \quad \mu_{F \cap G}(\omega) = min(\mu_F(\omega), \mu_G(\omega))$$

Définition 6 (Union)

$$\forall \omega \in \Omega, \quad \mu_{F \cup G}(\omega) = max(\mu_F(\omega), \mu_G(\omega))$$

Les définitions précédentes possèdent un certain degré d'arbitraire, bien qu'elles soient assez conformes aux intuitions. Elles coïncident avec les définitions des opérations sur les ensembles classiques, lorsque M est $\{0,1\}$. Les définitions des opérations ensemblistes floues ne sont pas uniques, d'autres définitions sont possibles.

Une des questions importantes qu'on peut se poser sur les sous-ensembles est : comment peut-on définir les fonctions d'appartenance ? L'exemple du sous-ensemble flou *jeune* montre bien cette difficulté.

Notons que pour les sous-ensembles flous, contrairement à la théorie des ensembles classiques, on n'a plus la relation du tiers-exclu, c'est-à-dire :

$$\forall A \subseteq \Omega, \quad A \cap \overline{A} \neq \phi$$
$$\forall A \subseteq \Omega, \quad A \cup \overline{A} \neq \Omega$$

Possibilité et nécessité

Avant de définir ces deux mesures, on va d'abord introduire la notion de mesure de confiance [43].

Définition 7 (Mesure de confiance) *Soit Ω un ensemble référentiel (i.e. un événement toujours vrai). Une mesure de confiance g sur Ω est une application de $\mathcal{P}(\Omega) \mapsto [0,1]$ qui est croissante pour l'inclusion :*

$$\forall A, B \subseteq \Omega, \ A \subseteq B \Rightarrow g(A) \leq g(B) \qquad (III.2)$$

Les parties A de Ω telle que $g(A) = 0$, (resp. $g(A) = 1$) sont appelées des événements impossibles (resp. certains).

La relation (III.2) traduit le fait que si l'événement A implique l'événement B, alors on a toujours au moins autant de confiance en B qu'en A. De cette relation, on déduit :

$$\forall A, B \subseteq \Omega, \quad g(A \cup B) \geq \max(g(A), g(B)) \qquad (III.3)$$

$$\forall A, B \subseteq \Omega, \quad g(A \cap B) \leq \min(g(A), g(B)) \qquad (III.4)$$

Définition 8 (Possibilité et Nécessité) *les cas où les égalités sont atteintes correspondent aux mesures de possibilité et de nécessité.*

Une mesure de possibilité est une mesure de confiance Π vérifiant :

$$\forall A, B \subseteq \Omega, \quad \Pi(A \cup B) = \max(\Pi(A), \Pi(B))$$
$$\Pi(\emptyset) = 0$$

Une mesure de nécessité est une mesure de confiance N vérifiant :

$$\forall A, B \subseteq \Omega, \quad N(A \cap B) = \min(N(A), N(B))$$
$$N(\Omega) = 1$$

En forçant l'égalité (III.3), on prend en effet une attitude prudente qui est un jugement subjectif et qui engage peu son auteur.

On peut facilement déduire que si N est une mesure de nécessité, la mesure définie par (avec $\overline{A}$ complémentaire de A) :

$$\forall A \subseteq \Omega, \quad \Pi(A) = 1 - N(\overline{A})$$

est une mesure de possibilité. Ceci traduit qu'un événement est nécessaire lorsque son événement contraire est impossible (on retrouve la même dualité de possible et nécessaire dans la logique modale).

Des définitions précédentes, on déduit les relations suivantes :

$$\max(\Pi(A), \Pi(\overline{A})) = 1 = \Pi(\Omega) \qquad (III.5)$$

$$\min(N(A), N(\overline{A})) = 0 = N(\emptyset) \qquad (III.6)$$

La relation (III.5) traduit en effet que l'un des deux événements contraires est au moins possible, tandis que la relation (III.6) veut dire que les deux événements contraires ne peuvent pas être nécessaires simultanément.

Possibilités et probabilités

Les probabilités sont plutôt considérées comme objectives, alors que les possibilités sont plus subjectives.

La connaissance de la probabilité d'un événement détermine complètement celle de son contraire, c'est-à-dire :

$$P(A) + P(\overline{A}) = 1 \qquad (III.7)$$

Alors qu'en possibilité, les relations liant A et $\overline{A}$ sont plus faibles, on a seulement :

$$\Pi(A) + \Pi(\overline{A}) \geq 1$$

$$N(A) + N(\overline{A}) \leq 1$$

Remarque : Les possibilités, comme les probabilités peuvent être caractérisées par une distribution.

Quand l'ensemble Ω est fini, toute mesure de possibilité peut être définie à partir de ses valeurs sur les singletons de Ω

$$\Pi(A) = \sup\{\pi(\omega) \mid \omega \in A\}$$

donc

$$\pi(\omega) = \Pi(\{\omega\})$$

π est une application de Ω dans $[0, 1]$.

Mesure de confiance d'un événement flou

Un événement mal défini peut être caractérisé par un sous-ensemble flou F. Les mesures de confiance définies plus haut peuvent être étendues à l'évaluation de la connaissance d'un événement flou.

Définition 9 (Probabilité d'un événement flou) *Soit (Ω, A, P) un espace de probabilité, où A est une $\sigma-$algèbre sur Ω, et P une mesure de probabilité, et soit F un sous-ensemble flou décrit par sa fonction d'appartenance μ_F. Zadeh a défini la probabilité de F par :*

$$P(F) = \int_{\Omega} \mu_F(x)\, dP(x)$$

C'est l'espérance mathématique de la fonction d'appartenance. On vérifie que :

$$P(\overline{F}) = 1 - P(F)$$

On peut aussi définir la possibilité et la nécessité d'un événement flou F, toujours selon Zadeh :

Définition 10 (Possibilité d'un événement flou)

$$\Pi(F) = \sup_{\omega \in \Omega} \min(\mu_F(\omega),\ \pi(\omega))$$

On peut vérifier que l'axiome de possibilité (III.5) est valide. La définition précédente est donc une mesure de confiance.

Par la dualité, on définit la mesure de nécessité d'un événement flou :

Définition 11 (Nécessité d'un événement flou)

$$N(F) = \inf_{\omega \in \Omega} \max(\mu_F(\omega),\ 1 - \pi(\omega))$$

Possibilités et raisonnement approché en systèmes experts

Les deux principaux mécanismes d'inférence dans un système expert, qui sont l'inférence déductive et la combinaison d'informations provenant de différentes sources, se font donc, dans un système où on traite les données partielles, en présence de prémisses incertaines et imprécises.

Rappelons qu'un sous-ensemble flou est défini par une fonction dite d'appartenance : de $\Omega \rightarrow [0,1]$. Dans la suite, on note $\mathcal{P}$ l'ensemble des propositions ; p une proposition quelconque ; A un sous-ensemble flou de Ω (référentiel ou l'univers de discours ou frame of discernment) ; $\omega \in \Omega$; X une variable à valeur dans Ω. Une proposition telle que :

"*X est A*"

spécifie d'une façon précise ou imprécise la valeur de X. Quand A est un singleton de Ω, la proposition élémentaire p = "*X est $\{\omega\}$*", est précise. Toute proposition non élémentaire est donc imprécise.

Une règle floue peut être exprimée de la façon suivante :

"*si X est A, alors Y est B.*"

où $A \subseteq \Omega$ et $B \subseteq \Omega'$ (Ω' un autre référentiel, et $Y \in \Omega'$). La mesure de possibilité de Y est donnée par sa distribution :

$$\forall t \in \Omega',\ \pi_Y(t) = \sup_{s \in \Omega} \{\pi_{Y|X}(t,s) * \pi_X(s)\} \qquad (III.8)$$

où $\pi_{Y|X}(t,s)$ est la distribution de possibilité conditionnelle de Y par rapport à X, et $*$ est une norme triangulaire dont on donnera la définition plus tard. (III.8) est l'analogue de la probabilité conditionnelle :

$$\forall t \in T,\ P_Y(t) = \sum_{s \in S} P_{Y|X}(t,s) \cdot P_X(s)$$

On notera dans la suite :

$$\pi_{Y|X}(t,s) = \mu_A(s) *\!\!\rightarrow \mu_B(t) \qquad (III.9)$$

Voici quelques exemples des opérations $*$ et $*\!\!\rightarrow$:

$$a * b = \min(a,b)$$
$$\Rightarrow a *\!\!\rightarrow b = \begin{cases} 1 & \text{si } a \leq b \\ b & \text{si } a > b \end{cases} \text{ (implication de Gödel)}$$
$$a * b = a \cdot b$$
$$\Rightarrow a *\!\!\rightarrow b = \begin{cases} 1 & \text{si } a = 0 \\ \min(1, \frac{b}{a}) & \text{sinon} \end{cases} \text{ (implication de Goguen)}$$
$$a * b = \max(0, a + b - 1)$$
$$\Rightarrow a *\!\!\rightarrow b = \min(1, 1 - a + b) \text{ (implication de Lukasiewicz)}$$

Il ne s'agit là que les implications les plus courantes, il en existe d'autres. Le choix des implications ne dépend que des attitudes de celui qui utilise ces opérations.

On peut maintenant essayer de voir comment sont combinées les différentes propositions et l'inférence floue.

Dans le raisonnement avec prémisses complexes, on est amené à introduire des règles d'inférence comportant des prémisses et des conclusions vagues. En particulier, on peut introduire la règle de "*modus ponens généralisé*" (A' signifie en gros que A' est presque A, de même pour B'.) :

Si X est A, alors Y est B $\quad$ X est A'

Y est B'

où $\mu_{B'} = \pi_Y$ est calculée à partir de (III.8) et de (III.9) :

$$\forall t \in \Omega',\ \mu_{B'}(t) = \sup_{s \in \Omega} \{\mu_A(s) *\!\!\rightarrow \mu_B(t) * \mu_{A'}(s)\}$$

et on note symboliquement par $B' = A' \circ (A *\!\!\rightarrow B)$. En effet dans ce schéma d'inférence, A' peut être considéré comme pas très différent de A, mais A' n'est pas nécessairement A. Les prédicats A, B, A' sont flous en général.

On peut encore noter que le "*modus ponens classique*" est un cas particulier du "*modus ponens généralisé*", quand $A = A'$ et quand les propositions sont des propositions précises.

La combinaison des différents faits dans la théorie des possibilités reste encore un domaine ouvert à des discussions. Le premier cas de combinaison d'informations est le suivant : on dispose de deux (ou plusieurs) sources qui donnent l'une "X est A_1" et l'autre "X est A_2" avec $\pi_i = \mu_{A_i}$, $i = 1, 2$. Dans le cas où les sources sont fiables, c'est-à-dire où il y a cohérence entre les données, on peut utiliser la formule suivante :

$$\forall s \in \Omega,\ \pi_{12}(s) = \frac{\pi_1(s) \cap \pi_2(s)}{\sup_{s \in \Omega} \{\pi_1(s) \cap \pi_2(s)\}}$$

avec $\cap$ intersection des sous-ensembles flous, et en particulier $\cap$ = min. Le dénominateur sert à normaliser π_{12}. Par contre, quand les sources sont très douteuses, on peut utiliser à la place de $\cap$ une opération de l'union des sous-ensembles flous, en général "max". Mais cette utilisation est dangereuse, et dégrade rapidement les résultats.

L'autre cas de combinaison est celui où il y a plusieurs règles de la forme "Si X est A_i, alors Y est B_i, $i = 1, n$." L'ensemble des règles peut être représenté par $\cap_i A_i *\!\!\rightarrow B_i$ avec $\cap_i$ = min dans la plupart des cas. Le *modus ponens généralisé* s'écrit alors par :

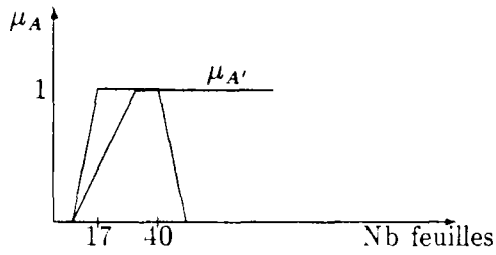
$$\forall t,\ \mu_{B'} = \sup_s \{\min_{i=1,n} (\mu_{A_i}(s) *\!\!\rightarrow \mu_{B_i}(t)) * \mu_{A_i}\}$$

Le *modus ponens généralisé* peut sembler attrayant, mais il pose aussi de nombreux problèmes tels que le choix de l'implication qui reste complètement arbitraire, et également le problème de réalisation pratique [11, 72, 71]. On peut aussi se poser la question sur son fondement théorique qui ne semble pas être très convainquant.

Exemple :

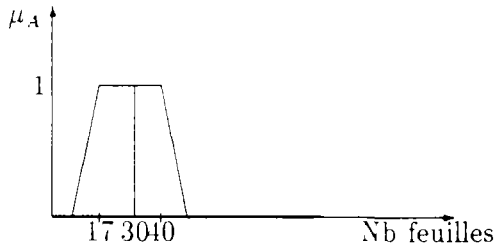
Règle : Si le nombre de feuilles est entre 17 et 40, alors bouquet foliaire.

Observation : La plante a beaucoup de feuilles.



Règle : Si le nombre de feuilles est entre 17 et 40, alors bouquet foliaire.

Observation : La plante a 30 feuilles.



Par contre, ce schéma de déduction ne permet par exemple, à partir d'une règle telle que "Si la tomate est rouge, alors elle est mûre" et d'un fait tel que "la tomate est très rouge", de déduire le fait "la tomate est très mûre, si on n'explicite pas la relation d'ordre entre "mûre" et "rouge". Ceci est dû au fait que dès qu'une partie de A' n'est pas dans A, il apparaît un niveau d'indétermination, en particulier quand $\exists s \in \Omega, \mu_{A'}(s) = 1$ et $\mu_A(s) = 0$, alors B' est totalement indéterminé.

III.2.4 Théorie de l'évidence

La théorie de l'évidence (ou de la croyance) de Shafer est apparue au début des années 70 [89]. Elle est essentiellement basée sur la théorie des probabilités.

Dans cette théorie, interviennent les notions de *plausibilité*, notée Pl et de *crédibilité*, notée Cr. On retrouve une relation de dualité entre ces deux notions que l'on verra plus tard. La suite ne constitue qu'une brève introduction de cette théorie.

Soit $\mathcal{P}$ l'ensemble fini de propositions, $\mathcal{O}$ la proposition toujours fausse, et $\mathcal{I}$ la proposition toujours vraie. Les mesures de plausibilité et de crédibilité sont définies à partir d'une fonction m de $\mathcal{P}$ dans [0,1] telle que :

$$m(\mathcal{O}) = 0$$
$$\sum_{p \in \mathcal{P}} m(p) = 1$$

Définition 12 (Mesure de crédibilité Cr) *Cr, basée sur m, est donnée par :*

$$\forall q \in \mathcal{P},\ Cr(q) = \sum_{p \to q} m(p)$$

$m(p)$ représente la part de croyance associée à la proposition p. m n'est pas une mesure de confiance au sens défini plus haut, mais seulement une pondération relative. Toute proposition p, telle que $m(p) > 0$ est appelée une *proposition focale.*

La mesure de plausibilité Pl est définie par dualité, à partir de Cr :

$$\forall p \in \mathcal{P}\ Pl(p) = 1 - Cr(\overline{p})$$

ce qui donne en terme de m :

$$\forall q \in \mathcal{P},\ Pl(q) = \sum_{p \not\to \overline{q}} m(p)$$

Ces deux mesures correspondent en effet aux bornes supérieure et la borne inférieure d'un intervalle de probabilité, si la distribution de probabilité n'est connue que pour les événements focaux. On vérifie que dans le cas où la probabilité d'un événement ϕ est connue et que $o \in E_i$ on a :

$$Pl(E_i) \leq P(\phi) \leq Cr(E_i)$$

Quand les événements focaux sont des événements élémentaires, on a $Pl = P = Cr$.

Un autre cas particulier est quand les événements focaux s'impliquent les uns les autres, i.e. $E_1 \subseteq E_2 \subseteq \ldots \subseteq E_n$, Les mesures de plausibilité et de crédibilité sont les mesures de possibilité et de nécessité.

III.2.5 Modèle de raisonnement approximatif dans MYCIN

MYCIN est un système expert bien connu pour le diagnostic des maladies des infections sanguines. Ce système utilise un modèle de raisonnement approximatif pour le problème des données partielles. Dans la suite, on expose le principe de ce modèle qui a été décrit dans [90].

MYCIN utilise deux coefficients "mesure de croyance" $MB[h, e]$ (mesure of Belief) et "mesure de non-croyance" $MD[h, e]$ (mesure of Disbelief), où h est une hypothèse et e une évidence. Ils sont définis par les relations suivantes :

$$MB[h, E] = \begin{cases} 1 & \text{si } P(h) = 1 \\ \frac{\max(P(h|E), P(h)) - P(h)}{\max(1,0) - P(h)} & \text{sinon} \end{cases}$$

$$MD[h, E] = \begin{cases} 1 & \text{si } P(h) = 0 \\ \frac{\min(P(h|E), P(h)) - P(h)}{\min(1,0) - P(h)} & \text{sinon} \end{cases}$$

où P(h) est la probabilité que l'expert donne à une hypothèse h. Et à partir de ces définitions, une troisième définition a été introduite pour résumer MB et MD, c'est le facteur de certitude CF :

$$CF[h, E] = MB[h, E] - MD[h, E]$$

Les valeurs de MB et MD sont comprises dans [0,1] et celle de CF dans [-1,1].

Combinaison des MB, MD et CF

Les évidences acquises d'une façon incrémentale :

$$MB[h, S_1 \& S_2] = \begin{cases} 1 & \text{si } MD[h, S_1 \& S_2] = 1 \\ MB[h, S_1] + MB[h, S_2](1 - MB[h, S_1]) & \text{sinon} \end{cases}$$

$$MD[h, S_1 \& S_2] = \begin{cases} 1 & \text{si } MB[h, S_1 \& S_2] = 1 \\ MD[h, S_1] + MD[h, S_2](1 - MD[h, S_1]) & \text{sinon} \end{cases}$$

La conjonction des hypothèses :

$$\begin{aligned} MB[h_1 \& h_2, E] &= \min(MB[h_1, E], MB[h_2, E]) \\ MD[h_1 \& h_2, E] &= \max(MD[h_1, E], MD[h_2, E]) \end{aligned}$$

La disjonction des hypothèses :

$$\begin{aligned} MB[h_1 \vee h_2, E] &= \max(MB[h_1, E], MB[h_2, E]) \\ MD[h_1 \vee h_2, E] &= \min(MD[h_1, E], MD[h_2, E]) \end{aligned}$$

La règle d'inférence :

$$\begin{aligned} MB[h, S_1] &= MB'[h, S_1] \cdot \max(0, CF[S_1, E]). \\ MD[h, S_1] &= MD'[h, S_1] \cdot \max(0, CF[S_1, E]). \end{aligned}$$

où MB' (resp. MD') est le degré de croyance (resp. non-croyance) en h quand S_1 est connue avec la certitude (règle donnée par l'expert).

Par la définition de CF et par les relations précédentes, on obtient en combinant les & et $\vee$ et en notant $CF'[h, S_1 \& S_2 \& (S_3 \vee S_4)] = X$ (CF' au même sens donné pour MB') :

$$\begin{aligned} &CF[h, S_1 \& S_2 \& (S_3 \vee S_4)] \\ &\quad = X \cdot \max(0, CF[S_1 \& S_2 \& (S_3 \vee S_4), E]) \\ &\quad = X \cdot \max(0, MB[S_1 \& S_2 \& (S_3 \vee S_4), E] \\ &\qquad\qquad - MD[S_1 \& S_2 \& (S_3 \vee S_4), E]) \end{aligned}$$

Un exemple dans Mycin

Une règle dans le système MYCIN :

If : (1) the stain of the organism is gram positive,
and (2) the morphology of the organism coccus,
and (3) the growth confirmation of organism in chains
then : there is suggestive evidence (CF = 0.7) that the identity of the organism is streptococcus.

Si la réponse à la question "Did the organism grow in clumps, chains, or pairs ? " est Chains(.6) Pairs(.3) Clumps(-.8), le système déduira automatiquement les facteurs suivants :

$$\begin{aligned} MB[chains, E] &= 0.6, \quad MD[chains, E] = 0., \\ MB[pairs, E] &= 0.3, \quad MD[pairs, E] = 0., \\ MB[clumps, E] &= 0.0, \quad MD[clumps, E] = 0.8. \end{aligned}$$

Pour la règle ci-dessus, on a

$$CF'[H_1, S_1 \& S_2 \& S_3] = 0.7,$$

avec S_1 = "*gram positive*", S_2 = "*coccus*" et S_3 = "*in chains*". On donne

$$CF[S_1, E] = 1, \quad CF[S_2, E] = 1$$

Avec les MB et MD calculés ci-dessus, MYCIN va trouver

$$CF[S_3, E] = 0.6.$$

En utilisant les règles de combinaison définies plus haut, on a

$$\begin{aligned} & CF[H_1, S_1 \& S_2 \& S_3] \\ &= 0.7 \cdot \max(0, CF[S_1 \& S_2 \& S_3, E]) - 0 \\ &= 0.7 \cdot 0.6 \\ &= 0.42 \end{aligned}$$

i.e.

$$\begin{aligned} MB[H_1, S_1 \& S_2 \& S_3] &= 0.42 \\ MD[H_1, S_1 \& S_2 \& S_3] &= 0. \end{aligned}$$

III.2.6 Conclusion sur les méthodes de traitement d'informations partielles

On a parcouru quelques unes des méthodes existantes pour le traitement des informations partielles. On a pu constater que la théorie de probabilité, bien que solidement fondée, n'est pas suffisante en raison de son cadre un peu trop restrictif... Il reste des points à éclaircir dans la théorie des possibilités et de l'évidence qui peuvent poser de véritables problèmes d'interprétation des résultats. Le *Modus ponens généralisé* de Zadeh n'est pas facilement utilisable, ni convaincant. Quant à la méthode utilisée dans MYCIN, elle est plutôt empirique, ce qui empêche de l'appliquer dans n'importe quel domaine, a fortiori dans le notre. D'autres approches existent, notamment celle de [101] qui donne un cadre théorique rigoureux, mais qui poserait des problèmes de mise en œuvre et d'application pratique.

En conclusion donc, dans le domaine d'audit thermique auquel on s'intéresse ici, on ne cherchera pas à appliquer ces méthodes à tout prix, mais plutôt à essayer de voir ce qu'on peut faire de ces informations partielles. Ce qui suit reviendra sur ces points plus en détail.

III.3 Les données partielles en audit thermique

III.3.1 Remarques générales et un exemple

Les données manipulées lors d'un audit thermique de bâtiments existants sont comme on a vu partielles, c'est-à-dire incertaines, imprécises et parfois lacunaires. Ces données se présentent sous des formes diverses. On peut prendre quelques exemples.

Considérons une paroi opaque d'un bâtiment existant, différents cas peuvent se produire.

1. On connaît la composition de la paroi (avec certitude). Les imprécisions liées à son coefficient K de la conductance thermique globale peut être le résultat des causes suivantes :

- caractéristiques thermo-physiques (température, humidité etc...) des différents matériaux ;
- épaisseur des différents matériaux ;
- vieillissement et état physique de la paroi ;
- échanges thermiques superficiels...

Pour estimer l'erreur sur la conductance, on peut la formuler par "le coefficient K est de $K \pm \delta K$ avec un intervalle de confiance à $X\%$". Des études récentes [6] ont proposé pour une paroi opaque les valeurs suivantes : "$\frac{\delta k}{K} = 13\%$ avec un intervalle de confiance à 95%" pour un mode d'analyse particulier ;

2. Il se peut que l'on ne connaisse pas la composition de la paroi. Il y deux façons pour estimer la valeur du coefficient K :

   - on peut à partir d'un certain nombre de constatations (sur l'époque de construction, le site géographique, l'aspect général, l'épaisseur totale, la connaissance des matériaux de construction et leur mise en œuvre etc...) supposer une composition possible. Dans ce cas, on dira "il est probable que la composition de paroi est ...". On peut introduire dans ce cas un coefficient d'assertion pour quantifier l'incertitude sur l'assertion avancée ;
   - on peut également pour contourner la difficulté liée à la composition dire ceci "la paroi, d'une épaisseur totale de Ecm, est constituée de matériaux lourds et est non isolée". A partir de cette assertion, qui est certaine, on peut obtenir un coefficient K qui aura une imprécision beaucoup plus élevée que dans le premier cas.

Cet exemple montre l'existence de deux aspects des données, à savoir incertitude et imprécision, mais également l'antagonisme entre l'incertitude et l'imprécision.

Les imprécisions liées à certaines données peuvent être appréciées selon la nature de ces données. Par exemple, sur les métrés, en connaissant l'origine des données (mesure sur site par un tel instrument ou relevé sur plan), on peut déduire une imprécision. C'est notamment le cas de l'exemple montré ci-dessus sur la paroi opaque.

Il peut aussi arriver qu'il manque certaines données nécessaires au calcul de consommation théorique, et que le diagnostiqueur ne dispose pas de moyen suffisant (manque de temps par exemple) pour revenir sur le site. Dans ces cas il est nécessaire de pouvoir estimer ces données manquantes par la connaissance d'autres paramètres ou bien par la proposition des valeurs les plus souvent rencontrées dans le type de bâtiment en question. Les informations manquantes doivent être également prises en compte dans un outil d'audit.

III.4 Traitement de données en audit

Les données en audit thermique présentent les deux aspects d'incertitude et d'imprécision, et parfois peuvent être manquantes.

Pour les informations de nature numérique, nous allons prendre une représentation déjà utilisée dans des études antérieures [6]. Pour chaque donnée imprécise, on suppose l'existence d'une loi de distribution normale de probabilité. Une valeur V sera donc de la forme $V \pm \delta V$ avec un intervalle de confiance à $x\%$.

En ce qui concerne l'incertitude sur les assertions, nous introduisons un *coefficient d'assertion* pour quantifier la certitude qu'a le diagnostiqueur lorsqu'il avance son assertion. Ce coefficient peut être gradué sur une échelle entre 0 et 1.

III.4.1 Traitement des imprécisions

Les outils existants de l'audit ne proposent pas de méthodes pour prendre en compte cet aspect de données. Certains auteurs [20] utilisent la méthode de calcul des erreurs, qui est décrite ci-dessous, pour affirmer l'impossibilité d'introduire dans le calcul thermique les imprécisions de données. On peut considérer les données imprécises en audit sont des variables aléatoires ayant le plus souvent une loi proche de loi normale. La méthode de calcul des erreurs n'est pas adaptée dans ce cas. Nous utilisons donc la méthode de propagation des erreurs que nous décrivons également ci-dessous.

Pour illustrer ces méthodes, nous prenons l'exemple la consommation C. C est fonction de n données X_i :

$$C = C(X_1, X_2, \ldots, X_n)$$

Calcul d'erreur

C'est un calcul d'incertitude. On a en effet pour la consommation C, avec ΔC l'intervalle d'erreur sur C :

$$\left| \frac{\Delta_C}{C} \right| = \sum \left| \frac{\Delta_{X_i}}{X_i} \right|$$

Ainsi, en prenant par exemple $C = \frac{0{,}024\ GV\ Dj}{\rho}$
et avec

$$\left| \frac{\Delta_G}{G} \right| = 15\%$$
$$\left| \frac{\Delta_V}{V} \right| = 4\%$$
$$\left| \frac{\Delta_{Dj}}{Dj} \right| = 12\%$$
$$\left| \frac{\Delta_\rho}{\rho} \right| = 8\%$$

on arrive à $\frac{\Delta_C}{C} = 39\%$, et le diagnostiqueur peut affirmer que la valeur vraie de la consommation est comprise entre $0,6C$ et $1,4C$; intervalle si large qu'il retire toute signification à la valeur obtenue de C. En effet, les erreurs sur les données obéissent au principe de compensation statistique des erreurs [17].

Méthode de propagation des erreurs

Les résultats récents [6, 42] font l'hypothèse que chaque donnée imprécise peut être considérée comme une variable aléatoire gaussienne et qu'on peut supposer une distribution normale pour les erreurs sur les paramètres (c'est-à-dire ici les X_i). La méthode consiste à cumuler les variances de chaque paramètre, pondérées par la sensibilité de la grandeur calculée au paramètre considéré, pour obtenir la variance de la loi de distribution de la valeur vraie de la grandeur calculée autour de la valeur calculée. On distingue :

- la valeur vraie, inconnue, C_v,
- la valeur estimée ou calculée, C_e,
- la valeur constatée, par relevés ou autre, C_c.

En reprenant la même fonction $C = C(X_1, X_2, \ldots, X_n)$, on va calculer l'écart-type σ de C en fonction des écarts-types σ des X_i, pondérés par la sensibilité de C à une variation de X_i seul. On montre qu'on peut écrire (avec X_i des variables indépendantes)

$$\sigma_C^2 = \sum (\frac{\partial C}{\partial X_i})^2 \sigma_{X_i}^2$$

formule générale s'appliquant à toute fonction $C(X_i)$.

Dans le cas particulier où $C(X_i)$ est une fonction multiplicative des X_i et où les variables sont indépendantes, la valeur de σ_C est donnée par :

$$(\frac{\sigma_C}{C})^2 = \sum (\frac{\sigma_{X_i}}{X_i})^2$$

On cherche la probabilité $P(p)$ d'avoir :

$$C_v \in [C_e - p.\sigma; C_e + p.\sigma]$$

l'intervalle $(\pm p\sigma)$ est l'intervalle de confiance dont le seuil de confiance est $P(p)$. Ce seuil est généralement pris égal à 95%, c'est-à-dire que si le diagnostiqueur adopte un intervalle de $(\pm 2\sigma)$ (pour une distribution normale), il sait qu'il n'a plus que 5% de chance de se tromper, c'est-à-dire de craindre que C_v sorte de l'intervalle $(C_e \pm 2\sigma)$. Ceci s'applique à la consommation comme à toutes les autres variables.

En posant $\sigma_x = \frac{\Delta_x}{2}$ (intervalle de confiance à 95%), il vient avec la même expression de C :

$$(\frac{\sigma_C}{C})^2 = (\frac{\sigma_G}{G})^2 + (\frac{\sigma_V}{V})^2 + (\frac{\sigma_{Dj}}{Dj})^2 + (\frac{\sigma_\rho}{\rho})^2 \qquad (III.10)$$

En reprenant les valeurs $\frac{\Delta_{X_i}}{X_i}$ précédentes, on trouve $\frac{\Delta_C}{C} = 20\%$, c'est-à-dire que le diagnostiqueur peut douter fortement de la fiabilité de son estimation de C_e, lorsque $[C_e - C_c] > 20\%$.

Dans ce cas, le calcul permet de répondre à 3 types de questions :

1. si le diagnostiqueur ne possède pas de valeur constatée de la consommation, quel est le degré de fiabilité de la consommation calculée ? Quelle confiance pourra-t-il apporter à ses modifications et aux calculs des nouvelles performances ?

2. sinon, qu'apporte la comparaison entre valeur estimée et valeur constatée ? Peut-on expliquer un éventuel écart ?

3. sur quels paramètres faut-il jouer prioritairement pour améliorer la précision et la fiabilité de l'estimation ? Quel gain global amène l'amélioration de la précision sur tel paramètre ?

Ce calcul permet de prendre en compte la compensation partielle des imprécisions de chaque paramètre. Le problème restant est la connaissance de l'écart-type de la distribution de la valeur vraie X_v du paramètre X autour de la valeur estimée X_e, qui est sauf exception inconnue. Ceci peut parfois être résolu par une étude spécifique mais relève la plupart du temps d'un dire d'expert [42]. Nous créerons donc une base de connaissance "experte" sur les imprécisions qui affectent les différentes données.

De plus, il faudra veiller à la cohérence interne des valeurs des imprécisions, pour dans la mesure du possible se maintenir tout au long du calcul au même niveau de précision.

III.4.2 Les incertitudes

En ce qui concerne les assertions, nous introduisons un *facteur d'assertion* (ou coefficient d'assertion) pour mesurer la certitude qu'a le diagnostiqueur vis-à-vis d'une assertion qu'il avance. Ce facteur est défini par une échelle numérique (l'intervalle $[0, 1]$), il peut également être qualitatif (sûr, probable, possible, vraisemblable, etc). Cette dernière qualification peut trouver évidement une correspondance avec l'échelle numérique.

La propagation des facteurs d'assertion amène rapidement à des résultats très incertains, donc peu utilisables. Aussi, les facteurs d'assertion ne sont ni propagés (ce qui pose encore des problèmes de fonds) ni cumulés. En cas de résultat manifestement faux, ils permettent une remise en cause des données, en commençant par les plus douteuses.

Nous utilisons les différents modèles thermiques (cf II.2), pour avoir les différents types de données nécessaires et les différents modes d'accès et de réalisation d'un même calcul. En cas de donnée importante manquante, si le choix d'une valeur est trop arbitraire, le calcul est mené selon plusieurs scénarios. Le choix définitif se fait à la fin, en recoupant toutes les informations.

III.4.3 Les données lacunaires

Comme on vu précédemment, un diagnostiqueur peut souvent se trouver dans des situations où certaines informations manquent. La solution idéale serait de revenir sur le site en procédant à des mesures ou aux observations nécessaires. Mais cette solution est rarement envisageable. On peut d'une manière générale proposer deux solutions pour pallier les problèmes posés par cette situation :

1. chercher à utiliser s'il est possible, un autre modèle de calcul qui n'utilise pas cette information. Par utilisation de ce nouveau modèle on peut se trouver dans une situation où il peut y avoir à nouveau d'autres informations manquantes, dans ce cas il vaut mieux revenir au premier modèle et utiliser la deuxième solution. Il peut également se poser un problème de cohérence de données, il faut faire attention lors du bouclage de bilan ;

2. prendre une valeur par défaut. Cette valeur par défaut peut être proposée par le diagnostiqueur, s'il s'estime être sûr compte tenu du type de bâtiment en question. Dans le cas contraire, l'outil de diagnostic doit pouvoir proposer les valeurs les plus courantes dans le cas de bâtiment traité.

III.4.4 Intégration du traitement dans le modèle de représentation par objet

La prise en compte et l'intégration du traitement des données partielles ont donc un double intérêt :

1. la possibilité d'avoir des résultats plus réalistes, et une meilleure cohérence entre les diverses informations fournies par l'auditeur et les résultats escomptés. Ceci se traduit dans le calcul par le choix des modèles physiques adaptés à la situation ;

2. la remise en cause des données imprécises et incertaines sera facilitée dans le recalage de bilan. Par la donnée des imprécisions et des incertitudes, on peut plus facilement localiser les paramètres à revoir pour obtenir une bonne concordance entre la consommation théorique calculée et la consommation facturée. Cette remise en cause des paramètres doit s'accompagner d'un calcul de l'influence des paramètres en question sur le résultat final.

Par cette considération, un outil de diagnostic doit posséder les possibilités de traitement suivantes :

- une structure de contrôle efficace et souple qui permettra de choisir en fonction de l'objectif donné au départ (niveau de l'audit, précision voulue), et des précisions des données directement utilisées dans le calcul, le modèle physique approprié. Elle doit également permettre le retour en arrière dans le cas des données inconnues ou de non-cohérence ;

- tous les calculs doivent se faire avec les imprécisions ;

- les coefficients d'assertion doivent être conservées tout en long du calcul ;

- il faut fournir les moyens nécessaires pour calculer l'influence de chaque paramètre sur le résultat final.

Chapitre IV

Modèles de représentation : acteurs et objets

L'objet s'oppose au sujet ; il se distingue de la sensation en ce qu'il est une totalité de sensation, à saisir par notre entendement. La discipline, c'est toujours de se priver, et de maintenir fortement sa pensée sur son objet.

— *Barrès*

La représentation et la manipulation des connaissances en intelligence artificielle, et le développement de gros systèmes en génie logiciel ont conduit à chercher les nouveaux modèles de représentation et les nouveaux paradigmes de programmation. Le modèle dit orienté objets et celui dit d'acteurs en sont deux exemples[1].

Ces deux paradigmes de programmation offrent de nombreuses caractéristiques intéressantes tant au point de vue de la représentation des connaissances qu'au point de vue des exigences du génie logiciel telles que la facilité de conception, la modularité, la maintenabilité...

Bien qu'ils reposent sur un certain nombre de concepts assez voisins, les deux modèles présentent néanmoins des caractéristiques différentes. Il nous semble intéressant de parcourir l'ensemble des concepts utilisés dans ces deux modèles et de faire une comparaison afin de dégager les éléments pouvant satisfaire les besoins en ce qui concerne notre recherche proprement dite.

On peut d'une manière rapide résumer quelques éléments qui différencient le modèle d'objets du modèle d'acteurs :

- les modèles d'objets utilisent généralement le concept de classe-instance, alors que les modèles d'acteurs sont basés sur le concept de prototype [70]. Ceci a pour conséquence

[1] Les termes *langages orientés objets* et *programmation orientée objet* correspondent aux termes anglo-saxons de *object oriented language* (or *programming*). Nous utiliserons dans notre texte plutôt les terms *langages à objets* ou *programmation par objets*.

sur le mécanisme de création de nouveaux objets (ou acteurs) ;

- les mécanismes de partage de connaissances sont différents, héritage pour les objets ou délégation pour les acteurs ;
- l'utilisation des continuations est fondamentale dans les modèles d'acteurs, alors qu'elle est absente dans les modèles d'objets en général ;

Le développement actuel, particulièrement la recherche de nouveaux modèles dans le domaine de calcul parallèle, tend à rapprocher ces deux paradigmes de programmation sous la dénomination unique de *programmation orientée objet*. Par exemple, les modèles inspirés des acteurs intègrent pour la plupart les avantages nés de la programmation orientée objet.

Dans ce qui suit, le terme *modèle d'objets* est réservé à tous les modèles présentant des caractéristiques voisines de SMALLTALK [53], et celui de *modèle d'acteurs* est utilisé pour les modèles proches de celui défini par C. Hewitt [58].

La première partie est consacrée au problème général de représentation des connaissances, et plus spécifiquement par le *formalisme objet*, ensuite on fait une description du modèle de SMALLTALK en nous limitant à des concepts de base. La troisième partie est destinée à présenter d'une manière détaillée le modèle d'acteurs, et la notion de continuation et son utilisation y seront discutées.

IV.1 Représentation des connaissances

Les recherches menées en intelligence artificielle depuis les années 50, ont naturellemnt conduit à ce que les problèmes liés à la représentation[2] et la manipulation des connaissances soient devenus un domaine de recherche à part entière. En effet, les recherches en intelligence

[2] Les sciences cognitives sont considérées par certains [102] comme ayant connu quatre étapes de développement qui correspondent à des approches différentes concernant le mécanisme de cognition :

- la première période dite de *jeunes années* (1940–1956) a donné lieu à des nombreuses échanges d'idées entre les différentes disciplines. La cybernétique en est le pionnier ;
- la seconde est celle de ce qu'on appelle les *cognitivistes*. La cognition est considérée comme le traitement de l'information : la manipulation des symboles à partir des règles. Les connaissances sont représentées sous formes de règles logiques. Cette approche a eu une grande influence sur le développement de l'intelligence artificielle ;
- ensuite a vu le jour le développement des *connexionnistes*. La cognition est maintenant une émergence d'états globaux dans un réseau de composants simples. Il y a des règles locales qui gèrent les opérations individuelles et des règles de changement gèrent les liens entre les éléments. Dans cette approche comme dans celle précédente, la notion de *représentation* joue un rôle clé dans le processus de l'intelligence ;
- la quatrième étape est celle des *évolutionnistes* qui ont une vue assez différentes des deux précédentes. En effet, les connaissances ne sont pas nécessairement représentables sous une forme bien définie, mais sont acquises au fur à mesure. La cognition est une action productive : l'historique du couplage structurel qui enacte un monde. Elle fonctionne par l'entremise d'un réseau d'éléments inter-connectés, capable de subir des changements structuraux au cours d'un historique non interrompu.

[81] donne également des éléments de synthèse et de réflexion intéressants sur les problèmes concernant l'intelligence.

artificielle, notamment dans le domaine de vision et de traitement de langages naturels ont conduit à considérer l'importance dans un système de raisonnement des façons de représenter et de manipuler les connaissances.

Par deux concepts différents du monde, on arrive à deux façons différentes de représenter les connaissances : la première qui met l'accent sur les relations entre les objets est appelé *formalisme relationnel*, la seconde qui considère les objets comme des entités indépendantes possédant des propriétés internes, est appelé *formalisme objet* [49]. Le formalisme *relationnel* consiste d'une manière générale à déduire des énoncés existants de nouveaux énoncés avec des règles d'inférence.

En programmation, il existe également deux approches opposées que l'on appelle *programmation procédurale* et *programmation déclarative*. Certes, on peut trouver des liens avec les deux formalismes cités, mais cette opposition est différente.

IV.1.1 Représentation de connaissances par formalisme logique

Les objets n'ont pas d'existence réelle dans le formalisme relationnel (ou logique), mais seulement comme participants à un ensemble d'énoncés dispersés dans la base de connaissances. Le raisonnement consiste à manipuler ces énoncés afin d'en déduire de nouveaux.

Le formalisme logique a conduit au développement des systèmes de production, des systèmes de démonstration automatique de théorèmes, et plus tard des *systèmes experts*, basés sur les mécanismes d'inférences en logique, ainsi qu'un langage de programmation maintenant bien connu et très utilisé PROLOG [51].

En effet, le développement de ces systèmes trouve sa base théorique dans la logique. Les systèmes formels fournissent les outils nécessaires à la réalisation effective de ces systèmes. Nous allons brièvement présenter le principe des *systèmes experts*.

Systèmes experts

L'utilisation du formalisme logique comme moyen de représentation de connaissances a donné lieu au développement des systèmes à base de règles, notamment les *systèmes experts* (S.E.)[3].

Le développement de ces systèmes a commencé vers le début des années 70, et est incarné par le système MYCIN [90, 47]. Et ils suscitent un engouement dans certains milieux professionnels, parce qu'ils répondent à des besoins précis pour des tâches jusqu'alors très peu informatisées.

Il n'existe pas une définition précise des systèmes experts [4]. En général, on peut les caractériser par un certain nombre d'éléments.

Composants d'un S.E.

Un système expert est constitué

[3] Il existe des systèmes experts qui ne sont pas à base de règles.

[4] Voici l'une des définitions possibles : "Les systèmes experts sont des programmes informatiques conçus pour raisonner habilement à propos des tâches requérant une expertise humaine".

- d'un *langage* d'expression des connaissances fournies par les experts, souvent sous forme proche du langage naturel ;
- d'une *base de connaissances* pour accueillir les connaissances spécifiques d'un domaine d'application ;
- d'un *moteur d'inférence*, c'est un programme relativement général qui exploite les connaissances précédentes.

En plus de ces éléments, un système expert doit également comporter des caractéristiques telles que explication du raisonnement, l'acquisition de nouvelles connaissances.

Base de connaissances

Deux types de connaissances sont en général distinguées :

- connaissances *assertionnelles* (appelées couramment *faits*) décrivant des situations considérées soit comme établies soit à établir ;
- connaissances *opératoires* ou *règles* qui représentent le savoir-faire du domaine.

Les règles comportent essentiellement deux parties : prémisses et conclusions. Une partie de la règle est appelée partie *déclencheur* et l'autre *corps*. Sous leur forme extérieure, elles sont en général représentées de la façon suivante :

```
   Si condition 1
et si condition 2
   ...
et si condition n
alors conclusion
```

Pour un domaine d'application donné, la constitution de la base des connaissances reste le point délicat de la réalisation du système expert.

Moteur d'inférence

Un moteur d'inférences consiste à déclencher les règles de la base de connaissances en présence de certains faits. Rappelons qu'une règle est représentable par la forme suivante :

```
une règle = <déclencheur> + <corps>
```

Quand on lance le moteur d'inférence, celui-ci essaie de déterminer s'il existe des règles déclenchables, c'est la phase d'*évaluation*. S'il en existe, il déclenche une des règles possibles et déduit de nouveaux faits, c'est la phase d'*exécution*. Ceci est le cycle de base d'un moteur d'inférence.

Dans la phase d'évaluation, on peut distinguer trois étapes :

- *sélection* ou *restriction* qui détermine, à partir d'un état (avec une base de faits déjà établie BF), et de la base des règles BR, un sous-ensemble $F1$ de BF et $R1$ de BR qui sont utilisés dans la phase suivante ;

- *filtrage (pattern-matching)*, le moteur d'inférence compare la partie déclencheur des règles de $R1$ par rapport à l'ensemble de $F1$, ce qui produit un sous-ensemble $R2$ dont les règles sont déclenchables. $R2$ est appelé *ensemble de conflits* ;

- *résolution de conflits*, cette étape consiste à choisir une règle (ou un sous-ensemble $R3$) de $R2$ pour être déclenchée.

La phase d'exécution consiste donc à déclencher la règle choisie, et réalise les actions définies dans la partie corps de la règle qui peuvent soit lancer un nouveau cycle de moteur, soit produire d'autres actions comme par exemple produire de nouveaux faits.

Retour-arrière

Si l'ensemble R3 est vide, un moteur d'inférence peut effectuer un retour-arrière (backtracking), c'est-à-dire qu'il reconsidère la résolution de conflits et en annulant les effets produits par le déclenchement de la précédente règle.

Différents types de moteurs d'inférence

On peut distinguer plusieurs types de moteurs d'inférence. Selon son mode de déduction : on peut avoir un fonctionnement à

- chaînage-avant. C'est-à-dire, on part de la base des faits initiale et avec la base des règles, on essaie de déduire de nouveaux faits. Le déclenchement de règles se fait donc avec la partie prémisse des règles ;

- chaînage-arrière. On part d'un but à démontrer. Le déclenchement de règles se fait donc par la partie conclusion des règles.

Selon la stratégie de recherche, on peut avoir une recherche par

- profondeur d'abord, largeur ensuite ;

- largeur d'abord, profondeur ensuite.

On peut également distinguer les moteurs d'inférences selon s'il accepte les variables dans les règles ou non. Dans le cas où il n'y a pas de variables dans les règles (ou faits), on parle souvent de moteurs d'ordre 0, qui correspondent en effet à la logique des propositions, dans le cas où on peut avoir des variables dans les règles ou faits, on parle de moteurs d'ordre 1, qui correspondent à la logique des prédicats du premier ordre.

Exemple

On donne un exemple très simple d'une base de règles avec deux faits établis, et on essaie de voir comment fonctionnent le chaînage avant et arrière.

Base de règles :

R1 $A \rightarrow E$
R2 $B \rightarrow D$
R3 $H \rightarrow A$
R4 $E \wedge G \rightarrow C$
R5 $E \wedge K \rightarrow B$
R6 $D \wedge E \wedge K \rightarrow C$
R7 $G \wedge K \wedge F \rightarrow A$
R7 $G \wedge K \wedge F \rightarrow A$

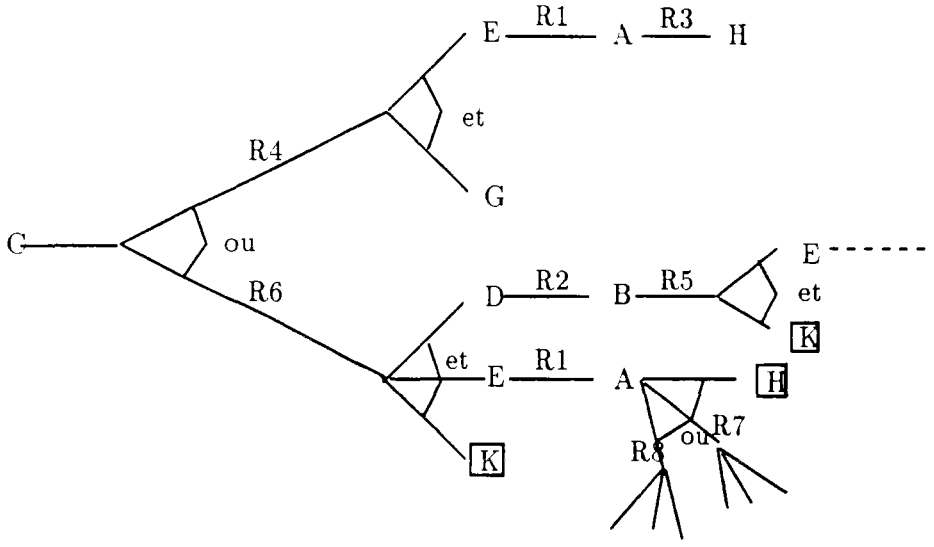
Base de faits initiale : H, K

Chaîne de dérivation avant :

$H \rightarrow A$ R3
$A \rightarrow E$ R1
$E \wedge K \rightarrow B$ R5
$B \rightarrow D$ R2
$D \wedge E \wedge K \rightarrow C$ R6

Chaîne de dérivation arrière :

Le but à démontrer est C. Voici l'arbre de dérivation :



D'autres types de différence existent entre les moteurs d'inférence, tels que la non-monotonie etc...

Notons qu'il existe des systèmes appelés générateurs de systèmes experts qui fournissent en général un moteur d'inférence et les schémas pour la constitution de la base des connaissances. Ces systèmes offrent dans la plupart des cas la majeure partie des possibilités citées ci-dessus.

Conclusion

Les systèmes experts semblent bien adaptés dans les domaines où les connaissances à représenter sont assez limitées, c'est-à-dire dans un domaine très spécifique. La réalisation d'un système expert général paraît à ce jour difficile et dénué d'intérêt.

Le passage des connaissances implicites des experts aux connaissances explicites compréhensibles par l'ordinateur semble la difficulté essentielle de réalisation de S.E.

IV.1.2 Représentation de connaissances par formalisme objet

Le formalisme objet met l'accent sur les propriétés internes des objets et les façons avec lesquelles ils interagissent entre eux et avec le monde extérieur.

Les recherches en intelligence artificielle, notamment en traitement des langages naturels, ont mis en avant le formalisme objet, car on cherche à modéliser le plus souvent les propriétés des objets physiques qui sont nombreux et complexes.

Ce formalisme trouve en effet sa racine dans le développement de psychologie cognitive. Piaget [82], par l'intermédiaire de sa théorie d'*accommodation, d'assimilation et d'équilibration*, voit l'intelligence comme une construction mentale du réel. Le réel est traduit intérieurement sous forme d'un ensemble de "schèmes".

L'un des premiers modèles de représentation de connaissance par objets est sans doute le *réseau sémantique*. Ensuite sont développés les modèles de frames et d'acteurs.

Les langages à classe sont, quant à eux, nés du besoin propre de la programmation dont le précurseur est SIMULA67. De nombreuses tentatives existent pour intégrer l'ensemble des concepts utilisés dans ces différents modèles [5, 73]. Nous nous plaçons à la fois sur le plan de la représentation de connaissances et de la programmation.

Réseau sémantique

Les réseaux sémantiques sont les premiers modèles proposant le mécanismes d'héritage pour la représentation de connaissances. Le but de réseaux sémantiques était de représenter la signification du mot des langages naturels sous la forme d'une mémoire associative.

Un réseau sémantique est constitué des nœuds typés qui représentent les *faits*, et des arcs orientés et étiquetés qui représentent les *prédicats* (relations) entre les nœuds.

Les réseaux sémantiques proposent généralement deux méthodes d'inférence sur les connaissances représentées : l'héritage et le filtrage.

Les réseaux sémantiques peuvent aussi être vus comme une liste de couples <attribut, valeur> : les *attributs* sont des *relations* et les *valeurs* des pointeurs vers d'autres entités, i.e. d'autres nœuds.

Depuis la formalisation des réseaux sémantiques, un grand nombre de modèles ont été proposés pour y intégrer de nouveaux concepts et mécanismes. Plusieurs langages ont été développés sur la base de réseaux sémantiques : par exemple KL-ONE, KRS, bien que ces langages intègrent également souvent des concepts issus de *frames*.

Frames

Le modèle de *frames* a été introduit par Minsky vers 1970. Ce modèle est largement influencé par des modèles développés dans le cadre de psychologie cognitive comme celui de Selfridge. Dans la "société d'esprits" (*society of minds*) [78] proposée par Minsky, on voit clairement cette influence.

On peut également considérer que le formalisme par *frames* est une synthèse de différents modèles de représentation existants (réseaux sémantiques, systèmes de production etc...).

Un frame est en effet une structure de données qui représente un objet *typique*, ou une situation *stéréotype*. C'est un réseau hiérarchisé de nœuds et de relations. Deux niveaux de nœuds se distinguent : les uns supérieurs qui représentent des informations considérées comme toujours vraies, et les autres inférieurs qui possèdent des *attributs* ou *slots*, auxquels sont associées les valeurs. La valeur d'un attribut est une donnée quelconque qui peut être un autre frame. Les frames incluent des informations déclaratives et procédurales caractérisées par les attributs qui peuvent être considérées comme généralisation des couples <attributs, valeurs> des réseaux sémantiques.

Un frame est donc une liste de couples <attributs, {facette}> où chaque facette correspond à un élément de description d'un attribut. Ces facettes contribuent à la spécification de données implicites, de contraintes, de valeurs par défaut, et peuvent contenir des *démons*, i.e. des procédures qui sont déclenchées lors de la manipulation des valeurs des attributs.

Les langages développés à partir des concepts de frames sont nombreux, on peut citer KRL, FRL, UNITS (qui sert de base de développement pour le système KEE) [73] et SHIRKA [84] etc...

Langages à classes

Les modèles de programmation par objets peuvent également classés dans la catégorie de modèles de représentation de connaissances. Cependant le modèle tel que celui de SMALLTALK manque certaines propriétés pour manipuler les connaissances : les valeurs par défaut, les démons par exemple.

Le développement des systèmes de plus en plus importants en génie logiciel a également conduit à introduit le paradigme de programmation par objets. Certains langages ont été développés spécialement pour satisfaire ce besoin. Les langages de programmation tels que C++ [94], et Eiffel [75] etc...en sont des exemples bien connus.

Les exigences de génie logiciel sont nombreuses tant au point de vue de développement de logiciel qu'au point de vue de maintenance de systèmes, différentes des préoccupations de l'I.A.. La modularité, la maintenabilité constituent les éléments essentiels dans ce domaine.

Langages d'acteurs

Les langages d'acteurs avaient également été au début destinés à être des modèles de représentation de connaissances. Leur développement tend actuellement vers les architectures des machines massivement parallèles, tant pour la représentation de connaissances

distribuées que pour la programmation de ces machines parallèles.

IV.2 Etude d'un modèle d'objets

Dans cette partie, on étudie spécialement un modèle de langages orientés objets. Les aspects concernant la manipulation des connaissances au sens de frames tels que les *démons*, l'utilisation des valeurs par *défaut* ne sont pas considérés.

Les langages orientés objets sont apparus au début des années 70, popularisés notamment par SMALLTALK, et dont le précurseur est SIMULA. En utilisant les notions de *classe* et d'*héritage* introduites par SIMULA, et en introduisant celle de *message*, SMALLTALK est devenu un des langages objets les plus connus.

Signalons que SMALLTALK est développé vers le début des années 70. C'est un langage de programmation, mais aussi un système d'exploitation et un environnement de programmation. L'interface utilisateur de SMALLTALK est particulièrement intéressante, car fait partie intégrante du système qui est accessible et modifiable par l'utilisateur.

D'autres langages ont aussi été réalisés dans des conditions différentes. Dans la préoccupation des recherches en I.A., les langages sont dans la plupart des cas réalisés par une extension sur LISP, ou certains sur PROLOG. Dans la famille lispienne, on peut notamment citer LOOPS, FLAVORS, CLOS(COMMON LISP OBJECT SYSTEM) [73] etc....

Par rapport au modèle de SMALLTALK, CLOS introduit certains nouveaux concepts, par exemple la notion de fonction générique [41] qui remplacent le concept de messages.

Les modèles d'objets reposent sur les principaux concepts suivants :

- objet ;
- classe ;
- message ;
- héritage ;
- méta-classe.

Chacun des éléments sera expliqué par la suite.

IV.2.1 Objet

L'*objet* est le concept de base dans la programmation orientée objet. Le concept d'objet peut être vu de différentes manières.

La notion d'objet unifie la notion d'états et de procédures dans une même entité. Un objet est composé par un état privé (les variables internes de l'objet) et un ensemble d'opérations avec lesquelles l'objet peut être manipulé. Les variables internes sont souvent appelées *champs*, et les opérations appelées *méthodes*.

Comme on le verra au chapitre suivant, notre représentation de l'objet consiste simplement en une fermeture au sens utilisé dans les langages fonctionnels. Dans ce cas un objet

est simplement représenté par un environnement où les variables internes sont liées, et par un ensemble de procédures définies dans cet environnement.

Exemple :

Nous prenons l'exemple de représentation d'un point en 2D. On peut définir ce point en 2 dimensions par les éléments suivants :

- les coordonnées X et Y définissant la position du point dans un système de repère donné ;
- une méthode **afficher** permettant d'afficher ce point sur un dispositif (écran par exemple) ;
- une méthode `bouger-de(dx, dy)` permettant de bouger le point avec `dx` et `dy` ;

Le point en 2D que nous considérons peut être représenté par la figure IV.1.

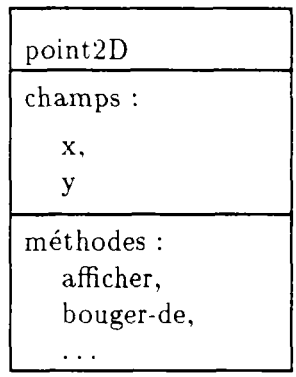


Figure IV.1: Représentation d'un point 2D

IV.2.2 Messages

On a vu qu'en effet un objet est une entité de calcul fermée dont l'état interne n'est accessible que par l'intermédiaire de l'objet lui-même. Le moyen que possède un objet pour communiquer avec le monde extérieur est l'envoi et la réception de messages. Autrement dit les messages permettent l'activation des méthodes d'un objet. Un message spécifie l'opération à effectuer, mais non comment cette opération est effectuée.

L'ensemble des messages qu'un objet peut répondre constitue l'*interface* de l'objet avec les autres objets du système. Les messages assurent la modularité des objets, car l'état interne d'un objet ne peut être modifié que par lui-même, les autres objets ne peuvent, par l'intermédiaire des messages, que demander à l'objet en question les opérations, mais non dicter la façon d'accomplir cette opération.

Dans l'exemple précédent, les messages que peut recevoir l'objet `un-point2D` sont `afficher` et `bouger-de(dx, dy)`. Dans la définition de l'objet telle qu'elle est décrite

au chapitre suivant, l'envoi de message peut simplement être vu comme une invocation de procédure dans un environnement donné.

Un message est composé d'un receveur, du sélecteur de méthodes et des arguments éventuels pour appliquer la méthode. Il est généralement de la forme :

```
(send receveur selecteur args ...)
```

Ainsi pour un-point2D, on peut lui envoyer les messages suivants :

```
(send un-point2D 'afficher)
(send un-point2D 'bouger-de 3 4)
```

Remarque : Dans l'exemple de un-point2D, la méthode **afficher** ne modifie pas l'état local de l'objet, alors que **bouger-de** doit modifier les valeurs **x** et **y** pour qu'à la réception d'un autre message **afficher**, le point s'affiche au bon endroit, c'est-à-dire la position modifiée. Une seconde remarque concernant les messages qu'un objet peut envoyer à soi-même. On peut dans la définition de la **bouger-de** avoir envie que le point s'affiche après avoir modifié sa position, dans ce cas le point envoie le message **afficher** à lui-même.

IV.2.3 Classe

Les objets ayant des propriétés similaires forment une *classe*. Tout objet dans ce modèle est une *instance* d'une classe. Une classe définit le comportement de ses instances (méthodes) et les champs qui auront des valeurs particulières pour chaque instance.

La notion de classe généralise la notion de *type abstrait*.

Exemple :

Reprenons le même exemple de point en 2D. On peut définir une classe **POINT2D** à partir de laquelle tous les points seront instanciés.

A partir de cette classe **POINT2D**, on peut instancier un objet **un-point2D** simplement par

```
(un-point2D (instance - de POINT2D avec x = 3 et y = 4))
```

Le point ainsi défini a initialement pour coordonnées 3 et 4.

IV.2.4 Héritage

L'héritage est un mécanisme essentiel dans un modèle d'objets qui permet de partager les connaissances entre les différents objets de différentes classes organisées d'une manière hiérarchique.

En effet, dans le modèle SMALLTALK toute classe est une *sous-classe* d'une autre classe. Cette sous-classe *hérite* de cette autre classe (*super-classe*) l'ensemble des champs et des méthodes. Une sous-classe est une spécialisation de la super-classe, alors que la super-classe est une généralisation.

Considérons l'exemple suivant : on définit une classe **point2D**, et on veut définir une autre classe que l'on appelle **point2D-colore**. Cette nouvelle classe aura pour champs les

coordonnées `x`, `y` et `couleur`, et pour méthodes `afficher` et `bouger-de`. Avec le mécanisme d'héritage, on peut définir `point2D-colore` comme une sous-classe de `point2D`. Les champs `x`, `y` seront définis par la super-classe, ainsi que la méthode `bouger-de`, par contre le champs `couleur` et la méthode `afficher` doivent être définis dans la nouvelle classe (on peut penser que le mécanisme d'afficher sur un dispositif tel qu'écran peut être différent selon qu'il s'agit d'un point couleur ou noir.)

Fonctionnement de l'héritage

Au point de vue méthodologique, l'héritage fonctionne d'une manière très simple. En effet, les instances d'une sous-classe héritent toutes les propriétés des classes supérieures hiérarchiquement.

Il existe différentes sortes d'héritage : simple ou multiple, si on considère le nombre de super-classes que peut avoir une classe ; statique ou dynamique si on veut savoir si les propriétés sont héritées au moment de création de l'objet ou au moment d'utilisation de ces propriétés. Ces différences ont des conséquences sur l'implémentation du modèle et son utilisation dans la structuration des objets.

La multiplicité de l'héritage pose des problèmes de conflit de noms. Différentes stratégies existent pour résoudre ce problème. L'héritage simple est suffisant dans la plupart des cas. Pour une discussion sur ce point, on peut notamment se reporter à [22, 44].

La différence entre l'héritage statique et dynamique pose des problèmes sur les ressources en temps et en mémoire. L'héritage statique réduit le temps de recherche de méthodes, mais utilise plus de mémoire, alors que l'héritage dynamique comporte d'une manière contraire.

Self et Super

On voit que pour appliquer correctement les méthodes (dans l'environnement de l'objet considéré), il nous faut introduire la notion du receveur du message que l'on appelé `self`[5]

L'exemple donnée par [53] montre clairement le fonctionnement de `self`. On suppose une classe `ONE` sous-classe de `OBJECT` (objet de racine de SMALLTALK), une deuxième classe `TWO` sous-classe de `ONE`.

```
Class ONE subclass of OBJECT
  method:
    test :  return 1
    result : send self test
Class TWO subclass of ONE
  method:
    test : return 2
```

[5] En SMALLTALK, `self` est appelé une pseudo-variable. Les autres pseudo-variables sont par exemple nil, true, false. La différence d'une pseudo-variable par rapport à une variable est que sa valeur ne peut pas être modifiée par une affectation.

Supposons `E1` et `E2` sont des instances respectives de `ONE` et `TWO`. Quand on envoie un message `result` à `E1`, le résultat retourné est 1, et quand on envoie le même message à `E2` le résultat est 2. Dans chaque envoi de message, la pseudo-variable `self` est liée à l'objet receveur du message `result`, ce qui permet à chaque fois d'aller chercher la bonne valeur de `test`.

Une autre pseudo-variable `super` comporte de la même façon que `self` à la différence que `super` cherche la méthode dans la super-classe de l'object receveur du message.

La sémantique de l'héritage peut poser certains problèmes, pour une discussion sur la sémantique formelle de l'héritage, on peut se rapporter par exemple à [63, 31].

IV.2.5 Méta-classe

L'introduction du concept de classe détruit dans un sens l'uniformité de concepts dans un modèle d'objets. Or toute entité du modèle doit être un objet, une classe ne peut faire l'exception. On considère donc une classe comme une instance d'une autre classe que l'on appelle *méta-classe*.

Il est bien évident que cette méta-classe doit également être une instance d'une autre classe, ce qui introduirait une méta-méta-classe. De cette manière, on peut continuer jusqu'à l'infini. SMALLTALK résout ce problème par l'introduction de la classe particulière `Metaclass` en plus des classes `Classe` et `Object`. La classe `Object` est la racine de l'arbre d'héritage, donc la super-classe par défaut de toutes les classes. La classe `Classe` est une méta-classe permettant de créer de nouvelles classes. La classe `Metaclass` est une classe dont toutes les méta-classes sont instances. En effet, chaque classe possède une méta-classe qui est créée automatiquement lors de la création de la classe. La méta-classe `Metaclass class` est une instance de `Metaclasse`, cette circularité permet d'éliminer le problème de régression à l'infini. La relation d'instanciation des classes en SMALLTALK est représentée par la figure IV.2.

L'uniformité de concept impose de considérer les classes comme objets de première classe dans le modèle, ce qui impose l'utilisation de méta-classe. La méta-classe est utile à deux niveaux :

- niveau d'implémentation, elle permet de décrire et contrôler l'implémentation des objets au niveau utilisateur ;
- niveau utilisateur, elle permet de définir les méthodes sur les classes, donc d'envoyer les messages, et aussi créer des variables d'instances au niveau des classes [29, 18].

L'introduction de méta-classe pose également des problèmes. On peut considérer le parallèle entre la notion de *classe-objet* et celle de *type-valeur*. Le concept de classe généralise celui de type dans les langages classiques. La méta-classe correspond donc à *typer* les types, ce qui revient à considérer les types comme des valeurs.

Dans la pratique, peu de langages considèrent le type comme valeur, car il existe une certaine difficulté pour manipuler de la même façon les *méta-types* que les types ordinaires. On peut cependant signaler le système de type à 2 niveaux de Cardelli [23] où est introduite la notion de *kinds*. Chaque type est un *kind*. Ce système de kind-type-value permet

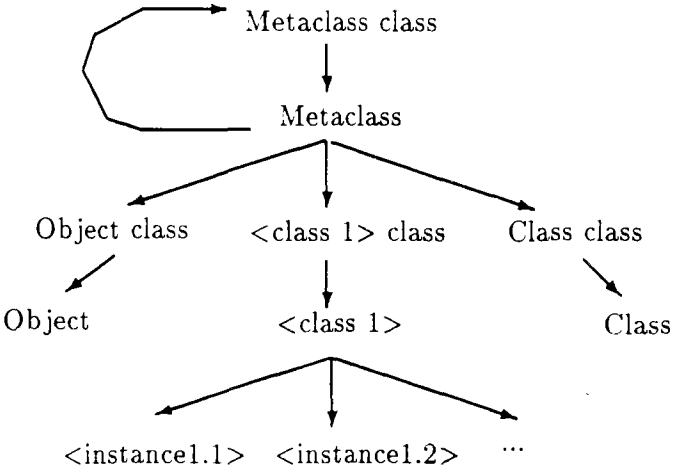


Figure IV.2: Relation d'instanciation en SMALLTALK

facilement de traiter d'une façon plus satisfaisante les problèmes que pose l'introduction de méta-classes.

IV.2.6 Conclusion sur les objets

Nous avons présenté les principaux concepts des modèles objets. Nous pouvons résumer les principales propriétés des langages orientés objets par les éléments suivants :

- facilité de conception. Le formalisme objet est en effet assez proche de la représentation qu'on fait du monde réel ;

- modularité. Les objets sont en effet basés sur le mécanisme d'encapsulation. L'état interne de chaque objet n'est en effet manipulable que par lui-même. La définition d'une classe d'objets est complètement indépendante des autres objets. La modularité est donc une caractéristique inhérente de ce genre de modèle. Le fait qu'un objet ne communique avec l'extérieur que par des messages assure également à l'objet une bonne intégrité. Du fait de cette interface, l'implémentation effective des méthodes peut être changée facilement sans modifier les autres éléments du système. Ces aspect contribue aussi à ce que la programmation par objet constitue un outil puissant et souple de prototypage ;

- partage des codes. Le mécanisme d'instanciation permet d'utiliser le code par un ensemble d'objets. Et le mécanisme d'héritage permet de réutiliser facilement les objets déjà définis.

IV.3 Modèles d'acteurs

Parallèlement au développement des *objets*, C. Hewitt a introduit la notion d'*acteur* dans [58]. Le développement des acteurs a donné lieu à des implémentations des langages expérimentaux tels PLASMA, ACT1 etc...

Bien qu'ayant pour origine des préoccupations typiques de l'intelligence artificielle (IA) et s'étant réalisée dans des langages (ACT...) dédiés à l'IA, l'idée des acteurs a eu deux développements notables :

- D'une part, l'étude des mécanismes des appels de procédures a conduit à la réalisation d'un LISP à liaison statique où les continuations sont accessibles au programmeur, c'est SCHEME (cf. chapitre V).
- D'autre part, la formalisation des systèmes d'acteurs en a fait un des "modèles" calculatoires de la concurrence, parallélismes à communications asynchrones, adapté à des architectures distribuées de processeurs.

IV.3.1 Concepts de base

C. Hewitt a défini dans [58] les concepts fondamentaux des acteurs. La métaphore utilisée est une communauté de scientifiques qui travaillent chacun sur des sujets précis et qui s'échangent des informations entre eux. Un acteur possède sa propre capacité de traitement et de stockage. Au lieu d'utiliser la notion de flux global de contrôle, le modèle d'acteur utilise la communication entre chacune des entités dont le contrôle est assuré par eux-même. Chaque acteur est autonome dans son existence.

La structure interne d'un acteur n'est manipulable que par lui-même, et opaque vis-à-vis d'autres acteurs. Les acteurs s'envoient des messages entre eux. Un acteur est responsable de l'action à effectuer lorsqu'il reçoit un message. Son comportement est déterminé par ses *accointances* (acquaintances) qui sont les acteurs qu'il connaît directement et à qui il peut envoyer des messages, et ses *scripts* qui déterminent l'action à effectuer vis-à-vis d'un message reçu.

Les communications sont asynchrones, i.e. un acteur peut envoyer un message à un autre acteur à tout moment sans se préoccuper de l'état, ni du statut du receveur. Certains modèles d'acteurs vont jusqu'à considérer les messages comme des acteurs à part entière. Ceci peut être vu de la façon suivante : quand un acteur envoie un message, c'est un *messager* qui s'en charge pour que le message arrive à son destinataire. L'envoi de message peut être représenté par un messager qui connaît le destinataire, l'expéditeur et une *enveloppe* du message. Ce messager peut posséder d'autres connaissances dans certains cas, par exemple en cas d'erreur, le destinataire pour le traitement d'erreur.

Les acteurs sont des entités de calcul actives. Les calculs sont dirigés par les événements. Un événement se produit, quand un message est accepté par un acteur.

Le modèle d'acteurs utilise plutôt le concept de prototype que celui de classe.

Une des caractéristiques fondamentales des acteurs vient de l'utilisation de la continuation, c'est-à-dire lorsqu'un acteur envoie un message, il peut spécifier le destinataire du résultat sans interrompre sa propre activité. Cette notion de continuation est plus générale

que la continuation technique de SCHEME [59] que l'on détaillera dans le chapitre suivant. C'est cette caractéristique qui fait que le système d'acteurs est particulièrement intéressant pour le problème de calcul concurrent. Cet aspect ne sera pas développé dans cette thèse, on s'intéressa uniquement à l'utilisation de la continuation pour les problèmes de contrôle de calculs.

Le modèle d'acteurs unifie les notions de procédures, de données et de structures de contrôle par l'utilisation de communication.

IV.3.2 Deux exemples

Nous donnons ici deux exemples de programmation en acteurs. Les exemples sont écrits dans une sorte de pseudo-code.

Le premier exemple, qui est un exemple classique, et qui montre la capacité du modèle d'acteurs à traiter les structures récursives, est celui de calcul de la factorielle d'un nombre entier.

La définition de l'acteur `factorielle` est :

```
factorielle with accointances self
  (communication with n: integer and u: customer
    (if n = 0
        then reply to u with 1
        else
          (let b = new – actor customer with accointances n and u
                (send (n - 1) and b to self))))
```

L'acteur `factorielle` ainsi défini connaît lui-même (accointance `self`, et on suppose que `self` est une variable qui dénote l'acteur recevant le message). Les messages acceptables sont ceux comportant un entier n et un client (customer) u. La réponse à un message accepté est la suivante : si le nombre n est égal à 0, alors renvoyer la valeur 1 au client u spécifié dans le message, sinon un nouvel acteur b ayant le comportement de **customer** et les accointances n et u est créé, et l'acteur `factorielle` envoie le message $n - 1$ et b à lui-même. L'acteur **customer** peut être défini par :

```
customer with accointances n and u
  (communication with k: integer
    (reply (n * k) to u))
```

Le comportement de l'acteur **customer** à la réception d'un message avec un entier k est simplement le suivant : il multiplie son accointance n et la valeur du message k et renvoie le résultat à son accointance u qui par définition de **customer** est également un acteur du type **customer**. Pour le calcul de factorielle de 3, on fait l'appel suivant : (send 3 and customer to factorielle) dont **customer** est un acteur qui utilisera la valeur de factorielle de 3 pour sa propre activité, par exemple il peut être simplement un acteur qui imprime le résultat sur l'écran.

Le fonctionnement de ce calcul peut être représenté graphiquement par la figure IV.3.

Le deuxième exemple sur l'implémentation de la structure de données de pile permet de montrer la capacité des acteurs à représenter les objets structurés.

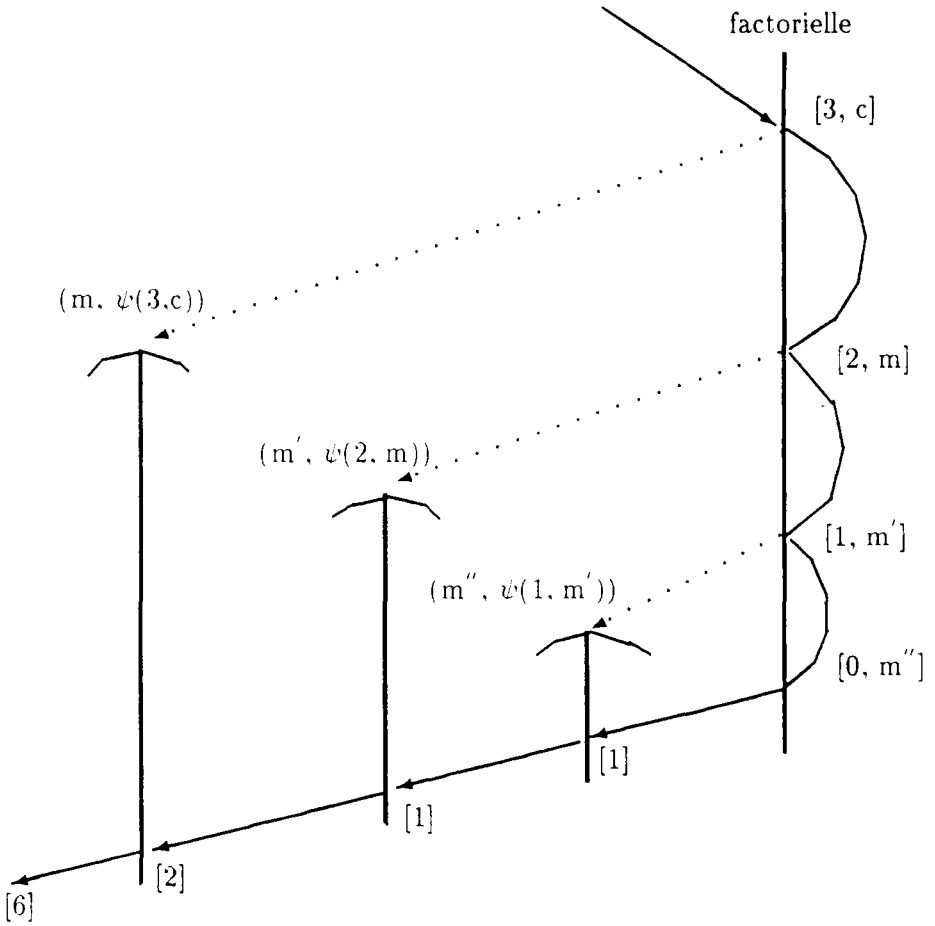


Figure IV.3: Calcul de factorielle

La pile (**stack**) est représentée par une liste chaînée (un premier élément et une autre pile), elle possède deux méthodes : **push** permet d'ajouter un nouvel élément à la pile, et **pop** permet de retirer le premier élément de la pile. D'autres méthodes peuvent évidement être ajoutées. Et la valeur **NIL** est supposée prédéfinie.

```
stack with accointances CONTENT and LINK
  (communication
   (if operation is push with NEW-CONTENT
      then
        let s2 = (new stack with accointances CONTENT and LINK)
         (become stack with accointances NEW-CONTENT and s2))
   (if operation is pop with CUSTOMER and CONTENT <> NIL
      then
        become forwarder to LINK
```

send CONTENT *to* CUSTOMER))

Dans le cas d'un message avec **push** et un acteur **x**, on crée une nouvelle pile ayant pour accointances cet acteur **x** et l'ancienne pile, et le comportement de l'ancienne devient celui de la pile créée (par **become**). Si le message est **pop**, le contenu de la pile est renvoyée au client, et tous les autres messages arriverait à cette pile sera avancés vers la pile **s** (par **forwarder** qui est un acteur permettant d'avancer le message au prochain acteur présent dans une liste).

IV.3.3 Etude de quelques modèles et langages d'acteurs

Les recherches depuis PLASMA ont donné plusieurs modèles basés sur le concept d'acteurs. Dans la lignée de PLASMA, au M.I.T. le développement se porte sur une série de langages ACT1, ACT2 et également sur les machines d'acteurs APIARY, et d'autres projets tels Ether etc .... dans le but de réaliser des langages de programmation basés sur acteurs. Parallèlement, l'étude de PLASMA a continué en France par exemple [87, 74]. D'autres recherches ont été menées en inspirant du formalisme d'acteur, le travail de Yonezama sur le langage ABCL [104], dont nous donnerons des éléments de base, pour la programmation parallèle, en est un exemple.

Nous allons d'abord donner quelques éléments historiques dans les recherches menées au M.I.T., ensuite nous allons présenter les langages ACT1, ACT2 et le dernier modèle d'Agha, enfin nous estimons intéressant de présenter une autre direction de recherche dérivée des acteurs qui est celle de ABCL.

Historique

PLASMA est la première tentative de réalisation d'un langage d'acteurs à partir du modèle d'acteurs proposé par C. Hewitt. Une implémentation complète a été réalisée à Toulouse.

L'objectif de PLASMA est de savoir s'il est possible de réaliser un langage basé sur le modèle d'acteurs.

PLASMA possède certaines facilités pour la communication, mais ne distingue pas les différentes sortes de messages. Les structures de contrôle sont réalisées pas les transmissions de messages qui est l'idée fondamentale développée par Hewitt. Le langage possède également des structures de données simples telles que les nombres, les séquences et les packages. La notion de package ressemble à celle de record en PASCAL, permettant l'encapsulation.

A partir de cette première implémentation d'un langage d'acteurs, sont ensuite développés les le langage ACT1, et les systèmes utilisant la notion d'acteur, pour la représentation de connaissance OMEGA, et pour la résolution de problèmes hautement concurrent ETHER. Le projet APIARY, lui, propose une nouvelle architecture de machines basée sur acteurs. Une simulation d'une telle machine a été réalisée. Les développements de tous ces systèmes et langages sont menés d'une façon indépendante, et dans le souci d'intégrer l'ensemble, il est nécessaire de développer un nouveau langage. Le langage ACT2 est le fruit de cette intégration, il a été écrit en SCRIPT, un langage écrit en LISP qui fournit les interfaces nécessaires à l'architecture d'APIARY.

Langages Act1, Act2

Généralité sur Act1

ACT1 est développé au M.I.T., implémenté en MACLISP. C'est une réalisation directe du modèle d'acteurs.

ACT1 ne manipule que les acteurs. Les nombres, processus, messages etc...sont des acteurs. Un acteur peut être caractérisé par les éléments suivants :

- les accointances qui déterminent l'état local de l'acteur. La liste des accointances peut évoluer au cours du temps ;
- les scripts qui déterminent le comportement de l'acteur vis-à-vis des messages reçus ;
- un mandataire (proxy) à qui l'acteur délégue tous les messages incompris. Ici la notion de délégation remplace celle de l'héritage dans les langages d'objets tels que SMALLTALK.

Création d'acteur en Act1

La création d'un nouvel acteur est réalisée par une copie *différentielle* d'un acteur existant. Chaque acteur répond au message `create` pour créer une nouvelle copie de lui-même. L'acteur créé partage les scripts de l'acteur existant, par contre on peut ajouter ou modifier les accointances de l'acteur à créer en passant en argument dans le message `create` celles-ci.

On peut également créer un acteur par le message `extend`. Un acteur existant recevant le message `extend` crée un acteur ayant de nouveaux scripts et accointances et dont il est le mandataire.

D'autres types de création d'acteurs existent par l'usage de `combine` qui combine les scripts de deux acteurs existants, et de `contrast` qui crée un acteur ayant pour scripts la différence de scripts de deux acteurs.

Un acteur prédéfini `OBJECT` possède un certain nombre de scripts permettant de répondre par défaut à des messages tels `print`, `create`, `extend`, `eval` etc.... Tous les acteurs déléguent directement ou indirectement les messages précédents à `OBJECT`.

Messages dans Act1

Dans ACT1, les messages sont également des acteurs. Ceci permet d'associer au message même des méthodes par défaut, soit stockées dans `OBJECT`, soit dans le message en question. Par exemple, `OBJECT` possède une méthode `print` permettant d'imprimer le contenu d'un acteur, et certains acteurs possèdent une méthode `pretty-print`. Quand on envoie le message `pretty-print` à un acteur, au cas où ce dernier ne comprend pas, on peut spécifier le traitement par défaut dans ce message :

> Si mon receveur comprend `pretty-print`
> alors lui donner l'ordre `pretty-print`
> sinon lui donner l'ordre `print`

Dans `OBJECT`, on implémente une méthode agissant de la façon suivante :

Si je n'ai pas de méthode pour traiter un message
alors je demande à ce message quelle est sa méthode par défaut

A la réception de message, l'identification de méthode à employer est effectuée par un filtrage (pattern-matching) au lieu d'utiliser les mot-clés comme en SMALLTALK par exemple.

Plusieurs sortes de messages sont également distingués dans ACT1. Il y a principalement les messages de **demande** (request) et le message de **retour** (**reply**). Dans chaque message, on peut spécifier la continuation de succès (acteur pour **reply**) et d'échec (acteur `complaint`).

Parallélisme en Act1

ACT1 est en effet conçu pour supporter le parallélisme autour d'une architecture parallèle de processeurs APIARY. Le langage est parallèle par nature. ACT1 offre des constructions facilitant la programmation concurrente [68].

Les objets prédéfinis d'Act1

Il existe un certain nombre d'objets de bas niveaux qui sont prédéfinis dans ACT1, tels que les nombres. Ces acteurs sont simulés par des objets et procédures LISP. Etant un langage de recherche, ACT1 ne possède pas un grand nombre d'objets prédéfinis.

Le langage Act2

ACT2 est la suite logique de ACT1 [97], en intégrant les possibilités dégagées par la recherche sur OMEGA. ACT2 se veut un langage extensible, pour supporter les différents besoins de génie logiciel, de l'I.A.... Nous donnons uniquement quelques éléments intéressants du langage.

ACT2 fournit un mécanisme d'abstraction (par **define**), permettant d'unifier les structures de données, les procédures et les structures de contrôle. Les objets du langage sont tous des objets de première classe, y compris les environnements que l'on peut accéder, modifier et abstraire.

Comme dans ACT1, l'acceptation des messages est faite par le mécanisme de filtrage (pattern-matching). Ce dernier est aussi utilisé pour l'extraction des informations d'une description des instances, l'authentification, la comparaison des acteurs, les liaisons des variables des acteurs etc... C'est un mécanisme fondamental.

Dans ACT2, sont distingués deux sortes d'acteurs : acteur sérialisé et non sérialisé. Les acteurs non sérialisés sont des acteurs qui ne changent pas d'état en cours de leur vie. Ces acteurs peuvent être donc créés et multipliés, et peut recevoir plusieurs messages en même temps. Les acteurs sérialisés, quant à eux, peuvent changer d'état interne et doivent donc avoir une synchronisation pour la réception de messages.

Le fait que ACT2 est construit à partir d'une sémantique claire, lui permet d'exploiter entièrement sa fondation qui est bien définie et spécifiée formellement.

Modèle d'Agha : fonctionnement

L'un des modèles d'acteur les plus récents a été proposé par Agha [3]. Ce modèle essaie d'unifier la facilité et les concepts de programmation par objects et l'élégance de la programmation fonctionnelle. Le modèle présente un point de vue élégant et intéressant, et possède une sémantique assez claire.

Nous allons décrire dans la suite les principaux éléments de ce modèle, et surtout le mécanisme de traitement de message appelée *remplacement de comportement* (behavior replacement).

Maintenant, un acteur peut être défini en spécifiant les éléments suivants :

- une *adresse* (*mail adress*) à laquelle correspond une *file d'attente* (*mail queue*) ;
- son *comportement* (*behavior*), qui est une fonction des communications acceptées.

Le mécanisme de traitement de messages, appelé *remplacement de comportement (behavior replacement)*, et proposé par [3] est illustré par la figure IV.4.

L'acteur **X**, en recevant un message (le n^{ieme} de la file d'attente), crée un acteur **X+1** ayant le même comportement qui le remplace pour traiter le message suivant. Pendant ce temps, il traite le message en créant éventuellement de nouvelles tâches (envoi de messages à d'autres acteurs etc...) ou/et de nouveaux acteurs (sur la figure l'acteur Y). Ce mécanisme est en effet une utilisation de continuation, et permet donc à un acteur de traiter des messages en parallèle.

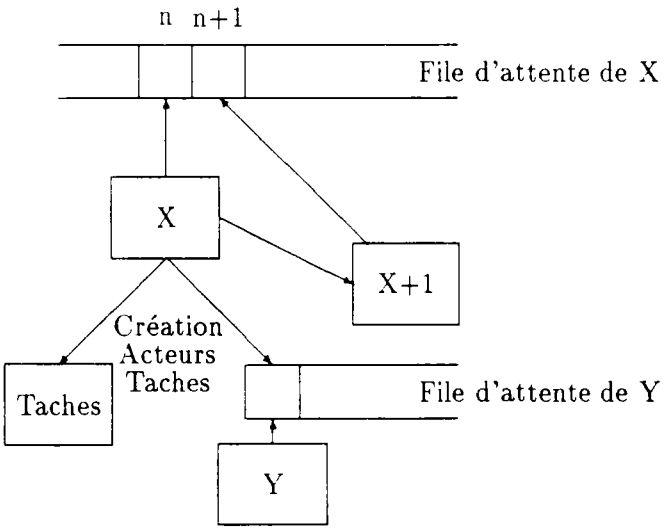


Figure IV.4: Traitement de messages par acteur

Ce mécanisme de remplacement de comportement offre une alternative élégante à la notion des *états* des objets. Par ce mécanisme, le modèle propose une solution intéressante

au problème de l'"histoire" des objets. Une analogie est faite avec les *streams* [3].

Ce modèle s'inspire largement de CCS (Calculus for Communicating Systems) de Milner [76].

Les éléments de base du modèle

Les constructions de base dans ce modèle sont :

- *définition de comportement*. Le comportement d'un acteur est défini par deux listes d'identificateurs, dont le premier est une liste d'accointances qui sont données au moment de création de l'acteur, et la seconde est une liste de communications. L'acteur exécute les commandes dans l'environnement défini par les liaisons des identificateurs dans les deux listes ;

- *création d'acteurs*. La création est faite à partir d'une définition de comportement en instanciant la liste des accointances. Tous les acteurs créés parallèlement par un acteur se connaissent mutuellement ;

- *création de tâches*. Les tâches sont créées par les envois de messages, c'est-à-dire en spécifiant le destinataire et le contenu du message ;

- *déclaration des réceptionnistes* dans un système d'acteurs. Dans ce modèle, dans le soucis d'augmenter la modularité des programmes, il a été introduit la notion de *réceptionnistes*, c'est-à-dire dans un système d'acteurs seuls peuvent recevoir les messages de l'extérieur les acteurs appelés réceptionnistes. Du fait de l'évolution dynamique du système, le nombre de réceptionnistes peut changer au cours de l'exécution ;

- *déclaration des acteurs externes*. Comme pour le réceptionniste, les acteurs d'un système ne peuvent envoyer des messages à l'extérieur du système que si ces derniers sont explicitement déclarés comme externes. Le but d'utilisation des acteurs externes est le même que celle des réceptionnistes ;

- comportement par *défaut* des acteurs ;

- *les commandes*. Les différentes catégories de commandes sont

  - les conditionnelles ;
  - les envois de messages par `send` ;
  - le remplacement de comportement par `become` ;
  - les liaisons locales par `let`.

Le remplacement de comportement ne pose aucun problème aux acteurs non sérialisés (ceux qui ne changent pas de comportement au cours de leur vie). Par contre pour les acteurs sérialisés, le remplacement acquiert une certaine synchronisation. Ce problème est résolu par l'introduction des acteurs *insensibles* (*insensitive actors*). Pour illustrer le

fonctionnement de ces acteurs, nous allons prendre l'exemple donné par [3] du compte bancaire (bank acount).

Sur un compte bancaire, on peut réaliser les opérations de dépôt, de retrait et de demande du solde. Une protection doit être mise au point pour le cas où le montant de retrait dépasse celui du solde.

Dans cet exemple, on définit un acteur appelé `checking-acc`. S'il reçoit un message de retrait et si le montant de retrait est supérieur au solde du compte, l'acteur checking-acc devient un acteur insensible ayant pour accointances un acteur `buffer` et un autre acteur appelé `overdraft-processor`. L'acteur insensible consiste à mettre tous les messages dans le `buffer`, et s'il reçoit un message du type `become`, i.e. de remplacement de comportement, il continuera à traiter tous les messages stockés dans `buffer`.

Le modèle d'Agha permet à partir d'un noyau minimum de réaliser des mécanismes offerts par d'autres langages tels que l'évaluation retardée par une simple extension de syntaxe.

Modèle ABCM, et le langage ABCL

Généralités

La préoccupation du modèle ABCM et de son langage ABCL [104] est un peu différente de celle des recherches menées au M.I.T. ABCL utilise le concept de base, mais ne cherche pas à être uniforme pour tous les éléments. Le nom ABCL (An object Based Concurrent Language) montre bien les intérêts qu'ont les auteurs pour la concurrence et objets orientés (dans cette partie, nous utilisons le terme objet et acteur indifféremment).

Les objets

Un objet au temps t est représenté par un état, et un ensemble de scripts.

Les opérations de base qu'un objet peut exécuter sont de quatre sortes :

- les passages de messages ;
- la création d'autres objets ;
- la déréférenciation et la mise à jour de son état interne ;
- les opérations telles que arithmétiques ou autres sur les valeurs stockées dans la mémoire interne.

Pour définir un objet, il faut connaître les éléments suivants :

- comment est représentée sa mémoire locale ;
- dans quelles conditions les messages sont acceptés ;
- quelles sont les séquences d'actions à effectuer si un message est accepté.

Un objet possède une unique file d'attente pour les messages qui sont traités d'une façon séquentielle.

Un objet est toujours dans l'un des états suivants : dormant, actif ou attente (dormant, active or waiting). Un objet est initialement dans l'état dormant. Un objet à l'état dormant est activé par la présence dans sa file d'attente des messages acceptables par cet objet. Un objet avec un état local ne peut traiter plus d'un message à la fois. Quand l'objet actif finit de traiter tous les messages dans sa file d'attente, il redevient dormant.

Il peut arriver à un objet d'attendre des messages spécifiques pour continuer le traitement d'autres messages présents dans sa file d'attente, alors cet objet entre dans le mode d'attente (waiting). En passant du mode actif au mode attente, l'objet réalise une opération spéciale qui est spécifiée par la construction `wait-for` en ABCL. Cette construction spécifie en effet les conditions que doit satisfaire un message pour réactiver l'objet. Ce mode de fonctionnement est en effet très voisin des acteurs insensibles d'Agha.

La différence entre le mode attente et le mode dormant réside dans la manière de traiter les messages. En mode dormant, tous les messages se trouvant avant l'arrivée d'un message acceptable sont rejetés, alors que dans le mode d'attente, tous ces messages restent où ils sont.

Les messages

Les messages en ABCL satisfont généralement les conditions suivantes :

- un objet ne peut envoyer un message à un autre objet que si le premier connaît le second. Cette relation de connaissance peut être dynamique ;
- les messages sont asynchrones, c'est-à-dire qu'un acteur peut envoyer à tout moment un message à un autre acteur sans savoir l'état et le mode de ce dernier (sauf pour le "now type") ;
- les messages arrivent toujours à leur destinataire en un temps fini et sont stockés dans l'unique file d'attente associé à l'objet en question ;
- les arrivées de messages sont supposées linéaires ;
- l'ordre de transmission est respecté à l'arrivée ;
- il n'y pas d'utilisation d'une *horloge* globale.

Trois types de passages de messages sont également distingués qui sont *passé, maintenant et future* (past, now and futur).

- type de passage passé (past type). C'est le type de message "envoyer et ne pas attendre" (send and no wait). Quand un objet O est activé, et envoie un message M à T. O continue la suite de calcul qu'il doit exécuter immédiatement après avoir envoyer le message, il ne se préoccupe pas de savoir si M est arrivé à T. Ce type de message augmente considérablement la concurrence des calculs ;

- type maintenant (now type). C'est le type "envoyer et attendre". O attend non seulement que M soit arrivé à T, mais aussi la réponse que renvoie T à la question posée. Il est à noter que la réponse n'est pas nécessairement renvoyée par T, car T peut déléguer M à un autre acteur ;

- type future (futur type). C'est le type renvoyez moi plus tard. La réponse est à renvoyer à O, mais O n'a pas besoin de cette réponse pour continuer le calcul. Au moment où il en a besoin, il regarde dans un objet privé crée au moment de transmission de M appelé *future object* si le résultat est présent. L'utilisation de ce type de transmission permet d'économiser le temps de calcul par rapport à l'utilisation directe du type maintenant dans certain nombre de cas.

Afin de simuler l'interruption du traitement d'un message par un acteur (le cas courant d'une personne travaille au bureau et est interrompue par le téléphone), ABCL introduit deux modes pour les messages : ordinaire et express-mode. Ce dernier peut être vu comme un cas simple du mécanisme d'interruption.

En combinant les trois types et ces deux modes, on obtient ainsi six façons différentes d'envoi de messages.

Destination de retour

En ABCL, le receveur du message connaît l'acteur qui lui a envoyé le message. Il doit connaître le destinataire pour la réponse qui n'est pas toujours l'expéditeur, comme dans le cas du message de type *now*. L'objet qui reçoit la réponse d'un message est appelé *destinataire de retour*. Dans le message, une variable appelée *variable de destination* est spécifiée pour que le receveur puisse savoir quel est le destinataire, ceci quel que soit le type de message. Et cela a une importance sur le fait que l'on n'a pas à savoir à quel type de message que le script a affaire, lorsqu'on définit ce dernier.

Dans le cas où le receveur T délégue le message à un autre objet T', la variable de destination est celle spécifiée par le message de départ. Il n'est donc pas nécessaire pour la réponse de repasser par T.

Création d'acteurs

Les nouveaux objets peuvent être créés dynamiquement au cours de calcul. Il existent deux moyens principaux de créations de nouveaux objets. Le moyen le plus courant est de définir un objet permettant d'initialiser les nouveaux objets en recevant un message spécifique. Ce mécanisme est proche de celui utilisé en SMALLTALK, à la différence qu'ici la notion de classe est implicite.

L'autre façon de créer de nouveaux objets est de faire des copies des objets existants, c'est un mécanisme proche de ACT1.

Conclusion sur ABCL

Le modèle ABCM et son langage correspondant ABCL ont une vue plus pragmatique de l'utilisation du modèle d'acteurs en combinant les concepts des langages orientés objets

et ceux des acteurs pour construire un environnement de programmation concurrente pour des architectures de machines multiprocesseurs.

Le langage montre des possibilités intéressantes sur le mécanisme de synchronisation par la distinction de trois modes des objets et de trois types des messages.

IV.3.4 Délégation ou héritage

Le mécanisme de partage des connaissances dans les modèles d'acteurs est en général celui de délégation. Ce choix est dû essentiellement à l'utilisation de concept de *prototype* comme base de création d'objets. Par rapport au modèle de SMALLTALK, le concept de classe n'existe pas dans les langages d'acteurs, et plus généralement dans tous les langages basés sur les prototypes [100]. Un prototype [9] est une instance d'exemple standard, tous les nouveaux objets sont produits par une copie et/ou une modification du prototype plutôt qu'une instantiation d'une classe.

Le fonctionnement de la délégation est intuitivement très simple : chaque acteur possède un *mandataire* (proxy) (qui correspond en effet à la super-classe de l'objet). Quand un acteur reçoit un message, il essaie de traiter ce message par ses propres méthodes. S'il n'existe pas de méthode appropriée, il envoie ce message à son mandataire (proxy).

Par rapport au mécanisme d'héritage, la délégation se différencie par le fait qu'elle est également un envoi de message, alors que l'héritage est un mécanisme câblé à l'avance qui est un concept différent de ceux déjà utilisés dans le modèle. Néanmoins, [93] montre l'équivalence de ces mécanismes, i.e. on peut réaliser l'héritage à partir de la délégation et vice versa.

La réalisation de délégation est en principe très facile, mais peut poser également certains problèmes dans le traitement du calcul parallèle. Dans le modèle d'Agha, ce mécanisme est volontairement laissé à côté.

Pour une discussion détaillée de différentes méthodes de partage de connaissances, on peut se reporter à [19]. Dans [19], est également proposé un autre mécanisme pour pallier les problèmes dans un contexte de parallélisme. Ce mécanisme appelé *query-recipe* repose sur le principe suivant : un objet, au lieu de déléguer le message à son mandataire, reçoit de ce mandataire la méthode appropriée pour traiter le message.

Nous ne développons pas ces problèmes en détail, car ils ne sont pas fondamentaux dans notre recherche. Comme on verra plus tard, nous adaptons le mécanisme de délégation pour utiliser les connaissances communes entre les différents objets, et la création de nouveaux objets est intimement liée à la délégation (voir partie implémentation).

IV.4 Conclusion

Les deux formalismes de représentation de connaissances ont donné des modèles de programmation particulièrement intéressants. Il existe de nombreuses tentatives pour intégrer ces deux formalismes dans un même modèle, et ceci a donné naissance à des langages hybrides qui permettent une structuration en objet et des inférences sur les connaissances.

Nous nous intéressons dans cette recherche particulièrement au formalisme objet, du fait que la plupart des connaissances manipulées dans le cadre d'un diagnostic thermique sont exprimables sous forme de modèles mathématiques bien précis. La réalisation d'un système expert sera sans doute bien adaptée dans certains cas particuliers comme le pré-diagnostic, ou certains outils de diagnostic destinés au grand public dans le cadre de l'habitat individuel.

Le modèle d'objet présente un intérêt particulier dans le cadre de l'outil que l'on cherche à réaliser, car l'organisation des objets physiques peut directement être transcrite dans ce modèle.

Les acteurs sont un cas particulier des objets (une spécialisation au sens des modèles objets), ce sont des objets actifs. Bien que les recherches actuelles dans ce domaine soient plus intéressées par son utilisation en calcul parallèle (les acteurs, comme on a vu, sont des entités calculatoires naturellement parallèles), il nous paraît clairement utile et intéressant d'adapter la structure de contrôle dans le cas de l'audit pour pouvoir manipuler d'une façon simple et élégante la multiplicité des modèles thermiques présents dans le calcul.

Combiner l'approche objet et acteur sera donc d'un grand intérêt au point de vue d'organisation des objets et de calculs.

Chapitre V

Langage Scheme et Continuation

Les idées formées par l'intelligence pure n'ont qu'une vérité logique, leur vérité possible, leur élection est arbitraire (...) Seule l'impression si chétive que me semble la matière, si invraisemblable la trace, est un critérium de vérité et à cause de cela mérite seule d'être appréhendée par l'esprit, car elle est seule capable, s'il sait en dégager cette vérité, de l'amener à une plus grande perfection et de lui donner une pure joie.

— *M. Proust*

Nous allons dans ce chapitre donner quelques éléments essentiels du langage SCHEME. Les principales différences de SCHEME par rapport aux LISP classiques seront mises en évidence. La notion de continuation sera développée en utilisant SCHEME. L'utilisation des extensions de nouvelles formes syntaxiques sera expliquée.

Le but de notre exposé étant de présenter le style de programmation en SCHEME, et de familiariser avec les principales fonctions et syntaxes utilisées dans le texte et quelques concepts introduits par SCHEME, notre présentation ne sera donc pas formelle.

V.1 Généralités

Inventé par G. J. Sussman et G. L. Steele Jr., vers 1975, SCHEME[1] [85] est un dialecte de LISP[2], car il hérite de LISP la clarté et la simplicité due à la structure de représentation des objets (données et procédures) par des listes. Par contre, sa lexicalité de liaison des variables et sa structure de bloc s'inspirent d'ALGOL60. SCHEME est essentiellement un LISP

[1] Prononcé [ski :m]. La version de SCHEME utilisée est celle de *ChezScheme* sur Sun-3. La différence de cette implémentation par rapport aux autres telles que MACSCHEME ou PCSCHEME réside essentiellement dans la définition de macros.

[2] LISP est l'acronyme de LISt Processing.

fonctionnel, mais offre également l'affectation, et quelques structures de contrôle. C'est le premier LISP à avoir adopté la liaison statique, et aussi le statut d'objet de première classe pour les procédures et les continuations. Il y a un seul environnement lexical pour toutes les variables. Il est également proprement récursif terminal, c'est-à-dire qu'il traite toutes les récursions terminales comme des processus itératifs.

SCHEME trouve son origine dans le développement des acteurs. Il a influencé le développement de COMMON LISP [92] (et a été influencé par celui-ci), notamment en ce qui concerne les liaisons statiques.

La simplicité, la clarté sémantique et la souplesse font de SCHEME un langage d'enseignement et de recherche par excellence. Un nombre de plus en plus grand d'applications se sont développées autour de ce langage [8].

Nous allons dans la suite développer les principaux éléments du langage.

V.2 Les éléments de base du langages

V.2.1 Interprète

Comme tous les LISP, SCHEME est d'abord un langage interprété dans son utilisation courante. Son mode de fonctionnement consiste donc à exécuter continuellement la boucle lecture-évaluation-écriture (read-eval-print).

Nous utilisons les notations suivantes pour la suite de la présentation : une expression à évaluer sera précédée d'un point d'interrogation (?), la valeur retournée par $\Longrightarrow$.

V.2.2 Les expressions de base

Une expression en SCHEME consiste en un symbol simple (un atom), ou une liste (une s-expression).

Listes

La liste est la structure de base en LISP, et est la seule structure pour tous les objets du langage (données, appel de procédures, les formes syntaxiques). Une liste est de la forme suivante :

```
(<exp1> <exp2> ...)
```

Comme tous les autres LISP, SCHEME offre les constructeurs et les sélecteurs de base pour la manipulation des listes : `cons`, `car`, `cdr` etc.

Commentaires

Les commentaires sont compris entre un point virgule (;) et la fin de ligne. Exemple :

```
?   ; ceci est un commentaire
```

Constantes

Les constantes sont des nombres, des chaînes de caractères, des caractères, des valeurs booléennes (`#f et #t`)[3] et des expressions quotées. Par exemple, les expressions suivantes sont évaluées en elles-même :

```
? 222  ; un nombre
? #j   ; un caractère
? #t   ; booléen   ou #f
? "chaine de caracteres"
```

Et les expressions quotées :

```
? 'tre.uhs*hgs ; une expression quotée
? (quote abcd) ;
```

Variables

Les identificateurs de variables sont formés de caractères alphabétiques. Par exemple, les identificateurs suivants sont valides : `+, s3, un-long-identificateur`. Il existe un petit nombre de *mots-clé* réservés qui ne peuvent pas être utilisés comme nom de variables[4] :

`define`	`let`	`let*`
`letrec`	`cond`	`else`
`case`	`if`	`lambda`
`quote (')`	`quasiquote (`)`	`unquote (,)` .
`unquote-splicing (,@)`	`set!`	`begin`
`and`	`or`	`do`
`delay`		

Une variable peut dénoter une expression simple, ou une procédure. Un seul espace de nom est utilisé pour tous les types de variables. Ceci constitue l'une des différences essentielles avec COMMON LISP.

Appel fonctionnel

La syntaxe de l'appel fonctionnel est :

```
(proc arg1 ...)
```

La procédure et les arguments sont d'abord évalués, ensuite les arguments sont passés dans la procédure. Par exemple

```
? (+ 5 7)              ⟹ 12
? (* 6 7 (+ 7 8 9))    ⟹ 1008
```

`+`, `*` sont deux variables qui dénotent les procédures d'addition et de multiplication.

L'ordre d'évaluation de la procédure et des arguments n'est pas spécifié, ceci constitue une des différences avec les LISP classiques.

[3] Les valeurs #f, nil, () (liste vide) sont considérées comme fausse, toutes les autres valeurs sont vraies.

[4] Même si dans certaines implémentations de SCHEME, on peut utiliser ces mots-clé pour les variables, il est fortement déconseillé de les utiliser car cela peut créer des ambiguïtés

Formes spéciales

Il existe un certain nombre de formes spéciales prédéfinies telle que `lambda`. On peut également définir les nouvelles formes spéciales (syntaxes ou macros). On reviendra sur cette question plus tard. Les identificateurs utilisés pour les formes spéciales (mots-clé) prédéfinies ne sont pas utilisables pour les variables.

Lambda expressions

Les lambda expressions sont très importantes, car elles permettent de définir des fonctions. La syntaxe d'une lambda expression est

(*lambda* <parametres> <corps>)

Une lambda expression est évaluée en une procédure. Par exemple

```
? (lambda (x) (* x x))
      ;fonction à 1 argument et retourne le carré
⟹ #<procedure>
? ((lambda (x)
      (* x x))
   5)
⟹ 25 ;; l'application de la fonction à l'argument 5
? (define square
     (lambda (x) (* x x)))
⟹ square  ; ou non spécifiée
? (square 5)  ⟹ 25
```

<parametres> sont de l'une des formes suivantes :

- (<variable$_1$ > ...). Le nombre de paramètres de la procédure est fixe. Quand la procédure est appelée, les variables sont liées aux arguments correspondants ;
- <variable>. La procédure prend un nombre quelconque d'arguments. Par exemple

```
? ((lambda x x) ;; fonction identité
   1 2 3 4)     ;; appliquée à 1 2 3 4
⟹ (1 2 3 4)
```

- (<variable$_1$ > ... <variable$_{n-1}$ > . <variable$_n$ >).

  La valeur stockée dans la <variable$_n$> sera la liste des arguments restants après la liaison des arguments à des variables avant le point (.). Par exemple

```
? ((lambda (a b . c) ;; fonction retournant la liste c
      c)
   2 3 4 5 6)
⟹ (4 5 6) ; a est lié à 2, b à 3 et c à (4 5 6)
```

<corps> est une séquence d'expression.

Conditionnelle

La conditionnelle est réalisée par la forme spéciale **if**

```
(if <test>
    <exp1>
    <exp2>)
```

L'expression <test> est d'abord évaluée, si elle est **#t**, <exp1> est évaluée et sa valeur est retournée, sinon <exp2> est évaluée.

Il existe d'autres formes spéciales dérivées[5] pour les conditionnelles : cond et **case** sont les formes que l'on utilise souvent dans notre implémentation et dont nous donnons la syntaxe.

Voici la syntaxe de cond

```
(cond <clause1> ...)
```

où chaque <clause> est de la forme

```
(<test> <expression> ...)
```

et la dernière clause peut être (**else** est un mot-clé)

```
(else <expression1> <expression2> ...)
```

Chaque <test> est évalué successivement, si un des <test> retourne la valeur **#t**, la séquence des expressions est évaluée et la valeur de la dernière <expression> est retournée. Si aucun de <test> n'est satisfait, la clause **else** est toujours évaluée, si elle est présente.

Par exemple :

```
? (cond ((> 1 2) 'un)
        ((< 1 2) 'deux)
 ⟹ deux
? (cond ((> 1 1) 'un)
        ((< 1 1) 'deux)
        (else 'ok))
 ⟹ ok
```

La syntaxe de **case** est la suivante :

```
(case <cle>
  <clause1>
  ...
```

où chaque <clause> est de la forme

```
((<datum1> ...) <expression1> ...)
```

et la dernière clause peut être une else-clause comme dans cond. <cle> est d'abord évaluée, ensuite sa valeur est comparée à chaque de <datum> d'une clause, si l'un de <datum> est

[5] Les formes dérivées ne sont pas indispensables, mais permettent une utilisation plus aisée de certaines formes.

égale à cette valeur la suite des expressions est évaluée et la valeur de la dernière expression est retournée. Si aucun test n'est satisfait, else-expression est évaluée.

Par exemple

```
? (define f
     (lambda (n)
       (case (+ 1 n)
         ((2 4 6 8) 'paire)
         ((1 3 5 7) 'impaire)
         (else 'trop-grand))))
 ⟹ f
? (f 1)          ;; n est lié à 1
 ⟹ paire
? (f 2)
 ⟹ impaire
? (f 16)
 ⟹ trop-grand
```

Définition

Un programme en SCHEME consiste en une séquence d'expressions et de définitions. La forme spéciale **define** permet la définition au Top-Level. La syntaxe de **define** est la suivante :

```
(define <variable> <expression>)
```

Cette définition n'est valide que si c'est en Top-Level (c'est le cas en *ChezScheme*). La définition a pour effet de lier la variable à l'expression qui peut être une valeur constante, fonctionnelle ou autre.

Par exemple

```
? (define dix 10)
 ⟹ dix
? (+ dix 1)
 ⟹ 11
? (define premier car)
 ⟹ premier
? premier
 ⟹ #<procedure car>
? (premier '(a b c))
 ⟹ a
? (define f
     (lambda (x y)
        (+ x y)))
 ⟹ f
? (f 3 5)
 ⟹ 8
```

Remarque : SCHEME permet aussi de définir les procédures par une syntaxe simplifiée. Au lieu de

```
(define f
   (lambda (x ..)
    ...))
```

on peut écrire

```
(define (f x ...)
   ...)
```

Nous utiliserons dans notre texte que la première forme.

Liaison locale des variables

Une lambda expression définit en effet un nouvel environnement où les paramètres seront liés aux arguments passés quand la procédure est invoquée. Une lambda expression permet donc de définir les liaisons locales des variables ; certaines formes en sont dérivées qui sont `let, let*, letrec`. Les trois formes ont une syntaxe semblable.

```
(let ((id val) ...)
  corps)
```

qui est équivalente à

```
((lambda (id ...)
   corps)
 val ...)
```

`let` a pour effet de lier la variable `id` à la valeur d'évaluation `val`, et cette liaison n'est valide qu'à l'intérieur du bloc de `let`. L'ordre de chaque liaison de `(id val)` n'est pas spécifié.

`let*` fonctionne de la même façon que `let`, mais il y a un ordre d'évaluation, donc la clause suivante peut utiliser la liaison établie par les clauses précédentes. Cette forme est équivalente à

```
(let ((id1 val1))
  (let ((id2 val2))
    ...
      corps))
```

`letrec` est similaire à `let`, mais permet de définir les objets mutuellement récursifs. Par exemple la définition des fonctions paire et impaire pour les nombres entiers :

```
? (letrec
    ([pair?
      (lambda (n)
        (if (= n 0)
            #t
            (not (impair? (- n 1)))))]
     [impair?
```

```
        (lambda (n)
          (if (= n 1)
              #t
              (not (pair? (- n 1)))))])
    (pair? 5)
⟹ #f
```

Cette forme

```
(letrec ([id1 val1] ...)
  e1 e2 ...)
```

peut être écrite en terme de `let` et de `set !` (voir affectation) de la façon suivante :

```
(let ([id1 #f] ...)
  (set! id val)
  e1 e2 ...)
```

`letrec` est préféré à `let` quand il s'agit de liaison de procédures locales.

Affectation

L'affectation est réalisée par la forme spéciale `set !`[6].

```
(set! <variable> <expression>)
```

`<variable>` doit être déjà liée dans l'environnement courant, sinon une erreur est signalée. `<expression>` est évaluée, et sa valeur sera stockée dans `<variable>`. La valeur retournée par `set !` n'est pas spécifiée.

Séquences

Les expressions de SCHEME peuvent être évaluées en séquence. La forme spéciale `begin` permet d'effectuer cette opération :

```
(begin
  <expression1>
  <expression2>
  ...)
```

Chaque expression est évaluée successivement, et le résultat de la dernière est retournée.

V.2.3 Récursion, itération

Une procédure récursive est une procédure qui fait appel à elle-même. Par exemple :

```
? (define length
    (lambda (l)
      (if (null? l)
```

[6] Les fonctions d'affectation sont en générale terminées par !, et les prédicats (exemple eq ?) se terminent par ?.

```
        0
        (+ 1 (length (cdr l))))))
```

Une récursion terminale est un appel récursif qui se produit en dernier appel. L'exemple précédent n'est pas une récursion terminale, car après l'appel récursif de `length`, il y a un autre appel à la fonction `+`. Toute récursion terminale est transformée en une itération (le mécanisme est expliqué dans la partie sur la continuation).

Il existe également des formes pour réaliser les itérations, bien que cela ne soit pas nécessaire théoriquement. La forme `do` est pratique :

```
(do ([id val update] ...)
    (test res ...)
    exp ...)
```

qui retourne la valeur de la dernière `res`. Les `id` sont initialisés avec `val`, et sa nouvelle valeur est donnée par `update` après chaque itération. Après la liaison des `id`, le `test` est évalué, s'il est vrai, la séquence des `res` est évalué, sinon la séquence des `exp` est évaluée et l'itération continue.

Remarque : L'application d'une même procédure à une liste d'objets peut être réalisée par la fonction `map` sans nécessairement recours à la récursion, ni à l'itération.

V.3 Quelques concepts exprimés en Scheme

V.3.1 Les objets de premières classes

SCHEME est le premier langage à introduire la notion d'objet de première classe pour les procédures et les continuations.

Un objet de première classe signifie que l'objet peut être

- créé dynamiquement, et détruit seulement si plus aucune référence ne le concerne (par le garbage-collector ou glâneur de cellules en français) ;
- anonyme ou nommé et stocké dans une structure de données ;
- passé en argument d'une procédure ou retourné comme le résultat d'une procédure.

Les procédures en SCHEME ont donc les mêmes droits que n'importe quel objet du langage tels que les entiers ou les listes.

Les continuations sont également des objets de première classe, ceci permet de manipuler directement les structures de contrôle telles que les échappements. Sur ce point, on revient plus tard.

V.3.2 Liaison statique, fermeture

Comme on a vu, SCHEME est le premier LISP à adopter la liaison statique. Ce mode de liaison est intimement lié aux concepts de *bloc* et et de *portée de variables*. On a vu

que toutes les formes de liaison des variables sont dérivables de **lambda** (sauf **define**). L'évaluation d'une lambda expression crée une fermeture.

Une fermeture (*closure*) est formée d'un *environnement* et d'une procédure. Un environnement contient les liaisons de variables. Dans une fermeture, toutes les variables libres de la lambda expression sont liées dans cet environnement, et au moment de l'appel de la fermeture, l'environnement est augmenté de nouvelles liaisons formées par les paramètres et les arguments. La recherche de liaison s'effectue donc simplement de la façon suivante : on cherche d'abord la liaison dans l'environnement du bloc courant, et ensuite dans l'environnement du Top-Level.

L'utilisation de fermeture permet de résoudre le problème bien connu de *funarg*.

Voici un exemple simple :

```
? (define twice  ;; appliquer f deux fois
    (lambda (f)
      (lambda (x)
        (f (f x)))))
⟹ twice
? ((twice 1+) 5)
⟹ 7
```

En effet, la première application de `(twice 1+)` crée une fermeture avec l'environnement où `f` est lié à `1+`, et la procédure `(lambda (x) (f (f x)))`. Quand on applique cette dernière procédure, `f` est lié. Par contre, dans un LISP à liaison dynamique, on aura un petit problème pour `f`, voici la même définition en LE-LISP :

```
(de twice (f)
    (lambda (x)
      (funcall f (funcall f x))))
⟹ twice
? (funcall (twice '1+) 5)
⟹ eval: f variable indéfinie
```

LE-LISP offre cependant comme d'autres LISP à liaison dynamiques des fonctions pour construire des fermetures.

V.3.3 Evaluation retardée et Streams

Le mécanisme de l'évaluation des arguments dans les applications de procédures est de type appel par valeur comme on a vu plus haut, c'est-à-dire que les paramètres sont d'abord évalués avant d'être passés à la procédure.

Un autre mécanisme d'évaluation des paramètres fonctionne de la façon suivante : quand on applique une procédure à des arguments, les arguments ne seront évalués que lorsqu'ils sont demandés par la procédure. C'est le mécanisme d'évaluation retardée (ou paresseuse). On peut simuler l'évaluation *paresseuse* en SCHEME par l'utilisation de la forme **delay**. Et **force** permet l'évaluation de cette expression.

```
(delay object)
```

```
⟹ delay-object
 (force delay-object)
⟹ résultat d'évaluation de l'objet
```

A partir du mécanisme de l'évaluation retardée, on peut réaliser les structures infinies en utilisant les *streams*. Exemple, si on suppose qu'il existe **cons-stream** :

```
? (define entiers-positifs
    (letrec ((construire
                (lambda (x)
                   (cons-stream x (construire (1+ x))))))
      (construire 1)))
⟹ #<procedure>
? (car-stream entiers-positifs)
⟹ 1
? (car-stream (cdr-stream entiers-positifs))
⟹ 2 ...
```

Le concept de streams offre un modèle alternatif intéressant à l'utilisation de l'affectation qui pose des problèmes quand il s'agit de programmation concurrente [1].

V.4 Continuation

On a vu que l'une des caractéristiques importantes du modèle d'acteur était l'utilisation des continuations. Une façon simple de voir la continuation dans un système d'acteur peut être la suivante : un acteur A envoie un message à un autre acteur B, en lui disant que le résultat du message est destiné à un troisième acteur C. Un message peut donc comporter un destinataire du résultat. Il est évident que pendant que l'acteur B traite le message, l'acteur A peut poursuivre ses propres activités, dans une implémentation parallèle.

Nous allons dans cette partie décrire d'une manière plus générale cette notion essentielle dans les langages de programmation.

La continuation d'un calcul est la représentation de ce qui reste à faire après un calcul. La continuation est en effet la pile d'exécution de ce calcul.

Dans les langages classiques, la continuation n'est pas accessible aux programmeurs.

La continuation permet de réaliser certaines constructions d'une façon simple et élégante, notamment le problème d'échappement, le retour en arrière [57] et aussi les coroutines [56].

V.4.1 Exemple d'utilisation de continuation : retour en arrière intelligent

L'exemple suivant montre l'utilisation de continuation dans un problème de retour-arrière (figure V.1).

Dans cet exemple, les calculs précédents nous conduisent à l'étape A où par exemple la variable X est liée à la valeur a. Deux choix sont possibles pour continuer le calcul, on prend d'abord le choix B qui lie la variable Y à la valeur b. Mais la suite du calcul nous

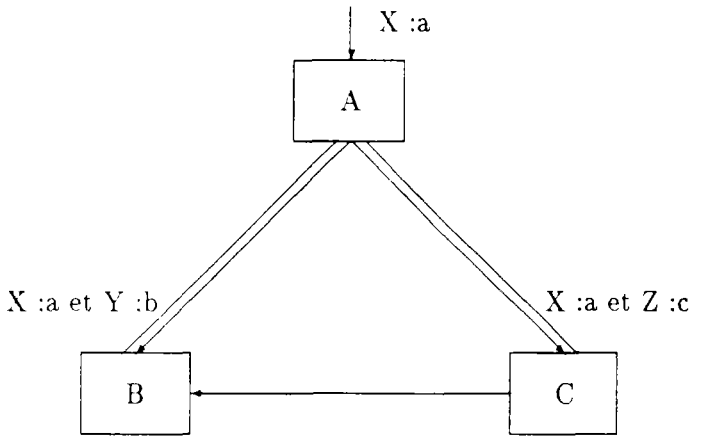


Figure V.1: Exemple d'utilisation de continuation

révèle que ce choix n'était pas bon (par exemple, la solution choisie présente un coût trop élevé par rapport à l'objectif fixé). Dans ce cas on "remonte" le calcul pour choisir C. Il peut arriver que la solution C soit pire que celle de B, on est alors conduit à revenir en B. Dans un backtracking classique, on remonte d'abord en A, et on refait tous les calculs en B. Mais si on a sauvegardé la continuation avant d'aller en C, il suffit d'invoquer cette continuation en B, c'est-à-dire qu'on n'aura pas à refaire les calculs.

V.4.2 Programmation par continuation

L'interprète de SCHEME peut être considéré comme formé de trois éléments : un environnement, l'expression à évaluer et une continuation. Donc après l'évaluation de chaque expression, on peut spécifier ce qu'il faut faire du résultat de l'évaluation. On peut programmer dans ce style.

Par exemple, l'expression

```
(+ 1 (* 2 3))
```

peut être vue de la façon suivante :

```
((lambda (valeur)
     (+ 1 valeur)
  (* 2 3)))
```

En effet `(lambda (valeur) (+ 1 valeur))` est ce qu'il reste à évaluer après l'évaluation de `(* 2 3)`, forme donc la continuation (ou appelé parfois *contexte d'évaluation*) [91]. Dans le style de programmation par continuation, on passe la continuation en argument de l'expression à évaluer. L'exemple précédent s'écrit `(* 2 3 (lambda (v) (+ 1 v)))`..

Reprenons l'exemple de factorielle développé dans le chapitre IV. La définition classique en SCHEME :

```
(define fact
  (lambda (n)
```

```
(if (= n 0)
    1
    (* n (fact (1- n))))))
```

Par le style de passage de continuation, on peut écrire[7]

```
(define cfact
  (lambda (n k)
    (if (= n 1)
        (k 1)
        (cfact (1- n) (lambda (x) (k (* n x)))))))
```

Pour appeler `cfact`, il faut fournir une continuation initiale `k0` qui peut être la fonction identité. Ici l'appel récursif est devenu un appel récursif terminal, ce qui sera traité d'une manière itérative. Mais cette transformation ne veut pas dire que le nouveau code soit plus efficace, en effet à chaque appel, il y une création de nouvelle continuation, donc une création de nouvelle objet dans le *tas* (heap).

V.4.3 Manipulation de continuation en Scheme

Comme on a déjà dit plus haut, SCHEME offre la continuation comme objet de première classe du langage. La fonction qui permet de capturer la continuation courante d'un calcul est la fonction `call-with-current-continuation` (ou en abrégé `call/cc`).

L'appel de la fonction `call/cc` est réalisé par

```
(call/cc proc)
```

où `proc` est une procédure à un argument. L'argument de `proc` est appelé *continuation*. Et la continuation est également une procédure à un argument. `proc` est de la forme suivante :

```
(lambda (k) (...))
```

`call/cc` retourne le résultat de l'application de `proc` à la continuation courante. La valeur retournée par `call/cc` est donc la valeur retournée par `proc`.

Exemple :

```
? (call/cc
    (lambda (k)
      (k 5)))
⟹ 5
? (call/cc
    (lambda (k)
```

[7] Il sera plus correct d'utiliser `rec` dans la définition d'une fonction récursive :

```
(define fact
  (lambda (n)
    ((rec f (lambda (x)
              (if (= x 0) 1 (* x (f (1- x))))))
     n)))
```

Car dans cette définition, l'appel récursif ne s'effectue que dans l'environnement local de `fact`.

```
        (+ 1 (k 5))))
⟹ 5
```

Dans le deuxième exemple, la continuation courante est appliquée à la valeur 5 qui est retournée par la procédure. Cet exemple illustre l'utilisation de continuation comme échappement.

Fonction escaper

A partir de `call/cc`, on peut définir une fonction **escaper** permettant de réaliser l'échappement. La procédure **escaper** prend en argument une procédure.

Soit `p` une procédure quelconque prenant une liste d'arguments `args`, et `f` est une procédure quelconque, l'expression

```
(f ((escaper p) args))
```

est évaluée à `(p args)`.

Exemple : L'expression `(+ 3 ((escaper *) 4 5))` est évaluée à 20. Seule la multiplication `(* 4 5)` est exécutée, l'évaluation de `(lambda (v) (+ 3 v))` est abandonnée.

Cette fonction **escaper** peut être facilement définie à partir de `call/cc`. Voici une définition possible :

```
(define escaper
   (lambda (proc)
      (lambda args
        (**escaper-toplevel**
          (apply proc args))))
(define **escaper-toplevel** #f)
(letrec ([receiver
          (lambda (k)
            (set! **escaper-toplevel** k))])
  (call/cc receiver))
```

On peut également utiliser la continuation après l'appel de `call/cc`

```
(let ([x (call/cc
            (lambda (k)
              k))])
  (x (lambda (i) "hello")))
⟹ "Hello"
```

La continuation courante après l'appel de `call/cc` est : lier la valeur de `x`, appliquer `x` à `f` (avec `f = (lambda (i) ...)`). `x` est d'abord lié à la valeur retournée par `(call/cc ...)` qui est la continuation courante, on applique cette continuation courante à la procédure `f`, ce qui a pour effet de lier `x` de nouveau à une nouvelle continuation qui retourne `f`. L'application de `f` à elle-même donne "Hello".

Un autre exemple pour utiliser la continuation, on peut récupérer la continuation au fond d'une pile d'exécution :

```
(define fact
  (lambda (n)
     (if (= n 0)
         (call/cc
           (lambda (k)
              (set! reentrant k)
              (k 1)))
         (* n (fact (1- n))))))
? (fact 4)
⟹ 24
? (reentrant 5)
⟹ 120
? (reentrant 2)
⟹ 48
```

V.4.4 Comparaison des arbres binaires par continuation

On donne un exemple de continuation dans un problème classique de comparaison des arbres binaires en informatique.

Soit à comparer plusieurs arbres binaires. S'ils sont identiques, on donne une réponse, sinon signaler la différence et arrêter la comparaison. Considérons quatre arbres (A, B, C, D), l'algorithme classique de comparaison est le suivant :

- Pour chaque éléments de {A, B, C, D}, aplatir l'arbre binaire en liste simple ;
- Comparer les listes {A', B', C', D'}.

Cet algorithme a besoin d'effectuer deux parcours des +arbres.

On peut résoudre ce problème par l'utilisation de continuation, en effectuant un seul parcours des arbres. Intuitivement on peut procéder de la façon suivante : parcourir l'arbre A, et si on a une feuille, on prend la valeur de de cette feuille, et on envoie cette valeur à l'arbre B pour la comparer à la valeur de la feuille correspondante de B, en sauvegardant la continuation dans A, c'est-à-dire la suite des éléments à comparer. B procède de la même façon que A vis-à-vis de C etc... jusqu'à D. S'il y a une différence entre les feuilles, on arrête, sinon on continue. E peut envoyer un message à A pour que A continue la comparaison du reste.

Chaque arbre possède des connaissances à qui il peut envoyer des messages, et également des variables locales et des méthodes permettant de répondre à des messages.

Nous considérons les arbres suivants :

```
(define a '((1 . 2) . ((3 . 4) . (18 . (6 . (7 . (8 . (9 . 10))))))))
(define b '(1 . ((2 .((3 .(4 . (5 . 6))) . (7 . (8 . 9)))) . 10 )))
(define c '((1 .(2 . 3)) . (4 . (5 . (6 . (7 . (8 . (9 . 10))))))))
(define d '(((1 . 2) . 3) . (4 . (5 .(6 .(7 .(8 .(9 . 10))))))))
(define end 100)
```

On introduit un objet **start** permet de démarrer et d'arrêter. Ceci se passe de la façon suivante :

start	connaît	fringe-a
fringe-a	...	fringe-b
fringe-b	...	fringe-c
fringe-c	...	fringe-d
fringe-d	...	start

Pour la première fois, **start** envoie le message **start** à **fringe-a**, et ainsi de suite jusqu'à **fringe-d**, et on revient à **start**. Ensuite **start** envoie le message **compare** comme précédemment, et ce jusqu'à une erreur ou jusqu'à la fin.

Un arbre quelconque peut être écrit en SCHEME de la manière suivante :

```
(define fringe
 (lambda (next tree)
    (let ([control '#f] ;; sauvegarde de continuation
          [value -1]    ;; valeur envoyée par le message
          [prod 'start]) ;;  le message à envoyer à son prochain
     (letrec
       ([start
         (lambda (v)
           (letrec ([f
              (lambda (x)
                 (cond
                   [(pair? x) (f (car x)) (f (cdr x))]
                   [(= x end-t)
                    ((force next) 'end x)] ;; terminer normalement
                   [else
                    (call/cc (lambda (k)
                      (set! control k)
                      (if (eq? prod 'start) (set! value v))
                      (if (= value 0) (set! value x))
                      (if (= value x)
                          ((force next) prod x))
                          ((force next) 'stop (cons x value))))]))])
            (f (cons tree end-t))))]

       [compare (lambda (v)
                  (set! value v)
                  (set! prod 'compare)
                  (control v))]
       [stop (lambda (val)
                ((force next) 'stop val))]
       [end (lambda (val)
                ((force next) 'end val))])
     (lambda (msg . args)
```

```
        (case msg
          [start (apply start args)]
          [end (apply end args)] [compare (apply compare args)]
          [stop (apply stop args)])))))))
```

Chaque fringe est défini donc comme une fermeture.

Voici celui qui permet de déclencher la comparaison et de la terminer :

```
(define starter
 (lambda (next)
   (let ([l ()]
         [state #f]
         [counter 0])
     (letrec
       ([begin
         (lambda ()
           (call/cc (lambda (k)
             (set! state k);; save the continuation
             (compare 0))))]
        [compare
         (lambda (x)
           (set! counter (+ 1 counter))
           (if (= counter 1)
               (apply (force next) '(start 0))
               (begin (set ! l (cons x l))
                      (apply (force next) '(compare 0)) )))]
        [start
         (lambda (x)
           (set! counter (+ counter 1))
           (set ! l (cons x l))
           (apply (force next) '(compare 0)))]

        [end
         (lambda (x)
           (state (printf "voici le resultat  s" (reverse l))))]
        [stop
         (lambda (x)
           (state (printf "une error  s" (cons (reverse l) x))))])
      (lambda (msg . args)
        (case msg
         [start (apply start args)]
 [begin (apply begin args)]
 [stop (apply stop args)]
 [compare (apply compare args)]
 [end (apply end args)]
```

```
))))))
```

Et voici les quatre arbres et le *starter* :

```
(define fringe-a (fringe (delay fringe-b) a))
(define fringe-b (fringe (delay fringe-c) b))
(define fringe-c (fringe (delay fringe-d) c ))
(define fringe-d (fringe (delay start) d ))
(define start (starter (delay fringe-a)))
```

Remarque : Il est nécessaire d'introduire une évaluation retardée pour pouvoir définir ces objets d'une façon mutuellement récursive.

Pour commencer il suffit de faire `(start 'begin)`.

V.5 Exemple d'utilisation d'extension de syntaxe

Nous allons dans cette section expliquer l'utilisation de la forme spéciale pour l'extension de syntaxe (macro) en SCHEME. On donnera un petit exemple de l'objet illustré au chapitre IV.

V.5.1 Extension de syntaxe

La plupart des versions de SCHEME offre le mécanisme pour la définition de nouvelle forme syntaxique. Nous utilisons **extend-syntaxe** de *ChezScheme*.

La syntaxe de **extend-syntaxe** est la suivante :

```
(extend – syntaxe
 (name key ...)
 (pattern fender expansion) ...))
```

où **name** est le nom de la nouvelle forme à définir. **key** ... sont des mots-clé qu'on peut utiliser (par exemple **else** dans **cond**). **pattern** spécifie la forme de la nouvelle syntaxe. **expansion** est la forme de sortie que l'on veut obtenir. **fender** est rarement utilisé, il sert à préciser des contraintes supplémentaires pour les entrées. On peut utiliser les trois points de suspensions (...) pour décrire une suite semblable d'expressions. Voici l'exemple de **let**

```
(extend – syntaxe (let)
   [(let ([x v] ...)  ;; pattern c'est l'entrée
         e1 e2 ...)
    ((lambda (x ...)  ;; expansion
       e1 e2 ...)
     v ...)])
```

On peut définir les formes syntaxiques récursivement.

V.5.2 Un objet simple

La façon la plus simple de définir un objet en SCHEME est de considérer l'objet comme une fermeture, c'est-à-dire un environnement et une procédure qui permet de traiter les messages. L'objet est de la forme suivante :

```
(let ([id val] ...) ;; les variables locales
  (letrec ([msg-key action] ...) ;; les méthodes
    (lambda (msg . args) ;; la procédure
      (case msg
       [...] ...))))
```

L'exemple de un-point2d est simplement défini par :

```
(define un-point2D
  (let ([x 0] [y 0])
    (letrec ([afficher
              (lambda () (draw x y))]
             [bouger-de
              (lambda (dx, dy)
                (set! x (+ x dx))
                (set! y (+ y dy))
                (un-point2D 'afficher))])
      (lambda (msg)
        (case msg
          [afficher (apply afficher)]
          [bouger-de (apply bouger-de args)]
          [else (error "erreur ")])))))
```

Cette définition a pour effet de lier la variable un-point2D à une procédure avec un environnement où les variables x, y sont initialement liées à la valeur 0.

Il est donc immédiat de construire une nouvelle syntaxe pour la définition des classes.

```
(extend-syntax (define-class)
   [(define-class (name)
       ([id1 val1] ...)
       ([id2 val2] ...))
    (define name
       (lambda ()
          (let* ([id1 val1] ...)
             (letrec ([id2 val2] ...)
                (lambda (msg . args)
                   (case msg
                      [id2 (apply id2 args)] ...
                      [else
                       (error 'name "invalid message  s"
                          (cons msg args))])))))))]
   [(define-class (name)
       ([id2 val2] ...))
    (define-class (name)
       ()
       ([id2 val2] ...))])
```

Dans ce cas, l'envoi de message peut être réalisé par :

```
(extend-syntax
 (send)
 [(send receiver selector arg ...)
  (apply (receiver selector) (list arg ...))])
```

Nous ne discuterons pas les problèmes posés par l'utilisation de l'héritage. Par contre l'utilisation du mécanisme de délégation sera décrit au chapitre VI. Pour les problèmes concernant la programmation par objets et les problèmes liés à l'implémentation des objets en SCHEME, on peut se reporter aux références suivantes : [80, 8].

V.6 Résumé

SCHEME est un langage à la fois simple et puissant, et permet de simuler un grand nombre de styles de programmation (par objet, logique etc...), et les concepts informatiques modernes.

Son extensibilité nous permet facilement de construire un nouveau modèle de programmation. L'utilisation de continuation comme objet de première classe du langage offre une grande souplesse pour implémenter de nouvelles structures de contrôle puissantes telles que le retour-arrière intelligent, co-routines. Tout ceci sera exploité dans la réalisation de notre maquette.

Chapitre VI

Implémentation et application

Retour le mouvement de la Voie
Faiblesse sa coutume
Toutes choses sous le ciel naissent de ce qui est
Ce qui est de ce qui n'est pas

— *Lao-Tzeu*

Ce chapitre décrit la réalisation effective de la maquette informatique d'un outil destiné à l'audit thermique de bâtiment existant. La maquette comprend deux parties distinctes : un module de saisi géométrique rapide appelé *métreur* et un module de calcul de consommation finale de chauffage et de bouclage de bilan. Les deux modules sont interfacés par l'intermédiaire d'un fichier de résultats produit par le premier. Nous décrivons essentiellement la seconde partie de la maquette. Pour la réalisation du métreur, on peut se reporter au [16]. Dans la suite, la *maquette* désignera donc uniquement la deuxième partie.

La réalisation de cette seconde partie de la maquette comprend en effet deux niveaux distincts :

- implémentation en SCHEME d'un modèle de programmation simple, inspiré du modèle d'acteurs et du modèle d'objets. La section VI.1 décrit cette implémentation ;
- utilisation du modèle de programmation ci-dessus pour réaliser la maquette d'un outil d'audit. La programmation est décrite dans la section VI.2.

Nous allons donc d'abord décrire l'implémentation d'un modèle de programmation et ensuite la programmation des objets de l'audit dans ce modèle en prenant en compte les aspects incertains et imprécis des données traitées.

VI.1 Implémentation d'un modèle de programmation

Un acteur est un objet *actif* qui communique avec l'extérieur par envoi et réception de messages. L'aspect *actif* d'un acteur est essentiel pour le différencier des modèles d'objets au sens des langages à objets (cf IV.3).

Rappelons que l'on peut définir un acteur par les éléments suivants :

- son *identificateur* ou son *adresse* permettant de le distinguer des autres acteurs du système ;
- ses *acquaintances (accointances)* ou connaissances, ce sont les autres acteurs du système que l'acteur connaît. Les acquaintances déterminent l'état interne d'un acteur ;
- son *script* qui détermine le comportement de l'acteur vis-à-vis des messages reçus (analogue des *méthodes* des langages à objets) ;
- son *mandataire* (proxy) à qui il délégue tous les messages incompris.

Un des intérêts des acteurs réside dans l'utilisation de la *continuation*, qui se traduit dans le modèle d'acteurs par le fait qu'un acteur peut envoyer un message à un deuxième acteur, sans attendre le retour du résultat que peut recevoir un troisième acteur spécifié dans le message. Dans un système d'audit, on peut utiliser ce mécanisme pour traiter le problème de choix de méthodes (cf. partie suivante).

Notre implémentation est très simplifiée, et ne possède pas de mécanismes de traitement de messages puissants comme dans les ACT1 [69], ACT2 [97]... par filtrage (pattern-matching). Elle essaie de combiner l'aspect *objet* des langages à objets, et l'aspect *contrôle* des acteurs.

L'implémentation consiste essentiellement en des extensions de syntaxe (macros) en SCHEME, un dialecte à liaison lexicale de LISP, décrit au chapitre V. Le choix du langage hôte pour implémenter le modèle se justifie par les caractéristiques particulièrement intéressantes de SCHEME, notamment la *continuation* comme objet de première classe [85, 45, 8, 26].

VI.1.1 Caractéristiques du modèle

Le modèle implémenté permet la définition et la création des acteurs. Un acteur est donc défini par son état interne, son script et son mandataire. Un acteur créé à partir d'un acteur existant partage les scripts de ce dernier. Le mécanisme de partage est réalisé par la *délégation*. Il existe deux façons d'envoyer des messages. Les accointances d'un acteur peuvent être soit des acteurs définis dans le système, soit des valeurs de SCHEME. Les scripts sont des lambda-expressions de SCHEME. Ces choix induisent bien entendu une non-uniformité dans le système, mais augmente considérablement la lisibilité du code de programme et l'efficacité. La suite décrit les principaux éléments du système implémenté.

VI.1.2 Définition des acteurs

Les objets que l'on veut définir comportent donc un nom, une liste d'acquaintances (les variables locales), un script (une liste de méthodes définissant ainsi le comportement de l'acteur à la réception de message), et une liste de mandataires (dans notre utilisation courante, un seul mandataire suffit).

La définition d'un acteur est réalisée par la forme syntaxique **`def-act`** :

```
(def-act nom
  (accointances [id-a val-a] ...)
  (scripts [id-s val-s] ...)
  (proxy p ...))
```

dans cette syntaxe les identificateurs `accointances`, `scripts`, `proxy` sont des mot-clés.

La définition en SCHEME de cette forme syntaxique est la suivante :

```
(extend-syntax
 (def-act accointances scripts proxy)
 [(def-act name
    (accointances (var val) ...)
    (scripts (scr vcr) ...)
    (proxy pro))
  (define name
    (letrec
        ([self
          (letrec
             ([list-accointances
               (make-env (list (cons (quote var)
                                     (delay val)) ...))]
              [list-scripts
               (make-env (list (cons (quote scr)
                                     (delay vcr)) ...))]
              [proxy (list pro)])
            (lambda (msg);; les messages sont :
              (case msg
                [accointances
                 (lambda (instance . args) list-accointances)]
                [set-var!
                 (lambda (instance . args)
                   (force
                    (list-accointances 'set-var! (car args)
                                       (delay (cadr args)))))]
                [else
                 (let ([v-acc (list-accointances 'get-var msg)])
                   (if v-acc ;;msg correspond à une accointance
                       (lambda (instance . args)
                         (force (cdr v-acc)))
                       (let ([v-scr
                              (list-scripts 'get-var msg)])
                         (if v-scr;;msg correspond a un script
                             (if (eq? msg 'reply)
                                 (lambda (instance . args)
                                   (apply (force (cdr v-scr)) args))
```

```
                    (lambda (instance . args)
                      (call/cc
                       (lambda (k)
                         (send self
                               'set-var! 'controle k)
                         (apply (force (cdr v-scr))
                                (cons instance
                                      args))))))
                (lambda (instance . args)
                  (if (null? (car proxy))
                      (printf " s introuvable  %" msg)
                      (delegue-f
                       (lambda (p)
                         (send-as p msg instance args))
                       proxy))))))]))))])
      self))])
```

Un acteur ainsi défini possède les propriétés suivantes :

- l'accès ou la modification des accointances se fait par un envoi de message ;
- de nouveaux scripts peuvent être ajoutés ;
- un message inconnu est automatiquement délégué à son mandataire *proxy*, s'il n'y a pas de proxy, une erreur est signalée. Dans ce dernier cas, on peut bien entendu invoquer la continuation d'échec.

L'environnement contenant les valeurs des accointances et des scripts des acteurs est représenté par une a-list (association list). Ceci est réalisé par la fonction **make-env** qui se trouve en annexe A.

L'exécution de la macro

(def – act **name** ...)

a pour effet de définir (par le **define** ordinaire de SCHEME) **name** avec comme valeur la valeur de **self** dans le (**letrec** ...), qui est une fermeture formée de l'environnement avec une liste d'accointances (des a-listes) et une liste de méthodes et une liste de mandataires, et d'une procédure prenant en argument un message (**msg**). Le **delay** permet l'évaluation différée de son argument, donc la définition mutuelle des acteurs. La fonction **delegate** permet de déléguer un message inconnu aux mandataires de l'acteur. Cette fonction est en effet un envoi de message comme on le verra dans la suite. La forme **send** et la procédure **send-as** seront définies plus tard (cf. partie VI.1.4).
(**case msg** ...) est une forme conditionnelle qui signifie que

- si le message est **accointances**, on donne la liste des accointances de l'acteur ;
- si le message est **set-var !** avec deux arguments **id** et **val**, on modifie la valeur de l'accointance **id** ;

- dans les autres cas
  - si le message correspond à une accointance, on rend la valeur de cette accointance ;
  - si message correspond à un script, on invoque la méthode correspondante ;
  - dans les autres, s'il y a des proxy, on délégue le message successivement à ses proxy, sinon on signale une erreur.

L'utilité de la fonction `call/cc` sera expliquée dans la suite.

Par cette forme, on définit en effet un prototype des objets qui engendre implicitement une classe d'objets. L'intérêt de l'utilisation de prototype est que l'on peut créer de nouveaux objets à partir de n'importe quel objet existant, ce qui nous permet comme on le verra d'utiliser les valeurs de l'objet existant comme valeur par défaut.

VI.1.3 Création à partir d'un acteur existant

A partir des acteurs existants, on peut créer de nouveaux acteurs par la forme syntaxique `new-actor` :

```
(new – actor nom-actor
 (accointances [id-a val-a] ...))
```

Les accointances sont optionnelles. L'acteur créé a pour accointances la liste des accointances données (éventuellement vide), pour scripts une liste vide et pour proxy l'acteur de référence `nom-actor`.

```
(extend-syntax
 (new-actor)
 [(new-actor act-ref
             (accointances (var val) ...))
  (letrec ([self
            (letrec
                ([list-accointances
                  (make-env
                   (list (cons (quote var) (delay val)) ...))]
                 [list-scripts
                  (make-env (list '() '()))]
                 [proxy (list act-ref)])
              (lambda (msg)
                (case msg
                  ;; meme que dans le def-act ...
                  )))])
    self)])
```

Remarque : On distingue souvent deux sortes de création d'acteurs dans les systèmes d'*acteurs purs* : copie conforme de l'acteur existant et copie par extension. Le mécanisme

que nous utilisons est en effet plus proche de la création par l'extension, car les accointances et le mandataires ne sont pas copiés au moment de la création.

L'acteur ainsi créé par cette forme spéciale a pour accointances (respectivement scripts) les accointances (respectivement scripts) spécifiées dans les arguments qui peuvent éventuellement être vides, et pour proxy l'acteur à partir duquel ce nouvel acteur est créé. Donc si les accointances et le script de cet acteurs sont vides, son comportement et son état sont identiques à l'acteur de référence.

Tous les acteurs possèdent une méthode permettant de modifier ses accointances. L'accès à l'accointance est très simple : si l'acteur en question possède cette accointances, il renvoie la valeur, sinon il délégue à son mandataire, c'est donc la valeur de l'accointance de son mandataire (direct ou indirect) qui est retournée. Par contre pour la modification, si l'accointance existe, on modifie directement sa valeur, sinon on ajoute à la liste d'accointances cette nouvelle accointance.

VI.1.4 Envois de messages

Les envois de messages sont réalisés par une extension de syntaxe **send**. On peut cependant distinguer deux sortes d'envois de messages : **send** et **send-as**. Voici leur syntaxe avec **sk** pour la *continuation de succès* et **fk** pour la *continuation d'échec* (failure continuation) :

```
(send acteur msg (customer sk) argslist fk)
(send-as acteur msg self argslist fk)
```

Customer est optionnel et permet de spécifier l'acteur à qui le résultat est destiné. **send-as** ressemble à **send-super** de certains modèles objets qui spécifie l'objet auquel on se réfère pour capturer les variables locales et lequel peut être différent de l'acteur receveur du message. Et **send** peut être défini par **send-as** :

```
(extend-syntax
 (send customer)
 [(send target msg (customer sk) args ...)
  (send sk 'reply (send target msg args ...))]
 [(send target msg args ...)
  (send-as target msg target (list args ...))])
```

et **send-as** est défini par :

```
(define send-as
   (lambda (component selector composite largs)
     (apply (component selector) (cons composite largs))))
```

Dans cette fonction, **component** est le receveur du message, et **composite** permet de capturer les variables locales dans le traitement de message d'une façon correcte. **send-as** retourne la valeur de l'application de la procédure locale **selector** définie dans **component**.

Avec ce mécanisme d'envoi de message, il faut, dans les arguments de chaque méthodes (une lambda-expression), ajouter l'objet auquel le message fait explicitement référence pour les variables locales. Toutes les méthodes auront donc la forme suivante :

```
(lambda (self arg ...)  ;; self designe l'objet reference des
  corps)                ;; variables locales
```

Remarque : `sk` et `fk` ne sont pas les mêmes types d'objets : le premier est un acteur défini par `def-act`, et le second est une continuation (donc une procédure) de SCHEME. On peut effectivement faire en sorte que `fk` soit du même type que `sk`, mais cela complique l'implémentation et n'apporte rien de fondamental. Nous avons également dans la syntaxe de `send` fait le choix de mettre `fk` comme un argument ordinaire du message, ceci facilite l'implémentation, mais oblige l'utilisateur à écrire dans la définition de méthode `fk` dans les arguments, c'est une contrainte supplémentaire.

VI.1.5 Continuation

Il existe des cas où on a plusieurs façons de calculer un paramètre selon la disponibilité de certaines informations. Le choix de méthodes et le retour éventuel en cas d'échec peuvent être programmé en utilisant la fonction `call/cc`. Mais pour que l'utilisateur du système n'ait pas à manipuler cette fonction directement, on définit un alternateur `alt/lt` pour réaliser l'application successive d'une liste de méthodes jusqu'à obtenir un résultat.

Dans la définition précédente de l'envoi de message, on peut distinguer trois cas qui nécessitent des traitements différents :

1. message simple. Le message spécifie le destinataire du message, le sélecteur et les arguments éventuels.

   ```
   (send target msg arg1 ... argn)
   ```

   dans ce cas le résultat est retourné à celui qui envoie le message (l'expéditeur). Avec la continuation courante de l'envoi de message, l'expéditeur continue. Ce cas ne pose donc pas de problème ;

2. message spécifiant le traitement d'échec.

   ```
   (send target msg  arg1 ... argn fk)
   ```

   Le résultat est toujours retourné à l'expéditeur, mais au cas d'echec, le contrôle est transféré à `fk` ;

3. message spécifiant la continuation de succès et d'echec.

   ```
   (send target msg (customer sk) fk arg1 ... argn fk)
   ```

   Au cas de succès, le contrôle est transféré à `sk`, c'est donc le `customer` qui est le destinataire du résultat. Au cas d'echec, `fk` sera invoquée.

Pour illustrer ces différents cas et les problèmes posées au niveau de l'implémentation, nous allons prendre l'exemple suivant :

Exemple :

Soient A, B, C, D les acteurs définis par le code suivant avec la syntaxe de **send** celle donnée précédemment. Et on suppose que chaque acteur délégue le message **read** à un acteur READ qui n'est pas présenté ici et dont la définition est immédiate. Cette méthode a deux arguments : identificateur de variable (id) et la continuation d'echec, et a pour effet d'affecter la valeur lue (sur clavier) à la variable locale id. Le mandataire de chacun de ces acteurs (qui est read) est omis dans le code.

Le but de cet exemple est de montrer le fonctionnement pour alterner le calcul entre B et C, c'est-à-dire en cas d'echec, pour aller de B à C et éventuellement de retourner de C en B.

Avec les syntaxes définies précédemment, voici comment on peut définir les quatre acteurs de la façon suivante :

```
(def-act A
   (accointances (v1 1)) ;; v1 a pour valeur val1
   (scripts
     (m (lambda (self fk0)  ;; methode m
          (let ((x (send self 'read v1 fk0)))  ;; fk = fk0
            (send b 'm (customer D)  x 3 C)))))) ;; sk= D   fk = C
```

A possède une accointance v1 qui a pour valeur initiale 1. La méthode m demande la valeur de v1 par la méthode read. Si la valeur lue est est différente de echec, il envoie le message m à B avec les arguments x (valeur lue) et 3 ainsi que la continuation de succès (sk = C) et d'echec (fk = D). S'il y a une erreur (echec, on peut définir echec si la valeur n'est pas une valeur numérique, ceci n'est pas important), alors la continuation d'echec initiale (fk0) est invoquée.

```
(def-act B
  (accointances (v 2))
  (scripts
    (m (lambda (self sk x y fk)
         (let ((z (send self 'read v fk)))
           (+ x y z))))))

(def-act C
  (accointances (v 3))
  (scripts
    (m (lambda (self sk x y fk)
         (let ((z (send self 'read v fk)))
           (* x y z))))))

(def-act D
  (accointances (v 4))
  (scripts
```

```
(reply
  (lambda (self value) ;; imprime simplement la valeur
    (printf " s " value))) ))
```

L'acteur A envoie à B le message **m** en indiquant sk = D, et fk = C. Si B répond par succès, le résultat est envoyé à D qui imprime ce résultat (simplement). Sinon, un message est envoyé à C qui peut être de l'une des formes suivantes selon si on veut réaliser ou non un retour éventuel en B en cas d'echec en C :

```
(send C 'm (customer D) x y E) ;; E permet de signaler l'erreur
ou (send C 'm (customer D) x y B) ;; plutôt l'état courant de B
```

Dans le second message, en cas d'echec, il faut reprendre le calcul laissé en B. La continuation d'echec n'est pas exactement B, mais la continuation courante en B au moment d'envoyer le message à C. Il pose donc un problème de bien récupérer l'état exacte laissé en B.

Donc pour l'alternateur, il faut sauvegarder la continuation courante de B, ce qui n'est pas encore réalisé par la définition de **send** précédente.

A partir des éléments précédents, nous définissons une nouvelle syntaxe pour pouvoir alterner entre plusieurs acteurs de même type. Dans cette définition, on simplifie la continuation d'echec en une continuation simple au lieu d'un acteur défini par **def-act**.

La syntaxe est la suivante :

```
(alt/lt (act1 act2 ...) msg instance fk args ...)
```

ou

```
(alt/lt liste-acteurs msg instance fk args ...)
```

Il réalise les actions suivantes : il envoie d'abord le message **msg** à **act1** (ou le premier de **liste-acteurs**), si **act1** réussit à traiter le message, le résultat est envoyé à l'acteur **sk**, sinon on sauvegarde l'état de calcul de **act1**, et on continue par **act2** jusqu'à la dernière. Si tous ces acteurs échouent, on revient à **act1**, on continue en prenant les valeurs sauvegardées.

Sa définition est la suivante :

```
(extend-syntax
 (alt/lt)
 [(alt/lt list-act msg instance fk args ...)
  (call/cc
   (lambda (sk)
     (let ([first-act (car list-act)]
           [rest-act (cdr list-act)])
       (printf " ... %")
       (sk
        (send-as first-act msg instance
                 (list args ...
                       (lambda (fk1)
```

```
                          (alt rest-act msg instance sk fk1 args ...))
                        )))) ))]
  [(alt/lt (act1 ...) msg instance fk args ...)
   (let ([list-act (list act1 ...)])
     (alt/lt list-act msg instance fk args ...))])
```

avec `alt` défini de la façon suivante :

```
(extend-syntax
 (alt)
 [(alt actor-list msg instance sk fk args ...)
  (do
      ([l actor-list (cdr l)])
      ([null? l]
       (printf " .......   %")
       (fk 'any))
    (call/cc
     (lambda (fk)
       (printf " ....  "))
     (sk (send-as (car l) msg instance (list args ... fk))))))])
```

Dans la définition de `def-act`, nous avons en effet sauvegarder la continuation courante de calcul au moment du traitement de message en affectant l'accointance `control` la valeur de cette continuation. La procédure suivante prend en argument une continuation et une valeur et retourne une procédure dont l'argument sera ignoré et qui invoquera la continuation passée avec la valeur en question. Cette procédure permet donc au cas de retour de retrouver la continuation et l'ancienne valeur qui sera utilisée par défaut.

```
(define return-state
  (lambda (fk old-value)
    (lambda (ignore)
      (printf " %")
      (printf "Invocation de la continuation pour le retour ! ! %")
      (printf " %")
      (fk old-value))))
```

L'utilisation de ces macros reste encore un peu complexe, car elles nécessitent dans la définition des méthodes de mentionner explicitement ce qu'il faut prendre comme valeur par défaut et la continuation à invoquer, mais elles montrent la potentialité d'une telle réalisation qui permettrait à un utilisateur non familiarisé avec les concepts utilisés d'employer ses structures de contrôle d'une façon aisée.

Il faut également noter que la sauvegarde de la continuation est coûteuse en mémoire (donc également en temps). Il nous paraît intéressant de pouvoir distinguer les acteurs dont cette sauvegarde est nécessaire de ceux qui n'ont pas besoin de manipuler les continuations.

VI.1.6 Quelques méthodes communes à tous les objets

On définit un objet appelé `obj` comportant un certain nombre de scripts qui seront utilisés par d'autres objets par le mécanisme de délégation.

Actuellement, dans cet objet de racine, on a défini plusieurs scripts, dont certains sont généraux et d'autres sont liés aux objets de l'audit de bâtiment, il serait bon de définir un autre objet qui possède les scripts communs aux objets de l'audit.

L'objet `obj` possède actuellement très peu de méthodes qui sont :

- `make-list` permet de construire une liste d'acteurs comme accointance d'un acteur ;
- `find-accointances` ; cette méthode permet de lister toutes les accointances d'un objet ainsi que les accointances de ses objets composants.
- `create` permet de créer un objet comme accointance ;
- ...

VI.2 Programmation des objets de l'audit

Cette partie décrit de façon détaillée le problème de la programmation des objets de l'audit dans le modèle d'acteurs défini dans la partie précédente.

La programmation des objets consiste simplement à traduire les objets décrits dans le chapitre II par les macros définies ci-dessus.

VI.2.1 Choix de l'implémentation des variables

Pour faciliter l'écriture de programme, pour les raisons telles que la lisibilité du programme, et aussi pour utiliser pleinement la puissance offerte par SCHEME, on définit les variables de l'audit par une nouvelle structure, qui comporte un certain nombre de champs. Elle est de la forme suivante :

```
VAR ::= (Type Valeur Domaine Coefficient-Assertion Historique)
```

dont **type** spécifie si la variable est certaine ou incertaine, et **valeur** peut être numérique (dans ce cas, elle peut comporter une imprécision) ou non, **historique** est une valeur symbolique telle que **calculé, saisi...** qui permet de savoir comment est obtenue la valeur de la variable. **domaine** définit le domaine de variation de la variable, il peut être sous forme d'intervalle (exemple : $[0, 5]$), ou une valeur symbolique (**entier, symbole...**) ou également sous forme de valeurs discrètes (exemple : {isolé, surisolé}). **coefficient-assertion** prend la valeur dans l'intervalle $[0, 1]$ (on utilisera la valeur 2 pour désigner une assertion certaine et dont le type n'est pas donné quand la variable est définie par **make-var**).

Cette structure est définie par une extension de syntaxe **make-var**. On aurait pu utiliser la forme spéciale **define-structure** de SCHEME. Il est également possible de définir ces variables comme les acteurs par **def-act**, mais il nous semble plus simple et plus clair de les définir par une telle structure. A partir de cette définition, on peut maintenant définir

les opérations manipulant ces objets, telles que +, -, *, /..., qui sont appelées **audit+**, audit-, audit*, audit/...L'écriture de ces fonctions ne présente pas de difficulté, on trouve le code de ces fonctions dans l'annexe, d'autres opérations et prédicats tels que eq ?, > ?, < ? etc..., peuvent aussi être définis de cette façon.

La définition de syntaxe de **make-var** est la suivante :

```
(extend-syntax
 (make-var type)
 [(make-var (type typ) val dom unit his coa)
  (list typ val dom unit his coa)]
 [(make-var (type typ) val dom unit his)
  (list typ val dom unit his 1)]
 [(make-var val dom unit his coa)
  (let ([typ
         (cond [(= coa 2) 'certaine]
               [else 'incertaine])])
    (make-var
     (type typ) val dom unit his coa))]
 [(make-var val dom unit his)
  (make-var val dom unit his 2)])
```

Les définitions des sélecteurs tels que **var-type** et **var-dom** etc...se trouvent en annexe A.2.

VI.2.2 Les objets de l'audit

Les objets physiques sont décrits dans le chapitre II. Cette partie montre la programmation de ces objets dans le système défini précédemment. En effet, il existe un seul objet au début du calcul **bâtiment**. Cet objet contient un certain nombre d'autres objets qui sont les **zones-thermiques**, **climat**, **système-production**. Notons qu'on pourrait définir l'objet **climat** comme objet à part, et l'objet **bâtiment** se trouve dans un environnement climatique.

On considère que le bâtiment est multizone, avec un système de production identique pour tout le bâtiment.

La programmation des objets consiste en effet simplement à transcrire les objets physiques décrits dans le chapitre II par les macros définies précédemment. Cette transcription ne pose aucun problème particulier quand il s'agit d'objets simples, c'est-à-dire des objets dont les méthodes ne nécessitent pas d'utiliser plusieurs modèles de calcul. La suite donne donc la définition de l'objet **bâtiment**. Dans le cas de l'utilisation possible de plusieurs modèles, on donne un exemple de la programmation avec la forme spéciale **alt/lt** définie précédemment.

Les méthodes communes à tous les objets d'audit

Tous les objets de l'audit possèdent un certain nombre de méthodes communes. Pour les définir, on introduit un objet appelé **obj-audit** à qui tous les objets déléguent les méthodes

mentionnées. Les méthodes suivantes sont définies pour l'instant :

- `read` permet la lecture des variables de l'audit ;
- recherche de l'identité de l'objet ;
- `find-acc` définit les méthodes permettant de rechercher les paramètres avec un coefficient d'assertion faible ;

Définition de Bâtiment

La définition de l'objet `bâtiment` est la suivante (certaines parties du code ne sont pas présentées en raison de la longueur, voir annexe) :

```
(def-act batiment
  (accointances
   [nom 'batiment]
   [type-batiment
    (make-var 'collectif '(collectif individuel
                                    tertiaire) () 'defaut)]
   [lieu (make-var '26 'entier () 'defaut)]
   [site (make-var 'plat-degage
                   '(ile cotier plat-degage) () 'defaut)]
   [niveau-audit (make-var 'intermediaire
                           '(rapide intermediaire
                                    approfondi) () 'defaut)]
   [climat (new-actor climat)]
   [equipement (new-actor equipement)]
   [nombre-zones (make-var 1 'entier () 'defaut)]
   [liste-zones #f]
   [consommation-calculee (make-var 0 'reel 'kWh 'defaut)])
  (scripts
   [consommation
    (lambda (self fk)
      (let*
          ([equip (send
                   self 'create 'equipement fk)]
           [clim (send self 'create 'climat fk)]
           [n (var-val (send self 'read 'nombre-zones fk))]
           [l0 (send self 'make-list n zone 'liste-zones 'zone)]
           [result (audit/
                    (audit* 24
                            (audit/somme-de
                             10
                             (lambda (z)
```

```
                                (send z 'consommation self fk))))
                        (send equip 'rendement self fk))])
        (var-set! result 'unit 'kWh)
        (send self 'set-var! 'consommation-calculee result fk)))]
    [consommation-facturee
     (lambda (self fk) ......)]

    [comparaison
     (lambda (self fk) ;; compare les deux consommations et
       ;; donne une liste de parametres avec un ca faible
       ]
     [debut
      (lambda (self fk)
        (printf " % Donnez le nom du batiment: ===> ")
        (let ([nom-projet (read)])
          (printf "Donnees cree par le metreur? %")
          (printf " O(ui)/N(on)  % ===> ")
          (let ([rep (read)]
                [fich-metre (string+symbol nom-projet ".met")]
                [fich-saisi (string+symbol nom-projet ".acc")]
                [fich-resul (string+symbol nom-projet ".res")])
            (if (or (eq? rep 'O) (eq? rep 'oui))
                (begin ;;affecter les variables nom, fichier-metreur
                  (call-with-input-file fich-metre
                    (lambda (p)
                      (lire-bat self nom-projet fk p)))))
            (saisie-generale self fk)
            (send self 'comparaison fk))))])
    (proxy obj))
```

L'objet **bâtiment** possède un certain nombre d'accointances définies par **makevar** et quatre objets définis par **def-act**. L'accointance **consommation-calculee** permet de stocker le résultat du calcul[1]. Les scripts de l'objet bâtiment sont principalement **comparaison** et **consommation-théorique**. Le premier demande la consommation théorique et la consommation facturée, et les compare, ensuite il signale l'écart et donne une liste des paramètres douteux ; le second demande à ses zones thermiques les déperditions, à l'installation le rendement. Le script **debut** permet d'initialiser certains paramètres et de réaliser l'interface avec le métreur automatique.

Cette définition crée en effet un bâtiment type. Quand on lance le calcul de l'audit d'un bâtiment, on crée un nouveau bâtiment à partir de ce prototype, et les valeurs des variables données dans la définition du bâtiment type seront proposées comme des valeurs

[1] Un certain nombre de résultats intermédiaires ne sont pas stockés dans cette maquette. Il sera nécessaire de le faire dans la version future, car on est amené à donner les résultats intermédiaires, e.g., les consommations énergétiques par poste.

par défaut. L'utilisation du prototype a donc pour l'avantage d'utiliser facilement les valeurs par défaut. Ceci est largement exploité dans notre maquette, par exemple pour une liste des zones thermiques, on crée la première zone, les autres zones ont pour valeurs par défaut des variables celles de la première.

Un exemple d'utilisation de `alt/lt`

On a défini précédemment une forme spéciale **alt/lt**, voici un exemple d'utilisation :

L'objet **paroi-opaque** possède une méthode pour le calcul du coefficient des déperditions surfaciques K.

```
(def-act paroi-opaque
  (accointances ......)
  (scripts ......
    [deperditions
     (lambda (self bat zone fk)
       (let* (.... )
        (letrec ([K
                  (lambda ()
                    (let ([comp
                           (send self 'create 'composition fk)])
                      (let
                          ([res
                            (send comp 'K bat zone self fk)])
                        (send self 'set-var! 'k-calcule res fk))))]
                 corps de la fonction)))])
  (proxy paroi))
```

Le **self** désigne le receveur du message. Dans le corps de calcul des déperditions, on définit (par **letrec**) une fonction locale sans argument **K**. Cette fonction **K** envoie un message **create** à **self** (la paroi en question) pour créer un acteur **composition** comme accointance de **self**. Ensuite, un message est envoyé à cet acteur **composition** pour connaître la valeur de K. Le résultat retourné par **composition** est stocké dans l'accointance **k-calculé** par le message **set-var !**.

L'acteur **composition** peut être défini par :

```
(def-act composition
  (accointances .....
   [K-A (new-actor K-ACT)]
   [k-calcule '()])
  (scripts
   [K
    (lambda (self bat zone paroi fk)
      (let ([K-pov (send self 'K-A fk)]
            [kc (send self 'k-calcule fk)])
```

```
        (if (null? kc)
            (begin (send self 'set-var! 'K-A K-pov fk)
                   (let ([res
                          (send-as K-pov 'K self
                                   (list bat zone paroi fk))])
                     (send self 'set-var! 'k-calcule res fk)))
            kc))] )
    (proxy obj))
```

Dans cette définition, composition possède dans sa liste des accointances un acteur K-ACT à qui un message est envoyé pour le calcul du coefficient K. Notons que l'on ne fait pas de calcul si la valeur de K a déjà été calculée.
La définition de l'acteur K-act est :

```
(def-act k-act
  (accointances)
  (scripts
   [k
    (lambda (composition bat zone paroi fk)
      (printf " ... %" )
      (let ([niveau-audit
             (send bat 'read 'niveau-audit fk)])
        (let ([ch (if (equal? (var-val niveau-audit) 'rapide)
                      (list k1-act k2-act)
                      (list k2-act k1-act))])
          (alt/lt ch 'k composition fk bat zone paroi))))]
   [reply
    (lambda (x)
      (printf "Retour du resultat a l acteur K  s  %" x)
      ((send self 'controle) x))])
  (proxy obj))
```

L'acteur K, en fonction du niveau d'audit, exécute la macro alt/lt, avec K1-act ou K2-act qui définissent les deux méthodes de calcul du coefficient K. Les définitions de ces deux acteurs sont :

```
(def-act k1-act;; la premiere methode
  (scripts
   [k
    (lambda (composition bat zone paroi fk0)
      (printf " methode 1. (source COSTIC) simplifiee %")
      (let* ([fk1 (send self 'controle)] ;continuation courante
             [nom (send composition 'nom fk0)]
             [isol (send composition 'read 'isolation fk0)]
             [type-mur (send composition 'read 'type-mur fk0)]
             [ep (send composition 'read 'epaisseur fk0)]
```

```
                [ep-isol
                 (if (eq? isol 'sur-isolee)
                     (send composition 'read 'ep-isol fk0)
                     ())])
          (let ([res
                 (k-costic isol ep type-mur ep-isol)])
            (if (echec? res)
                (fk0 (return-state fk1 res))
                res))))])
    (proxy obj))
  (def-act k2-act;; la deuxieme methode
    (scripts
     [k
      (lambda (composition bat zone paroi fk0)
        (printf " % methode 2. DTU  %")
        (let* ([fk1 (send self 'controle)]
               [nom (send composition 'nom fk0)]
               [n (var-val
                    (send composition 'read 'nombre-couches fk0))]
               [l1 (send composition
                         'make-list n couche 'liste-couches 'couche)])
          (let ([res ;;;
                 ([(k-cstb l1 'couche bat zone paroi fk0)])
                 (audit/somme-de
                  l1
                  (lambda (ci)
                    (audit/
                     (send ci 'epaisseur fk1)
                     (send ci 'lambd bat zone paroi fk1) )))])
            res)))]);; meme que k1-act
    (proxy obj))
```

Les deux fonctions K-CSTB et K-COSTIC sont définies ailleurs. Il serait très intéressant de pouvoir séparer la définition de ces fonctions et la définition des acteurs qui utilisent ces fonctions, et ce dans le soucis de pouvoir interfacer facilement avec les logiciels existants de calcul en thermique ou réutiliser les modules écrits dans d'autres langages. Il paraît assez difficile de séparer ces fonctions de la définition des acteurs.

VI.2.3 Interface avec le métreur

Une maquette du module spécialisé de saisi géométrique (métreur automatique) a été réalisée en Turbo-Pascal [16]. L'interface entre la partie métreur et la partie décrite précédemment est réalisée par un fichier de résultats du métreur.

La lecture du fichier est réalisée par une fonction **lire-bat** qui crée tous les objets

possibles à partir des données transmises. Les codes de fonctions de lecture se trouvent en annexe.

VI.2.4 Intégration du traitement de données partielles

Les incertitudes

Comme on a vu dans la définition des variables d'audit, toute variable possède un coefficient d'assertion (CA). Pour les variables de type certain, le CA est toujours égal à 1. L'information concernant l'incertitude n'est pas propagée au cours du calcul, elle est simplement mémorisée pour être utilisée lors du bouclage de bilan.

La maquette ne fait pas l'usage de cette information pour déterminer le choix des méthodes. Il nous semble tout à fait possible d'intégrer cette possibilité dans la maquette, en élargissant le contexte du choix.

Les imprécisions

Le calcul des imprécisions est effectué automatiquement par les opérations généralisées (`audit+`, `audit*` `etc...`), donc les imprécisions sont propagées en cours de calcul. Ceci ne pose aucun problème de réalisation pratique.

Rappelons très brièvement les hypothèses et la méthode de calcul.

Les variables considérées sont des variables aléatoires gaussiennes avec une loi de distribution normale. En prenant l'intervalle de confiance à 95% ($\sigma_x = \frac{\delta_x}{2}$), on a pour une expression de C du type

$$C = \frac{24GVDj}{\rho}$$

$$(\frac{\sigma_C}{C})^2 = (\frac{\sigma_G}{G})^2 + (\frac{\sigma_V}{V})^2 + (\frac{\sigma_{Dj}}{Dj})^2 + (\frac{\sigma_\rho}{\rho})^2$$

avec σ écart-type.

C'est cette formule qui est utilisée dans le calcul actuel des imprécisions. Notons également que si on décide de changer les méthodes de calcul, il suffit de reprogrammer les opérations généralisées.

Les informations lacunaires

Le traitement des informations lacunaires est difficile. Dans le cas courant, l'absence d'une information peut bloquer la poursuite du calcul. Il est toujours possible de prendre une valeur par défaut, dans ce cas il faut alors savoir si cette valeur par défaut n'est pas trop loin de la vraie valeur. Dans le cas où il existe plusieurs façons de calculer un paramètre, le manque d'une information peut être moins grave, car on peut toujours essayer d'autres méthodes (qui donnent certes des résultats moins précis par exemple, mais qui permettent de ne pas bloquer le calcul). Pour traiter d'une façon satisfaisante ce problème, une bonne connaissance de la pratique des experts est très souhaitable.

VI.2.5 Fonctionnement de la maquette, exemples

Cette partie de la maquette fonctionne d'une façon interactive. Le programme est dirigé par *besoin* ou *but*. Pour lancer le programme, il suffit de créer un nouveau bâtiment et de lui envoyer un message **debut**. Quelques questions générales sont posées. Si le bâtiment en question avait déjà été saisi par le métreur, le programme lit le fichier résultat créé par le métreur et crée ses **zones-thermiques**. Ensuite il lance la méthode du calcul de la consommation. Chaque fois que le programme a besoin de connaître une donnée, il regarde si cette donnée existe déjà, si oui, il prend la valeur et continue le calcul, dans le cas contraire, il la demande à l'utilisateur.

Un exemple est donné à l'annexe D.

Les résultats donnés par la maquette n'ont qu'une valeur indicative, car certains calculs ne sont pas effectués.

VI.3 Quelques remarques

Le modèle d'acteurs offre des caractéristiques intéressantes en ce qui concerne le traitement d'informations partielles. L'implémentation décrite ici est très simpliste par rapport aux modèles généraux, mais montre déjà que l'on peut d'une façon simple traiter les problèmes liés à l'audit thermique de bâtiment où il est nécessaire de considérer la disponibilité des informations et les différents niveaux de précision de données et de résultats.

La maquette actuelle comporte de nombreuses lacunes tant au point de vue de l'implémentation des acteurs qu'au point de vue des objets et des méthodes programmés.

VI.3.1 Sur le modèle implémenté

Le modèle que l'on a implémenté est un modèle simplifié d'acteurs, et il essaie de combiner les caractéristiques des acteurs et des objets pour permettre de programmer d'une façon simple les objets du bâtiment. Les principales limitations sont les suivantes :

- l'envoi de message par continuation n'est pas implémenté d'une façon générale, pour l'instant on stocke l'état courant de calcul de chaque acteur, ce qui augmente sensiblement les occupations de mémoire. La solution est sans doute de distinguer deux sortes d'objets, les premiers sont des objets ordinaires qui ne provoquent pas de retour, ni ne reçoivent les résultats des messages des autres ; les seconds sont ceux qui provoquent des retours ;
- l'utilisation de l'alternateur **alt/lt** reste encore assez complexe, car elle nécessite d'expliciter par l'utilisateur la continuation à invoquer avec la valeur en cas de retour ;
- les messages et les scripts (généralement) ne sont pas des acteurs du système comme les autres, mais ceci n'est pas essentiel ;
- il existe une seule façon de création d'acteurs par **new-actor**, en effet on peut penser créer deux sortes d'acteurs pour mieux partager les connaissances communes entre les acteurs ;

- Pour être un modèle de programmation générale, il manque certaine souplesse en ce qui concerne la définition de méthodes. Comme on voit dans la définition de l'objet `bâtiment`, la liste des méthodes est très longue, ceci augmente la difficulté de mise au point de programme. Il faut donc étendre le modèle pour pouvoir définir des méthodes d'un objet d'une façon séparée et plus souple (par exemple : par une forme du genre `add-method`). Pour être utilisé, il faut sûrement optimiser les occupations mémoires, ceci est complètement laissé à côté dans la stade actuelle.

VI.3.2 Sur les objets programmés

En ce qui concerne la programmation des objets, il faut remarquer qu'une fois les objets et les modèles physiques explicités, la réalisation est immédiate. Par contre nous n'avons pas pu réaliser tous les objectifs fixés au départ, notamment sur les points suivants :

- les modèles thermiques du chapitre II ne sont pas tous utilisés, surtout en ce qui concerne les relations existantes entre les différents paramètres ;
- la maquette n'effectue pas encore la remise en cause des données après le bouclage de bilan, seul pour l'instant est effectué le listage des paramètres ayant un coefficient d'assertion faible ;
- le choix des méthodes est déterminé par un contexte qui est le niveau d'audit considéré. Il est évidement nécessaire d'élargir ce contexte aux autres paramètres, notamment les coefficients d'assertion des paramètres liés directement au calcul ;
- l'interface utilisateur reste très pauvre.

Conclusion et Perspectives

A l'époque actuelle on fait grand cas des livres.
Les livres ne sont fait que des mots.
Les mots ne valent que par les idées.
Les idées ont une origine qui ne peut s'exprimer par les mots.
Les discours que le monde transmet par les livres et qu'il considère comme précieux sont sans valeur.

— *Tchouang Tzeu*

Le diagnostic thermique de bâtiments existants pose un certain nombre de problèmes spécifiques quant à la réalisation des outils informatiques. Ces problèmes sont, jusqu'à présent mal pris en compte, liés à la nature partielle des données disponibles (données imprécises, informations qualitatives, optimisation du coût d'accès à l'information...), et à la complexité des objets à manipuler (nombre d'objets, complexité géométrique, dépendance forte entre les objets etc...).

L'objectif de notre recherche est donc de contribuer à l'élaboration d'une nouvelle génération d'outils d'aide au diagnostic thermique ou à la gestion énergétique de bâtiments.

Le développement récent de l'informatique offre de nombreux outils de programmation puissants, comme par exemple la programmation logique (systèmes experts), les modèles basés sur le concepts d'objets et d'acteurs. Le modèle que l'on s'est proposé d'expérimenter, et qui repose sur le concept d'acteurs, ouvre des perspectives intéressantes dans le traitement des informations partielles (choix des méthodes de calculs au cours du traitement des données de l'audit en fonction du contexte).

Nous avons réalisé une implémentation expérimentale d'un modèle de programmation intégrant des caractéristiques issues des modèles objet et acteur. Cette implémentation nous a permis de réaliser une maquette d'un outil destiné à l'audit thermique des bâtiments existants, en prenant notamment en compte la nature des données telles qu'elles.

La maquette réalisée à l'issue de cette recherche a permis de tester :

- Les nouveaux modèles de programmation tels que les objets et les acteurs, qui sont bien adaptés au problème de l'audit. La structuration par objets permet une meilleure organisation des entités descriptives d'un bâtiment existant, tandis que les acteurs

offrent une grande souplesse pour contrôler le déroulement des calculs en fonction des contextes, des exigences sur les résultats. En ce qui concerne la structuration des objets de bâtiment, le modèle objet est un modèle naturel, et facilite grandement la mise en œuvre de l'outil et augmente la modularité du programme.

- La gestion des imprécisions et des incertitudes constitue une amélioration appréciable de ce type d'outil, qui respecte l'approche des "auditeurs". Elle permet de rompre avec l'aspect souvent factice des calculs menés de façon déterministe et d'obtenir des conclusions plus nuancées et plus réalistes (fourchette de valeurs, remise en cause des données les moins sûres, en cas de non concordance entre consommations calculées et observées).

- En ce qui concerne le traitement de données et l'exploitation complète des modèles physiques existants, la structure de contrôle offerte par le modèle d'acteur permet de programmer d'une façon plus et plus élégante. Bien que ces modèles soient actuellement utilisés dans un contexte de calcul parallèle, il nous semble tout à fait intéressant d'appliquer ces notions dans un contexte de traitement des informations partielles où un certain nombre de modèles physiques peuvent être utilisés selon les objectifs du diagnostic.

- La distinction de différents niveaux de diagnostic et leur traitement correspondant est une étape importante dans la réalisation future des outils informatiques.

Dans la réalisation du modèle de programmation, l'accent a surtout été mis sur l'utilisation de la continuation. Nous avons pour cela un langage de programmation simple et élégant qui offre la continuation courante comme objet de première classe du langage, SCHEME. Ceci nous a permis de nous intéresser au problème des structures de contrôle. L'utilisation de la continuation permet de réaliser certaines structures d'une façon facile et élégante. Il faut noter que cette utilisation demande évidement un effort important de compréhension et de pratique.

Il conviendrait de noter que la réalisation décrite dans cette thèse est expérimentale. Nous nous sommes volontairement limités à un modèle simplifié.

Avant d'envisager une poursuite à ce travail, il conviendra de définir le ou les domaines d'application de l'outil à réaliser à partir de la maquette existante. Les deux grandes pistes de départ restent toujours viables : la gestion énergétique de patrimoine et le diagnostic thermique rapide de bâtiment (c'est en effet dans ces deux cas que le traitement des données partielles se présente avec le plus d'accuité). Il faudra également définir les types de bâtiments concernés : bâtiments résidentiels collectifs, petit tertiaire sans climatisation, groupes scolaires...

L'outil est destiné aux professionnels, spécialistes ou non d'un domaine du bâtiment, ou à un public plus large (particulier assisté d'un artisan par exemple). Dans le premier cas, l'approche des incertitudes et des imprécisions, développé à l'occasion de cette thèse, permettrait de rendre un réel service pour l'exploitation cohérente de l'ensemble des informations dont dispose un gestionnaire. Dans le deuxième cas, la recherche pourrait consister

en la réalisation d'un outil d'audit télématique performant et étendu à d'autres types de bâtiment que la maison individuelle.

Le plan de travail pourrait consister à définir les perspectives d'un prototype dédié à partir de la maquette actuelle. Ce travail serait réalisé en liaison avec des professionnels, pour une étude plus approfondie des différents modes d'investigation mis en œuvre (mesures, analyse de factures, ratios, entretien...) et de l'optimisation du coût d'accès à l'information (comment obtenir la meilleure précision sur la consommation à coût global constant ?). Cette étude permettrait à la fois la conception et la validation de l'outil. Une autre direction de recherche serait de scinder la maquette actuelle en deux outils : d'une part le module "modèle d'acteurs", écrit en SCHEME, et d'autre part le module "métreur", écrit en PASCAL, qui peut constituer un module indépendant à lui seul. Son développement autonome serait en effet possible. Il pourrait après être relié à une interface graphique, un outil télématique ou un outil de gestion... Dans tous les cas de figure, des améliorations de l'interface homme-machine, actuellement assez rudimentaire, sont à prévoir.

Enfin, le modèle d'acteurs est utilisable comme un outil général de programmation dans d'autres domaines, en particulier la conception de bâtiment. En effet, la conception comprend également la prise en compte des données imprécises (dans ce cas, non encore complètement définies) et non figées (le nombre augmentant avec l'élaboration du projet). Dans ce cas il est bien évidement nécessaire d'avoir un ensemble d'outils de développement qui faciliterait la mise en œuvre.

Comme il est dit au cours de l'exposé, le modèle d'acteur est actuellement destiné à la recherche des modèles de traitement dans un environnement parallèle, cet aspect pourrait également être intéressant dans les développements futures des outils informatiques puissants.

Annexe A

Définition du modèle

La première partie de cet annexe contient toutes les nouvelles syntaxes pour la défintion des objets décrites au chapitre VI. La seconde partie contient toutes les fonctions et syntaxes utilitaires.

A.1 Codes des extensions de syntaxe des acteurs

Définition de `def-act` :

```
(extend-syntax
 (def-act accointances scripts proxy)
 [(def-act name
    (accointances (var val) ...)
    (scripts (scr vcr) ...)
    (proxy pro))
  (define
    name
    (letrec
        ([self
          (letrec ([list-accointances
                    (make-env
                     (list (cons (quote var) (delay val)) ...))]
                   [list-scripts
                    (make-env
                     (list (cons (quote scr) (delay vcr)) ...))]
                   [proxy (list pro)])
            (lambda (msg) ; ; les messages sont :
              (case msg
                [accointances
                 (lambda (instance . args) list-accointances)]
                [set-var !
                 (lambda (instance . args)
                   (force (list-accointances
                    'set-var ! (car args) (delay (cadr args)))))]
                [set-scr !
                 (lambda (instance . args)
```

```
                (list-scripts
                 'set-var! (car args) (del_, (cadr args))))]

              [else
               (let ([v-acc (list-accointances 'get-var msg)])
                 (if v-acc ; ;msg correspand a une accointance
                     (lambda (instance . args)
                       (force (cdr v-acc)))
                     (let ([v-scr (list-scripts 'get-var msg)])
                       (if v-scr ; ;msg correspond a un script
                           (if (eq? msg 'reply)
                               (lambda (instance . args)
                                 (apply (force (cdr v-scr)) args))
                               (lambda (instance . args)
                                 (call/cc
                                  (lambda (k)
                                    (send self 'set-var! 'controle k)
                                    (apply (force (cdr v-scr))
                                           (cons instance args))))))
                           (lambda (instance . args)
                             (if
                              (null? (car proxy))
                              (printf "~s introuvable ~%" msg)
                              ; ; ici il faut mettre (fk (printf "  "))
                              (let ([f (lambda (p)
                                         (send-as p msg instance args))])
                                (delegue-f f proxy)))))))))])
      self))]

 [(def-act name (scripts (scr vcr) ...) (proxy pro))
  (def-act name (accointances) (scripts (scr vcr) ...) (proxy pro))]
 [(def-act name (accointances (var val) ...) (scripts (scr vcr) ...))
  (def-act name (accointances (var val) ...) (scripts (scr vcr) ...)
    (proxy ()))]
 [(def-act name (accointances (var val) ...) (proxy pro))
  (def-act name (accointances (var val) ...) (scripts) (proxy pro))]
```

Création de nouvel acteur :

```
(extend-syntax (new-actor)
 [(new-actor act-ref (accointances (var val) ...) (scripts (scr vcr) ...))
  (letrec
      ([self
        (letrec ([list-accointances
                  (make-env
                   (list (cons (quote var) (delay val)) ...))]
                 [list-scripts
                  (make-env
                   (list (cons (quote scr) (delay vcr)) ...))]
                 [proxy (list act-ref)])
```

```
          (lambda (msg)
            (case msg
              [accointances
               (lambda (instance . args) list-accointances)]
              [set-var!
               (lambda (instance . args)
                 (force (list-accointances
                         'set-var! (car args) (delay (cadr args)))))]
              [set-scr!
               (lambda (instance . args)
                 (list-scripts
                  'set-var! (car args) (delay (cadr args))))]

              [else
               (let ([v-acc (list-accointances 'get-var msg)])
                 (if v-acc ;;msg correspand a une variable
                     (lambda (instance . args)
                       (force (cdr v-acc)))
                     (let ([v-scr (list-scripts 'get-var msg)])
                       (if v-scr
                           (if (eq? msg 'reply)
                               (lambda (instance . args)
                                 (apply (force (cdr v-scr)) args))
                               (lambda (instance . args)
                                 (call/cc
                                  (lambda (k)
                                    (send instance 'set-var! 'controle k)
                                    (apply (force (cdr v-scr))
                                           (cons instance args))))))
                           (lambda (instance . args)
                             (if
                              (null? (car proxy))
                              (printf "~s introuvable ~%" msg)
                              (let ([f (lambda (p)
                                         (send-as p msg instance args))])
                                (delegue-f f proxy))))
                           ))))]))])
    self)]
 [(new-actor act-ref (accointances (var val) ...))
  (new-actor act-ref (accointances (var val) ...) (scripts))]
 [(new-actor act-ref (scripts (var val) ...))
  (new-actor act-ref (accointances) (scripts (var val) ...))]
 [(new-actor act-ref)
  (new-actor act-ref (accointances) (scripts))] )
```

Envoi de messages

```
(extend-syntax
 (send customer)
 [(send target msg (customer sk) args ...)
```

```
  (send sk 'reply (send target msg args ...))]
 [(send target msg args ...)
  (send-as target msg target (list args ...))])

(define send-as
  (lambda (component selector composite largs)
    (apply (component selector)
           (cons composite largs))))
```

Alternateur

```
(extend-syntax
 (alt)
 [(alt actor-list msg instance sk fk args ...)
  (do ([l actor-list (cdr l)])
      ([null? l]
       (printf " Plus de methodes a essayer, on effectue un ~%")
       (printf "  retour a la premiere methode ~%")
       (printf " L'ancienne valeur (par defaut) sera gardee ~%")
       (fk 'any))
    (call/cc
     (lambda (fk)
       (alt-message)
       (sk (send-as (car l) msg instance (list args ... fk))))))])

(define alt-message
  (lambda ()
    (printf "   On essaie les autres methodes ... ~%")
    (printf "~%")))

(extend-syntax
 (alt/lt)
 [(alt/lt (act1 ...) msg instance fk args ...)
  (call/cc
   (lambda (sk)
     (let ([list-act (list act1 ...)])
       (let ([first-act (car list-act)]
             [rest-act (cdr list-act)])
         (printf " La premiere methode est invoquee   ~%")
         (sk (send-as
              first-act msg instance (list args ...
               (lambda (a) (alt rest-act msg instance sk a args ...))
               ))))
       )))]
 [(alt/lt list-act msg instance fk args ...)
  (call/cc
   (lambda (sk)
    (let ([first-act (car list-act)]
          [rest-act (cdr list-act)])
      (printf " La premiere methode est invoquee   ~%")
```

```
      (sk (send-as first-act
                   msg instance (list args ...
                    (lambda (a) (alt rest-act msg instance sk a args ...))
                    ))))    ))])

(define return-state
  (lambda (fk old-value)
    (lambda (b)
      (printf "~%")
      (printf "Invocation de la continuation pour le retour ! !~%")
      (printf "~%")
      (fk old-value))))

; ; on definit l'environnement par des alistes
(define make-env
  (lambda (alist)
    (lambda (msg . args)
      (case msg
        [add (set ! alist (cons (cons (car args) (cadr args)) alist))]
        [set-var !
         (let ([t1 (assq (car args) alist)])
           (if (null ? t1)
               (begin
                 (set ! alist
                       (cons (cons (car args) (cadr args))
                             alist))
                 (cadr args))
               (begin (set-cdr ! t1 (cadr args)) (cadr args))))]
        [get-var (let ([t (assq (car args) alist)])
                   (if (not t)
                       ()
                       t))]
        [get-all alist]
        [else (let ([t1 (assq msg alist)])
                (if t1
                    (cdr t1)
                    ()))])))

; ; quelques fonctions utilitaires
(define delegue-f
  (lambda (f lproxy)
    (if (null ? lproxy)
        ()
        (or (f (car lproxy))
            (delegue-f f (cdr lproxy))))))

(define member-cdr
  (lambda (x l)
    (cond [(null ? l) ()]
          [(eq ? x (car l)) l]
          [else (member-cdr x (cdr l))])))
```

```
   ;; La continuation d'echec au top-level

(define fk0
  (lambda (x) (call/cc
                (lambda (k) (k x)))))
```

A.2 Les fonctions utilitaires

```
;; projet MA'TH   les fonctions utilitaires
;; cree le 5-09-89 modifie le 27-02-90

;; somme d'une liste de valeurs
(define somme-de
  (lambda (liste nom action)
    (do ([i 1 (1+ i)]
         [l liste (cdr l)]
         [som 0 (begin
                  (printf "***** Pour la ~s eme ~s : *****~%" i nom)
                  (set! som
                        (+ som (action (car l)))))])
        ([null? l] som))))

;;calcul des imprecisions
;;representation des valeurs sous forme de (Xe, DeltaX/Xe)

(define make/delta
  (lambda (v d)
    (cons v d)))

(define valeur-moy
  (lambda (x)
    (if (atom? x)
        x
        (car x))))

(define delta;; le delta
  (lambda (x)
    (if (atom? x)
        0
        (* (car x) (cdr x)))))

(define delta%;;le delta/valeur%
  (lambda (x)
    (if (atom? x)
        0
        (cdr x))))

(define delta+
  (lambda (x y)
```

```
    (let ((ve (+ (valeur-moy x)
                 (valeur-moy y))))
      (if (= 0 ve)
          (make/delta 0 0)
          (make/delta
           ve
           (/ (sqrt (+ (square  (delta x))
                       (square (delta y))))
              ve))))))

(define delta-
  (lambda (x y)
    (let ((ve (- (valeur-moy x)
                 (valeur-moy y))))
      (if (= 0 ve)
          (make/delta 0 0)
      (make/delta
       ve
       (/ (sqrt (+ (square  (delta x))
                   (square (delta y))))
          ve))))))

(define delta*
  (lambda (x y)
    (let ((ve (* (valeur-moy x)
                 (valeur-moy y))))
      (if (= 0 ve)
          (make/delta 0 0)
      (make/delta
       ve
       (/ (sqrt (+ (square (* (valeur-moy y)
                              (delta x)))
                   (square (* (valeur-moy x)
                              (delta y)))))
          ve))))))

(define delta/
  (lambda (x y)
    (let ((ve (/ (valeur-moy x)
                 (valeur-moy y))))
      (make/delta
       ve
       (/ (sqrt (+ (square (/ (delta x) (valeur-moy y)))
                   (square (/ (* (delta y) (valeur-moy x))
                              (square (valeur-moy y))))))
          ve)))))

(define delta-sqrt
  (lambda (x)
```

```
    (let* ([ve (valeur-moy x)]
           [d (delta x)]
           [ve1 (sqrt ve)])
      (make/delta ve1 (* (sqrt d) (/ (* 2 ve1)))))))

;;-------------------------------------------
(define echec?
  (lambda (aud-var)
    (cond [(atom? aud-var)
           (equal? aud-var 'echec)]
          [(pair? aud-var)
           (if (atom? (cdr aud-var))
               (equal? (cdr aud-var) 'echec)
               (equal? (cadr aud-var) 'echec))]
          [else
           (or (equal? (var-val aud-var) 'echec)
               (< (var-cfa aud-var) .5))])))
            ;; on peut modifier ce predicat pour intergrer d'autres
            ;; cas d'echec

;; conversion d'un nombre reel en reel avec N decimales
(define convert-to-n-dec
  (lambda (x n)
    (let ([tenexpt (expt 10 n)])
      (exact->inexact
       (/ (round (* x tenexpt))
          tenexpt)))))

;; Definition du constructeur de variable audit
;; constructeur
(extend-syntax
 (make-var type)
 [(make-var (type typ) val dom unit his coa)
  (list typ val dom unit his coa)]
 [(make-var (type typ) val dom unit his)
  (list typ val dom unit his 1)]
 [(make-var val dom unit his coa)
  (let ([typ (cond [(= coa 2) 'certaine]
                   [else 'incertaine])])
    (make-var (type typ) val dom unit his coa))]
 [(make-var val dom unit his)
  (make-var val dom unit his 2)])

;; les selecteurs
(define var-type car)
(define var-val cadr)
(define var-dom caddr)
(define var-unit cadddr)
(define var-hist
```

```
  (lambda (var)
    (car (cddddr var))))
(define var-cfa
  (lambda (var)
    (if (or (eq? (var-type var) 'certaine)
            (eq? (var-type var) 'certaine-imprecise))
        1
        (cadr (cddddr var)))))
(define var-coefficient-assertion var-cfa)

(define var?
  (lambda (var)
    (and (not (null? var))
         (not (atom? var))
         ;;(= (length var) 5)
         (symbol? (car var)))))

(define var-set!
  (lambda (var id new-val)
    (case id
      [type (set-car var new-val)]
      [val (set-car! (cdr var) new-val)]
      [dom (set-car! (cddr var) new-val)]
      [unit (set-car! (cdddr var) new-val)]
      [hist (set-car! (cddddr var) new-val)]
      [cfa (set-car! (cdr (cddddr var)) new-val)])))
;; quand on fait (var-set! var 'cfa .9), il faut changer le type de var

(define var-print
  (lambda (var)
    (let ([typ (var-type var)]
          [val (var-val var)]
          [dom (var-dom var)]
          [unit (var-unit var)]
          [his (var-hist var)]
          [cfa (var-cfa var)])
      (cond
       [(and (eq? typ 'certaine) (atom? val))
        (if unit
            (printf " ~s(~s)~% obtenu (ou propose) par ~s ~%" val unit his)
            (printf " ~s~%  obtenu (ou propose) par ~s ~%" val his))
        (printf-domaine dom)]
       [(and (eq? typ 'incertaine) (atom? val))
        (if unit
            (printf "~s(~s)~%  obtenu (ou propose) par ~s ~%" val unit his)
            (printf "~s~%  obtenu (ou propose) par ~s ~%" val his))
        (printf-domaine dom)
        (printf " et le coefficient d'assertion : (par defaut ~s) ~%" cfa)]
       [(and (eq? typ 'certaine) (pair? val))
```

```
        (if unit
            (printf "~s +- ~s% (~s)~%      obtenu (ou propose) par ~s ~%"
                    (car val) (cdr val)
                    unit his)
            (printf "~s +- ~s%~%       obtenu (ou propose) par ~s ~%"
                    (car val) (cdr val)
                    his))
        (printf-domaine dom)]
       [(and (eq? typ 'incertaine) (pair? val))
        (if unit
            (printf
             "~s +- ~s% (~s)~%      obtenu (ou propose) par ~s~%"
             (car val) (cdr val) unit his)
            (printf
             "~s +- ~s%~%       obtenu (ou propose) par ~s~%"
             (car val) (cdr val) his))
        (printf-domaine dom)
        (printf "    et le coefficient d'assertion : ~s ~%" cfa)]
       [else (printf "Type inconnu~%")])))

(define printf-domaine
  (lambda (dom)
    (printf "    le domaine de valeur possible est :~% ")
    (cond [(atom? dom) (printf "    ~s ~%" dom)]
          [(list? dom)
           (letrec ([pf
                     (lambda (l)
                       (cond [(null? (cdr l)) (printf "~s" (car l))]
                             [else
                               (printf "~s," (car l)) (pf (cdr l))]))])
             (printf "  {") (pf dom) (printf "}~%"))])))

(define var-read #f)
(letrec
    ([read-def (lambda (ca)
                 (printf "===>? ")
                 (let ([rep (read)])
                   (if (eq? rep 'o) ca rep)))]
     [read-def-ca
      (lambda (ca)
        (let ([rep (read-def ca)])
          (cond [(or (eq? rep 'I) (eq? rep 'inconnu))
                 (message-screen
                  "On prendre un CA = ~s~%" ca)]
                [(or (not (number? rep)) (< rep 0) (> rep 1))
                 (message-screen
                  "Le coefficient assertion doit etre compris [0, 1]
                  Redonnez le CA  ~%" ())
                 (read-def-ca ca)]
```

```
            [else rep]))))]
    [filtre
     (lambda (x l)
       (cond
        [(null? l) #f]
        [(atom? l)
         (case l
           [real (if (real? x) x #f)]
           [reel (if (real? x) x #f)]
           [entier (if (integer? x) x #f)]
           [string (if (string? x) x #f)]
           [symbol (if (symbol? x) x #f)]
           [else #f])]
        [(and (= (length l) 3) (or (eq? (cadr l) '..) (eq? (cadr l) '/)))
         (if (and (number? x) (>=? x (car l)) (<=? x (caddr l))) x #f)]
        [else (cond
               [(appartenir? x l) x]
               [(and (integer? x) (<=? x (length l)))
                (list-ref l (- x 1))]
               [else #f])]))]

    [read-pair-cert
     (lambda (old-val dom unit)
       (let ([r1 (read-def (car old-val))])
         (if (filtre r1 dom)
             (let ([r2 (read-def (cdr old-val))])
               (if (filtre r2 '(0 .. 50))
                   (list (make/delta r1 r2) dom unit 'saisi)
                   (list 'echec dom unit 'saisi)))
             (list 'echec dom unit 'saisi))))]
    [read-pair-incert
     (lambda (val dom unit coefa)
       (let ([rep-pair (read-pair-cert val dom unit)])
         (if (eq? 'echec (car rep-pair))
             (append rep-pair (list coefa))
             (append rep-pair (list (read-def-ca coefa))))))]
    [read-atom-cert
     (lambda (old-val dom unit)
       (let ([rep (filtre (read-def old-val) dom)])
         (if rep
             (list rep dom unit 'saisi)
             (list 'echec dom unit 'saisi))))]
    [read-atom-incert
     (lambda (val dom unit coefa)
       (let ([rep (read-atom-cert val dom unit)])
         (if (eq? 'echec (car rep))
             (append rep (list coefa))
             (append rep (list (read-def-ca coefa))))))])

(set! var-read
```

```
        (lambda (id obj-id var)
          (let ([typ (var-type var)]
                [val (var-val var)]
                [dom (var-dom var)]
                [unit (var-unit var)]
                [his (var-hist var)]
                [cfa (var-cfa var)])
            ; ; (var-print var)
            (if (appartenir? his '(calcule metreur saisi))
                var
                (begin
                  (printf "~%")
                  (printf "=======>~%")
                  (printf "~s : ~%" obj-id)
                  (printf "  ~s est =  " id)
                  (var-print var)
                (cons
                 typ
                 (if (atom? val)
                     (if (eq? typ 'incertaine)
                         (read-atom-incert val dom unit cfa)
                         (read-atom-cert val dom unit))
                     (if (eq? typ 'incertaine)
                         (read-pair-incert val dom unit cfa)
                         (read-pair-cert val dom unit)))))))))

; ; definitions des operations arithmetiques pour variables du type var
(define audit/somme-de
  (lambda (liste action)
    (do ([i 1 (1+ i)]
         [l liste (cdr l)]
         [som 0 (begin
                  (set! som
                        (audit+ som (action (car l)))))])
        ([null? l] som))))

(define audit*
  (lambda (v1 v2)
    (operation-audit-var v1 v2 '*
                         (delta* (var-val2 v1)
                                 (var-val2 v2)))))

(define audit/
  (lambda (v1 v2)
    (operation-audit-var v1 v2
                         '/ (delta/ (var-val2 v1)
            (var-val2 v2)))))

(define audit+
```

```
  (lambda (v1 v2)
    (operation-audit-var v1 v2 '+
     (delta+ (var-val2 v1)
             (var-val2 v2)))))

(define new-audit+
  (lambda (v1 . v2)
    (let ([l (cons v1 v2)])
      (audit/somme-de l (lambda (x) x)))))

(define audit-
  (lambda (v1 v2)
    (operation-audit-var v1 v2 '-
     (delta- (var-val2 v1)
             (var-val2 v2)))))

(define audit-sqrt
  (lambda (x)
    (let* ([val (var-val x)]
           [val1 (delta-sqrt val)])
      (make-var val1 'calcule 'm (var-cfa x)))))

(define operation-audit-var
  (lambda (v1 v2 op res)
    (cond [(and (var? v1) (var? v2))
           (let ([unit1 (var-unit v1)] [unit2 (var-unit v2)])
           (make-var
            res
            (max-interval (var-dom v1) (var-dom v2))
            (unit-op op unit1 unit2)
            'calcule
            (min-coefficient-assertion
             (var-cfa v1) (var-cfa v2))))]
          [(var? v1)
           (make-var res (var-dom v1) (var-unit v1) 'calcule
                                 (var-cfa v1))]
          [(var? v2)
           (make-var res (var-dom v2) (var-unit v2) 'calcule
                         (var-cfa v2))]
          [else res])))

; ; calculs des unit\'es, non defini completement
(define unit-op
  (lambda (op u1 u2)
    (cond
     [(and (null? u1) (null? u2)) #f]
     [(null? u1) u2]
     [(null? u2) u1]
     [(eq? op '*)
```

```
      (string->symbol (string-append
                       (symbol->string u1) (symbol->string u2)))]
     [(eq? op '/)
      (string->symbol
       (string-append (string-append (symbol->string u1) "/")
                      (symbol->string u2)))]
     [else u1])))

(define var-val2
  (lambda (v)
    (cond [(atom? v) v]
          [(atom? (cdr v)) v]
          [(var? v) (var-val v)]
          [else v])))

(define max-interval
  (lambda (int1 int2)
    (cond [(eq? int1 int2) int1]
          [(string>? (symbol->string int1) (symbol->string int2)) int1]
          [else int2])))
(define min-coefficient-assertion min)

(define member?
  (lambda (x l)
    (cond [(null? l) #f]
          [(atom? l)
           (case l
             [real (real? x)]
             [reel (real? x)]
             [entier (integer? x)]
             [string (string? x)]
             [symbol (symbol? x)]
             [else #f])]
          [(and (= (length l) 3) ; ; interval du genre [-1 .. 1] ou [1 / 2]
                (or (eq? (cadr l) '..) (eq? (cadr l) '/)))
           (and (>? x (car l)) (<? x (caddr l)))]
          [else (or (appartenir? x l)
                    (and (integer? x)
                         (<? x (length l))))])))

(define appartenir?
  (lambda (x l)
    (if (null? l)
        #f
        (or (eq? x (car l))
            (appartenir? x (cdr l) )))
```

Annexe B

Définitions des objets de bâtiment

```
; ; projet MA'TH    defintion des objets BATIMENT CLIMAT
; ; cree le 30 novembre 1989,    modifie le 1 mars 1990

(def-act batiment
  (accointances
   [nom 'batiment]
   [type-batiment
    (make-var
     'residentiel-collectif
     '(residentiel-collectif individuel tertiaire) () 'defaut)]
   [lieu (make-var '26 'entier () 'defaut)]
   [site
    (make-var
     'plat-degage
     '(ile sommet-en-altitude cotier plat-degage pente-expose)
     () 'defaut)]
   [niveau-audit
    (make-var 'intermediaire '(rapide intermediaire approfondi) () 'defaut)]
   [climat (new-actor climat)]
   [equipement (new-actor equipement)]
   [nombre-zones (make-var 1 'entier () 'defaut)]
   [liste-zones #f]
   [consommation-calculee  (make-var 0 'reel 'kWh 'defaut)])
 (scripts
   [consommation
    (lambda (self fk)
      (message-screen "       Calcul de la consommation : " ())
      (let*
          ([equip (send self 'create 'equipement fk)]
           [clim (send self 'create 'climat fk)]
           [n (var-val (send self 'read 'nombre-zones fk))]
           [l0 (send self 'make-list n zone 'liste-zones 'zone)]
           [result
            (audit/
             (audit* 24
```

```
                    (audit/somme-de
                     10
                     (lambda (z) (send z 'consommation self fk))))
              (send equip 'rendement self fk))])
      (var-set! result 'unit 'kWh)
      (send self 'set-var! 'consommation-calculee result fk)))]

[consommation-facturee
 (lambda (self fk)
   (printf "~%")
   (printf " Consommation facturee ~%")
   (printf "~%")
   (random 160)
   (let ([cons-cal (send self 'consommation-calculee fk)])
     ((rec f
           (lambda (x)
             (if (and (> x 1/2) (< x 3/2))
                 (audit* (* 1.2 x) cons-cal)
                 (f (/ (random 160) 100)))))
      (/ (random 160) 100))))]

[comparaison
 (lambda (self fk)
   (let*
       ([c1 (send self 'consommation fk)]
        [c2 (send self 'consommation-facturee fk)]
        [c11 (cons
              (convert-to-n-dec (/ (car (var-val c1)) 1000) 1)
              (convert-to-n-dec (cdr (var-val c1)) 1))]
        [c22 (cons
              (convert-to-n-dec (/ (car (var-val c2)) 1000) 1)
              (convert-to-n-dec (cdr (var-val c2)) 1))]
        [r (convert-to-n-dec
            (* 100 (abs (car (var-val (audit/
                                       (audit- c1 c2) c2)))))
            0)])
     (printf  "Consommation calculee = ~s +- ~s% kWh~%
               Consommation facturee = ~s +- ~s% kWh~%
               dC/ C_facturee
               = ~s%~%" (car c11) (cdr c11) (car c22) (cdr c22) r)
     (cond
      [(> r 15)
       (printf "Le resultat est assez mauvais~%")
       (printf "On vous conseille de revoir certains parametres~%")
       (printf "Voici la liste des parametres dont le CA est faible~%")]
      [else
       (printf "Le resultat est acceptable, mais vous pouvez toujours~%")
       (printf " modifier certains parametres dont voici une liste~%")
       (printf " avec un CA que j'estime faible~%")])
```

```
        (let* ([acc (send self 'find-acc fk)]
               [facc (send self 'fichier-accoint fk)]
               [fres (send self 'fichier-results fk)]
               [port (open-output-file fres)]
               [port2 (open-output-file facc)])
          (send self 'find-incert port fk)
          (send self 'find-res port2 fk)
          (close-output-port port2)
          (close-output-port port))))]

   [debut
    (lambda (self fk)
     (explication)
     (printf "~% Donnez le nom du batiment : ===> ")
     (let ([nom-projet (read)])
       (printf "~% Lecture de donnees du fichier cree par le metreur ?~%")
       (printf "  O(ui)/N(on)  ~%  ===> ")
       (let ([rep (read)]
             [fichier-metre (string-append
                             (symbol->string nom-projet) ".met")]
             [fichier-saisi (string-append
                             (symbol->string nom-projet) ".acc")]
             [fichier-resul (string-append
                             (symbol->string nom-projet) ".res")])
         (if (or (eq? rep 'O) (eq? rep 'oui))
             (begin
               (send self 'set-var! 'nom nom-projet fk)
               (send self 'set-var! 'fichier-metreur fichier-metre fk)
               (send self 'set-var! 'fichier-accoint fichier-saisi fk)
               (send self 'set-var! 'fichier-results fichier-resul fk)
               (printf " Lecture du fichier metreur <~s>...~%" fichier-metre)
               (call-with-input-file fichier-metre
                 (lambda (p)
                   (lire-bat self nom-projet fk p)))))
         (saisie-generale self fk)
         (send self 'comparaison fk))))])
 (proxy obj))

; ; Definition de l'objet CLIMAT
(def-act climat
  (accointances
   [nom 'climat]
   [zone-climat-valeur
    (make-var 'zone-h2 '(zone-h1 zone-h2 zone-h3) () 'defaut)])
  (scripts
   [Dj-unifie
    (lambda (self bat fk)
      (let ([zm (var-val (send self 'zone-climat bat fk))])
          (cond
```

```
          [(eq? zm 'zone-h1) 2500] ; ; ce n'est pas la bonne formule
          [(eq? zm 'zone-h2) 2600]
          [(eq? zm 'zone-h3) 2700]))))]
   [edit-result
    (lambda (self fk)
      (printf "ZONE CLIMAT : ~s ~%"
              (var-val (send self 'zone-climat fk)))
      (printf "Dj-Unifies : ~s (degres-jours~%"
              (send self 'Di-unifie fk)))]
   [ensoleillement
    (lambda (self fk)
      (printf "L'ensoleillement de la Region, ...~%")
      1.)]
   [zone-climat
    (lambda (self bat fk)
      (let ([obn (send self 'nom fk)])
        (send self 'read 'zone-climat-valeur fk)))])
    (proxy obj))

(define debut
  (lambda ()
   (let ([b1 (new-actor batiment)]) ; ; creation d'un batiment
     (set! **current-bat** b1) ; ; temporaire
     (send b1 'debut fk0))))

(define saisie-generale
  (lambda (bat fk)
    (message-screen "~%      *** SAISIE GENERALE *** ~%" ())
    (let ([nom (send bat 'nom fk)])
    (send bat 'read 'type-batiment fk)
    (send bat 'read 'niveau-audit fk)
    (send bat 'read 'lieu fk)
    (send bat 'read 'site fk)  )))
```

Annexe C

Interface avec le métreur

```
; ; prjet MA'TH  interface avec le metreur
; ; cree le 10 decembre 89      modifie le 2 mars 1990
; ; LE Xiaohua

; ; lecture du fichier metreur   avec la creation des acteurs
(define boucle
  (lambda (n prefix symb f)
    (do ([i n (- i 1)]
         [l () (cons (f (string->symbol
                         (string-append
                          (if (string? prefix)
                              prefix
                              (symbol->string prefix))
                          "." (symbol->string symb)
                          (integer->string (1+ (- n i))))))) l)])
        ([= 0 i] (reverse l)))))

(define lire-bat
  (lambda (bat nom fk port)
    (let ([r4 (begin (read port) (read port))] ; ;nb-zones
          [r5 (begin (read port) (read port))]) ; ; niveau-audit (1 2 3)
      (let* ([lz (boucle r4 nom 'zone
                         (lambda (nom-zone)
                           (lire-zone nom-zone port)))]
             [n1 (cond [(= r5 1) 'rapide]
                       [(= r5 2) 'intermediaire]
                       [(= r5 3) 'approfondi])]
             [nv (make-var
                  n1 '(rapide intermediaire approfondi) () 'metreur)])
        (send bat 'set-var! 'niveau-audit nv fk)
        (send bat 'set-var!
              'nombre-zones (make-var r4 'integer () 'metreur))
        (send bat 'set-var! 'liste-zones lz)))))

(define lire-zone
```

```
(lambda (nomz port)
  (let*
      ([r0 (begin (read port) (read port))] ; ; numero zone
       [r1 (begin (read port) (read port))] ; ;date-construction
       [r2 (begin (read port) (read port))] ; ;volume
       [r3 (begin (read port) (read port))] ; ;hauteur
       [r4 (begin (read port) (read port))] ; ;nombre-etages
       [r5 (begin (read port) (read port))] ; ; nb-povs
       [r6 (begin (read port) (read port))] ; ;surface-plancher
       [r7 (begin (read port) (read port))] ; ; ori par 1
       [rr11 (begin (read port) (read port))] ; ;statut
       [lp (boucle r5 nomz 'paroi-verticale
                   (lambda (npov) (lire-pov npov port)))]
       [r9 (begin (read port) (read port))] ; ;nombre-cage-escalier
       [lce (boucle r9 nomz 'cage-escaliers
                    (lambda (nomc) (lire-cage nomc port)))]
       [surf (make-var (/ r6 r4) 'reel 'm2 'metreur)]
      ; ; plancher bas
       [p1 (read port)] ; ; chiffre indiquant contigute pb
       [p2 (read port)]
       [p3 (read port)]
       [ph1 (read port)] ; ; chiffre indiquant contigute pb
       [ph2 (read port)]
       [ph3 (read port)]
       [pbct #f] [phct #f] [comble #f])

    (cond
     [(= p1 1)
      (set ! pbct
           (list (cons
                  (generate-name
                   (list (string-last (symbol->string nomz)) "."
                         "zone" (integer->string p3)))
                  surf)))]
     [else (set ! pbct (list (cons p3 surf)))])
    (cond
     [(= ph1 7)
      (set ! phct
           (list (cons (generate-name
                        (list (string-last
                               (symbol->string nomz))
                              "."
                              "zone" (integer->string ph3)))
                       surf)))]
     [(= ph1 1) (set ! phct #f)]
     [(= ph1 2)
      (let ([nomcomble (generate-name (list nomz "." "comble"))])
        (set ! phct (list (cons nomcomble surf)))
        (set ! comble (lire-comble nomcomble port)))]
```

```
        [else
         (let ([taux-vent
                (cond
                 [(= ph1 3) 'fort]
                 [(= ph1 4) 'moyen]
                 [(= ph1 5) 'faible]
                 [(= ph1 6) 'faible])])
           (set!
            comble
            (new-actor
             comble-non-chauffe
             (accointances
              [taux-ventilation
               (make-var taux-vent '(fort faible moyen) () 'metreur)]))))
           (set! phct (list (cons ph2 surf))))])

       (let ([pb (construire-plancher-bas nomz surf pbct port)]
             [ph (construire-plancher-haut nomz surf phct port)]
             [st (if (= rr11 1) 'chauffe 'non-chauffe)])
         (new-actor
          zone
          (accointances
           [nom nomz]
           [date-construction (make-var r1 'entier () 'metreur)]
           [statut (make-var st '(chauffe non-chauffe) () 'metreur)]
           [volume (make-var r2 'reel 'm3 'metreur)]
           [hauteur (make-var r3 'reel 'm2 'metreur)]
           [nombre-etages (make-var r4 'entier () 'metreur)]
           [nombre-cages-escaliers r9]
           [liste-cages-escaliers lce]
           [nombre-parois-verticales
            (make-var r5 'entier '() 'metreur)]
           [liste-parois-verticales lp]
           [comble comble]
           [plancher-haut ph]
           [plancher-bas pb])))))))

(define lire-pov
  (lambda (nomp port)
    (let*
        ([r0 (begin (read port) (read port))] ; ; no pov
         [r1 (begin (read port) (read port))] ; ;long
         [r3 (begin (read port) (read port))] ; ;surf-opaque
         [r31 (begin (read port) (read port))] ; ; surf-vitree
         [r4 (begin (read port) (read port))] ; ;zone-contigue n.paroi
         [r41 #f] [r42 0.] [lcont #f] [surface-b 0] [donne-sur 0])
      (if (not (= r4 0))
          (let ([num (begin (read port) (read port))]
                [nombat (string-f (symbol->string nomp))])
```

```
            (set! r41 (string->symbol
                       (string-append nombat "." "zone"
                                      (integer->string num))))
            (set! r42 (begin (read port) (read port))) ; ; n zone et surface
            (set! lcont
                  (list (cons r41 (make-var r42 'reel 'm2 'metreur))))))
       (let ([r6 (read port)]) ; ; donne-sur 0 :ext 1 :autre bat chauff 2 : autr
         (cond
          [(eq? r6 'surface_contigue_batiment_chauffe)
           (set! surface-b r3)
           (set! donne-sur 1)]
          [(eq? r6 'surface_contigue_batiment_non_chauffe)
           (set! surface-b r3)
           (set! donne-sur 1)]
          [else ()])
         (let ( ; ;[r7 (begin (read port) (read port))] ; ;orintation
               [lpv (boucle 1 nomp 'paroi-vitree
                            (lambda (nmpv)
                              (lire-pv nmpv r31 port)))])
           (new-actor
            paroi-verticale
            (accointances
             [nom nomp]
             [longueur (make-var r1 'reel 'm 'metreur)]
             [surface-opaque (make-var r3 'reel 'm2 'metreur)]
             [zone-contigue-paroi (make-var r4 'entier () 'metreur)]
             [liste-contiguites lcont]
             [donne-sur (make-var donne-sur '(0 1) () 'metreur)]
             [val-orientation ; ; a revoir
              (make-var 'ese '(ese ene o n se so ne no) () 'metreur 1)]
             [nombre-parois-vitrees (make-var 1 'entier () 'metreur)]
             [liste-parois-vitrees lpv])))))))

(define lire-pv
  (lambda (nomp surf-vit port)
    (new-actor
       paroi-vitree
       (accointances
        [nom nomp]
        [val-surface (make-var surf-vit 'reel 'm2 'metreur)]))))

(define construire-plancher-haut
  (lambda (nom surf lc port)
    (let ([nomp (generate-name (list nom "." "plancher-haut"))])
      (new-actor
       plancher-haut
       (accointances
        [nom nomp]
        [liste-contiguites lc]
```

```
        [donne-sur (make-var 0 '(0 1) () 'metreur)]
        [surface surf])))))

(define construire-plancher-bas
  (lambda (nom surf lc port)
    (let ([nomp (generate-name (list nom "." "plancher-bas"))])
      (new-actor
       plancher-bas
       (accointances
        [nom nomp]
        [liste-contiguites lc]
        [donne-sur (make-var 0 '(0 1) () 'metreur)]
        [surface surf])))))

(define lire-cage
  (lambda (nomc port)
    (let ([r1 (begin (read port) (read port))]
          [r2 (begin (read port) (read port))]
          [r3 (begin (read port) (read port))])
      (new-actor cage-escalier
                 (accointances [nom nomc]
                               [surface r1])))))

(define lire-comble
  (lambda (nomc port)
    (let ([vol (read port)]
          [statut (read port)]
          [ouv (read port)])
      (new-actor
       comble-chauffe
       (accointances
        [nom nomc]
        [volume (make-var vol 'reel () 'metreur)]
        [ouverture ouv])))))

(define string-f
  (lambda (s)
    (let ([l (string->list s)])
      (letrec
          ([f
            (lambda (l res)
              (cond
               [(null? l) (list->string (reverse res))]
               [(char=? (car l) #\.) (list->string (reverse res))]
               [else
                (f (cdr l)
                   (cons (car l) res))]))])
        (f l '())))))
```

```
(define string-last
  (lambda (s)
    (let ([l (string->list s)])
      (letrec
          ([f (lambda (l1)
                (cond
                 [(null? l1) ()]
                 [(char=? #\. (car l1))
                  (cdr l1)]
                 [else (f (cdr l1))]))])
        (list->string (reverse (f (reverse l))))))))
```

Annexe D

Exemple

L'exemple présenté est celui du bâtiment de l'Ecole des Mines de Paris.

Fichier résultat produit par le métreur et utilisé par le module de calcul :

```
Nombre-de-zones : 5
Niveau-daudit : 2
Pour-la-zone 1
    date-de-construction 1900
    volume  1.1e+4
    hauteur  9.0e+0
    nombre-detages 3
    nombre-parois-opaques-verticales 4
    surface-plancher  3.7e+3
    orientat-par-1 5
    statut 1
       paroi-n 1
       longueur-de-paroi  3.5e+1
       surface-opaque-verticale  3.2e+2
       surface-vitree  9.2e+1
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 2
       longueur-de-paroi  3.5e+1
       surface-opaque-verticale  3.2e+2
       surface-vitree  9.2e+1
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 3
       longueur-de-paroi  3.5e+1
       surface-opaque-verticale  3.2e+2
       surface-vitree  9.2e+1
       contiguite-sur-paroi 3 avec-la-zone 2
       contiguite-sur-surf  2.4e+1
       paroi-donnant-sur-exterieur
       paroi-n 4
       longueur-de-paroi  3.5e+1
       surface-opaque-verticale  3.2e+2
       surface-vitree  9.2e+1
       contiguite 0
       paroi-donnant-sur-exterieur
  nombre-de-cage-descalier 0
3 zone en-contact-avec-le-sol
2 zone combles-chauffes-vol  2.4e+3   ouvert  2.0e+1

Pour-la-zone 2
    date-de-construction 1900
    volume  7.0e+2
    hauteur  3.0e+0
    nombre-detages 1
    nombre-parois-opaques-verticales 4
    surface-plancher  2.3e+2
    orientat-par-1 5
    statut 2
       paroi-n 1
       longueur-de-paroi  8.0e+0
       surface-opaque-verticale  2.4e+1
       surface-vitree  3.9e+0
       contiguite-sur-paroi 1 avec-la-zone 1
       contiguite-sur-surf  2.4e+1
       paroi-donnant-sur-exterieur
       paroi-n 2
       longueur-de-paroi  2.9e+1
       surface-opaque-verticale  8.7e+1
       surface-vitree  3.9e+0
```

```
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 3
       longueur-de-paroi  8.0e+0
       surface-opaque-verticale  2.4e+1
       surface-vitree  3.9e+0
       contiguite-sur-paroi 3 avec-la-zone 3
       contiguite-sur-surf  2.4e+1
       paroi-donnant-sur-exterieur
       paroi-n 4
       longueur-de-paroi  2.9e+1
       surface-opaque-verticale  8.7e+1
       surface-vitree  3.9e+0
       contiguite 0
       paroi-donnant-sur-exterieur
  nombre-de-cage-descalier 0
3 zone en-contact-avec-le-sol
2 zone combles-chauffes-vol  2.4e+3   ouvert  2.0e+1

Pour-la-zone 3
    date-de-construction 1650
    volume  1.0e+3
    hauteur  3.0e+0
    nombre-detages 1
    nombre-parois-opaques-verticales 8
    surface-plancher  3.5e+2
    orientat-par-1 5
    statut 1

       paroi-n 1
       longueur-de-paroi  2.0e+0
       surface-opaque-verticale  6.0e+0
       surface-vitree  4.8e+0
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 2
       longueur-de-paroi  1.9e+1
       surface-opaque-verticale  5.7e+1
       surface-vitree  4.8e+0
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 3
       longueur-de-paroi  3.3e+1
       surface-opaque-verticale  8.7e+1
       surface-vitree  4.8e+0
       contiguite-sur-paroi 3 avec-la-zone 2
       contiguite-sur-surf  2.4e+1
       paroi-donnant-sur-exterieur
       paroi-n 4
       longueur-de-paroi  1.0e+1
       surface-opaque-verticale  3.0e+1
       surface-vitree  4.8e+0
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 5
       longueur-de-paroi  3.5e+1
       surface-opaque-verticale  9.3e+1
       surface-vitree  4.8e+0
       contiguite 0
       paroi-donnant-sur-exterieur

       paroi-n 6
```

```
       longueur-de-paroi  2.9e+1
       surface-opaque-verticale  8.7e+1
       surface-vitree  4.8e+0
       contiguite-sur-paroi 6 avec-la-zone 4
       contiguite-totale  8.7e+1
       paroi-donnant-sur-exterieur
       paroi-n 7
       longueur-de-paroi  2.8e-40
       surface-opaque-verticale  3.0e+1
       surface-vitree  4.8e+0
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 8
       longueur-de-paroi  9.2e-41
       surface-opaque-verticale  3.0e+1
       surface-vitree  4.8e+0
       contiguite 0
       paroi-donnant-sur-exterieur
  nombre-de-cage-descalier 0
3 zone en-contact-avec-le-sol
1 zone-sous-la-zone 5

Pour-la-zone 4
    date-de-construction 1650
    volume  7.0e+3
    hauteur  3.0e+0
    nombre-detages 1
    nombre-parois-opaques-verticales 4
    surface-plancher  2.3e+3
    orientat-par-1 5
    statut 1
       paroi-n 1
       longueur-de-paroi  8.0e+1
       surface-opaque-verticale  2.4e+2
       surface-vitree  8.0e+0
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 2
       longueur-de-paroi  2.9e+1
       surface-opaque-verticale  8.7e+1
       surface-vitree  0.0e+0
       contiguite-sur-paroi 2 avec-la-zone 3
       contiguite-totale  8.7e+1
       paroi-donnant-sur-exterieur
       paroi-n 3
       longueur-de-paroi  8.0e+1
       surface-opaque-verticale  2.4e+2
       surface-vitree  8.0e+0
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 4
       longueur-de-paroi  2.9e+1
       surface-opaque-verticale  8.7e+1
       surface-vitree  0.0e+0
       contiguite 0
       paroi-donnant-sur-exterieur
  nombre-de-cage-descalier 0
6 zone enterree-ou-semi-enterree
1 zone-sous-la-zone 5

Pour-la-zone 5
    date-de-construction 1650
```

```
    volume  1.6e+4
    hauteur  6.0e+0
    nombre-detages 2
    nombre-parois-opaques-verticales 6
    surface-plancher  5.4e+3
    orientat-par-1 5
    statut 1

       paroi-n 1
       longueur-de-paroi  8.2e+1
       surface-opaque-verticale  4.9e+2
       surface-vitree  1.2e+2
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 2
       longueur-de-paroi  1.9e+1
       surface-opaque-verticale  1.1e+2
       surface-vitree  1.8e+1
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 3
       longueur-de-paroi  3.3e+1
       surface-opaque-verticale  2.0e+2
       surface-vitree  5.0e+1
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 4
       longueur-de-paroi  1.0e+1
       surface-opaque-verticale  6.0e+1
       surface-vitree  0.0e+0
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 5
       longueur-de-paroi  1.2e+2
       surface-opaque-verticale  6.9e+2
       surface-vitree  2.5e+1
       contiguite 0
       paroi-donnant-sur-exterieur
       paroi-n 6
       longueur-de-paroi  2.9e+1
       surface-opaque-verticale  1.7e+2
       surface-vitree  1.3e+2
       contiguite 0
       paroi-donnant-sur-exterieur
  nombre-de-cage-descalier 0
1 zone-sur-la-zone 4
2 zone combles-chauffes-vol  3.1e+3   ouvert  4.6e+1
```

Fichier résultat contenant les informations sur les coefficients d'assertion :

```
Objet : testemp.equipement.production.chaudiere-fioul ; etat-entretien =  mal  avec ca =  1
Objet : testemp.zone1.paroi-verticale1 ; val-orientation =  ese  avec ca =  1
Objet : testemp.zone1.paroi-verticale1.composition ; nombre-couches =  1  avec ca =  0.7
Objet : testemp.zone1.paroi-verticale1.paroi-vitree1 ; etancheite =  mauvaise  avec ca =  0.9
Objet : testemp.zone1.paroi-verticale2 ; val-orientation =  ese  avec ca =  1
Objet : testemp.zone1.paroi-verticale3 ; val-orientation =  ese  avec ca =  1
Objet : testemp.zone1.paroi-verticale4 ; val-orientation =  ese  avec ca =  1
Objet : testemp.zone1.porte1 ; calfeutrement =  mal-calfeutree  avec ca =  0.9

Objet : testemp.zone2.paroi-verticale1 ; val-orientation =  ese  avec ca =  1
Objet : testemp.zone2.paroi-verticale1.paroi-vitree1 ; etancheite =  mauvaise  avec ca =  0.9
```

```
Objet : testemp.zone2.paroi-verticale2 ; val-orientation = ese avec ca = 1
Objet : testemp.zone2.paroi-verticale4.composition ; nombre-couches = 1 avec ca = 0.7
Objet : testemp.zone2.paroi-verticale3 ; val-orientation = ese avec ca = 1
Objet : testemp.zone2.paroi-verticale4.composition ; nombre-couches = 1 avec ca = 0.7
Objet : testemp.zone2.paroi-verticale4 ; val-orientation = ese avec ca = 1
Objet : testemp.zone2.paroi-verticale4.composition ; nombre-couches = 1 avec ca = 0.7
Objet : testemp.zone2.porte1 ; calfeutrement = mal-calfeutree avec ca = 0.9
Objet : testemp.zone3.paroi-verticale1 ; val-orientation = ese avec ca = 1
Objet : testemp.zone3.paroi-verticale1.composition ; nombre-couches = 1 avec ca = 0.7
Objet : testemp.zone3.paroi-verticale1.paroi-vitree1 ; etancheite = mauvaise avec ca = 0.9
Objet : testemp.zone3.paroi-verticale2 ; val-orientation = ese avec ca = 1
Objet : testemp.zone3.paroi-verticale3 ; val-orientation = ese avec ca = 1
Objet : testemp.zone3.paroi-verticale4 ; val-orientation = ese avec ca = 1
Objet : testemp.zone3.paroi-verticale5 ; val-orientation = ese avec ca = 1
Objet : testemp.zone3.paroi-verticale6 ; val-orientation = ese avec ca = 1
Objet : testemp.zone3.paroi-verticale7 ; val-orientation = ese avec ca = 1
Objet : testemp.zone3.paroi-verticale8 ; val-orientation = ese avec ca = 1

Objet : testemp.zone3.porte1 ; calfeutrement = mal-calfeutree avec ca = 0.9
Objet : testemp.zone4.paroi-verticale1 ; val-orientation = ese avec ca = 1
Objet : testemp.zone4.paroi-verticale1.composition ; nombre-couches = 1 avec ca = 0.7
Objet : testemp.zone4.paroi-verticale1.paroi-vitree1 ; etancheite = mauvaise avec ca = 0.9
Objet : testemp.zone4.paroi-verticale2 ; val-orientation = ese avec ca = 1
Objet : testemp.zone4.paroi-verticale3 ; val-orientation = ese avec ca = 1
Objet : testemp.zone4.paroi-verticale4 ; val-orientation = ese avec ca = 1
Objet : testemp.zone4.porte1 ; calfeutrement = mal-calfeutree avec ca = 0.9
Objet : testemp.zone5.paroi-verticale1 ; val-orientation = ese avec ca = 1
Objet : testemp.zone5.paroi-verticale1.composition ; nombre-couches = 1 avec ca = 0.7
Objet : testemp.zone5.paroi-verticale1.paroi-vitree1 ; etancheite = mauvaise avec ca = 0.9
Objet : testemp.zone5.paroi-verticale2 ; val-orientation = ese avec ca = 1
Objet : testemp.zone5.paroi-verticale3 ; val-orientation = ese avec ca = 1
Objet : testemp.zone5.paroi-verticale4 ; val-orientation = ese avec ca = 1
Objet : testemp.zone5.paroi-verticale5 : val-orientation = ese avec ca = 1
Objet : testemp.zone5.paroi-verticale6 ; val-orientation = ese avec ca = 1
Objet : Testemp.zone5.porte1 ; calfeutrement = mal-calfeutree avec ca = 0.9
```

Bibliographie

[1] Abelson H. and Sussman G.J. with Julie Sussman. *Structure and Interpretation of Computer Programs.* M.I.T. Press, 1985.

[2] AFME and COSTIC. *Guide du Diagnostic Thermique.* Eyrolles Paris, 1987.

[3] Agha Gul. *Actors : A Model of Concurrent Computation in Distributed System.* serie : Artificial Intelligence. MIT Press, 1986.

[4] Atelier d'évaluation H2E 85. *Catalogue des Outils de Conception Thermique des Bâtiments.* PYC Edition, Paris, 1986.

[5] Bailly C et al. *Les Langages Orientés Objets : Concepts, Langages et Applications.* Cepadues-Edition Toulouse, 1987.

[6] Battini J.P., Brejon P., and Dubernet J.P. Elaboration d'un logiciel de diagnostic thermique des bâtiments tertiaires. (rapport intermédiaire), Convention tripartie FRME/ITHEF/ARMINES, 1986.

[7] Bernard. Les ratios et leur utilisation en pré-diagnostic. In *Séminaire International : Audit Energétique dans les Bâtiments Existants*, pages 205–214. AFME, PCA : Bâtiment Econome, 20-21 juin 1988.

[8] BIGRE N° 65. Putting Scheme to work, Juillet 1989.

[9] Borning, A.H. Classes versus prototypes in object-oriented languages. In *Proc. of ACM/IEEE Fall Joint Computer Conference*, pages 36–40, November 86.

[10] Bouchet J.A. Approche du diagnostic thermique par système expert. In *Séminaire International : Audit Energétique dans les Bâtiments Existants*, pages 357–364. AFME, PCA : Bâtiment Econome, 20-21 juin 1988.

[11] Bouchon B. and Desprès S. Propagation of uncertainties and inaccuracies in knowledge-based system. In *Lecturer Note in Computer Science 286 : Uncertainty.* Springer-Verlag, 1987.

[12] B. Bourges. Calcul des degrés-jours mensuels à température de base variable. *revue CVC*, 87 : N°5, mai 87.

[13] Brejon, P. Le projet LEB : un serveur télématique sur les logiciels à l'usage des professionnels du génie climatique. In *Séminaire spécialisé : Outils d'aide à la conception et gestion (OACG '89)*. AFME, Novembre 1989.

[14] Brejon P. *Les Logiciels d'Energétique des Bâtiments. Développement, Evaluation Technique, Illustrations.* Thèse de Docteur-Ingénieur Ecole des Mines de Paris, Octobre 1988.

[15] Brejon P., Lalement R., Le X.H., and Rialhe A. Projet MA'TH (rapport intermédiaire). Rapport de recherche, Convention avec le programme IN.PRO.BAT. du Plan Construction et Architecture contrat n^o 88 61 409 ; et avec l'Agence Française pour la Maîtrise de l'Energie contrat n^o 8 04 0116., juillet 1989.

[16] Brejon P., Lalement R., Le X.H., and Rialhe A. Projet MA'TH (rapport final). Rapport de recherche, Convention avec le programme IN.PRO.BAT. du Plan Construction et Architecture contrat n^o 88 61 409 ; et avec l'Agence Française pour la Maîtrise de l'Energie contrat n^o 8 04 0116., mars 1990.

[17] Brejon P and Marchio D. Incertitudes liées aux données et méthodes d'audit énergétique des bâtiments. In *Séminaire International : Audit Energétique dans les Bâtiments Existants*, pages 367–380. AFME, PCA : Bâtiment Econome, 20-21 juin 1988.

[18] Briot, J.P. Programming with explicit metaclasses in Smalltalk-80. In *Proc. of OOPSLA '89*, pages 419–432. SIGPLAN Notices Vol. 24 Number 10, October 89.

[19] Briot J.P. and Yonezawa A. Inheritence and synchronization in concurrent OOP. In *European Conference on object oriented programming (ECOOP'87)*, pages 35–43, Juin 1987.

[20] Cadiergues, R. et al. Calcul des puissances et des économies en chauffage discontinu. *PROCLIM*, E N° 4, Sep. 1977.

[21] CAO-MIP. Spécification du système Krèpis. Rapport intremédiaire de recherche, CAO-MIP, 1987 Toulouse.

[22] Cardelli, Luca. A semantics of multiple inheritance. *Information and Computation*, 76 : 138–164, 1988.

[23] Cardelli, Luca and Longo, Giuseppe. A semantic basis for Quest. In *Proc. of 1990 ACM Conference on Lisp and Functional Programming*, pages 30–43. 7, June 27-29, 1990.

[24] Centre d'Energétique. Outils d'audit energétique de bâtiment. Point de repère. Synthèse du séminaire international d'audit energétique du 20-21 juin 1988., Plan Construction et Architecture, 1989.

[25] CEP and Dialogic. Documents internes. non publié.

[26] Charniak E., Riesbeck C., McDermott D.V., and Meehan J.R. *Artificial Intelligence Programming.* Lawrence Erlbaum Associats New-Jersey, second edition, 1987.

[27] Clinger W. Semantics of Scheme. *BYTE*, 1988 : 220–227, February 1988.

[28] CNEME. Caractéristique d'une méthode de diagnostic thermique adaptée au logement individuel. non publié, 1985.

[29] Cointe, P. Metaclasses are first class : the ObjVlisp model. In *OOPSLA '87*, pages 156–167. ACM, 1987.

[30] Cointe Pierre. *Implémentation et Interprétation des Langages Orientés Objets : Application aux Langages Smalltalk, ObjVlisp et Forme.* Thèse de Doctorat d'Etat, Université de Paris VI, 1984.

[31] Cook, William and Palsberg, Jens. A denotational semantics of inheritance and its correctness. In *Proc. of OOPSLA '89*, pages 433–443. SIGPLAN Notices Vol. 24 Number 10, October 89.

[32] COSTIC. Coefficient des déperditions des parois anciennes. non publié, 1985.

[33] Croquelois, F. Anagram : une méthode pour aider à la maîstrise des consommations d'énergie. In *Séminaire International : Audit Energétique dans les Bâtiments Existants*, pages 183–203. AFME, PCA : Bâtiment Econome, 20-21 juin 1988.

[34] CSTB. *règles Th-G.* DTU cahier 2256, 1988.

[35] CSTB. *Coefficients K des parois des bâtiments anciens.* CSTB, cahier $n°1682$, déc 1980.

[36] CSTB. *Règles Th-BV, règles de calculs du coefficient de besoins de chauffage des logements.* CSTB, juillet 88.

[37] CSTB. *Règles Th-C, règles de calcul du coefficient de performance thermique globale des logements.* cahier du CSTB, juillet 88.

[38] CSTB and AFME. *Isolation Thermique de l'Habitat Existant.* Technique et Economie d'énergie. CSTB, 198x.

[39] Cyssau, René. Le diagnostic instrumenté. L'évaluation de l'état initial : un exemple d'utilisation de la méthodologie. In *Séminaire International : Audit Energétique dans les Bâtiments Existants*, pages 225–246. AFME, PCA : Bâtiment Econome, 20-21 juin 1988.

[40] Delahaye JP. *Outils Logiques pour l'IA.* Eyrolles Paris, 1987.

[41] DeMichiel, Linda G. and Gabriel, Richard P. The Common Lisp Object System : An overview. In *Proc. of ECOOP '87*, pages 201–220. AFCET, Juin 87.

[42] Dubernet J.P. Sensibilité des résultats d'un diagnostic thermique à l'incertitude sur les données. Technical report, ARMINE/COSTIC, mai 1988.

[43] Dubois D and Prade H. *Théories des Possibilités : Application à la Représentation des Connaissances en Informatique.* série : Technique Avancée en Informatique. Masson, Paris, 1985.

[44] Ducournau, Roland and Habib, Michel. La multiplicité de l'héritage dans les langages à objets. *Technique et Science Informatique*, 8 N° 1 : 41–62, 1989.

[45] Dybvig Kent. *The Scheme Programming Language.* Prentice-Hall, 1987.

[46] Edf. *Degrés-jours à base de température quelconque période hivernale, année entière.* EDF, avril 74.

[47] Farreny H. *Sytèmes Experts : Principe et Exemples.* Cépadues Editions Toulouse, 1985.

[48] Farreny H. and Ghallab M. *Eléments d'Intelligence Artificielle.* Collection Traité de Nouvelles Techniques serie I.A. Hermès Paris, 1987.

[49] Ferber A. *Mering : Un Langage d'Acteurs pour la Représentation des Connaissances.* Thèse de Doctorat d'Ingénieur, Universite de Paris VI, 1983.

[50] Fontanel C. Un outil neuf sur le diagnostic thermique dans le résidentiel et le tertiaire. *Revue Chauffage Ventillation Conditionnement*, 1983 : 10 : 5–10, oct. 1983.

[51] Giannesini et al. *Prolog.* InterEditions, Paris, 1985.

[52] Gilles, René, Brasselet, Jean-Pierre, and Rio, André. Diagnostic énergétique détaillé et instrumenté assisté par ordinateur (DEDIAO). In *Séminaire International : Audit Energétique dans les Bâtiments Existants*, pages 381–390. AFME, PCA : Bâtiment Econome, 20-21 juin 1988.

[53] Goldberg A and Robson D. *Smalltalk-80, The Language.* Addison-Wesley, 1989.

[54] Hamilton G. and Harrison A. Expert systems for the construction and building services industry. Technical report, BSRIA Technical Note TN7/86, 1986.

[55] Harris J. et al. A comparaison of multifamily retrofits in the U.S. and Europe : mesured results and policy implications. In *Séminaire International : Audit Energétique dans les Bâtiments Existants*, pages 135–146. AFME, PCA : Bâtiment Econome, 20-21 juin 1988.

[56] Haynes C. Logic continuation. *The Journal of Logic Programming*, 1987 : 4 : 157–176, 1986.

[57] Haynes Christopher T., Friedman Daniel P., and Wand, Mitchell. Obtaining coroutines with continuations. *Computer Language*, 11, N°3/4 : 143–153, 1987.

[58] Hewitt C. Viewing control structure as patterns of passing messages. *Artificial Intelligence*, 8 : 323–363, 1977.

[59] Hieb, Robert and Kent Dybvig, R. Continuations and concurrency. In *Second ACM SIGPLAN Symposium on Principle and Pratice of Parallel Programming (PPOPP)*, pages 128–136, March 14-16, 1990.

[60] Hofstadter, Douglas D. *Gödel, Escher, Bach : an Eternal Golden Braid.* Basic Books, New York, 1979.

[61] IIRIAM. Projet SET. Communication personnelle.

[62] IIRIAM, AFME, CETE, and COMETEC. SET : Système expert d'aide au diagnostic thermique. In *Communications des Journées des 17,18/12/86 CAO et Bâtiment : Etat et Pespectives.* Plan Construction, 1986.

[63] Kamin, Samuel. Inheritance in Smalltalk-80 : A denotational defintion. In *Proc. of Fifteenth Annuam ACM SIGACT-SIGPLAN Symposium on Pronciples of Programming Languages*, pages 80–87. ACM, January 88.

[64] Kaufmann A. *Introduction à la Théorie des Sous-ensembles Flous.* Masson, Paris, 1973.

[65] Kyburg H.E. Representing knowldge and evidence for decision. In *Lectures Note in Computer Science 286 : Uncertainty.* Springer-Verlag, 1987.

[66] Lauriere J.L. *IA, Résolution de Problèmes par l'Homme et la Machine.* Eyrolles Paris, 1987.

[67] LE, X.H. Acteurs et audit thermique de bâtiment. In *Séminaire spécialisé : Outils d'aide à la conception et gestion (OACG '89).* AFME, Novembre 1989.

[68] Lieberman H. Thinking about lots of things at once without getting confused — Parallelism on ACT1. AI memo 626, AI Lab. MIT, 1981.

[69] Lieberman H. Delegation and inheritance : Two mechanisms for sharing knowledge in object-oriented systems. In *Troisièmes journées LOO*, pages 78–89, Juin 1986.

[70] Lieberman H. Using prototypical objects to implement shared behavior in object oriented systems. In *Proc. of OOPSLA '86*, pages 214–223, Octobre 1986.

[71] Magrez P. and Smets P. Fuzzy modus ponens : A new model suitable for application in knowledge based systems. *International Journal of Intelligent Systems*, 4, 1989.

[72] Martin-Clouaire R. Efficient deduction in fuzzy logic. In *Lectures Note in Computer Science 286 : Uncertainty.* Springer-Verlag, 1987.

[73] Masini G. et al. *Les Langages à Objets, Langages de classes, langages de frames, langages d'acteurs.* serie : Informatique Intelligence Artificielle. InterEditions, 1989.

[74] Maurel C. *Définition Opérationnelle de la Sémantique du Langage Plasma : Réalisation d'un Interpréte Méta-circulaire de Référence.* Thèse de Doctorat de 3eme, Université Paul Sabatier Toulouse, Déc. 1982.

[75] Meyer B. Tools for new culture : Lessons from the design of the Eiffel libraries. *Communication of the ACM*, 33 : 68–88, September 1990.

[76] Milner, Robin. *Communication and Concurrency.* Computer Science. Prentice Hall, 1989.

[77] Ministère de l'Urbanisme et du Logement. *Guide pour l'amélioration des logements existants.* éd. Moniteur, 1982.

[78] Minsky, Marvin. *The Society of Mind.* Touchstone, 1988.

[79] Montalban M. *Prise en compte de specifications en ingénierie, Application aux systèmes experts de conception.* Thèse Université de Nice, Nov. 1987.

[80] Norman A. and Rees J. Object oriented programming in Scheme. In *1988 ACM Conference on Lisp and Functional Programming*, pages 277–288, July 1988.

[81] Oléron, Pierre. *l'Intelligence de l'Homme.* Armand Colin, Paris, 1989.

[82] Piaget, Jean. *La Psychologie de l'Intelligence.* Armand Colin, Paris, 1967.

[83] Prade H. A synthetic view of approximate reasoning techniques. In *Proc 8th I.J.C.A.I.*, 1983.

[84] Rechenmann, François and Vignard, Philippe. Crika : Quand les règles rencontrent les schémas. Rapport de recherche n° 468, I.N.R.I.A., Décembre 1985.

[85] Rees J.A. and Clinger W. eds. The revised^3 report on algorithmic language Scheme. Technical report, Sigplan Notices 21,12, December 1986.

[86] Rialhe A. *Le Traitement des Données Partielles en Audit Energétique de Bâtiments.* Thèse de l'Ecole des Mines de Paris, Décembre 1991.

[87] Rivemale C. *Sémantique Dénotationnelle du Langage Plasma : Définition et Implémentation.* Thèse de Doctorat de 3eme, Université Paul Sabatier Toulouse, Déc. 1982.

[88] SGAO. Projet de recherche sur le couplage de Keops avec outils d'évaluation thermique et acoustique. Rapport final, Convention AFME-Plan Construction et Architecture, 1987 Paris.

[89] Shafer Glenn. *A Mathematical Theory of Evidence.* Princeton University Press, 1976.

[90] Shortliffe E.H. and Buchanan B.G. A model of inexact reasoning in medicine. *Mathmatical Biosciences*, 23 : 351–379, 1975.

[91] Springer G. and Friedman D.P. *Scheme and the Art of Programming.* The MIT Press, 1989.

[92] Steels JR, G.L. *Common Lisp the Language.* Digital Press, second edition, 1990.

[93] Stein, Lynn Andrea. Delegation is inheritance. In *Proc. of OOPSLA 87*, pages 138–146. ACM Sigplan oct. 87, Oct. 87.

[94] Stroustrup B. *The C++ Programming Language.* Addison-Wesley, 1988.

[95] Stroustrup B. What is "object-oriented programming" ? In *European Conference on object oriented programming (ECOOP'87)*, pages 51–70, Juin 1987.

[96] Subrahamian V.S. On the semantics of quantitative logic programs. *IEEE*, 1987 : 9, 1987.

[97] Theriault D. G. Issues in the design and implementation of ACT2. Technical report, TR N° 728. M.I.T. Artificial Intelligence Laboratory, June 1983.

[98] Tlili A. *Structure des Données de la Conception d'un Bâtiment pour une Utilisation Informatique.* Thèse de Docteur-Ingénieur ENPC Paris, Déc 1986.

[99] Turner R. *Logique pour l'Intelligence Artificielle.* Masson, 1986.

[100] Ungar, David and Smith, Randall B. Self : The power of simplicity. In *Proc. of OOPSLA 87*, pages 227–242. ACM Sigplan oct. 87, Oct. 87.

[101] van Emden M.H. Quantitative deduction and its fixpoint theory. *The Journal of Logic Programming*, 1986 : 1 : 37–53, 1986.

[102] Varela, Francisco J. *Connaître les Sciences Cognitives, tendance et perspectives.* Seuil, Paris, 1989.

[103] Winston P.H. *Artificial Intelligence.* Addison-Wesley, second edition, 1984.

[104] Yonezawa, A., editor. *ABCL, an Object-Oriented Concurrent System.* Computer Systems Series. The MIT Press, 1990.

[105] Zreik K. *Prise en Compte de Performance Acoustique en Phase d'APS de Bâtiment dans un Contexte de CAO.* Thèse de Docteur-Ingénieur ENPC Paris, Nov. 1986.