Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
UNIVERSITÉ DE NAMUR
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Utilisation d'OBLOG dans le développement transformationnel d'un système d'information concurrent

Pellez, Pierre

Award date:
1993

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Facultés Universitaires Notre-Dame de la Paix à Namur
Institut d'Informatique

UTILISATION D'OBLOG DANS LE DÉVELOPPEMENT TRANSFORMATIONNEL D'UN SYSTÈME D'INFORMATION CONCURRENT

Pierre PELLEZ

Promoteur : Eric Dubois

Mémoire présenté en vue de l'obtention
du diplôme de Licencié et Maître en Informatique

Année académique 1992-1993

Facultés Universitaires Notre-Dame de la Paix
Institut d'informatique
Rue Grandgagnage 21, B-5000 Namur
Tél. 081/72.49.83
Fax 081/72.49.67

UTILISATION D'OBLOG DANS LE DÉVELOPPEMENT TRANSFORMATIONNEL D'UN SYSTÈME D'INFORMATION CONCURRENT

Pierre Pellez

RÉSUMÉ • Le modèle de cycle de vie appelé approche transformationnelle a été principalement étudié et appliqué dans le cadre du développement de petits logiciels mono-utilisateur. C'est pourquoi, tout au long de ce mémoire, nous avons tenté de découvrir et d'appliquer les principes sous-jacents à cette approche sur un cas de taille réelle et présentant une dimension concurrente et distribuée : le cas de l'hôpital MortSubite.
Adoptant une découpe originale, la conception du logiciel est scindée en deux parties : la conception statique et la conception dynamique. L'aspect distribué et concurrent est "oublié" dans un premier temps en posant l'hypothèse d'un système mono-utilisateur (conception statique) puis il est réintroduit afin de fournir une solution complète (conception dynamique).
Pour réaliser la phase de conception, nous avons fait appel au langage OBLOG. Ce langage de haut niveau a été choisi en raison de sa puissance d'expression et de modélisation et de son caractère exécutable.

ABSTRACT • The life-cycle model called transformational approach has been studied and applied mainly for the development of small single-user softwares. This is the reason why all along this thesis we tried to discover and apply principles that underlie this approach on a real-size, distributed and concurrent case study called Hospital MortSubite.
The software design is divided in two parts according to an original position : the static design and the dynamic design. First the distributed and concurrent aspect is "forgotten" by assuming a single-user system (static design). Then it is re-injected in order to give a complete solution to the system requirements (dynamic design).
To carry out the software design we used the OBLOG language. This high-level language has been chosen for its expressiveness, its modelization power and its executability.

Mémoire de Licence et Maîtrise en Informatique
Septembre 1993
Promoteur : Eric Dubois

Je tiens à remercier vivement mon promoteur, le professeur Eric Dubois,
pour ses conseils, sa disponibilité et sa patience.

Merci à Naji Habra pour ses remarques constructives ainsi que pour l'aide
qu'il m'a apportée dans le développement de l'étude de cas.

Merci aussi à Jean-Marc Zeippen. C'est au cours d'un stage
au Portugal que j'ai appris OBLOG.
Il est pour beaucoup dans la réussite de ce stage.

Enfin, merci à tous celles et ceux qui m'ont aidé et soutenu
tout au long de de l'élaboration de ce mémoire.

TABLE DES MATIÈRES

INTRODUCTION **2**
1. Préambule 2
2. Etude de cas 3
3. L'approche transformationnelle 3
4. OBLOG 6
5. Plan de travail 7

ETUDE DES BESOINS **9**
I. Introduction 9
II. Etude d'opportunité 9
1. Cadre général 9
2. Organisation générale 9
3. Accueil 10
4. Hospitalisation 11
5. Facturation 11
6. Statistiques 12
7. Politique d'allocation 12
III. Modèle de structuration des traitements 13
1. Présentation du modèle 13
2. Le cas de l'hôpital 16
IV. Modèle de structuration des données 19
1. Introduction 19
2. Description des entités et des associations 19

SPÉCIFICATIONS FONCTIONNELLES **28**
I. Objectifs des spécifications 28
II. Construction des spécifications 29
III. Fonctions 32
1. Description des fonctions 32
2. Remarques 44
IV. Schéma conceptuel 45
1. Schéma Entités-associations 45
2. Contraintes 46

L'APPROCHE OBLOG **48**
I. Introduction 48
1. Le langage 48
2. L'outil 49
3. La méthode 49
II. Le langage OBLOGkernel 50
1. Objets, classes d'objets et types de donnée 50
2. Structure d'un objet 52
3. Comportement d'un objet 53

4. Le langage d'expression d'OBLOG-kernel 56
5. Types d'objet 56
III. Le langage OBLOG-light 58
1. Classes d'objets, métaclasses et types de donnée 58
2. Structure d'un objet 59
3. Comportement d'un objet 62
4. Le langage d'expression d'OBLOGlight 62
5. Types d'objets 63
6. Relations entre objets (classes d'objets) 64
IV. L'outil 67

CONCEPTION STATIQUE 71
I. Introduction 71
II. Objectifs de la conception statique 71
1. Architecture logique 72
1.1. Structuration 72
1.2. Abstraction 72
1.3. Modularisation 73
1.4. Place mémoire vs complexité des traitements 73
2. Spécification des modules 76
III. OBLOGstatic 77
1. Modifications 77
2. Traduction 80
3. Avantages d'OBLOGstatic 80
IV. Le développement 81
1. Le cadre 81
2. Les spécifications 81
2.1. Architecture globale 82
2.2. Les niveaux en détail 84

CONCEPTION DYNAMIQUE 96
I. Introduction 96
II. Objectifs de la conception dynamique 96
III. La concurrence en OBLOG 98
IV. Développement du cas 100
1. Transformations 100
1.1. Parallélisme 100
1.2. Le mécanisme de stored action 101
2. Evolution 103
2.1. Protections 104
2.2. "Cardinalité" des objets 106
2.3. Gestion de la concurrence 107
3. Optimisations 110
3.1. Allocation d'un lit 110
3.2. Allocation d'une prescription 115

CONCLUSION 117

BIBLIOGRAPHIE 121

Chapitre 1
Introduction

INTRODUCTION

1. PRÉAMBULE

"Le Génie Logiciel a pour objectif le développement et la maintenance d'un logiciel de "qualité", étant donné les contraintes de coûts et de délais, pour des applications de natures différentes et développées dans des environnements de natures différentes. Le Génie Logiciel repose sur l'utilisation maîtrisée d'un ensemble cohérent de principes, d'outils et de techniques." [Fai85].

Pour atteindre cette "qualité", le Génie Logiciel propose au développeur des démarches basées sur des modèles, des méthodes et des outils. Toute méthode de développement et de maintenance d'un logiciel repose sur l'identification d'un certain nombre d'activités réalisées selon un certain séquencement temporel.

Au travers de toute démarche, on peut distinguer les activités suivantes :
- l'*analyse des besoins* qui a pour objectif de définir le problème informatique aussi précisément que possible,
- la *conception du logiciel* qui a pour objectif de proposer une solution abstraite (c'est-à-dire indépendante de moyens informatiques particuliers) au problème posé,
- l'*implémentation du logiciel* qui a pour objectif de concrétiser la solution proposée à l'étape précédente dans un environnement matériel donné.

Différents agencements temporels de ces activités conduisent à la définition de différentes formes du cycle de vie d'un logiciel. La plus populaire est celle du *waterfall model* [Boe76] où l'on passe à l'étape suivante lorsque la précédente est entièrement terminée.
Cependant ce modèle est maintenant controversé et c'est pourquoi des cycles de vies alternatifs ont été proposés.
De ceux-ci, nous n'en retiendrons que deux :
- le cycle de vie basé sur le *prototypage* [Boe84] dans lequel l'adéquation du logiciel aux besoins du client est évaluée grâce à la construction d'un prototype dès la phase d'analyse,
- le cycle de vie du type *transformationnel* [Par83] dans lequel, à l'issue de chaque activité, une description *formelle* (basée sur des langages mathématiques et/ou logiques) du logiciel désiré est réalisée. C'est ce caractère formel qui fait de ce cycle de vie un modèle alternatif.

C'est ce dernier modèle qui va nous intéresser. C'est en effet dans le cadre de ce cycle de vie que nous allons résoudre le problème posé par notre étude de cas.

2. ETUDE DE CAS

L'approche transformationnelle a été principalement étudiée et appliquée dans le cadre du développement de petits logiciels mono-utilisateur. Cela a été moins le cas en ce qui concerne le développement de systèmes d'information distribués et concurrents. Et c'est pourquoi, à l'aide des principes de cette approche transformationnelle, nous allons tenter de mener une étude de cas dans lequel la dimension concurrente est présente.

Nous avons choisi le cas de l'*hôpital MortSubite* qui a été développé en grande partie par N. Habra [Hab93] et a servi de base à un travail pratique réalisé par les étudiants de 2ème licence, dans le cadre du cours de Méthodologie de Développement de Logiciels dispensé par le professeur E. Dubois. Ce cas traitant du développement d'un système d'information centralisé, il nous a alors été fort facile de l'étendre afin de lui ajouter les dimensions concurrente et distribuée qui lui manquaient.

L'hôpital MortSubite est un établissement hospitalier comme les autres. On trouve des services de soins, des médecins et des infirmières, des lits et des patients. Ces patients effectuent des transferts d'un lit vers un autre et ils subissent des prestations de soins prescrites par des médecins. Par ailleurs, certains patients, arrivés au terme de leur hospitalisation, quittent l'établissement et d'autres y entrent afin de se faire soigner.

Toutes ces transactions sont réalisées par les différents services concernés comme, par exemple, le service d'accueil, divers services de soins ou services techniques. Côté administratif, on y trouve aussi un service de comptabilité ainsi que la direction qui dirige l'ensemble des opérations.

Afin de bien gérer leur établissement, les responsables souhaitent qu'une série d'activités soient supportée par un système d'information. Chaque service disposera donc d'un ou plusieurs terminaux afin d'y encoder les opérations qu'il effectue ou plus simplement pour obtenir divers renseignements. Ces opérations seront effectuées de manière concurrente. Ainsi, un service pourra effectuer une prescription de soin alors qu'un deuxième réalise une admission et que deux autres "rivalisent" pour obtenir un lit à leur patient dans le cadre d'un transfert.

3. L'APPROCHE TRANSFORMATIONNELLE

Comme on peut le constater, le système d'information que nous allons développer peut être situé dans la catégorie "système d'information de gestion classique". Outre la dimension concurrente et distribuée, ce système d'information est, en effet, fort classique : il ne s'agit pas de gérer des opérations en temps réel, les souhaits du client sont clairs et complets et la "complexité" des traitements à réaliser par le système d'information est plus importante que l'aspect interface homme-machine[1]. C'est pourquoi nous avons préféré adopter l'approche transformationnelle plutôt que le prototypage.

---

[1] Ce qui ne signifie pas que l'interface doit être négligée...

Cette approche est caractérisée par cinq grandes phases :
- l'étude des besoins,
- les spécifications fonctionnelles
- la conception statique
- la conception dynamique
- l'implémentation

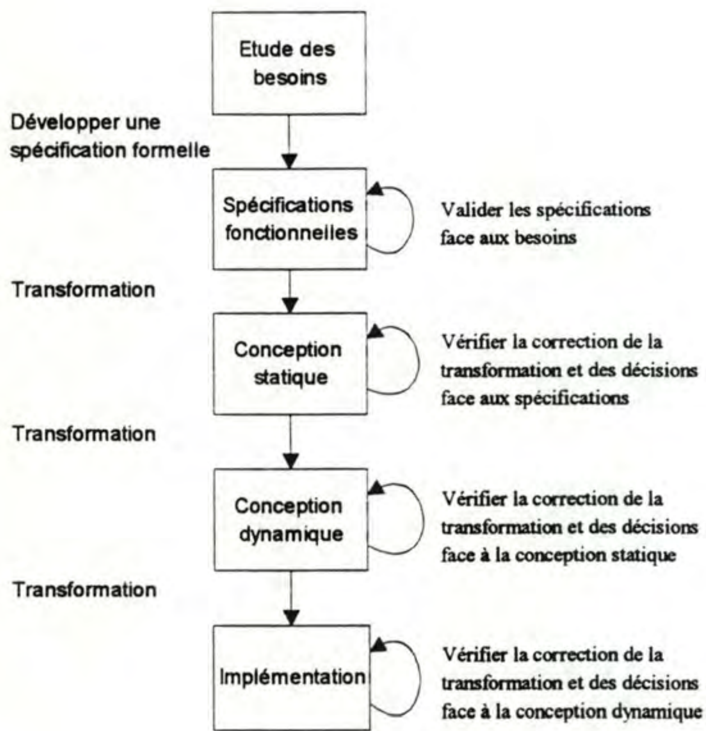


Figure 1 - L'approche transformationnelle

Cette division tend à grouper, dans une phase, un ensemble de décisions homogènes : chaque phase possède un objectif et seules les décisions relatives à cet objectif doivent y être prises.

En plus de l'activité qui consiste à produire la spécification formelle du produit, chaque phase implique d'autres activités telles que la validation, face au produit de la phase précédente, des décisions prises et des transformations effectuées sur la spécification.

Etude des besoins

Cette phase a pour objectif de définir les souhaits du client de manière semi-formelle, complète et structurée. Cette phase capitale conditionne l'adéquation du futur logiciel aux besoins du client.

Spécifications fonctionnelles

Partant d'un document semi-formel exprimant les objectifs du système d'information, il s'agit d'identifier et de spécifier précisément et rigoureusement (éventuellement, de manière formelle) le sous-système informatique (les fonctionnalités) qui fera l'objet du logiciel et ce, en faisant abstraction de l'organisation. Il s'agit ici de poser le problème aussi précisément que possible et non pas de lui trouver une solution.

Les fonctionnalités seront spécifiées en termes de relations entre les entrées et les sorties ainsi que de modifications sur les états (données persistantes).

Conception statique

Sur base de la description du problème (*What*), il faut ensuite se diriger vers une solution (*How*). Il s'agira donc de "raffiner" les spécifications fonctionnelles : des choix seront posés au niveau de la représentation des données persistantes, au niveau de l'interface homme-machine ou de la gestion des cas d'erreurs. Les spécifications qui, auparavant, étaient purement descriptives sont enchaînées afin de les rendre "exécutables".

Le volume des spécifications résultantes étant beaucoup plus important, il faut l'organiser dans une *architecture logique*. Trois principes méthodologiques sous-tendent cette organisation :

- la *structuration* : ce principe prône l'introduction de fonctions intermédiaires. Une fonctionnalité sera éclatée en plusieurs fonctions de granularité plus fine. Le même processus sera répété jusqu'à obtenir un niveau de granularité satisfaisant.
- l'*abstraction* : les fonctions obtenues sont regroupées en différents niveaux en fonction de leur "nature". Trois niveaux seront identifiés :
  - le *niveau 5* qui contient les fonctions dérivées des fonctionnalités,
  - le *niveau 4* qui contient les fonctions s'occupant de l'interface homme-machine et
  - le *niveau 3* qui contient les fonctions concernées par la gestion des informations persistantes

  Cette organisation est une *hiérarchie descendante* : un niveau offre des services aux niveaux supérieurs.
- la *modularisation* : au sein d'un niveau, les fonctions seront regroupées en modules et cela selon deux critères possibles :
  - orienté-fonction : un module regroupe des fonctions qui font "la même chose" ou
  - orienté-objet : un module regroupe des fonctions qui travaillent sur la même structure de données

La conception est dite *statique* car nous posons les hypothèses d'un système mono-utilisateur et de l'instantanéité d'une fonction. Ces hypothèses permettent d'éviter des problèmes liés à l'aspect distribué et concurrent du logiciel et requérant l'introduction de concepts tels que la synchronisation par exemple.

Le résultat de cette phase est une architecture globale composée de plusieurs modules organisés en niveaux. Ces niveaux fournissent des services à d'autres modules qui les *utilisent*.

Conception dynamique

Cette phase de conception détaillée consiste en un raffinement de l'architecture statique dans laquelle l'aspect distribué et concurrent est pris en compte. Nous devons, en effet, lever les hypothèses que nous avions posées afin que, au terme de cette étape de conception, nous disposions d'une solution abstraite au problème posé par le cahier des charges.

Dans l'architecture, les modules deviennent des *processus concurrents* et il faut alors imaginer des stratégies pour gérer cette concurrence. Nous verrons alors apparaître l'un ou l'autre module supplémentaire dont l'objectif sera de gérer l'interaction entre objets et l'accès "ordonné" aux ressources partagées par ces processus.

Le choix de séparer la conception statique de la conception dynamique est principalement motivé par deux observations :
- la description d'aspects temporels d'un logiciel semble être plus proche de la phase d'implémentation que d'une spécification fonctionnelle abstraite
- on observe que, dans la catégorie de systèmes d'information qui nous concerne, la concurrence n'est pas le facteur-clé qui conditionne les grandes décisions d'architecture, par opposition aux systèmes temps-réels par exemple.

Implémentation

Cette dernière phase a pour objectif de fournir, à partir d'une solution *abstraite* exprimée dans un langage de haut niveau, une solution *concrète*, exécutable sur une machine réelle. Il s'agit alors de faire le choix d'un ou plusieurs langages d'implémentation et d'adapter les spécifications issues de la phase de conception dynamique à ce(s) langage(s).

Cependant, avec l'apparition de générateurs de code, il devient possible d'éviter cette phase en laissant au générateur le soin de transformer les spécifications en un code exécutable.

4. OBLOG

Pour réaliser les deux phases de conception, nous avons fait appel au langage OBLOG (*OBJect LOGic*), langage développé par la société ESDI S.A. au Portugal.

OBLOG est un langage formel, orienté-objet, qui permet de modéliser un système d'information comme une collection d'objets concurrents qui interagissent entre eux. Outre un langage, l'approche OBLOG comprend également un outil (en fait, un atelier logiciel complet) et une méthodologie.

Ce langage, particulièrement adapté pour la conception de logiciels, possède aussi une sémantique opérationnelle : à partir de spécifications OBLOG, il est possible de générer du code exécutable[2]. C'est une des raisons pour lesquelles nous avons choisi d'adopter ce langage.

---

[2] Ce qui explique pourquoi nous ne traiterons pas la phase d'implémentation du logiciel.

De plus, l'objectif de ce mémoire étant de traiter un cas comportant une dimension concurrente, ce langage nous semblait tout à fait adapté. Par ailleurs, cet ouvrage espère contribuer à l'évaluation constructive des qualités ou défauts de l'approche OBLOG. Ce langage, fort prometteur, est en effet loin d'avoir acquis ses lettres de noblesse.

Concrètement, nous serons amenés à modifier la sémantique du langage afin qu'il puisse satisfaire aux exigences de la phase de conception statique. Cette phase posait l'hypothèse de fonctions mathématiques instantanées, ce qui n'est pas le cas en OBLOG.
Nous développerons alors l'architecture statique et nous spécifierons les fonctions de notre logiciel à l'aide de ce langage.
Nous entamerons alors la conception dynamique et nous verrons comment, à l'aide d'un concept OBLOG, nous pouvons assurer la cohérence des données partagées. Nous verrons aussi un exemple de module dont l'objectif est de gérer la concurrence.

5. PLAN DE TRAVAIL

La découpe en chapitre de ce mémoire suit tout naturellement les étapes de l'approche transformationnelle. Ainsi, le chapitre 2 sera consacré à l'étude des besoins. Le chapitre 3 traitera des spécifications fonctionnelles.
Le chapitre 4 présentera en détail le langage OBLOG et plus généralement, ce que nous avons appelé l'approche OBLOG. Les chapitres 5 et 6 seront consacrés respectivement à la conception statique et dynamique.
Le lecteur trouvera en annexe la spécification complète du cas de l'hôpital MortSubite.

Chapitre 2
Etude des besoins

ETUDE DES BESOINS

I. INTRODUCTION

Dans ce chapitre, nous allons exposer l'étude des besoins relative à l'hôpital MortSubite. Ce cas, initialement défini par N. Habra [Hab93], a été entendu afin d'y introduire une dimension concurrente. Nous allons d'abord présenter l'étude d'opportunité dans laquelle nous décrirons le fonctionnement général de l'hôpital (section II). Nous présenterons ensuite, à l'aide de deux modèles, les traitements que le système d'information doit accomplir et les données présentes dans le système d'information (section III et IV).

II. ETUDE D'OPPORTUNITÉ

1. CADRE GÉNÉRAL

Une société de service informatique veut développer un logiciel de gestion hospitalière répondant aux besoins de l'hôpital MortSubite.

La société envisage plusieurs autres contrats dans le domaine de la gestion hospitalière; des négociations sont déjà engagées avec d'autres hôpitaux de tailles et de besoins différents de ceux de l'hôpital MortSubite.

Des qualités essentielles du logiciel à développer sont donc:
- généralité : la plus grande indépendance possible par rapport aux données particulières de MortSubite.
- flexibilité : adaptation facile à des besoins spécifiques d'un autre hôpital.
- extensibilité : possibilité d'automatisation de nouvelles fonctions en relation avec celles qui sont réalisées, sans devoir changer ces dernières.

2. ORGANISATION GÉNÉRALE

L'hôpital MortSubite est subdivisé en 4 unités qui sont : les services de soins, les services administratifs, le service des urgences et les services techniques comme le montre le schéma suivant :

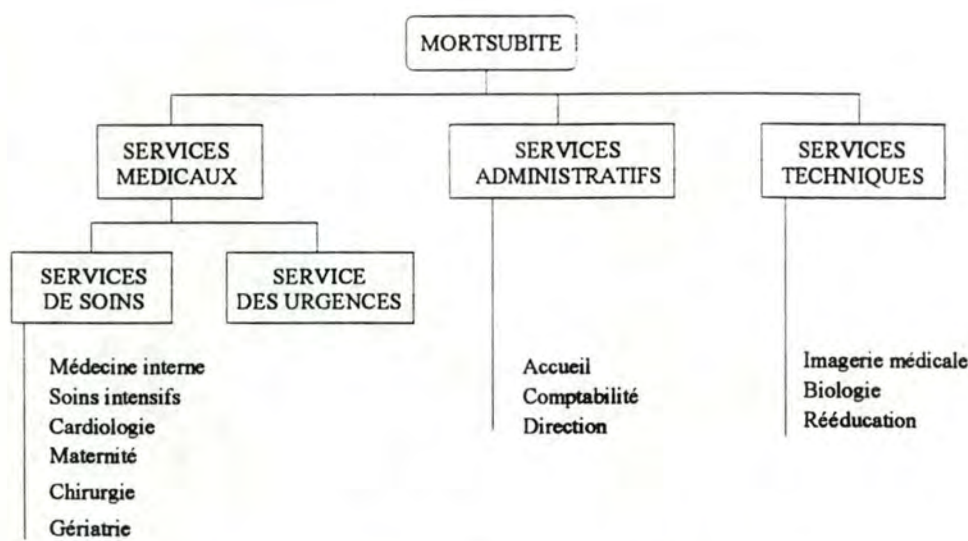


Figure 1 - Organisation générale de MortSubite

L'hôpital comprend 200 lits d'hospitalisation partagés entre les 7 services médicaux (6 services de soins et 1 service d'urgences). Le nombre de lits dans chaque service médical varie entre 8 et 24. Chaque service médical contient un certain nombre de chambres de différents types. Ainsi nous avons les types suivants:

- chambre à quatre lits,
- chambre à deux lits,
- chambre particulière.

Nous considérons que, pour un hôpital donné, la configuration des chambres et des lits est fixe. Dans chaque service médical un poste d'infirmières assure les soins quotidiens.

Le personnel de l'hôpital travaille en "feux continus" : trois "pauses" de 8 heures (6h-14h, 14h-22h, 22h-6h).

3. ACCUEIL

L'admission passe par le service d'accueil qui reçoit les informations nécessaires concernant le malade. Ce service attribue au patient un lit dans le service de soins correspondant. Le type de la chambre (1,2 ou 4 lits) correspondra au souhait du patient dans la mesure du possible. Il n'y a donc refus d'admission que dans le cas où le service demandé est complètement occupé.

Cependant, dans des cas d'urgence, l'admission se fera en partie au service des urgences où l'on demandera les informations minimales et elle sera complétée au service d'accueil lorsque le patient quittera les urgences pour entrer dans un service de soins ou lorsqu'il sortira directement de l'hôpital.

4. HOSPITALISATION

4.1. MOUVEMENTS INTERNES ET EXTERNES

Pendant son hospitalisation, le malade peut changer de service médical et/ou de lit une ou plusieurs fois (transfert en chirurgie, retour au service, changement de chambre dans le même service, etc). Il n'y a en aucun cas un transfert en direction des urgences.

Un malade peut effectuer plusieurs mouvements sur une même journée.

4.2. PRESTATIONS

Pendant son hospitalisation, le malade reçoit un certain nombre de soins et un certain nombre de médicaments [Prestations internes].
De même, il peut être envoyé une ou plusieurs fois aux services techniques (une heure de rééducation par jour, un scanner, etc) [Prestations externes].

Ces prestations font toujours l'objet d'une prescription d'un des médecins de l'hôpital.
Les infirmières du service concerné (médical ou technique) assurent le suivi de ces prescriptions.

Une prestation peut être exécutée (par une unité exécutrice) au plus un certain nombre de fois par pause. Par exemple, un scanner accepte au plus 32 patients par pause, une infirmière peut faire jusqu'à 80 piqûres par pause, une heure de rééducation "accepte" jusqu'à 8 patients par pause.

Afin d'aider les infirmières à planifier leur travail, on souhaite disposer, en début de pause, d'une liste des prestations à effectuer pour la pause en cours. Ainsi, pour les services médicaux, on leur fournira une liste des prestations internes et externes concernant leurs patients tandis que les services techniques disposeront d'une liste des prestations externes concernant l'ensemble des patients qu'ils vont recevoir.

Après une hospitalisation d'une durée quelconque, le malade quitte l'hôpital. Pour simplifier les choses, nous supposerons qu'un malade peut sortir n'importe quel jour de la semaine avec accord du médecin.

5. FACTURATION

La comptabilité facture toutes les prestations ainsi que tous les jours d'hospitalisation. Le prix d'une prestation dépend de son type, tandis que le prix d'une journée d'hospitalisation dépend lui du type de la chambre. Pour simplifier les choses, le type sera celui de la chambre que le patient occupe à 16h. Rappelons, en effet, qu'un patient peut effectuer plusieurs mouvements au cours d'une journée.

Pour chaque élément facturé (journée d'hospitalisation ou prestation) une partie est supportée par l'organisme assureur, et le reste par le patient.

Le service de comptabilité prépare mensuellement les factures concernant les malades sortant durant le mois écoulé et envoie ces factures aux patients et aux organismes assureurs. On ne facture donc que les hospitalisations terminées.

L'hôpital dispose de ses propres tarifs qui donnent :
- le prix plein unitaire pour chaque type de chambre,
- le prix plein unitaire pour chaque type de prestation,

où prix plein = montant à payer par le patient
+ montant supporté par l'organisme assureur.

On dispose aussi, et pour chaque organisme assureur connu, de l'adresse de cet organisme, ainsi que des prix unitaires remboursés par cet organisme pour chaque type de chambre ainsi que pour chaque type de prestation. Tous les tarifs ainsi que les parties supportées par les organismes assureurs peuvent changer.

6. STATISTIQUES

La direction de l'hôpital souhaite disposer de certaines statistiques comme :
- le taux d'occupation des lits de l'hôpital,
- le taux d'occupation des lits d'un service donné,
- le taux d'occupation des lits d'un type de chambre donné.

Elle souhaite aussi pouvoir calculer périodiquement les statistiques suivantes:
- le taux d'occupation de tout l'hôpital réparti sur les mois de l'année
- les taux d'occupation par service et par catégorie répartis sur les mois de l'année

Afin de simplifier le calcul des statistiques périodiques, on considérera l'état d'occupation de l'hôpital à une certaine date comme étant l'état d'occupation de l'hôpital à cette date et à une heure fixée.

7. POLITIQUE D'ALLOCATION

La direction demande aussi d'envisager la possibilité de traiter les demandes des services en parallèle. Afin de gérer la concurrence qui en résultera, elle propose les règles suivantes :

- les services ont une priorité de traitement :
  - "cas spéciaux" 0
  - urgences..........................1
  - soins intensifs..................2
  - cardiologie.......................3
  - chirurgie..........................3
  - médecine interne 3
  - maternité..........................3
  - gériatrie 3

  les "cas spéciaux" étant des demandes ponctuelles qui ne peuvent souffrir de retard et qu'il convient donc d'exécuter immédiatement.

- pour l'attribution des lits, on tentera de maximiser les demandes :
  - en tenant compte des priorités. La priorité sera celle du service demandeur dans le cas d'un transfert ou celle du récepteur dans le cas d'une admission
  - sans se limiter à l'examen des demandes sur un seul lit mais en étendant aux "précédents" (sur plusieurs lits)
  - une demande de priorité 0 doit être accordée impérativement

- pour la gestion des prestations, on tentera d'atteindre le "plein emploi" des ressources :
  - en tenant compte des priorités
  - en tenant compte de la capacité disponible pour cette prestation : les demandes refusées ou "éjectées" (accordées précédemment mais cédant leurs places) seront reportées à la pause suivante avec une plus grande chance d'être accordées
  - seules les demandes de priorité 0 sont autorisées pour la pause en cours (p). Elles seront exécutées sans attendre mais sans éjecter aucune demande prévue pour cette pause : elle seront "casées" parmi les demandes prévues
  - la gestion du planning est donc à faire sur les pauses p+1 et suivantes

III. MODÈLE DE STRUCTURATION DES TRAITEMENTS

1. PRÉSENTATION DU MODÈLE

La présentation du modèle de structuration des traitements est largement inspirée de [Bod89] pp. 50-63

1.1. DÉFINITIONS ET OBJECTIFS

Bodart et Pigneur [Bod89] proposent, dans le cadre de l'analyse fonctionnelle, un *modèle de structuration des traitements* qui doit permettre de représenter les fonctionnalités d'un système d'information (S.I.) à différents niveaux de détails. Ce modèle comprend un ensemble de concepts et de règles destinés à structurer un S.I. en une *hiérarchie d'agrégats fonctionnels* de traitements. Un *agrégat* comprend un ensemble de règles à suivre pour réaliser une fonctionnalité du S.I. A chaque agrégat seront associés la définition et l'objectif de cette fonctionnalité.

Ce modèle fournit des critères généraux pour procéder à la structuration d'un projet par *raffinements successifs* (structuration *arborescente*). Ces critères généraux sont complétés par des critères spécifiques attachés à une *nomenclature standardisée* de structuration d'un système d'information.

Bien que la structuration par raffinements successifs soit commune à de nombreuses méthodes d'analyse, l'originalité de la méthode réside dans l'existence de *niveaux prédéfinis* au sein d'une hiérarchie générale.

1.2. STRUCTURATION ARBORESCENTE DES TRAITEMENTS

Le modèle de structuration des traitements est basé sur le principe de décomposition arborescente d'un traitement : un traitement de niveau intermédiaire i provient de la décomposition d'un et d'un seul traitement de niveau i-1 et se découpe en n traitements de niveau i+1. La structuration arborescente établit une relation de *partition* entre traitements : tout traitement *fait partie* d'un traitement de niveau supérieur.

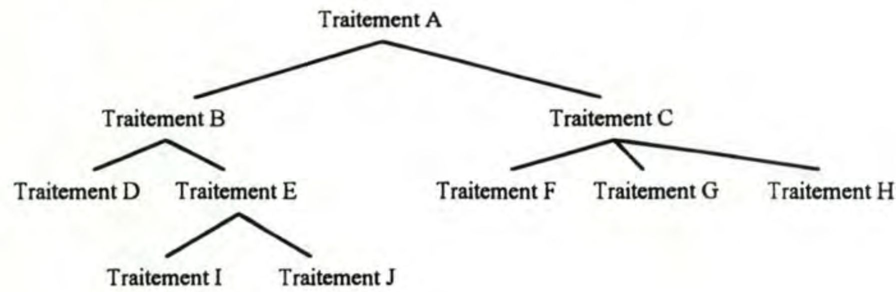


Figure 2 - Structuration arborescente et relation de partition

Remarquons qu'il ne faut pas confondre la relation *fait partie de* et la relation *utilise*[1]. En effet, cette dernière traduit un aspect de mise en oeuvre du S.I. et elle permet d'éliminer une partie de la redondance que peut créer la relation *fait partie de*. Ainsi, si l'on juge que les règles du traitement D et du traitement H sont voisines et pourraient, de ce fait, être réalisées par un même traitement K, alors on pourra dire que les traitements B et C *utilisent* le traitement K.

Nous reviendrons sur la relation *utilise* au chapitre traitant de la conception statique.

1.3. NOMENCLATURE STANDARDISÉE

Dans une décomposition arborescente, on peut distinguer des niveaux privilégiés, compte tenu de leur signification particulière. La figure 3 les représente.

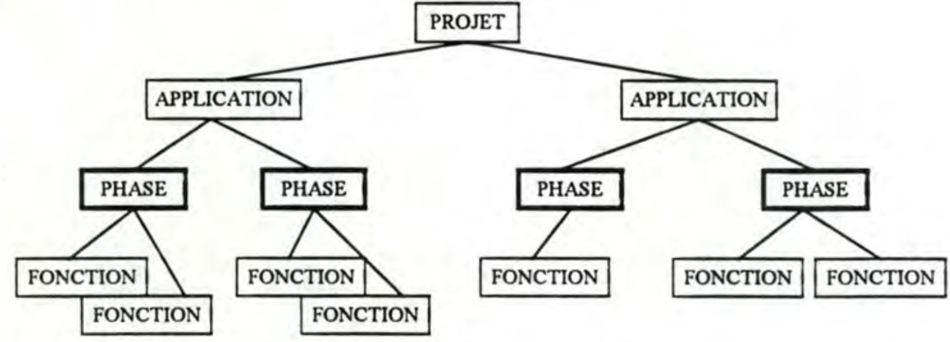


Figure 3 - Nomenclature standardisée

[1] Pour plus de détails, consulter [Par72a, Par72b]

Nous présenterons, de manière synthétique, les concepts de *projet*, d'*application* et de *phase* (le repère central) mais pas celui de *fonction* car nous ne spécifierons pas le cahier des charges avec une telle granularité. Pour plus de détails, consulter [Bod89].

Le concept de projet

Il s'agit de la partie du S.I. qui fait l'objet d'une analyse ou encore un sous-système du S.I. à réaliser. Le niveau projet est significatif du point de vue de la gestion de projet.

Il n'existe pas de critères d'identification précis mais on peut dire qu'un projet est associé aux objectifs opérationnels et/ou de gestion à atteindre dans un domaine de l'organisation.

Exemple : réorganisation du S.I. relatif aux traitements administratifs des commandes clients et des commandes fournisseurs.

Le concept d'application

Une application est un traitement quasi-autonome par rapport aux autres applications d'un projet : elle constitue une unité de planning dans la gestion d'un projet.

On l'identifiera donc grâce aux faits suivants :
- elle est en interaction faible avec d'autres applications
- elle ne communique avec les autres applications que de façon ponctuelle et par l'échange d'agrégats d'informations
- elle est liée à un flux homogène de messages et/ou à une structure d'information homogène

Exemple : deux applications seront identifiées dans le cadre du projet ci-dessus : traitement-commandes-clients (flux "clients") et traitement-réapprovisionnements-fournisseur (flux "fournisseur"). Celles-ci ne communiquent entre elles que via l'état du stock.

Le concept de phase

Traitement possédant une unité spatio-temporelle d'exécution : elle est entièrement exécutée dans un centre d'activité qui est homogène dans le temps et dans l'espace, doté de ressources humaines et/ou matérielles et pourvu de règles de comportement nécessaires à son fonctionnement.

Ce concept est central car,
- sur le plan informationnel, c'est un lieu d'identification de structures de données et de règles de traitements homogènes.
- sur le plan économique, elle est un lieu d'allocations des ressources
- sur le plan organisationnel, elle est un lieu de redéfinition des structures d'organisation.

Une phase sera identifiée par deux critères :
- l'unité spatiale d'exécution : pas de changement de lieu et/ou de ressource lors de l'exécution de la phase
- l'unité temporelle : pas d'interruption lors de l'exécution de la phase.

Exemple : l'application traitement-commandes-clients pourrait être décomposée en phases comme suit : préparation-bon-de-commande, enregistrement-d'une-commande, préparation-réquisition, exécution-réquisition, constitution-colis, etc.

2. LE CAS DE L'HÔPITAL

2.1. NIVEAU PROJET

L'objectif étant d'informatiser le fonctionnement de l'hôpital tel qu'il est décrit dans le cadre général (section II de ce chapitre), nous avons choisi de prendre la gestion de l'hôpital comme projet.

2.2. NIVEAU APPLICATION

Nous avons opté pour une découpe en application qui reflète la réalité organisationnelle. Nous aurons donc une application pour chaque service : l'accueil, la comptabilité, la direction, les urgences, les soins intensifs, la cardiologie, la chirurgie, la médecine interne, la maternité, la gériatrie, l'imagerie médicale, la biologie et la rééducation.

Notre choix est justifié par le fait que les services fonctionnent de façon plus ou moins autonome sans aucun échange d'information ou alors seulement via la mémoire du système.

L'objectif de chacune des applications est de gérer le fonctionnement du service concerné.

2.3. NIVEAU PHASE

Sur base des critères fournis dans la définition, nous avons identifié les phases de chaque application dégagée au point précédent. Nous donnerons le nom ainsi que l'objectif de chacune d'entre elles.

Application Accueil

Phases :
- admission-complète : introduire et/ou mettre à jour les informations concernant un patient lors de son admission pour une hospitalisation
- renseignements-malade : consulter la mémoire du système pour connaître des informations concernant un patient malade donné

Application Urgences

Phases :
- pré-admission-patient : introduire et/ou mettre à jour les informations minimales concernant un patient lors de son admission en urgences
- transfert-patient : transférer un patient des urgences vers un autre service.
- prescription-soin-patient : introduction d'une prescription dictée par un médecin pour un soin à administrer à un patient malade
- annuler-prescription-patient : annuler une prescription de soin
- information-patient : obtenir des informations sur un patient malade
- sortie-patient : transférer un patient vers la sortie
- exécuter-prescription-patient : exécuter une prescription de soin

Application Imagerie médicale

Phases :
- renseignements-patient : obtenir des renseignements sur un patient
- opérer-prescription-patient : opérer une prescription de soin

Application Soins intensifs & Médecine interne

Phases :
- information-malade : présenter des informations au sujet d'un malade
- prescrire-soin : prescrire un soin dicté par un médecin
- annuler-prescription : annuler une prescription
- exécuter-prescription : exécuter une prescription de soin
- introduction-sortie : introduire un mouvement de sortie d'un patient
- introduction-transfert : introduire un mouvement de transfert d'un patient vers un autre service.

Application Comptabilité

Phase :
- production-factures : facturer périodiquement les frais de séjours et de prestations
- interrogation-facture : connaître, à tout moment, ce que doit un patient (facture instantanée)

Application Cardiologie & Chirurgie

Phases :
- sortie : introduire la sortie d'un patient
- supprimer-prescription : annuler une prescription de soin pour un patient
- transfert : faire passer un patient du service de cardiologie vers un autre service.
- prescription-soin : prescrire un soin à administrer à un patient malade
- compte-rendu-malade : obtenir des renseignements sur un patient malade
- prestation-soin : prester une prescription de soin

Application Rééducation

Phases :
- information-patient : donner des information sur un patient
- exécuter-prescription-patient : exécuter une prescription de soin

Application Direction

Phases :
- rappel-prestations : production d'une liste de rappel de prestations par service. Cette liste reprend les prestations prévues pour la pause en cours à la date du jour
- demande-information-malade : obtenir des renseignements sur un malade
- statistiques-on-line : calculer le taux d'occupation instantané des lits de l'hôpital (taux d'occupation global, pour un service ou pour une catégorie de chambre)
- statistiques-périodiques : calculer périodiquement le taux d'occupation des lits par mois pour tout l'hôpital, par service ou par catégorie de chambre.

Application Maternité & Gériatrie

Phases :
- transfert-patient : déplacer un patient vers un autre service.
- prescription-soin-patient : introduire une prescription pour un soin à administrer à un patient
- annulation-prescription-patient : supprimer une prescription de soin
- information-patient : acquérir des information sur un patient
- sortie-patient : transférer un patient vers la sortie
- prestation-prescription-patient : exécuter une prescription de soin

Application Biologie

Phases :
- obtenir-info-malade : obtenir des information sur un patient malade
- accomplir-prescription-soin : accomplir une prescription de soin

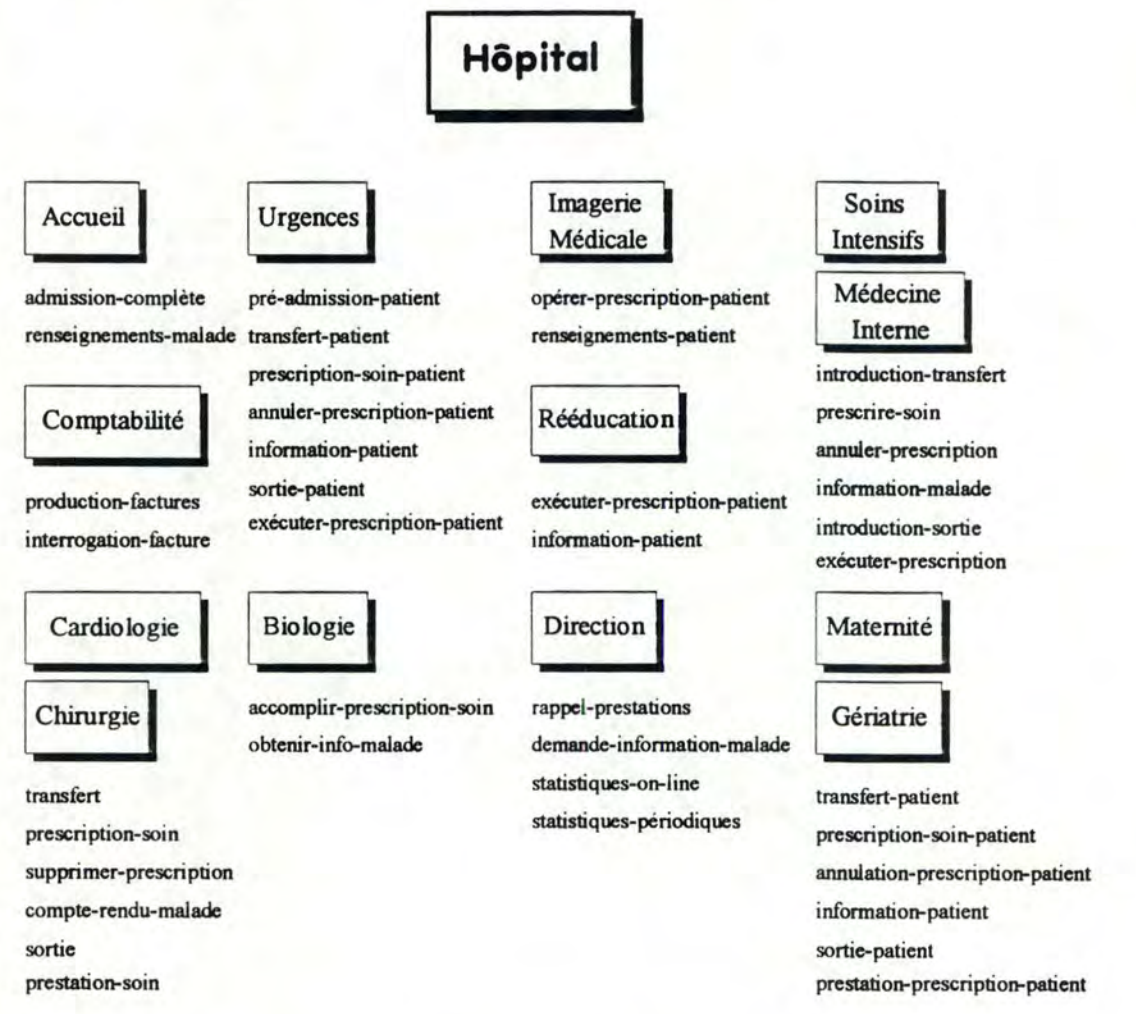


Figure 4 - Schéma récapitulatif

IV. MODÈLE DE STRUCTURATION DES DONNÉES

1. INTRODUCTION

Pour représenter les données présentes dans le cas de l'hôpital, nous nous inspirerons du modèle entités-associations. Le lecteur qui ne serait pas habitué à ce modèle pourra consulter [Che76] pour plus de détails. Nous nous contenterons d'un description des entités et associations; la définition du schéma et des contraintes sera réalisée au chapitre suivant.

2. DESCRIPTION DES ENTITÉS ET DES ASSOCIATIONS

2.1. ENTITÉS

2.1.1. Entité HOPITAL

Définition:

Organisme assurant des soins médicaux à des malades lors d'une ou de plusieurs hospitalisations au sein de cet hôpital.

Identifiant(s):

nom_hop : le nom de l'hôpital.

Attributs:

date_der_fact : la date de la dernière facturation effectuée.
date_der_stat : la date de la dernière fois qu'un calcul de statistiques périodiques a été fait.

2.1.2. Entité PATIENT

Définition:

Toute personne hospitalisée (ou ayant été hospitalisée) dans l'hôpital MortSubite.

Identifiant(s):

nr_dos : le numéro de dossier du patient, ce numéro est un identifiant attribué définitivement à la première hospitalisation de la personne (id1)
nom_pat : le nom du patient
pre_pat : son prénom (id2)
dnais : sa date de naissance

Attributs:

adr_pat : l'adresse du patient
phone : son numéro de téléphone
sexe : m ou f.
etat_civ : son état civil (célibataire, marié, veuf, ...)

2.1.3. Entité PRESTATION

Définition :

Type de soin ou de médicament pouvant être donné à un malade et donnant lieu à une facturation.

Identifiant(s):

code_prest : le code de la prestation.

Attributs:

nom_prest : le nom de la prestation.
type_prest : type de la prestation (interne ou externe).
cap_maxi : nombre maximum de fois que cette prestation peut être exécutée par une unité exécutrice sur l'espace d'une pause.
prix_plein_prest : le prix plein unitaire de la prestation.

Contraintes:

Si type_prest = interne alors est_effectuée_par_ext(prest) n'existe pas.
Si type_prest = externe alors est_effectuée_par_int(prest) n'existe pas.

2.1.4. Entité MEDECIN

Définition :

Toute personne ayant droit de prescrire une prestation ou un mouvement.

Identifiant(s):

nr_med : le numéro du médecin.

Attributs:

nom_med : le nom du médecin;
pre_med : le prénom du médecin;
adr_med : l'adresse du médecin.

2.1.5. Entité LIT

Définition :

Tout lit d'hospitalisation pouvant être attribué à une personne pendant toute (ou une partie de) son hospitalisation dans MortSubite.

Identifiant(s):

nr_lit : un numéro identifiant le lit dans l'hôpital.

2.1.6. Entité CHAMBRE

Définition :

Partie de l'hôpital contenant des lits d'hospitalisation et faisant partie d'un service médical.

Identifiant(s):

nr_ch : un numéro identifiant la chambre au sein de l'hôpital.

2.1.7. Entité CATEGORIE_DE_CHAMBRE

Définition :

Classe de chambres ayant toutes le même prix unitaire de séjour pour un lit.

Identifiant(s):

cat : le nom de cette catégorie (1L, 2L, etc).

Attributs:

prix_plein_lit_j : le prix plein d'un séjour d'une journée d'hospitalisation dans un lit d'une chambre de cette catégorie.

2.1.8. Entité SERVICE_MEDICAL

Définition :

Unité fonctionnelle de MortSubite comprenant des chambres qui contiennent des lits d'hospitalisation. Regroupe les services de soins et le service d'urgence.

Identifiant(s):

ser : le nom du service

2.1.9. Entité SERVICE_TECHNIQUE

Définition :

Unité fonctionnelle de MortSubite ne comprenant pas de chambres. Propose des prestations (radio, scanner) aux services médicaux.

Identifiant(s):

ser_tech : le nom du service

2.1.10. Entité MOUVEMENT

Définition :

Changement d'attribution de lit à une personne; un mouvement peut donc être d'un des 3 types:
- entrée: mouvement dont l'origine est extérieure à l'hôpital, et la destination est un lit d'hospitalisation.
- sortie : mouvement dont l'origine est un lit d'hospitalisation et la destination est extérieure à l'hôpital.
- transfert: mouvement dont l'origine et la destination sont des lits d'hospitalisation.

Identifiant(s):

nr_mv : un numéro identifiant le mouvement.

Attributs:

date_mv : la date effective du mouvement.
heure_mv : l'heure effective du mouvement
type_mv : (entrée, sortie, ou transfert).

Contraintes :

- Un mouvement de type entrée est associé à un seul "lit" par "destination".
- Un mouvement de type sortie est associé à un seul "lit" par "origine".
- Un mouvement de type transfert est associé à deux "lits" différents par "origine" et "destination".

2.1.11. Entité ORGANISME_ASSUREUR

Définition :

Association qui rembourse une partie des frais d'hospitalisation des personnes affiliées.

Identifiant(s):

nr_oa : un numéro identifiant l'organisme assureur.

Attributs:

nom_oa : le nom de l'organisme assureur.
adr_oa : l'adresse de l'organisme assureur.

2.1.12. Entité PRESCRIPTION_SOIN

Définition :

Exprime la prescription par un médecin d'un soin ou d'un médicament à administrer à un malade.

Attributs:

nr_pr : numéro d'ordre identifiant une prescription et attribué lors de la création de cette entité.
date_pr : la date prévue pour l'exécution de la prestation.
heure_pr : l'heure prévue pour l'exécution de la prestation.
état_pr : l'état actuel de la prescription (en attente ou exécutée).

2.2. ASSOCIATIONS

2.2.1. Association PR_PRESCRITE

Définition:

Associe une prescription de soins à la prestation concernée.

2.2.2. Association : PR_PRESCRITE_POUR

Définition:

Associe une prescription de soins au patient qui va la subir.

2.2.3. Association PR_PRESCRITE_PAR

Définition:

Associe une prescription de soins au médecin qui l'a prescrite.

2.2.4. Association AFFILIATION

Définition :

Associe un patient à l'organisme assureur auquel il est affilié.

Attributs:

nr_tit : le numéro de matricule du titulaire de l'assurance.
type_aff : le type de l'affiliation (vipo, salarié, indépendant...).

2.2.5. Association DEMANDE_MOUVEMENT

Définition :

Associe un mouvement au médecin demandeur de ce mouvement.

2.2.6. Association EFFECTUE_MVT

Définition :

Associe un mouvement au patient qui l'a effectué.

2.2.7. Association DESTINATION

Définition :

Associe un mouvement (du type entrée ou transfert) au lit de sa destination c-à-d. le lit où le patient se trouvera après le mouvement.

2.2.8. Association ORIGINE

Définition :

Associe un mouvement (du type transfert ou sortie) au lit de son origine c-à-d. le lit où le patient se trouvait avant le mouvement.

2.2.9. Association APPARTIENT

Définition :

Associe un lit à la chambre où il se trouve.

2.2.10. Association PARTIE_DE

Définition :

Associe une chambre au service de soins dont elle fait partie.

2.2.11. Association COMPORTE

Définition :

Associe un service de soins à l'hôpital dont il fait partie.

2.2.12. Association REMBOURSEMENT_CHAMBRE

Définition :

Associe une catégorie de chambres à un organisme assureur qui rembourse une partie du prix unitaire de séjour dans cette catégorie de chambre.

Attributs:

prix_remb_lit_j : le prix remboursé par l'organisme assureur pour un séjour d'une journée dans un lit d'une chambre de cette catégorie.

2.2.13. Association REMBOURSEMENT_PRESTATION

Définition :

Associe une prestation à un organisme assureur qui rembourse une partie du prix unitaire de cette prestation.

Attributs:

prix_remb_prest : le prix remboursé par l'organisme assureur pour cette prestation.

2.2.14. Association EST_DE_CATEGORIE

Définition :

Associe une chambre à la classe de chambres dont elle fait partie.

2.2.15. Association EFFECTUE_PREST_INT

Définition :

Associe une prestation interne à un service médical.

Attributs:

nb_unités_exéc : nombre d'unités exécutrices pour cette prestation dans ce service.

2.2.16. Association EFFECTUE_PREST_EXT

Définition :

Associe une prestation externe au seul service technique qui la propose.

Attributs:

nb_unités_exéc : nombre d'unités exécutrices pour cette prestation dans ce service.

Chapitre 3

Spécifications fonctionnelles

SPÉCIFICATIONS FONCTIONNELLES

Dans ce chapitre, nous commencerons par expliquer quels sont les objectifs des spécifications fonctionnelles (section I). Nous expliquerons comment nous avons construit les spécifications du cas de l'hôpital (section II) avant de finir par l'énoncé de ces spécifications (section III), le schéma entités-associations et les contraintes associés (section IV).

I. OBJECTIFS DES SPÉCIFICATIONS

On peut lire dans [Mey80] que "spécifier un *problème informatique*, c'est le *poser* aussi précisément que possible en s'interdisant de penser prématurément à sa *solution*".

La tâche qui incombe au spécifieur est donc de décrire ce que le logiciel doit réaliser et non pas comment il doit le réaliser (*What ≠ How*). On distinguera donc la spécification des propriétés attendues du logiciel et la résolution du problème.

On parle de spécifications *fonctionnelles* car on ne s'intéresse qu'aux seules fonctionnalités du logiciel en faisant abstraction de l'organisation et de l'aspect interface homme-machine. Cette abstraction permet de se concentrer sur l'essentiel[1] du logiciel.

La spécification des fonctions[2] sera une spécification *statique* comme le représente la figure 1 : l'effet d'une spécification fonctionnelle est décrit par les arguments et l'état que la fonction F accepte en entrée et par les résultats et l'état modifié (désigné par Etat') que la fonction fournit en sortie. [Ghe91] parlent de spécifications *descriptives* par opposition aux spécifications *opérationnelles* : "les spécifications descriptives essaient de décrire les propriétés désirées d'un système plutôt que son comportement désiré".

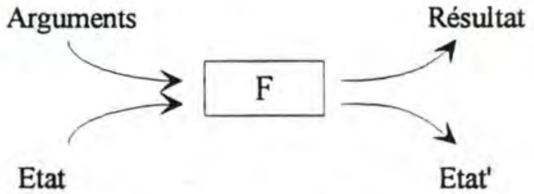


Figure 1 - Effet d'une spécification fonctionnelle

---

[1] Rappelons que notre logiciel se situe dans une catégorie particulière : l'interface homme-machine est relativement simple par rapport à la complexité des traitements.

[2] Notons qu'il ne faut pas confondre le concept de fonction introduit dans le cahier des charges et dans les spécifications fonctionnelles.

On peut encore lire dans [Dub92, Dub93] que les spécifications se construisent à partir du cahier des charges dont on extrait le sous-système informatique qui fera l'objet du logiciel.

Pour le cas de l'hôpital, nous avons donc dérivé les fonctionnalités des phases (au sens de la méthodologie IDA [Bod89]) dégagées dans le cahier des charges en faisant abstraction de l'organisation : les services et la hiérarchie n'existent plus et on essaie donc de trouver les fonctionnalités communes sous-jacentes aux phases. Remarquons que, comme dans le modèle de structuration des traitements (cfr. section II chapitre 2), le concept de phase joue un rôle central : c'est à partir des phases que l'on dérive les fonctionnalités.

L'étape de spécification fonctionnelle a pour but d'identifier et de décrire la *"substance"*, le "*noyau dur*" d'un logiciel. C'est pourquoi on tente d'éliminer une redondance (introduite notamment par l'utilisation de la relation *fait partie de* entre traitements) entre phases qui pourrait entraîner un nombre plus important d'erreurs ainsi qu'une perte de temps (on spécifie deux fois la même chose).
On peut distinguer deux types de redondance. Supposons que, comme l'illustre la figure 2, nous avons deux traitements TA et TB. TA comprend les traitements TA1 et TA2. TB comprend les traitements TB1 et TB2. Ce dernier comprend le traitement TA1. Les traitements TA et TB sont spécifiés indépendamment comme le montre la ligne pointillée verticale. Nous avons donc deux types de redondance : il se trouve que TB1 et TA2 sont en fait des traitements identiques (1er type) et le traitement TA1 est compris dans les deux traitements TA et TB2 (2ème type).

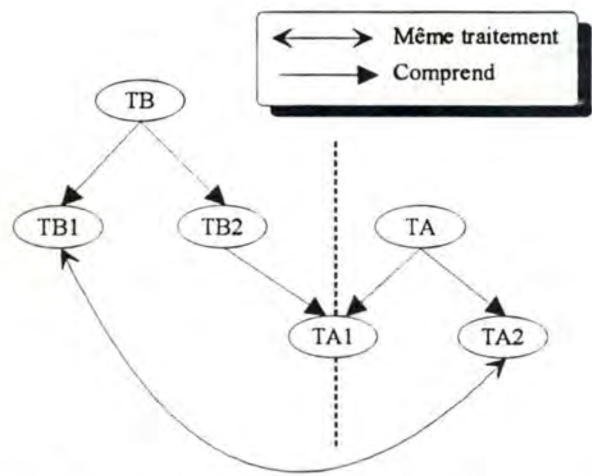


Figure 2 - Mise en évidence de deux types de redondance

Afin d'illustrer l'élimination de la redondance dans le cas de l'hôpital MortSubite, voici un tableau qui établit, pour chaque application, le lien entre les phases du cahier des charges et les fonctionnalités des spécifications fonctionnelles.

Applications	Phases	Fonctionnalités
Accueil	admission-complète renseignements-malade	admission-complète demande-info-malade
Urgences	pré-admission-patient transfert-patient prescription-soin-patient annuler-prescription-patient information-patient sortie-patient exécuter-prescription-patient	admission-pre introduction-transfert prescription-soin annulation-prescription demande-info-malade introduction-sortie prestation-d'une-prescription
Imagerie Médicale	renseignements-patient opérer-prescription-patient	demande-info-malade prestation-d'une-prescription
Soins Intensifs & Médecine Interne	information-malade prescrire-soin annuler-prescription exécuter-prescription introduction-sortie introduction-transfert	demande-info-malade prescription-soin annulation-prescription prestation-d'une-prescription idem idem
Comptabilité	production-factures interrogation-facture	idem idem
Cardiologie & Chirurgie	sortie supprimer-prescription transfert prescription-soin compte-rendu-malade prestation-soin	introduction-sortie annulation-prescription introduction-transfert idem demande-info-malade prestation-d'une-prescription
Rééducation	information-patient exécuter-prescription-patient	demande-info-malade prestation-d'une-prescription
Direction	rappel-prestations demande-information-malade statistiques-on-line statistiques-périodiques	idem demande-info-malade idem idem
Maternité & Gériatrie	transfert-patient prescription-soin-patient annulation-prescription-patient information-patient sortie-patient prestation-prescription-patient	introduction-transfert prescription-soin annulation-prescription demande-info-malade introduction-sortie prestation-d'une-prescription
Biologie	obtenir-info-malade accomplir-prescription-soin	demande-info-malade prestation-d'une-prescription

Figure 3 - Lien entre phases et fonctionnalités

A la figure 3, on peut voir que sur les 39 phases dégagées, seules 13 fonctionnalités sont reprises dans les spécifications fonctionnelles, soit une réduction d'un tiers du nombre de fonctions à spécifier.

Pour la spécification proprement dite des fonctions, nous avons opté pour une description *informelle* mais rigoureuse des spécifications. Ce choix se justifie par, d'une part, le travail déjà réalisé par N. Habra[3] et par, d'autre part, notre volonté de nous concentrer sur l'essentiel du mémoire : l'aspect concurrence dans le S.I.
Citons comme choix alternatifs la logique des prédicats du premier ordre étendue, les langages *Z* [Spi89], *VDM* [Bjø78], etc.

Il faut cependant noter que la construction de *bonnes* spécifications fonctionnelles fait partie de "l'art de spécifier" : mis à part quelques techniques, règles et outils, la détection de la redondance présente entre phases du cahier des charges, l'identification des bonnes fonctionnalités et la spécification correcte et précise de celles-ci sont fortement influencées par l'expérience et le niveau de qualification du spécifieur.

[3] Rappelons que nous n'avons fait qu'étendre un cas qui avait été développé par N. Habra. Une grande partie du travail avait donc déjà été effectué dans ce formalisme.

III. FONCTIONS

1. DESCRIPTION DES FONCTIONS

1.1. FONCTION ADMISSION_COMPLETE

Objectifs

Introduire et/ou mettre à jour les informations concernant un patient lors de son admission pour une hospitalisation.

Messages d'entrée

- nom_pat : le nom de la personne qui demande l'admission
- pre_pat : son prénom
- dnais : sa date de naissance
- ser : le service de soins demandé
- cat : la catégorie de chambre souhaitée
- nr_med : le numéro du médecin qui en cas d'admission sera responsable du mouvement d'entrée
- adr_pat : l'adresse du patient
- phone : son numéro de téléphone
- sexe : (M ou F).
- état_civ : son état civil
- nr_oa : identifiant de son organisme assureur.
- nr_tit : le numéro du titulaire de l'assurance.
- type_aff : son type d'affiliation à l'organisme assureur.

Les données "ser", "cat" et "nr_med" identifient des entités de l'hôpital.

Messages de sortie

Message1 : "le service est plein".
Message2 : "le patient est déjà malade dans MortSubite".
Message3 : <nr_dos, nr_lit>.

Actions sur le S.I.

E1 : Création d'une occurrence de "patient".
E2 : Mise-à-jour des attributs d'une occurrence de "patient" .
E3 : Création des entités et associations: "affiliation", "effectue_mvt", "demande_mouvement", "mouvement" (du type entrée), et "destination".

Règles de traitement

Dénotons par C(i) et E(i) les Conditions et Effets suivants:

C1: L'identifiant introduit correspond à un patient connu (c-à-d nr_dos existant).
C2: L'identifiant introduit correspond à un patient actuellement malade.
C3: Le service désiré est complètement plein.
C4: Les chambres du type souhaité dans le service "ser" sont toutes pleines.
E4: Production du message1.
E5: Production du message2.
E6: Attribution d'un lit dans une chambre du type souhaité.
E7: Attribution d'un lit dans une chambre d'un type autre que le type souhaité.
E8: Production du message3 .

Les règles de traitement sont illustrées par la table suivante:

	R1	R2	R3	R4	R5	R6	R7	R8
C1	V	F	-	-	V	V	F	F
C2	V	V	F	F	F	F	F	F
C3	-	-	V	V	F	F	F	F
C4	-	-	V	F	F	V	F	V
E1							X	X
E2					X	X		
E3					X	X	X	X
E4			X					
E5	X							
E6					X		X	
E7						X		X
E8					X	X	X	X
??		X		X				

1.2. FONCTION ADMISSION_PRE

Objectifs

Introduire les informations minimales concernant un patient lors de son admission en urgence.

Messages d'entrée

- nom_pat : le nom de la personne qui demande l'admission
- pre_pat : son prénom
- dnais : sa date de naissance
- nr_med : le numéro du médecin qui en cas d'admission sera responsable du mouvement d'entrée

Le "nr_med" identifie l'entité Médecin de l'hôpital.

Messages de sortie

Message1: "le service des urgences est plein".
Message2: "le patient est déjà malade dans MortSubite".
Message3: <nr_dos, nr_lit>.

Actions sur le S.I.

E1: Création d'une occurrence de "patient".
E2: Création des entités et associations: "effectue_mvt", "demande_mouvement", "mouvement" (du type entrée), et "destination".

Règles de traitement

Dénotons par C(i) et E(i) les Conditions et Effets suivants:

C1: L'identifiant introduit correspond à un patient connu (c-à-d nr_dos existant).
C2: L'identifiant introduit correspond à un patient actuellement malade.
C3: Le service des urgences est complètement plein.
E3: Production du message1.
E4: Production du message2.
E5: Attribution d'un lit dans le service des urgences.
E6: Production du message3.
??: impossible.

Les règles de traitement sont illustrées par la table suivante:

	R1	R2	R3	R4	R5
C1	V	F	-	V	F
C2	V	V	F	F	F
C3	-	-	V	F	F
E1					X
E2				X	X
E3			X		
E4	X				
E5				X	X
E6				X	X
??		X			

1.3. FONCTION ADMISSION_POST

Objectifs

Compléter les informations concernant un patient malade lorsqu'il quitte le service d'urgence pour sortir de l'hôpital ou être transféré dans un service de soins.

Messages d'entrée

- nr_dos : le numéro de dossier du patient
- adr_pat : l'adresse du patient
- phone : son numéro de téléphone
- sexe : (M ou F).
- état_civ : son état civil
- nr_oa : identifiant de son organisme assureur.
- nr_tit : le numéro du titulaire de l'assurance.
- type_aff : son type d'affiliation à l'organisme assureur.

Nr_dos identifie un patient malade dans le service des urgences.

Messages de sortie

/

Actions sur le S.I.

E1: Mise-à-jour des attributs d'une occurrence de "patient" .
E2: Création de l'association "affiliation".

Règles de traitement

Exécution de E1 et E2.

1.4. FONCTION INTRODUCTION_TRANSFERT

Objectifs

Introduction (si c'est possible) d'un mouvement du type transfert pour un patient malade à l'hôpital.

Messages d'entrée

- nr_dos : un numéro de dossier d'un patient malade.
- nr_med : le numéro du médecin demandeur du transfert.
- nr_lit_or : le numéro du lit d'origine, où le patient est sensé être avant le transfert.
- nr_lit_des : le numéro du lit de la destination voulue.

Les données "nr_dos", "nr_med", "nr_lit_or" et "nr_lit_des" identifient des entités de l'hôpital.

Messages de sortie

Message1: "le patient n'est pas dans le lit d'origine".
Message2: "le lit de destination est occupé".

Actions sur le S.I.

Création des occurrences de: "mouvement" de type transfert, "demande-mouvement", "effectue_mvt", "origine" et "destination".

Règles de traitement

Si le patient se trouve dans le lit d'origine, et si la destination est un lit non-occupé, alors il faut mettre le S.I à jour pour ajouter le mouvement demandé. Si, en plus, le patient est actuellement en urgence, alors il faut déclencher la fonction admission_post; sinon, il faut produire le message adéquat.

1.5. FONCTION INTRODUCTION_SORTIE

Objectifs

Introduction (si c'est possible) d'un mouvement du type sortie pour un patient malade à l'hôpital.

Messages d'entrée

nr_dos : le numéro de dossier d'un patient malade.
nr_med : le numéro du médecin demandeur du transfert.
nr_lit_or : le numéro du lit d'origine, où le patient est sensé être.

Les données "nr_dos", "nr_med" et "nr_lit_or" identifient des entités de l'hôpital.

Messages de sortie

"le patient n'est pas dans le lit d'origine".

Actions sur le S.I.

Création des occurrences de: "mouvement" de type sortie, "demande-mouvement", "effectue_mvt", et "origine".

Règles de traitement

Si le patient se trouve dans le lit d'origine, alors il faut mettre le S.I à jour pour ajouter le mouvement demandé. Si, en plus, le patient est actuellement en urgence, alors il faut déclencher la fonction admission_post; sinon, il faut produire le message adéquat.

1.6. FONCTION PRESCRIPTION_SOIN

Objectifs

Introduction d'une prescription dictée par un médecin pour un soin (au sens large : prestation interne ou externe) à administrer à un patient malade.

Messages d'entrée

nr_dos : le numéro du dossier du malade concerné.
nr_med : le numéro du médecin.
code_prest : le code de la prestation prescrite.
date_pr : la date d'exécution prévue.
heure_pr : l'heure d'exécution prévue.
priorité_0 : indique si oui ou non la demande porte la priorité 0.

Les données "nr_dos", "nr_med" et "code_prest" identifient des entités de l'hôpital.

La date et l'heure prévue représentent une pause qui est au moins supérieure à la pause en cours sauf si la demande est de priorité 0.

Actions sur le S.I.

Création d'une occurrence de "prescription_soins", "pr_prescrite_pour", "pr_prescrite_par", "pr_prescrite"

Règles de traitement

L'état de la prescription créée est "en_attente".

1.7. FONCTION PRESTATION_D'UNE_PRESCRIPTION

Objectifs

Confirmer l'exécution d'une prescription d'un soin.

Messages d'entrée

nr_pr : le numéro de la prescription. "nr_pr" 'identifie une prescription de soins en attente prévue pour la date du jour.

Actions sur le S.I.

Changement de l'attribut "état" de la prescription concernée.

1.8. FONCTION ANNULATION_PRESCRIPTION

Objectifs

Supprimer une prescription d'une prestation non exécutée (par exemple à cause d'un changement de traitement, de la sortie du malade,....)

Messages d'entrée

nr_pr : le numéro de la prescription concernée. "nr_pr" identifie une prescription de soins qui est en attente.

Actions sur le S.I.

Suppression de l'occurrence concernée de "prescription_soin", "pr_prescrite_pour", "pr_prescrite_par", "pr_prescrite"

1.9. FONCTION RAPPEL_PRESTATION

Objectifs

Production d'une liste de rappel de prestations par service (médical ou technique). Cette liste reprend les prestations prévues pour la pause en cours à la date du jour et non encore exécutées.
Chaque service médical recevra une liste de rappel des prestations internes et externes à effectuer sur ses patients. Chaque service technique recevra une liste des prestations externes ("commandées" par les services médicaux) à effectuer.

Messages de sortie

Services médicaux

La sortie est un ensemble de listes de rappels (une par service). Chaque élément de cet ensemble est composé du nom de service concerné et d'une liste des rappels le concernant. Cette liste est un ensemble de rappels où chaque rappel est un tuple de la forme: <nr_dos, nr_pr, date_pr, heure_pr, code_prest, nom_prest>.

Services techniques

La sortie est un ensemble de listes de rappels (une par service). Chaque élément de cet ensemble est composé du nom de service concerné et d'une liste des rappels concernant ce service.
Cette liste est un ensemble de rappels où chaque rappel est un tuple de la forme: <nr_dos, ser, nr_pr, date_pr, heure_pr, code_prest, nom_prest>.

Actions sur le S.I.

\

Règles de traitement

Pour tout service (médical ou technique) de l'hôpital il faut produire une liste de rappels. Cette liste contient les prescriptions à rappeler concernant les malades séjournant dans ce service. Une prescription est à rappeler si sa date et son heure prévue est inférieure ou égale à la pause en cours à la date du jour et si elle est en attente.

1.10. FONCTION DEMANDE_INFO_MALADE

Objectifs

Consulter la base de données pour connaître des informations concernant un patient malade donné.

Messages d'entrée

nom_pat : le nom du malade
pre_pat : son prénom
dnais : sa date de naissance

Messages de sortie

- "Message1": Les informations suivantes sur le malade même :
  "Message1A"
  • nr_dos : le numéro du dossier du malade cherché.
  "Message1B"
  • adr_pat : son adresse
  • phone : son numéro de téléphone
  • état_civ : son état civil
  • sexe : (M ou F).

- "Message2" qui contient les informations suivantes concernant l'organisme assureur du malade:
  • nr_oa : le numéro de l'organisme assureur
  • adr_oa : l'adresse de l'organisme assureur
  • nom_oa : le nom de l'organisme assureur
  • nr_tit : le numéro du titulaire de l'assurance
  • type_aff : son type d'affiliation (VIPO, ...)

- "Message3" qui contient des informations concernant les mouvements effectués par le malade lors de l'hospitalisation courante. "Message3" est une liste de mouvements dans laquelle chaque ligne représente un mouvement effectué par le malade concerné; une ligne est un tuple de la forme: <date_mv, heure_mv, type_mv, nr_med, nr-lit_or, nr_lit_des>.

- "Message4" est une liste de prestations dans laquelle chaque ligne représente une prestation effectuée pour le malade concerné; une ligne est un tuple de la forme:
  <nr_pr, date_pr, code_prest, nr_med>.

Actions sur le S.I.

\

Règles de traitement

Les informations fournies (message1A, 1B, 2, 3 et 4) en sortie concernent le patient malade identifié par le triplet fourni en entrée et ne se trouvant pas dans le service d'urgence.
Seuls les messages 1B, 3 et 4 seront fournis si le patient malade est en urgence.

1.11. FONCTION PRODUCTION_FACTURES

Objectifs

Facturer périodiquement les frais de séjour et de prestations concernant les patients sortis durant la période écoulée depuis la dernière facturation. Deux ensembles de factures seront produits :

- les factures des patients, chacune de ces factures reprend les frais à charge d'un patient à facturer
- les factures des organismes assureurs, chacune de ces factures reprend les frais à charge d'un organisme pour un patient affilié

Messages de sortie

- Un ensemble de factures destinées aux patients. Une facture de cet ensemble est composée des parties suivantes:
  - L'en-tête : un tuple de la forme <nr_dos, nom_pat, adr_pat, nom_oa, adr_oa>.
  - La partie "frais de séjour": une suite de lignes ayant chacune la forme:
    <cat, prix_plein_lit_j, prix_remb_lit_j, prix_pat_lit_j, date_debut_sej, date_fin_sej, duree_sej, cout_sejour_oa, cout_sejour_pat>.
  - La partie "frais de prestations": une suite de lignes ayant chacune la forme:
    <code_prest, nom_prest, date_pr, heure_prest, prix_plein_prest, prix_remb_prest, prix_pat_prest>.
  - Le total dû par le patient.
- Un ensemble de factures destinées aux organismes assureurs des patients à facturer. Une facture de cet ensemble a la même structure et le même contenu qu'une facture destinée au patient, sauf en ce qui concerne la dernière partie qui contient - dans ce cas - le total dû par l'organisme assureur.

Actions sur le S.I.

Mise à jour de l'attribut "date_der_fact" de l'hôpital.

Règles de traitement

- Toute "hospitalisation" terminée entre la date de la dernière facturation et la date du jour est une "hospitalisation à facturer". Le patient concerné est le "patient à facturer".

- A chaque "hospitalisation à facturer" correspond une et une seule facture dans l'ensemble des factures adressées aux patients et une et une seule facture dans l'ensemble des factures adressées aux organismes assureurs; ces deux ensembles ne contiennent que des factures relatives à des "hospitalisations à facturer". Les 2 factures correspondant à une hospitalisation d'un patient sont identiques sauf pour la partie "total".

- Toute paire de mouvements successifs et facturables de l' "hospitalisation à facturer" donne lieu à une et une seule ligne dans la partie "frais de séjour" de chacune des 2 factures concernant cette hospitalisation (séjour facturable).

- Toute prescription de soins concernant un "patient à facturer", avec un état "exécutée" et une date comprise entre le début et la fin de l' "hospitalisation à facturer" donne lieu à une et une seule ligne dans la partie "frais de prestations" de chacune des 2 factures concernant cette hospitalisation.

- Dans une facture:
  "prix_pat_lit_j" = "prix_plein_lit_j" - "prix_remb_lit_j"
  "duree_sej" = "date_fin_sej" - "date_debut_sej".
  "cout_sej_oa" = "duree_sej" * "prix_remb_lit_j"
  "cout_sejour_pat" = "duree_sej" * "prix_pat_lit_j"
  "prix_pat_prest" = "prix_plein_prest" - "prix_remb_prest".

Le total dû par le patient est la somme de tous les "cout_sej_pat" + la somme de tous les "prix_pat_prest".
Le total dû par l'organisme assureur est la somme de tous les "cout_sej_oa" + la somme de tous les "prix_remb_prest".

Rappelons qu'un patient peut effectuer plusieurs mouvements sur une journée. L'état d'occupation de l'hôpital à une certaine date sera donc l'état d'occupation de cet hôpital à la date et à une heure fixée : les sorties étant généralement avant 14h, on peut convenir que cette heure est 16h.

1.12. FONCTION INTERROGATION_FACTURE

Objectifs

Cette fonction permet de savoir à chaque instant ce que doit un malade donné jusqu'à la date de la demande (abstraction faite de ce que doit son organisme assureur). Ceci revient à produire une facture concernant un seul patient et reprenant les frais de son hospitalisation comme si cette hospitalisation se termine le jour même de la demande.

Messages d'entrée

nr_dos : le numéro du dossier du patient malade.

Messages de sortie

- La structure du message est identique à la structure d'une facture destinée à un patient décrite dans la fonction PRODUCTION FACTURES ci-dessus.

Actions sur le S.I.

\

Règles de traitement

En considérant le malade identifié par "nr_dos" comme un "patient à facturer", et considérant son hospitalisation courante comme une hospitalisation qui se termine la date du jour; la facture est parfaitement décrite par les règles de la fonction PRODUCTION FACTURES ci-dessus.

1.13. FONCTION STATISTIQUES_ON_LINE

Objectifs

Calcul on-line de différents taux d'occupation de lits dans l'hôpital MortSubite.

Messages d'entrée

- glob : indique que l'on désire le taux d'occupation global ou
- ser : le nom du service pour lequel on souhaite calculer le taux ou
- cat : le nom de la catégorie pour laquelle on souhaite calculer le taux

Messages de sortie

- Taux_glob: le taux d'occupation global de l'hôpital et pour tout service de l'hôpital
- Taux_ser: le taux d'occupation de ce service et pour toute catégorie de chambre
- Taux_cat: le taux d'occupation de cette catégorie de chambre.

Actions sur le S.I.

\

Règles de traitement

- Taux-glob = (100 * nbroc_hop) / lithop
  où:
  - nbroc_hop = le nombre de lits occupés dans MortSubite.
  - lithop = le nombre total de lits dans MortSubite.
- Taux-ser = (100 * nbroc_ser) / lit_ser
  où
  - nbroc_ser = le nombre de lits occupés du service.
  - lit_ser = le nombre total de lits du service.
- Taux-cat = (100 * nbroc_cat) / lit_cat
  où:
  - nbroc_cat = le nombre de lits occupés appartenant à des chambres de la catégorie ,
  - lit_cat = le nombre total de lits appartenant à des chambres de la catégorie.

1.14. FONCTION STATISTIQUES_PERIODIQUES

Objectifs

Calculer périodiquement quelques statistiques intéressantes concernant les séjours dans MortSubite. Nous envisageons actuellement le taux d'occupation par mois pour tout l'hôpital, le taux d'occupation par mois par service, et le taux d'occupation par mois par catégorie de chambre.

Messages de sortie

- Les taux d'occupation de l'hôpital répartis par mois, une suite de tuples :
  <mois, Taux-glob-pm, liste-taux-ser, liste-taux cat>
  où:
  liste-taux-ser est une suite de couple <ser, Taux-ser-pm>
  liste-taux cat est une suite de couple <cat, Taux cat-pm>

Actions sur le S.I.

Mise à jour de l'attribut "date_der_stat".

Règles de traitement

A chaque mois écoulé depuis la date du dernier calcul de statistiques "date_der_stat" correspond un et un seul tuple <mois, Taux-glob-pm, liste-taux-ser, liste-taux cat> dans lequel
- mois est le mois concerné
- Taux-glob-pm = (100 * nuitées_hop) / (journois * lithop)
  où:
  - nuitées = la somme, pour chaque lit de l'hôpital, du nombre de journées d'hospitalisation passées dans ce lit pendant ce mois.
  - jourmois = le nombre de jours du mois.
  - lithop = le nombre de lits dans MortSubite.
- A chaque service de l'hôpital correspond un et un seul couple <ser, Taux-ser-pm> de la suite liste-taux-ser dans lequel
  - ser est le service concerné
  - Taux-ser-pm= (100 * nuitées_ser) / (jourmois * lit_ser)

  où:
  a) nuitées_ser = la somme, pour chaque lit du service concerné, du nombre de journées d'hospitalisation passées dans ce lit pendant ce mois.
  b) jourmois = le nombre de jours du mois
  c) lit_ser = le nombre total de lits du service.
- A chaque catégorie de chambre de l'hôpital correspond un et un seul couple <cat,Taux-cat-pm> de la suite liste-taux-cat dans lequel
  - cat est la catégorie concernée
  - Taux-cat-pm = (100 * nuitées_cat) / (jourmois * lit_cat)

  où:
  a) nuitées_cat = la somme, pour chaque lit de la catégorie concernée, du nombre de journées d'hospitalisation passées dans ce lit pendant ce mois.
  b) jourmois = le nombre de jours du mois
  c) lit_cat = le nombre total de lits appartenant à des chambres de la catégorie.

Rappelons qu'un patient peut effectuer plusieurs mouvements sur une journée.
L'état de l'hôpital à une certaine date sera donc l'état de cet hôpital à la date et à une heure fixée : 16h.

2. REMARQUES

- L'aspect syntaxique a été omis intentionnellement dans les spécifications précédentes. Le schéma conceptuel ainsi que les messages de fonctions ne décrivent qu'une structure abstraite des données. La définition de formats adéquats pour les données permanentes ainsi que pour les messages externes est laissée pour la suite.

- Les spécifications précédentes couvrent la fonctionnalité et les données, l'interface utilisateur n'a pas été spécifiée. Néanmoins, on demande que l'interface soit "amicale" et "ergonomique". Ainsi, les modules qui couvrent cette interface sont à définir puis à réaliser. Ces modules assureraient aussi la validation syntaxique par rapport aux formats choisis dans le paragraphe précédent.

IV. SCHÉMA CONCEPTUEL

1. SCHÉMA ENTITÉS-ASSOCIATIONS

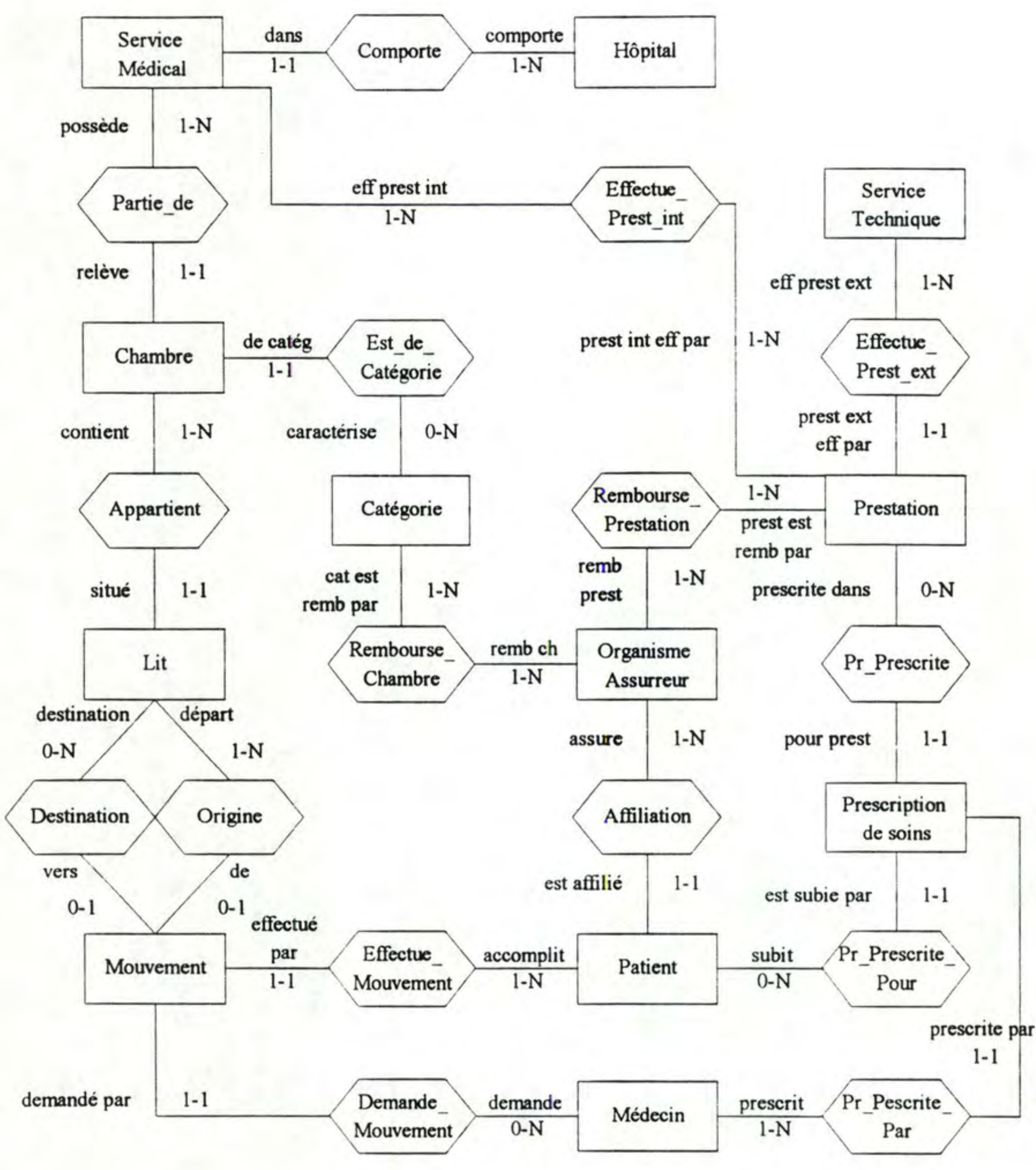


Figure 4 - Schéma E-A de l'hôpital

2. CONTRAINTES

- Un patient est toute personne admise (au moins une fois) à l'hôpital MortSubite, et ayant donc un numéro de dossier dans cet hôpital.

- Un patient malade est tout patient ayant actuellement un lit attribué.

- Un ancien malade est tout patient n'ayant pas actuellement un lit attribué.

- Un patient malade P occupe un lit L si le mouvement le plus récent effectué par ce malade a le lit L comme destination. On dit donc que le lit L est occupé par le patient P.

- Un malade se trouve dans la chambre C si ce malade occupe un lit appartenant à cette chambre.

- Un malade se trouve dans le service S si ce malade se trouve dans une chambre C faisant partie du service S.

- Une chambre est pleine si tous les lits appartenant à cette chambre sont occupés.

- Un service est plein si toutes les chambres appartenant à ce service sont pleines.

- Deux mouvements MV1 et MV2 effectués par un patient P sont des mouvements successifs et facturables si :
  - MV1 est le mouvement le plus proche effectué par le patient P avant une heure fixée
  - et MV2 est le mouvement le plus proche effectué par le patient P après la date et l'heure de MV1 + 24h

- Un séjour facturable du patient P dans le lit L est la période écoulée entre deux mouvements successifs et facturables MV1 et MV2 de ce patient:
  - MV1 ayant L comme destination,
  - et s'il existe un mouvement MV3 (non nécessairement différent de MV2) tel que :
    a) date/heure(MV1)<date/heure(MV3)<=date/heure(MV2) et,
    b) origine(MV3)=L
  - la date du mouvement MV1 est le début du séjour
  - et la date du mouvement MV2 est la fin du séjour.

- Deux séjours du patient P sont successifs si le début du deuxième séjour est égal à la fin du premier séjour.
- Une hospitalisation d'un patient P est la période écoulée entre D1 et D2 (D1 < D2) dont:
  - D1 est la date/heure d'un mouvement "entrée" effectué par le patient P;
  - D2 est
    a) soit la date/heure du mouvement "sortie" le plus proche de D1, on parle alors d'une hospitalisation terminée,
    b) soit la date/heure du jour. S'il n'existe aucun mouvement "sortie" postérieure à D1, on parle alors de l'hospitalisation courante.

Chapitre 4
L'approche OBLOG

L'APPROCHE OBLOG

I. INTRODUCTION

L'informatique est une discipline jeune et en constante évolution. La construction progressive d'un modèle du cycle de vie d'une application informatique reflète bien cette évolution.

A mesure que la taille des programmes grandissait, la nécessité est apparue de mettre en place une phase d'analyse du problème et de conception de la solution, bien distinctes de la phase de programmation proprement dite. Par la suite, c'est la maintenance qui a attiré l'attention, à un moment où on s'est rendu compte que cette activité exigeait une part disproportionnée des budgets informatiques.

Cet élargissement du cycle de vie, coûteux et difficile à gérer, est peut-être à l'origine d'une crise du développement de logiciels [ESDI92]. Signalons entre autres la dispersion des efforts, la distance "sémantique" entre les spécifications et l'application exécutable et le fait que les modifications sont généralement implémentées directement au niveau de la programmation sans être répercutées aux niveaux supérieurs.

L'approche OBLOG a pour ambition de rationaliser le cycle de vie :
- en intégrant les différents formalismes utilisés en un seul langage : OBLOG;
- en intégrant les différents outils de développement selon le concept IPSE *(⇨ Integrated Project Support Environment)*;
- en offrant un support méthodologique adéquat.

1. LE LANGAGE

OBLOG *(⇨ OBject LOGic)* est un langage destiné à supporter tout le développement d'un système d'information. En conséquence :
- il propose des concepts suffisamment de haut niveau pour pouvoir appréhender les entités du monde réel et supporte un formalisme graphique (diagrammatique) relativement simple qui peut être compris par des personnes non initiées;
- il fournit également des concepts de bas niveau, liés à la technologie, permettant de couvrir des détails d'implémentation d'une solution;
- il possède une sémantique formelle rigoureuse autorisant des vérifications de cohérence et de complétude;
- enfin, il possède une sémantique opérationnelle permettant la génération automatique du code correspondant aux spécifications (aussi bien du code C, SQL ou MOTIF).

OBLOG est un langage orienté-objet. A ce titre, il supporte les principes O-O habituels (intégration des données et du comportement, techniques d'agrégation et d'héritage,...) mais se distingue par une approche pleinement concurrente : le système d'information est une collection d'objets concurrents qui interagissent entre eux.

On le voit, les objectifs sont ambitieux. Le langage a subi plusieurs évolutions qui ont donné lieu à plusieurs versions. Concrètement, seul un sous-ensemble d'OBLOG, appelé OBLOGkernel, est aujourd'hui entièrement défini et supporté par un atelier logiciel. OBLOGkernel propose les notions de base du langage et ne peut prétendre supporter la phase d'analyse des besoins ni même le développement d'applications complexes. OBLOGkernel est décrit à la section II.

Dans le cadre de ce mémoire, nous utiliserons également OBLOGlight, qui sera implémenté dans la prochaine version de l'atelier logiciel. OBLOGlight est décrit à la section III.

2. L'OUTIL

OBLOG-CASE est l'atelier logiciel *(⇨ workbench)* qui supporte le langage OBLOG et permet d'articuler les spécifications OBLOG avec la technologie. Concrètement, OBLOG-CASE permet aux développeurs de :
- spécifier l'application dans l'éditeur graphique;
- concevoir l'interface utilisateur avec un éditeur de dialogue;
- construire la base de données à l'aide d'un éditeur de schémas logiques et physiques;
- vérifier la complétude et la cohérence de l'application;
- générer le code de l'application;
- etc...

OBLOG-CASE se veut un système ouvert et adopte l'approche IPSE, offrant une intégration verticale (intégration de plusieurs outils pour les différentes phases d'un projet) et une intégration horizontale (plusieurs personnes travaillent simultanément sur une vue unifiée des spécifications).

La première version, OBLOG-CASE v1.0[1], supporte le langage OBLOGkernel. Le code d'OBLOG-CASE a été spécifié et généré avec le langage OBLOGkernel, mettant ainsi en oeuvre une stratégie de *bootstrap*. OBLOG-CASE v1.0 est décrit à la section IV.

3. LA MÉTHODE

Après le *langage* et l'*outil*, la *méthode* constitue le troisième axe de l'approche OBLOG, qui se veut une approche "pratique". Un ensemble de règles méthodologiques sont proposées dans [ESDI93] et ont été testées durant le développement d'OBLOG-CASE v1.0. Ces règles concernent tant la gestion d'un projet que la conception de l'architecture d'une application. Leur développement dépasse le cadre de ce mémoire.

---

1 OBLOG-CASE V1.0 est commercialisé seulement depuis avril 1993.

II. LE LANGAGE OBLOG-KERNEL

La définition du langage est inspirée de [ESDI92b, ESDI93][2].

Pour illustrer les concepts du langage, nous réutiliserons des éléments du cas de l'hôpital.

1. OBJETS, CLASSES D'OBJETS ET TYPES DE DONNÉE

1.1. OBJETS

Le cas de l'hôpital fait apparaître plusieurs concepts : patient, prestation, lit, service de soins etc. Chacun de ces concepts correspond à des *objets* du monde réel. Les objets constituent les "briques" de base pour décrire un système en OBLOG.

Un objet OBLOG peut être décrit de la manière suivante :
- un objet possède une structure (voir section 2.) et un comportement (voir section 3.);
- un objet est créé et peut être détruit. Entre sa création et sa destruction, un objet peut accomplir des *actions*;
- un objet encapsule un *état*. Cet état est représenté par les *attributs* de l'objet;
- l'état d'un objet ne peut être changé que par les actions de cet objet;
- les objets interagissent entre eux au sein d'une *communauté*.

Chaque objet possède un *identifiant* interne unique qui ne dépend pas de son état (par opposition aux éventuels identifiants externes correspondant à des attributs ou groupes d'attributs de l'objet).

1.2. CLASSES D'OBJETS

Une *classe d'objets* représente un ensemble d'objets semblables. Plus précisément, tous les objets qui vérifient les mêmes propriétés structurelles et comportementales font partie d'une même classe d'objets. Ces propriétés forment la spécification de la classe d'objets. Chaque élément de la classe est appelé objet ou *instance* de la classe.

1.3. TYPES DE DONNÉE

Un *type de donnée* sert à représenter des entités statiques, qui, contrairement aux objets, n'ont pas de comportement. Un nombre entier, par exemple, n'a pas de comportement. On distingue:
- des types de données prédéfinis : `int`, `nat` (pour "*natural*"), `string`, etc.;
- des types de données déclarés par l'utilisateur : énumération et liste (liste de types de donnée ou liste de types d'objets).

---

2 Pour des raisons pratiques, nous nous sommes permis certaines libertés au niveau de la représentation graphique du langage.

1.4. COMMUNAUTÉ D'OBJETS

Une spécification OBLOG consiste en un ensemble de spécifications de classes d'objets. La communauté de toutes les classes d'objets est déclarée dans le *diagramme de communauté* (voir figure 1).

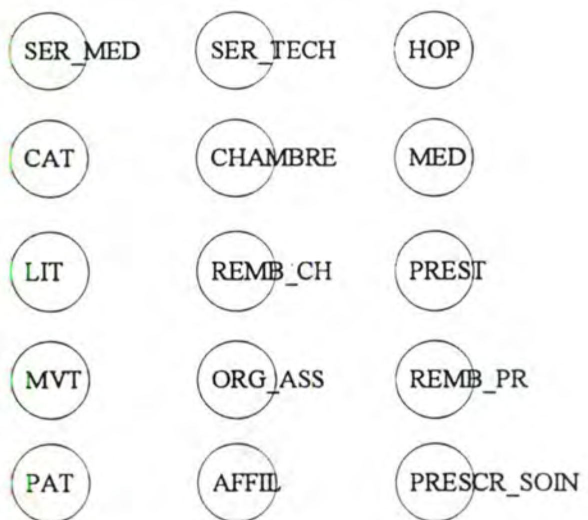


Figure 1 - Diagramme de communauté du cas de l'hôpital (partim)

Chaque classe d'objets est représentée par un cercle. Lorsque le chiffre "1" apparaît sous le nom de la classe, cela signifie qu'il ne peut exister plus d'une instance dans la classe (*single*).
Le diagramme de communauté comporte également la déclaration des types de donnée définis par l'utilisateur, représentés par un rectangle.

1.5. UNITÉS

Pour de larges applications, on préférera regrouper les spécifications de classes d'objets en plusieurs *unités*. A chaque unité correspond un diagramme de communauté.
Dans l'exemple de l'hôpital, on peut envisager de créer trois unités : une unité par niveau de structuration (voir conception statique), le niveau 3 correspondant en partie au diagramme de communauté de la figure 1.

2. STRUCTURE D'UN OBJET

La structure d'un objet est définie dans le *diagramme de déclaration* de la classe d'objets. Les attributs occupent la partie supérieure du diagramme; les actions se trouvent dans la partie inférieure (voir figure 2, les déclarations associées à la classe d'objets `PATIENT`).

2.1. ATTRIBUTS

Chaque attribut est déclaré par un nom et un codomaine qui définit les valeurs que l'attribut peut prendre. Un codomaine peut être :
- un type de donnée;
- un type d'objet;
- une action (voir le mécanisme de *rappel (⇨ call-back*), plus loin dans ce chapitre).

Les attributs décrivent les propriétés des objets (par exemple l'adresse d'un patient) aussi bien que des relations entre les objets (l'attribut `estAffilié` désigne l'instance de `AFFIL` à laquelle le patient est affilié).

2.2. ACTIONS

Les actions sont utilisées pour décrire le comportement d'un objet. Les actions sont classifiées en *actions de naissance* (qui créent une nouvelle instance dans une classe d'objets), *actions de mort* (qui détruisent l'instance) et *actions de mise à jour* (qui ne font que modifier l'état de l'instance). Les actions de naissance et de mort sont indiquées respectivement par une astérisque ("*") et une croix ("+") en-dessous de leur nom.

De plus, on distingue les actions *actives* et *passives*. Les actions actives, comme `incNrDosSuiv`, ont lieu de la propre intitiative de l'objet. Les actions passives, au contraire, ne peuvent avoir lieu qu'à la suite d'un stimulus extérieur. L'action `fixeAdr` est un exemple d'action passive, déclenchée à partir de l'objet `ADMISSION`. L'activité d'une action est indiquée par un point d'exclamation ("!") en-dessous de son nom.
Remarquons que l'action de naissance `birth` pourrait être active : le patient a l'initiative pour "se créer" lui-même (au commencement de la vie de la communauté d'objets).

Les actions peuvent avoir des *paramètres* (que l'on déclare de la même manière que les attributs). Les paramètres sont utilisés pour définir les effets de l'action sur les attributs ou pour échanger de l'information entre objets. L'action `fixeTel`, par exemple, a comme paramètre le numéro de téléphone du patient. Les actions qui ont des paramètres sont surmontées d'un triangle noir ("▼").

Enfin, précisons que les actions n'ont pas de durée; nous dirons qu'elles sont *atomiques*.

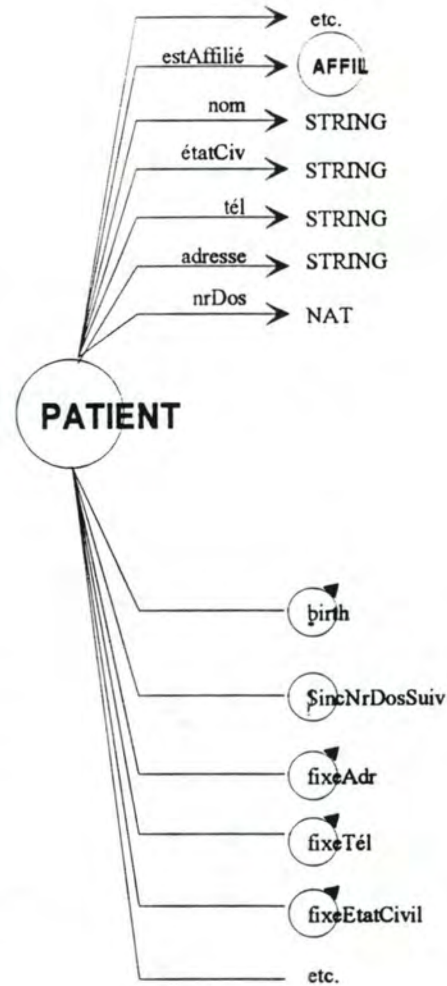


Figure 2 - Diagramme de déclaration de la classe d'objets PATIENT

3. COMPORTEMENT D'UN OBJET

Le comportement d'un objet est formé de trois éléments : les *effets* des actions, le *cycle de vie* et l'*interaction* de l'objet. Effets et interactions sont spécifiés à partir du diagramme de déclaration (cf figure 2); le cycle de vie est représenté par le *diagramme de comportement* (voir figure 3).

3.1. EFFETS DES ACTIONS

Les effets d'une action sur l'état (c'est-à-dire sur les attributs) sont décrits dans la clause *mise à jour d'attribut* associée à cette action. Cette clause comprend une liste d'affectations qui définissent les nouvelles valeurs des attributs après l'occurence de l'action. Ces affectations sont introduites par le symbole "$\Rightarrow$".
Dans l'action `birth`, les attributs `nom` et `nrDos` sont initialisés. L'action `incNrDosSuiv` incrémente le numéro de dossier qui sera attribué au patient suivant.

3.2. CYCLE DE VIE

La *vie* d'un objet est une séquence d'actions commençant par une action de naissance et se terminant éventuellement par une action de mort. Le *diagramme de comportement* décrit les cycles de vie possibles d'un objet sous la forme d'un graphe situation-transition (voir figure 3).

Une *situation* définit un ensemble d'actions possibles. Durant sa vie, l'objet est toujours dans une situation bien déterminée et ne peut accomplir que les actions possibles dans la situation courante. Suite à l'occurence d'un action possible, l'objet évolue d'une situation à une autre, ce qui correspond à une *transition*.

Une transition peut être inhibée par une *précondition*, en d'autres termes, une action dans une certaine situation ne peut avoir lieu que si la précondition associée est vérifiée.
Enfin, le diagramme de comportement comprend également les clauses d'*instanciation* des paramètres d'action.

A la figure 3, on peut lire que, après sa naissance, le patient se trouve dans une situation d'attente. Les actions passives `fixeTel`, `fixeAdr`, `fixeEtatCivil` et `fixeSexe` sont "accessibles".

Remarque : un diagramme de comportement peut laisser place à de l'indéterminisme. Ainsi, à la figure 3 : les actions peuvent avoir lieu dans un ordre quelconque (si ce n'est qu'aucune action ne peut avoir lieu avant la naissance ni après la mort) et les paramètres des actions prennent des valeurs arbitraires.

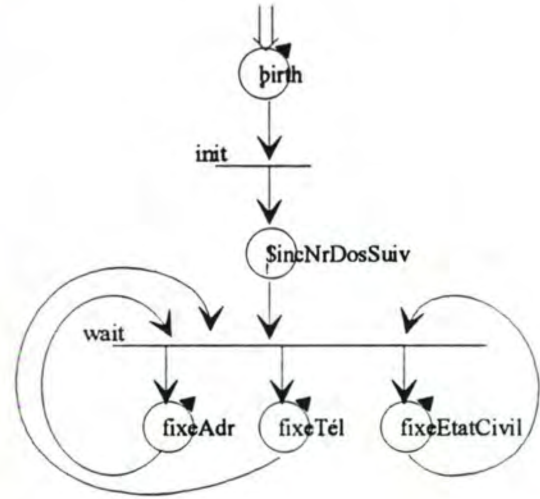


Figure 3 - Diagramme de comportement de la classe d'objets PATIENT

3.3. INTERACTION

Les objets interagissent entre eux via le mécanisme *d'appel d'action*. La clause *appel*, associée à une action, représente la liste des appels de cette action. Pour chaque appel (introduit par le symbole ">>"), il faut spécifier l'identifiant de l'objet appelé, l'action appelée (obligatoirement une action passive) ainsi que les éventuels paramètres réels de l'action.
L'appel synchronise la vie de l'objet appelant et de l'objet appelé. En effet, lorsqu'une action en appelle une autre, ces deux actions ont lieu simultanément, pour autant qu'elles soient toutes les deux possibles (dans le cas contraire, l'appel ne se produit pas et aucune des deux actions n'a lieu).

Un cas particulier d'appel concerne les actions de naissance. Nous avons vu qu'un objet avait la capacité de se créer tout seul. Dans d'autres cas, l'action de naissance est passive. La figure 4, par exemple, représente le fait que l'objet `Admission` ajoute un nouveau patient dans l'hôpital. L'action `creePat` appelle l'action `birth` d'une nouvelle instance de la classe `PATIENT` et lui passe comme paramètre les coordonnées du nouveau patient. L'identifiant de cette nouvelle instance est fourni au paramètre `nvPatient`.

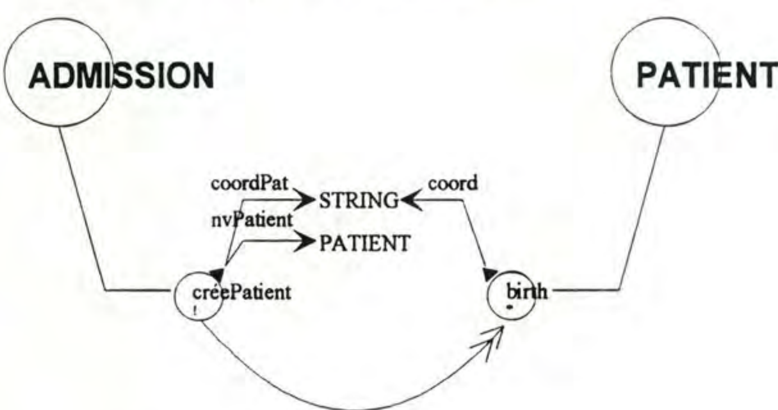


Figure 4 - Création d'une nouvelle instance de PATIENT

Signalons encore brièvement que :
- un appel peut être inhibé par une précondition;
- si l'identifiant de l'appel est une liste d'instances, chaque instance de la liste est appelée (mécanisme de *multicast*);
- une action peut appeler une action d'un objet appartenant à une autre unité;
- un appel vers ou de l'extérieur est un appel à ou de quelque chose en-dehors du monde OBLOG (par exemple une fonction C). Ce mécanisme est très intéressant puisqu'il permet d'interfacer facilement une application OBLOG avec d'autres applications;
- une action peut appeler une action dont la référence est stockée dans un attribut *(⇨ stored action)*. Ce mécanisme permet d'implémenter une architecture client-serveur, le client passant en paramètre la référence de l'action à rappeler *(⇨ to call back)* lorsque le service est terminé.

4. LE LANGAGE D'EXPRESSION D'OBLOG-KERNEL

Dans l'exemple développé jusqu'ici, nous avons utilisé des expressions en plusieurs endroits: mise à jour d'attribut, précondition de transition, instanciation de paramètres, etc.
Brièvement, signalons que les expressions sont formées au moyen :
- d'attributs et de paramètres;
- des opérateurs classiques sur les types de données;
- de la constante `SELF` qui, dans la spécification d'un objet, désigne l'objet lui-même;
- de requêtes *(⇨ queries)*.

Il existe deux types de requêtes : les requêtes d'instance et les requêtes de classe.

- Une requête d'instance (construite avec l'opérateur ".") renvoie la valeur d'un attribut d'un objet dont on possède la référence. Par exemple, la requête `pat.adr` renvoie la valeur de l'attribut `adresse` du patient référencé par `pat`.. Les requêtes d'instances peuvent être imbriquées (exemple : *`réf1.réf2.réf3.attr`*).

- Les requêtes de classe sont construites à l'aide des opérateurs `ONE`, `ALL` et `EXISTS`.
  Exemples :
  - `ONE[PATIENT|nom = "Dupont"]` renvoie une instance de `PATIENT` dont l'attribut `nom` vaut `"Dupont"`);
  - `ALL[PATIENT|nom = "Dupont"]` renvoie tous les patients Dupont;
  - `EXISTS[PATIENT|nom = "Dupont"]` indique s'il existe au moins un patient qui s'appelle Dupont.

  Comme les requêtes d'instance, les requêtes de classe peuvent être imbriquées.

5. TYPES D'OBJET

Comme nous l'avons déjà dit plus haut, le langage OBLOGkernel s'intègre dans un environnement de production de logiciels OBLOG-CASE v1.0 qui a pour objectif final la production automatisée de code. Pour réaliser ces logiciels, les utilisateurs peuvent faire appel à différentes technologies telles que des technologies d'interface homme-machine, d'impression ou de gestion de base de données.

OBLOGkernel propose une vue "unifiée" de ces technologies dans le sens où les différents concepts technologiques sont reliés à des concepts OBLOGkernel : on spécifiera les éléments technologiques comme des types d'objets *(⇨ kind-of)* [Zei92].

Les caractéristiques et les limitations d'une technologie définiront le sous-ensemble des concepts d'OBLOGkernel et la sémantique opérationnelle qui seront associés à un type d'objet [Zei92].

5.1. TYPE OBLOG (OBL)

Il s'agit d'un objet OBLOG classique tel qu'il a été présenté jusqu'à présent. L'éditeur associé est l'éditeur de spécifications *(⇨ OBLOG Editor)*.

5.2. TYPE DIALOGUE (DBX)

Ce type d'objet est destiné à réaliser l'interface homme-machine de l'application. A un objet sera associée une boîte de dialogue que l'on peut construire dans l'éditeur de dialogue (⇨ *DIALOG Editor*).

Les éléments de base de ce type d'objet sont :
- les étiquettes (⇨ *Labels*),
- les champs entrées/sorties (⇨ *Input Output Field*),
- les champs d'affichage uniquement (⇨ *Output Only Field*) et
- les boutons (⇨ *PushButtons*).

La relation entre la boîte de dialogue et "le monde OBLOG" est la suivante :

Eléments de la boîte de dialogue	Concepts d'OBLOGkernel
Etiquette	Attribut (de type LBL)
Champ Entrées/Sorties	Attribut (de type IOF)
Champ Affichage uniquement	Attribut (de type OOF)
Bouton	Action (passive, de type PBT)

A la figure 5, on peut voir un exemple de boîte de dialogue que l'utilisateur peut créer grâce à l'outil éditeur de dialogues. Un objet de type DBX sera généré par l'outil.

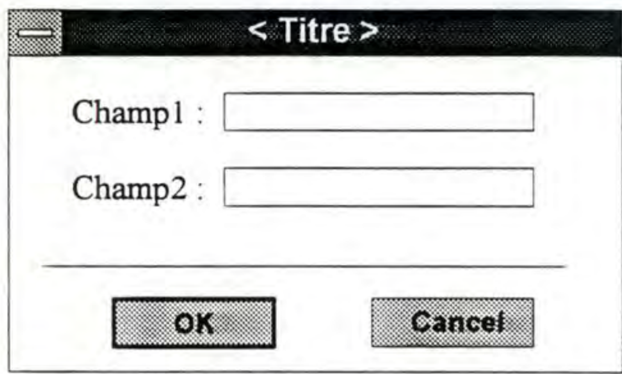


Figure 5 - Un exemple de boîte de dialogue

5.3. TYPE TABLE (TBL)

Les objets OBLOG sont volatiles : une fois l'exécution de l'application terminée, ils disparaissent. Les objets de type table sont, eux, persistants : la population des objets de ce type est conservée d'exécution en exécution. Elle est automatiquement chargée et sauvée respectivement en début et en fin d'exécution.

La relation entre les concepts "base de données" et les concepts OBLOGkernel est la suivante :

Concepts BD	Concepts d'OBLOGkernel
Table	*Classe d'objets TBL*
- record	- objet (instance)
- champ	- attribut
Opérations	*Actions*
- insertion	- naissance
- mise-à-jour	- mise-à-jour
- effacement	- mort

Les objets de type TBL ont un certain nombre de restrictions que nous ne détaillerons pas ici.

III. LE LANGAGE OBLOG-LIGHT

Comme son nom l'indique, OBLOGkernel constitue le noyau du langage OBLOG. La prochaine version d'OBLOG-CASE devrait fournir un langage beaucoup plus étendu, appelé OBLOGlight. Ce chapitre présente brièvement les principales extensions d'OBLOGlight que nous utiliserons au cours de la troisième partie. La définition du langage proposée ici est inspirée des documents préparatoires [ESDI92e, ESDI92f] disponibles au moment de la rédaction de ce mémoire ainsi que de divers exposés sur le sujet.

Les exemples utilisés pour illustrer les nouveaux concepts sont des exemples intuitifs portant sur une classe d'objets `PERSONNE`. Ces exemples sont présentés sous forme textuelle. Malgré les nombreux avantages de la représentation graphique, il est en effet apparu indispensable d'introduire, à l'occasion d'OBLOGlight, une représentation textuelle du langage (les deux représentations ne doivent pas nécessairement être exclusives mais peuvent coexister)[3].

1. CLASSES D'OBJETS, MÉTACLASSES ET TYPES DE DONNÉE

1.1. CLASSES D'OBJETS

- Lorsqu'une contrainte de cardinalité impose qu'une classe d'objets contienne au plus une instance *(⇨ single constraint)*, il n'est pas nécessaire d'identifier explicitement la seule instance de la classe. Pour référencer cette instance, on utilise directement le nom de la classe (en minuscules).

[3] Comme aucune représentation d'OBLOGlight n'est encore rigoureusement définie, il nous a fallu improviser certaines notations. Selon les cas, nous aménagerons les diagrammes d'OBLOGkernel pour les adapter aux nouveaux concepts ou nous utiliserons les notations textuelles introduites dans les sections suivantes.

- Un opérateur CLASS_OF est défini sur toutes les classes d'objets. Cet opérateur accepte en argument une instance et renvoie comme résultat le nom de la classe auquelle cette instance appartient.

1.2. MÉTACLASSES

A chaque classe d'objets est associée un *objet de classe (⇨ class object)* ou *métaclasse*. Cet objet particulier (de catégorie *single*) n'a pas de représentation propre et réside "à l'intérieur" de la spécification d'une classe d'objets. Il permet de spécifier des propriétés, non pas d'une instance particulière, mais de la classe d'objets tout entière (voir les attributs de la métaclasse, plus loin dans ce chapitre).

1.3. TYPES DE DONNÉE

OBLOGlight élargit sensiblement l'ensemble des types de donnée disponibles. Notons:
- parmi les types de donnée simples : date, time;
- parmi les constructeurs de types : RANGE, CP (produit cartésien), UNION, ARRAY.

Tous les types de donnée (excepté ARRAY) contiennent implicitement une donnée indéfinie, appelée NULL. Toute opération qui devrait fournir une valeur incompatible avec le type de donnée spécifié renvoie la valeur NULL.

2. STRUCTURE D'UN OBJET

2.1. ATTRIBUTS

- On distingue les attributs *variables*, que l'on peut modifier librement, des attributs *constants*, pour lesquels une valeur est donnée à la création de l'objet et ne peut jamais être modifiée.

- Un attribut peut être *dérivé*, ce qui signifie que sa valeur est calculée, selon une *règle de dérivation*, en fonction de la valeur d'autres attributs (et d'éventuelles préconditions). Exemple :

```
âge : nat
DER majeur : bool = âge ≥ 18.
```

- Une *contrainte* peut être associée à un attribut afin de restreindre l'ensemble des valeurs que l'attribut peut prendre. Exemple :

```
solde : int
    {? NOT majeur} ⇒ ≥ 0
```

  exprime que le solde d'une personne mineure ne peut être négatif (l'expression entre accolade est une précondition sur la contrainte).

- Les attributs de la métaclasse représentent des propriétés de la classe d'objets tout entière. Ces attributs sont préfixés du sigle "$". Exemple :

```
DER $nbPersonnes : nat = #(ALL[p:PERSONNE]).
```

- Les *clés (⇨ keys)* sont des attributs particuliers de la métaclasse. Une clé est une fonction qui, à partir d'un tuple de valeurs d'attributs identifiants, renvoie l'unique instance qui possède ces valeurs d'attributs. Exemple : la clé
```
$laPersonne(nom, prénom) : PERSONNE
```
renvoie une personne sur base des valeurs des attributs `nom` et `prénom`.

2.2. ACTIONS

- En plus des préconditions associées à des situations particulières, on peut associer à une action une *précondition globale*.

- De même qu'il existe des attributs dérivés, une action peut être *dérivée* d'une autre action. L'action dérivée se distingue par le fait que les valeurs de ses paramètres sont données par des règles de dérivation. Exemple : étant donné une action
```
!dépense(montant : nat),
```
on peut définir l'action dérivée
```
!dépenseTout IS dépense(solde).
```

2.3. INTERFACE

Le langage OBLOGkernel ne prévoit aucune restriction quant à l'utilisation des requêtes; l'état d'un objet peut être observé librement par la communauté. D'un point de vue méthodologique, il est évidemment souhaitable de construire des interfaces "propres" entre les objets en distinguant clairement la partie *privée* et la partie *publique* d'un objet.

OBLOGlight supporte explicitement ce principe d'*interface*. La déclaration d'un objet comprend deux volets : une *interface* et un *corps*. Le corps de la déclaration comprend tous les attributs et toutes les actions de l'objet, comme dans OBLOGkernel. L'interface de la déclaration consiste en un sous-ensemble du corps rendu public, c'est-à-dire :
- une liste d'attributs observables par la communauté;
- une liste des actions pouvant être déclenchées par la communauté[4].

Cependant, il est souvent intéressant d'offrir, non pas un sous-ensemble, mais une *vue* des attributs et des actions. C'est pourquoi l'interface d'un objet peut contenir des attributs et des actions dérivés, qui n'existent pas dans le corps mais qui sont dérivés à partir des attributs et des actions du corps.

Enfin, à chaque attribut ou action public peut être associée une précondition. L'attribut ne peut être observé, l'action ne peut être déclenchée, que si la précondition est vérifiée.

---

4 Les attributs et les actions rendus publics peuvent être renommés respectivement *observations* et *événements*.

```
OBJET PERSONNE

INTERFACE

Observations
  nom : string
  prénom : string
  $nbPersonnes : nat

Keys
  $laPersonne(nom, prénom) : PERSONNE

Events
  * nais
  reçoit(montant : nat)

BODY

Attributes
  CONST nom : string
  CONST prénom : string
  CONST dateNaiss := date
  DER âge : nat
      := calendrier.dateJour - dateNaiss
  DER majeur : bool
      := âge ≥ 18
  solde : int
      {NOT majeur} ⇒ ≥ 0
  DER $nbPersonnes : nat
      := #(ALL[p:PERSONNE])
  KEY $laPersonne(nom, prénom) : PERSONNE
      := ONE[p:PERSONNE|p.nom=nom AND p.prénom=prénom]

Actions
  * nais(nom, prénom : string)
      ⇒ nom := nais.nom
      ⇒ prénom := nais.prénom
      ⇒ dateNaiss := calendrier.dateJour
  reçoit(montant : nat)
      ⇒ solde := solde + reçoit.montant
  !emprunte
      >> banque.demandeCrédit(SELF)
  !dépense(montant : nat)
      {dépense.montant ≤ solde} ⇒ solde := solde - dépense.montant
      {dépense.montant > solde} ⇒ solde := solde -
      (dépense.montant * 1,08).
  !dépenseTout
      IS dépense(solde)
      ? solde > 0
  + !meurs
```

Figure 6 - Déclarations de la classe d'objets PERSONNE (représentation textuelle)

3. COMPORTEMENT D'UN OBJET

3.1. EFFETS DES ACTIONS

La mise à jour d'un attribut par une action peut être soumise à une précondition (entre accolades). Exemple :

```
!dépense(montant : nat)
    {dépense.montant ≤ solde} ⇒ solde := solde -
    dépense.montant
    {dépense.montant > solde} ⇒ solde := solde -
    (dépense.montant * 1,08).
```

3.2. CYCLE DE VIE

- Comme la mise à jour d'un attribut, l'instanciation d'un paramètre peut être soumise à une précondition.
- Le diagramme de comportement d'un objet peut s'avérer complexe. Dans certains cas, il est préférable de définir des *patrons (⇨ patterns)*, c'est-à-dire des morceaux du diagramme de comportement qui ont une signification par eux-mêmes ou qui se répètent en plusieurs endroits. (La notion de patron correspond, dans un langage de troisième génération, à la notion de procédure.)

3.3. INTERACTION

Dans OBLOGkernel, un appel est toujours synonyme de synchronisation : l'appel ne se produit que lorsque action appelante et action appelée ont lieu simultanément. OBLOGlight fournit également un mécanisme d'*appel asynchrone* : l'action appelante peut avoir lieu indépendamment de l'action appelée.

4. LE LANGAGE D'EXPRESSION D'OBLOGLIGHT

- Dans une expression, le nom d'une classe d'objets (en minuscules), désigne:
  - soit l'unique instance de la classe s'il s'agit d'une classe de catégorie *single*;
  - soit la métaclasse, dans le cas contraire.
- Le langage d'expression des requêtes est amélioré, notamment pour pouvoir manipuler, non seulement des classes d'objets, mais également des types de donnée. L'introduction de variables facilite l'expression de requêtes imbriquées.
  Exemple :
  - `ALL[(p.nom, p.prénom) OF p:PERSONNE | p.âge > 60]` renvoie la liste des noms et prénoms des personnes qui ont plus de 60 ans;
  - `ALL[p:PERSONNE | EXISTS[v:VOITURE | v.propriétaire = p]]` renvoie la liste des personnes qui possèdent une voiture.

5. TYPES D'OBJETS

5.1. TYPE RAPPORT

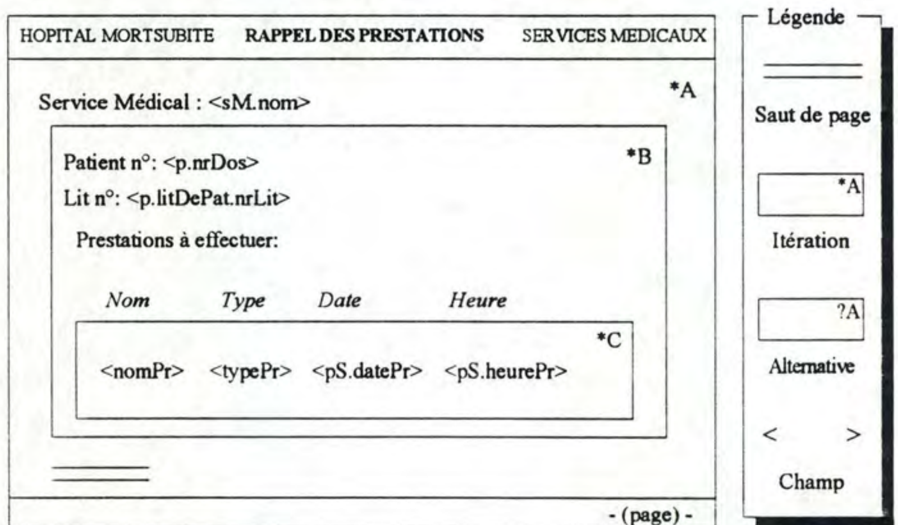
Le langage OBLOGlight offre, en plus des objets de type OBL, DBX et TBL, un type d'objets rapport (RPT).

Ces types d'objets sont destinés à fournir des rapports imprimés sur tout ou partie des objets présents dans l'application au moment de l'exécution. L'éditeur associé est l'éditeur de rapport (⇨ *REPORT Editor*).

Les différents concepts de l'éditeur de rapport sont :
- les champs
- les boucles
- les alternatives
- les opérations de synthèse sur les champs (ex : TOTAL, etc.)
- les sauts de page, en-tête, pieds de page

Le langage de requête peut être utilisé pour définir le contenu des champs, les conditions de boucles et d'alternatives.

La figure 7 illustre un exemple de rapport que l'utilisateur pourrait créer grâce à l'outil éditeur de rapports. Un objet de type RPT sera alors généré par l'outil.



A : ALL [sM : SER_MED]
B : ALL [p : PAT | p.patDansSer(sM)=TRUE]
C : ALL [pS : PRESCR_SOIN | pS.subitePar=p]

Figure 7 - Un exemple de rapport

Vu que rien de précis n'existe quant à l'intégration OBLOG, nous nous contenterons de signaler qu'une action de l'objet représentant ce rapport permettra de le déclencher.

5.2. OBJETS INTERNES

Dans certains cas, une action d'un objet peut être trop complexe pour être exprimée "en une fois" : il devient nécessaire de lui associer un véritable comportement. Avec le concept d'objet interne (⇨ *Inner Object*), une action peut être "conceptuellement" éclatée en un objet comme le schématise la figure suivante.

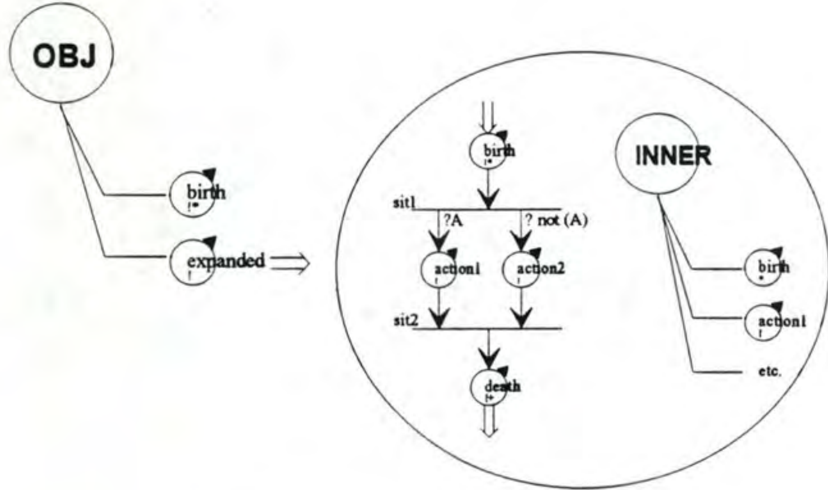


Figure 8 - Un exemple d'objet interne

L'exécution de l'action éclatée est atomique du point de vue de l'objet "maître" mais, du point de vue de l'objet interne, cette action débute à la naissance de l'objet interne et se termine à sa mort. Certains paramètres de l'action éclatée peuvent être communiqués à l'objet interne lors de l'action de naissance de celui-ci et les résultats seront communiqués à l'action éclatée par l'action de mort.

L'objet interne peut interagir avec d'autres objets mais l'objet "maître" ne peut pas se trouver dans la chaîne des appels.

6. RELATIONS ENTRE OBJETS (CLASSES D'OBJETS)

Le langage OBLOGlight introduit, en plus de la relation d'interaction, quatre catégories de relations entre objets (classes d'objets). Il s'agit de...

6.1. SPÉCIALISATION / GÉNÉRALISATION

Un objet (une classe d'objets) peut être spécifié(e) comme étant la spécialisation d'un autre objet (d'une autre classe d'objets) déjà spécifié(e) : un COMPTE EPARGNE peut être la spécialisation d'un COMPTE car il s'agit d'un type de compte particulier (p. ex., taux d'intérêt particulier, etc.).

L'objet spécialisé (aussi appelé aspect) hérite de la spécification de son "ancêtre". Aucune modification ne peut être faite sur la partie interface de la spécification héritée (observations, clés et événements) sauf sur le ou les événement(s) de naissance de cet objet. Il est seulement permis d'étendre l'interface : ajouter des observations, des clés ou des événements.

Le corps (l'implémentation) de l'objet peut être plus ou moins modifié selon le type de spécialisation que l'on choisit :
- spécialisation libre : modification libre du corps
- spécialisation avec mise-à-jour limitée : aucune règle de mise-à-jour d'attribut héritée ne peut être modifiée et aucune nouvelle règle de mise-à-jour d'attribut hérité ne peut être définie. Les actions et attributs hérités ne peuvent être changés.
- spécialisation limitée : correspond à la spécialisation avec mise-à-jour limitée accompagnée d'une redéfinition du diagramme comportemental limitée. Une transition héritée ne peut être que substituée par une transition plus complexe. Cette transition plus complexe est un patron construit à partir de nouvelles actions et contenant la transition héritée.

La généralisation est le pendant de la spécialisation et, de ce fait, n'a pas été implémentée dans le langage : il s'agit d'abstraire une spécification commune à une ou plusieurs classe(s) d'objets. La spécialisation multiple (plusieurs classes d'objets racine) n'est pas autorisée.

6.2. AGRÉGATION

Un objet peut être défini comme étant l'agrégation d'autres objets (composantes).
Une agrégation sera spécifiée par :
- la liste des composantes
- un mécanisme de sélection des instances des composantes
- des attributs qui ont des valeurs dérivées des observations des composantes
- les éléments de spécification classiques (actions, attributs, etc.)

Une agrégation peut changer dynamiquement ses composantes : une voiture composée de quatre roues peut changer de roue pour cause de crevaison.

6.3. CONTRAINTE D'INTÉGRITÉ

Une contrainte d'intégrité sera spécifiée par :
- la liste des classes d'objets sur lesquelles elle porte
- une liste de formules en logique des prédicats du premier ordre pouvant utiliser les observations des classes d'objets listées

Il est possible de spécifier que l'on souhaite que cette contrainte soit vérifiée (par l'outil) au moment de l'exécution ou non.

6.4. ASSOCIATION

Il s'agit d'un concept similaire à l'association du modèle Entités-Associations. L'association sera toujours binaire et sans attributs.

Elle sera spécifiée par :
- deux classes d'objets
- le nom de rôle joué par chaque classe d'objets
- une contrainte de cardinalité pour chaque partenaire de l'association définie par une borne inférieure (0-1) et une borne supérieure (1-N)

L'association sera transformée en un attribut mono- ou multi-valué selon la cardinalité du rôle joué par la classe d'objets. Cet attribut peut être soit un attribut normal soit un attribut dérivé.

La possibilité de vérification à l'exécution est offerte comme pour la contrainte d'intégrité.

1. SCHÉMA

Le langage OBLOGlight et l'outil *OBLOG-CASE v1.0* s'inscrivent dans un projet à plus long terme qui est destiné à offrir un atelier de conception de logiciels assistée par ordinateur *OBLOG-CASE* centré autour du langage OBLOG.
La figure suivante montre les différentes composantes de ce futur atelier. Seules quelques composantes font partie de la version actuelle de l'outil.

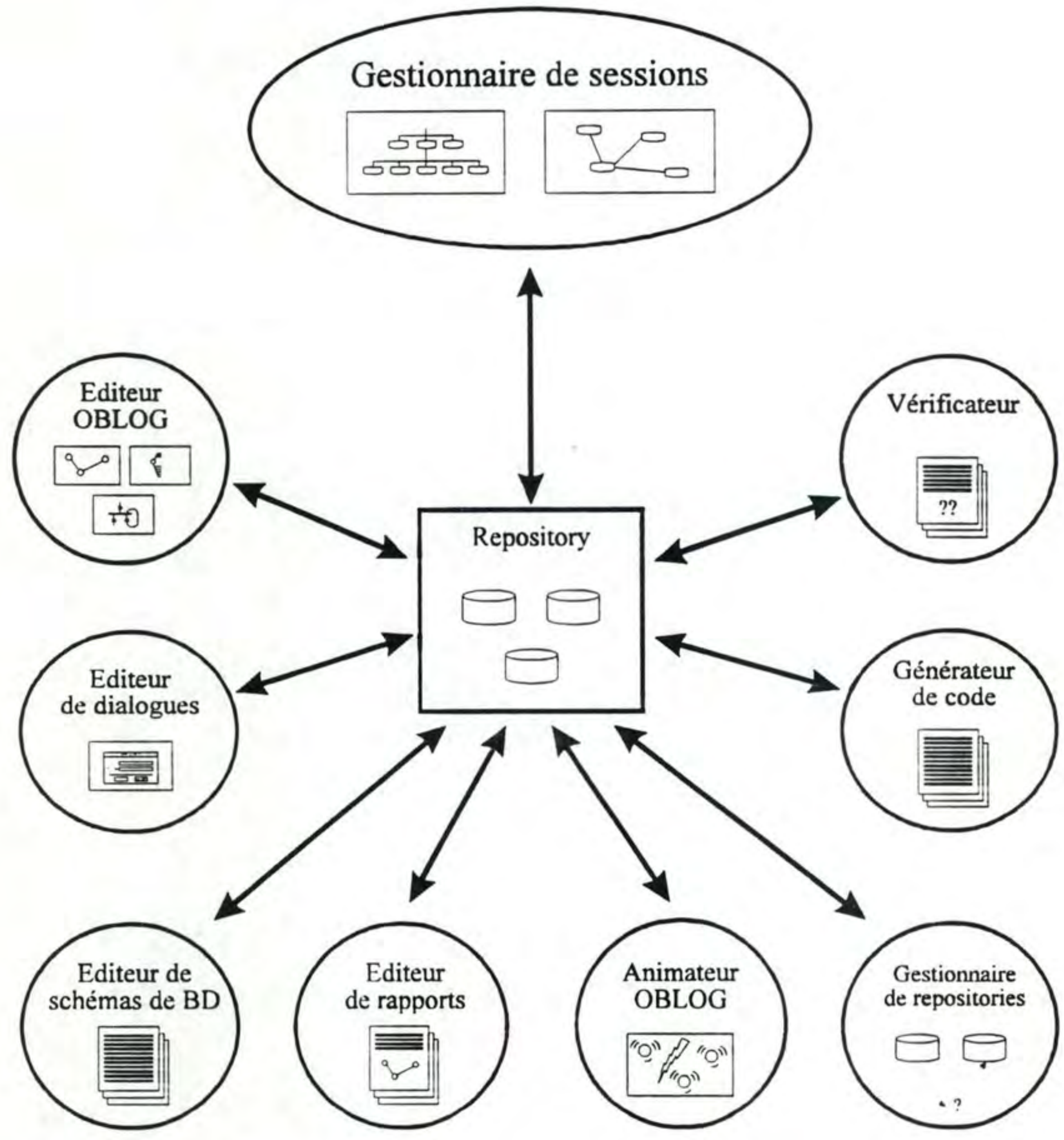


Figure 9 - Les composantes du futur atelier OBLOG-CASE

1.1. GESTIONNAIRE DE SESSIONS

L'utilisateur, au plus haut niveau d'abstraction du projet, manipulera les concepts d'*unités* ("containers" à objets), de *liens* entre unités (du type "utilise") et de *groupes* ("containers" à unités/groupes). Le gestionnaire de sessions offre deux diagrammes :
- le *diagramme de système* gère les groupes
- le *diagramme de projet* gère les unités et les liens entre les unités.

1.2. EDITEUR OBLOG

Activé dans le contexte d'une unité, l'éditeur offre les diagrammes suivants :
- le *diagramme de communauté* présente la population des classes d'objets et des types de données ainsi que les relations entre classes d'objets
- le *diagramme de déclaration* présente les déclarations associées à une classe d'objets ainsi que les effets des actions et les interactions de la classe d'objets avec le reste de la communauté
- le *diagramme de comportment* présente le cycle de vie d'un objet

1.3. EDITEUR DE DIALOGUES

Cet éditeur permet de créer des interfaces homme-machine ainsi que les objets OBLOG associés. Le code lié à la technologie interface (MOTIF, etc.) sera produit par l'outil.

1.4. EDITEUR DE SCHÉMAS DE BASE DE DONNÉES

Cet éditeur permettra de gérer plus finement la partie base de données du projet. L'outil génère automatiquement le code SQL DDL (définition statique) et SQL DML (manipulation dynamique) relatif au schéma logique de la BD.

D'autre part, afin d'optimiser son application ou de tirer parti de primitives puissantes offertes par le SGBD, l'utilisateur peut définir lui même le DDL/DML relatif au schéma physique de la BD.

1.5. EDITEUR DE RAPPORTS

Cet éditeur permet de concevoir des rapports imprimés sur la population des objets de l'application. L'utilisateur "dessine" son rapport dans l'éditeur de rapport et l'outil génère automatiquement un objet de type RPT qui implémente ce rapport.

1.6. ANIMATEUR OBLOG

Cette composante offre la possibilité de simuler l'exécution des spécifications OBLOG, de les tracer (⇨ *Trace*), de signaler à l'utilisateur des erreurs ou indéterminismes dans les spécifications, d'évaluer des expressions, de donner des valeurs aux paramètres, etc.

Il s'agit d'un outil de *debug* interactif semblable à ceux que proposent beaucoup de langages (Turbo Pascal, Turbo C++, etc.).

1.7. GESTIONNAIRE DE REPOSITORIES

L'ensemble des groupes, des unités et des spécifications est stocké dans une base de données appelée *repository*. Grâce au *gestionnaire de repositories*, l'utilisateur a la possibilité de créer et gérer ses propres *repositories*.

Il pourra aussi importer et exporter des spécifications via un *flat file* sous le format ASCII.

1.8. GÉNÉRATEUR DE CODE

L'outil génère, sur base des différentes spécifications, du code ANSI C, MOTIF, DDL/DML SQL et Postscript ainsi que différentes procédures pour réaliser le programme exécutable. Ce code devra être compilé et "linké" à l'aide des différentes procédures. Il est aussi possible de tracer l'exécution du code généré.

Le code généré sera une image exacte des spécifications (bien que l'équivalence ne soit pas formellement prouvée).

1.9. VÉRIFICATEUR

L'outil permet de vérifier :
- la complétude des spécifications OBLOG, des éléments d'interface et des éléments de base de données ainsi que leur cohérence
- la cohérence syntaxique des spécifications
- la sémantique des spécifications : boucles, indéterminisme dans le diagramme de comportement, etc.

1.10. OBLOG-CASE V1.0

Dans la version actuelle du produit, seules les fonctionnalités suivantes sont implémentées :
- gestionnaire de sessions
- éditeur OBLOG
- éditeur de dialogues (limité)
- gestionnaire de *repositories* (limité)
- générateur de code
- vérificateur (limité)

Chapitre 5

Conception statique

CONCEPTION STATIQUE

I. INTRODUCTION

Nous allons développer ici la conception statique. Pour ce faire, nous allons d'abord exposer les objectifs de cette étape (section II) et, plus particulièrement, les raisons pour lesquelles la conception est dite *statique* (section II.2). Nous verrons ensuite comment modifier le langage OBLOGlight (qui deviendra *OBLOGstatic*) afin qu'il puisse supporter la conception statique (section III). Suivra ensuite le développement du cas en OBLOGstatic (section IV).

II. OBJECTIFS DE LA CONCEPTION STATIQUE

"L'activité de conception est une phase fondamentale dans le processus qui transforme progressivement, à travers un certain nombre d'étapes intermédiaires, les *system requirements* en un produit final. Son résultat est un *software design*. Nous définissons un *software design* comme étant une décomposition du système en modules — description de ce que chaque module doit faire et des relations entre ces modules. Une telle description est souvent appelée *architecture* du logiciel ou *structure* du logiciel. Le but de l'activité de conception est donc la définition de l'architecture du logiciel." (Traduit de [Ghe91]).

Partant de nos spécifications fonctionnelles, nous allons les raffiner en réalisant :
- un enchaînement des fonctionnalités : les fonctionnalités ne sont plus simplement "déclarées", elles deviennent "exécutables"
- une implémentation des états (données persistantes) : des choix sont faits au niveau de la représentation des données (base de données, constantes de programmes, fichiers, ...), de la structure des données, etc.
- une gestion des erreurs : il importe de déterminer quels sont les cas d'erreurs ainsi que les traitements à réaliser lorsqu'ils surviennent
- l'interface homme-machine de l'application

Pour effectuer ces transformations, nous allons nous aider des primitives offertes par les divers logiciels tels que les logiciels système, les gestionnaires de base de données ou de gestion des interfaces homme-machine.

1. ARCHITECTURE LOGIQUE

"Le volume de la description résultant de l'étape de conception statique étant beaucoup plus important que celui issu des spécifications fonctionnelles, il importe d'organiser les spécifications au sein d'une *architecture* faite de différents *modules*" (adapté de [Dub93]).

Avant de présenter trois principes qui peuvent nous aider à construire une *bonne* architecture, remarquons que l'architecture est dite *logique* car elle dénote une solution exécutable sur une machine virtuelle, abstraite contrairement à une architecture *physique* qui désignerait une solution exécutable sur une machine réelle.

1.1. STRUCTURATION

Ce principe prône l'introduction de fonctions intermédiaires : "diviser pour régner". Une fonctionnalité `F` tirée des spécifications fonctionnelles sera donc "éclatée" en plusieurs fonctions `F1,F2,...,Fn` de granularité plus fine. Le même processus sera appliqué aux fonctions et ainsi de suite jusqu'à obtenir un niveau de granularité suffisant.
Les branches terminales de l'arbre de décomposition sont des fonctions de création, de modification et de consultation des différents états. Ces états sont désignés par l'ensemble `{E1,E2,...,En}`. On y trouve aussi des fonctions qui s'occupent de l'interface homme-machine ainsi que des fonctions qui effectuent des appels *"hardware"* (appels à des primitives offertes par la machine).

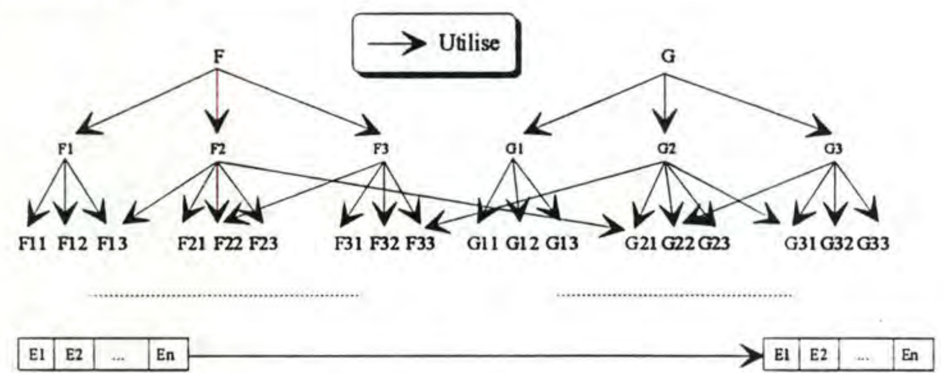


Figure 1 - Fonctions intermédiaires et relation Utilise

Les relations entre fonctions sont du type *utilise* [Par72a, Par72b]. Une fois encore, l'usage d'une telle relation permet d'éviter une redondance : si on suppose que, par exemple, les fonctions `F12` et `G33` sont identiques, le travail pour les développer ne sera pas fait deux fois.

1.2. ABSTRACTION

Les fonctions obtenues sont regroupées en différents niveaux selon leur "nature". Ainsi,
- au niveau 5, nous aurons les fonctions qui sont dérivées des fonctionnalités,
- au niveau 4, nous aurons les fonctions concernées par la gestion de l'interface H/M et
- au niveau 3, nous trouverons les fonctions traitant l'information persistante.

Citons, pour information, les deux derniers niveaux dont nous ne nous préoccuperons pas :
- niveau 2 : couche logiciels d'application (gestionnaire de BD, d'interface, etc.)
- niveau 1 : couche système opératoire

Il est clair que cette découpe tend à regrouper les fonctions selon la technologie qui leur sera associée plus tard.

Notons que l'architecture obtenue est du type hiérarchie "descendante" : un niveau offre des services aux niveaux supérieurs.

1.3. MODULARISATION

Au sein d'un niveau, les fonctions seront groupées en modules selon deux critères possibles [Ghe91] :

- orienté-traitement ou fonction : plusieurs fonctions sont dans un même module car elles font le même type de traitement, éventuellement sur différentes données.
  Ex : fonctions qui vérifient l'existence d'un patient, d'un service, etc.
  Un tel classement se justifie par le fait que, si des fonctions font plus ou moins la même chose, les moyens pour les mettre en oeuvre seront similaires.

- orienté-objet ou type abstrait : plusieurs fonctions sont ensemble car elles travaillent sur la même structure de données.
  Ex : fonctions qui créent, détruisent, modifient un patient.
  Le fait de localiser les fonctions travaillant sur une même structure présente, entre autres, l'avantage de pouvoir faire évoluer ces structures beaucoup plus facilement.

Ce principe tend à rendre l'architecture du logiciel facile à maintenir et à faire évoluer en cas de changements par exemple.

On peut alors se demander quel critère adopter. Une des grandes conclusions que nous avons tirées de notre stage est qu'il n'est pas de position meilleure que l'autre à tout point de vue et, de ce fait, qu'une approche combinant les deux positions est recommandée.

1.4. PLACE MÉMOIRE VS COMPLEXITÉ DES TRAITEMENTS

Il est des cas où, si l'on veut suivre les principes à la lettre, le développement de certaines fonctions sera "lourd" et inefficace.

Afin d'illustrer ce problème, nous nous proposons de prendre l'exemple du module `PLANNING`. Ce module regroupe différentes fonctions qui ont pour but d'allouer une prescription de soin en tenant compte des prescriptions prévues, des capacités des services qui proposent ces soins, ainsi que des fonctions qui renseignent sur le planning ou annulent une prescription déjà allouée.

Nous aurons un planning pour chaque couple (service de soins, prestation offerte par ce service). Ce planning stocke, pour chaque pause à venir, la liste des références des prescriptions de soins et leurs priorités ainsi que le nombre maximum de prescriptions que le service peut traiter, le nombre effectif de prescriptions prévues et le nombre de prescriptions venues en surcharge.

La structure de données sous-jacente est la suivante :

```
PLANNING_DT = LIST of PAUSE_CP
où
 PAUSE_CP =
  CP  [nbMax : NAT,
      nbEffectif : NAT,
      nbSurcharge : NAT,
      listePS : LISTE_PS]
 où
   LISTE_PS = LIST of
     CP [pS : PRESCR_SOIN,
        prior : NAT]
```

c'est à dire,

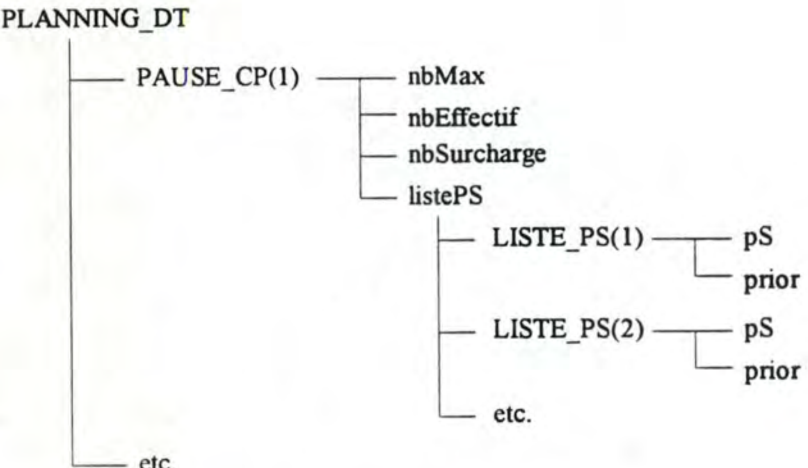


Figure 2 - La structure du planning

et nous avons un module (un objet) qui encapsule cet état :

```
object PLANNING
 state
    planning : PLANNING_DT;
    ...
 ...
endobject
```

Comme on peut s'en douter, la gestion de ce planning est capitale et complexe. Pour réaliser ses traitements, le planning doit faire appel à des fonctions se trouvant dans différents modules au niveau 5 et au niveau 3. De plus, puisque l'application peut être interrompue, il est nécessaire que l'information soit persistante donc ce module devrait se trouver au niveau 3.

Cette obligation est en conflit avec le principe d'abstraction (hiérarchie descendante) qui veut qu'un service d'un niveau i n'utilise que des services des niveaux i, i-1, i-2, etc. Dans notre cas, le problème pourrait être éludé en construisant le planning au début de l'application à l'aide d'une série de traitements, le planning étant maintenant au niveau 5 et respectant le principe d'abstraction. En effet, l'information présente dans le planning est tout à fait redondante avec l'information contenue dans d'autres modules au niveau 3.
Cependant, les traitements pour construire et gérer le planning en seront alourdis considérablement et donc il en sera de même pour l'efficacité de l'application.

La question qui se pose alors est la suivante : faut-il respecter à la lettre les principes qui régissent la méthode et ne pas tenir compte des exigences d'efficacité ? Cette question en soulève une deuxième qui est : faut-il, à ce stade du développement, se préoccuper de l'efficacité du logiciel ?

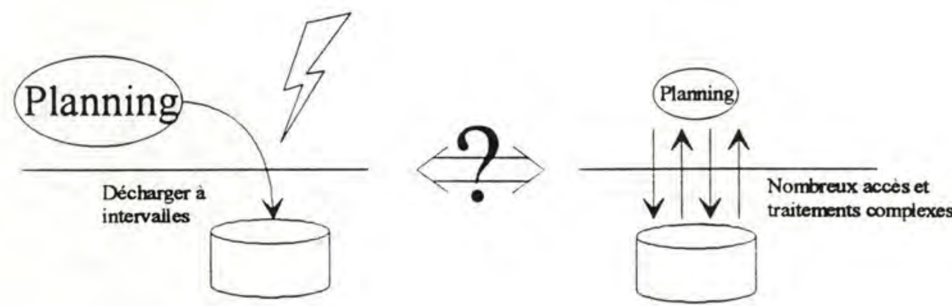


Figure 3 - Place mémoire vs complexité des traitements

Puisque nous nous dirigeons vers une solution, il nous semble que la réponse à la dernière question est "oui". En effet, selon [Bod89], *"il importe d'évaluer, le plus tôt possible dans le développement d'un projet, si la solution fonctionnelle envisagée est réalisable ou faisable par rapport aux ressources dont dispose l'organisation"*.

La réponse à la première question est moins évidente et n'est certainement pas tranchée. Un argument qui ferait pencher la balance en faveur du "oui" pourrait être le suivant : les principes énoncés tentent de guider le(s) concepteur(s) vers une "bonne" architecture, facile à maintenir et à faire évoluer. Si on ne les respecte pas ou si on tolère des exceptions, l'objectif ne sera pas atteint. Cependant, tout le monde conviendra que, même s'il est fondé, cet argument est trop strict et se révèle être purement "méthodologique".

Personnellement, compte tenu du problème, nous penchons plutôt pour une réponse négative. En effet, si nous devions réaliser le planning sans faire d'entorse aux principes, il nous semble que la spécification serait complexe et le risque d'erreur élevé, entraînant de ce fait une dégradation sensible des performances.

Cependant, il nous semble indispensable de répondre à cette question "au cas par cas" et de ne faire pencher la balance d'un côté ou de l'autre qu'après avoir évalué les deux solutions.

Remarquons enfin que si, dans le cas exposé ci-dessus, nous pouvons choisir de soit violer un principe soit d'alourdir le traitement, il est des cas où un tel choix n'est pas possible : il n'existe pas de solution qui respecte les trois principes. Cette impasse pourrait alors se révéler être l'indice d'une mauvaise modélisation. Argumenter une telle opinion dépasserait le cadre de ce mémoire. ▪

2. SPÉCIFICATION DES MODULES

Citons brièvement quelques langages que l'on pourrait utiliser pour mener à bien cette étape de conception statique : le langage *RAISE* [Rai92], la famille de langages *Larch* [Win85a, Win85b] et les langages basés sur le concept plus général de *types de données* ou *d'objets abstraits* [Ghe91].

Dans les langages que nous venons de citer, il n'existe pas de notion de temps d'exécution : le concept de fonction est purement mathématique et, dès lors, il n'y a pas de temps d'exécution lié à la fonction. De plus, nous posons l'hypothèse que le système développé est un système mono-utilisateur, mono-poste. C'est pourquoi cette première étape de la conception est qualifiée de *statique*.

Cependant, étant donné que nous nous dirigeons vers une solution, il peut être utile d'introduire un certain séquencement. Il s'agira d'un séquencement dit "séquencement de données" (⇨ *Data Flow*) par opposition au séquencement arbitraire.

Par exemple, supposons une spécification fonctionnelle telle qu'elle serait décrite par :

```
b=f(a) et c=g(b) et d=h(b)
```

alors, il convient de garder le séquencement de données suivant :

```
b=f(a) puis c=g(b) et d=h(b)
```

et d'éviter un séquencement arbitraire tel :

```
b=f(a) puis c=g(b) puis d=h(b)
```

On peut se demander pourquoi supposer un temps d'exécution nul alors que l'on sait pertinemment bien que ce n'est pas le cas. Les avantages de cette hypothèse peuvent être envisagés sous deux angles : l'angle théorique et l'angle pratique.

D'un point de vue théorique, si l'on considère que la ressource "processeur" est une ressource dont la vitesse de traitement est infinie (d'où le temps d'exécution nul), alors une fonction au sens informatique du terme devient une fonction au sens mathématique du terme et nous disposons d'un support théorique tel que, par exemple, la théorie des preuves [Goo79, Eri87]. Le développement d'un logiciel peut alors se faire de manière beaucoup plus cohérente : on passe de la logique des prédicats du premier ordre (dans l'étape de spécification fonctionnelle) aux fonctions mathématiques (dans l'étape de conception) [Fol92].

D'un point de vue pratique, l'hypothèse d'une ressource à capacité infinie évite de devoir prendre en compte des préoccupations relatives à la concurrence au sein du logiciel, problèmes requérants l'introduction de concepts tels que les *zones de mutuelle exclusion*, la *synchronisation*, etc. En effet, si un traitement s'effectue instantanément, il est impossible qu'un autre traitement vienne l'interrompre et donc, il ne faut pas se préoccuper de problèmes tels que la définition de zones critiques ou encore d'allocation à tour de rôle du "processeur", etc. ▪

Pour une illustration plus rigoureuse de cette hypothèse, le lecteur est invité à consulter [Win86].

Nous allons voir que nous ne devons modifier que très peu le langage OBLOGlight afin qu'il supporte cette phase de conception statique.

1. MODIFICATIONS

Dans le langage OBLOGlight, la relation *utilise* peut être implémentée via deux mécanismes :
- le *call & return* : un événement appelle un événement d'un autre objet et lui passe en paramètres un ou plusieurs événements auxquels il doit rappeler (adresses de retour : *stored event*). Cet objet peut procéder à quelques opérations et rappeler ensuite à une des adresses de retour.

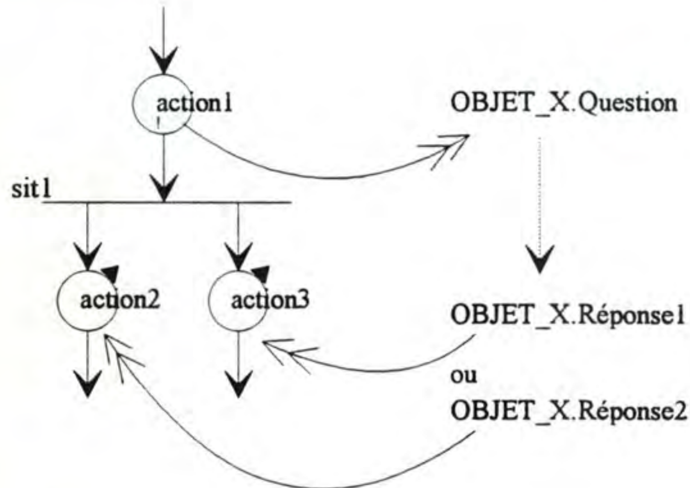


Figure 4 : Le mécanisme du "call & return"

Remarquons que l'objet appelé peut ne pas rappeler : il s'agit typiquement d'opérations qui ne nécessite pas de réponse et que l'on peut lancer puis ne plus s'en préoccuper. Dans ce cas, on parlera du mécanisme *call*.

- l'*observation* : mécanisme qui permet de consulter la valeur d'un attribut `attr` d'un objet `X` à partir d'un objet `Y` de façon tout à fait transparente. En effet, l'attribut observé peut être défini par une formule (requête, somme, etc.) qui sera calculée à chaque observation de cet attribut. Ainsi, `attr` sera par exemple défini par :

```
attr : LIST of Y
attr = ALL [Y|Y.refAObjetZ.uneObsDansObjetZ=TRUE]

(uneObsDansObjetZ est une observation de Z)
```

Le problème vient du fait que le mécanisme *call & return* introduit une *synchronisation* entre événements : l'objet qui appelle doit attendre que l'objet X le rappelle à l'événement Event2 ou Event3 pour poursuivre son exécution. Cette synchronisation induit la notion de temps d'exécution (non nul), ce qui est contraire à nos exigences.

Afin de résoudre ce problème, nous nous proposons d'expliquer par un exemple comparé, la façon détaillée dont fonctionne un appel OBLOG.

Supposons un module `X` quelconque mais tel que, dans sa spécification, il fasse appel à l'opération `MàJ` de l'instance `cpt` du module `CPTR`. Supposons encore que l'opération `MàJ` du module `CPTR` soit définie par `val1:=val1+1` et `val2=val2+2`, `val1` et `val2` faisant partie de l'état du module `CPTR`. Si l'on représente cette spécification d'une manière classique, cela donne la partie supérieure de la figure 5. La représentation OBLOG de cette spécification se passe de commentaires.

Cependant, il est intéressant de remarquer que l'appel à une action peut être décomposé en deux parties que nous qualifierons (intentionnellement) de *statique* et de *dynamique*. La partie statique consiste en la seule définition de l'effet de l'action c'est-à-dire la définition de la fonction, de l'opération de mise-à-jour `MàJ`. La partie dynamique peut, à son tour, être scindée en deux parties : une requête de mise à jour envoyée par l'objet `X` et une réponse fournie par l'objet `CPTR`. Cette réponse indique seulement que la mise à jour a été effectuée.

Pour la conception statique, nous ferons l'hypothèse que l'appel à une action est composé de la seule partie statique, ce qui est représenté par la relation *utilise* qui remplace la requête de mise-à-jour et la réponse.

Cette nouvelle sémantique "colle" tout à fait avec nos exigences et, comme le montre l'exemple comparé, seule la représentation graphique change entre les deux parties de la figure 5.

En résumé, lors de la conception statique, nous définissons la fonction/l'opération/l'action et lors de la conception dynamique, nous nous donnerons en plus le moyen de l'invoquer via n'importe quel mécanisme (appel d'action, de procédure, etc.) ▪

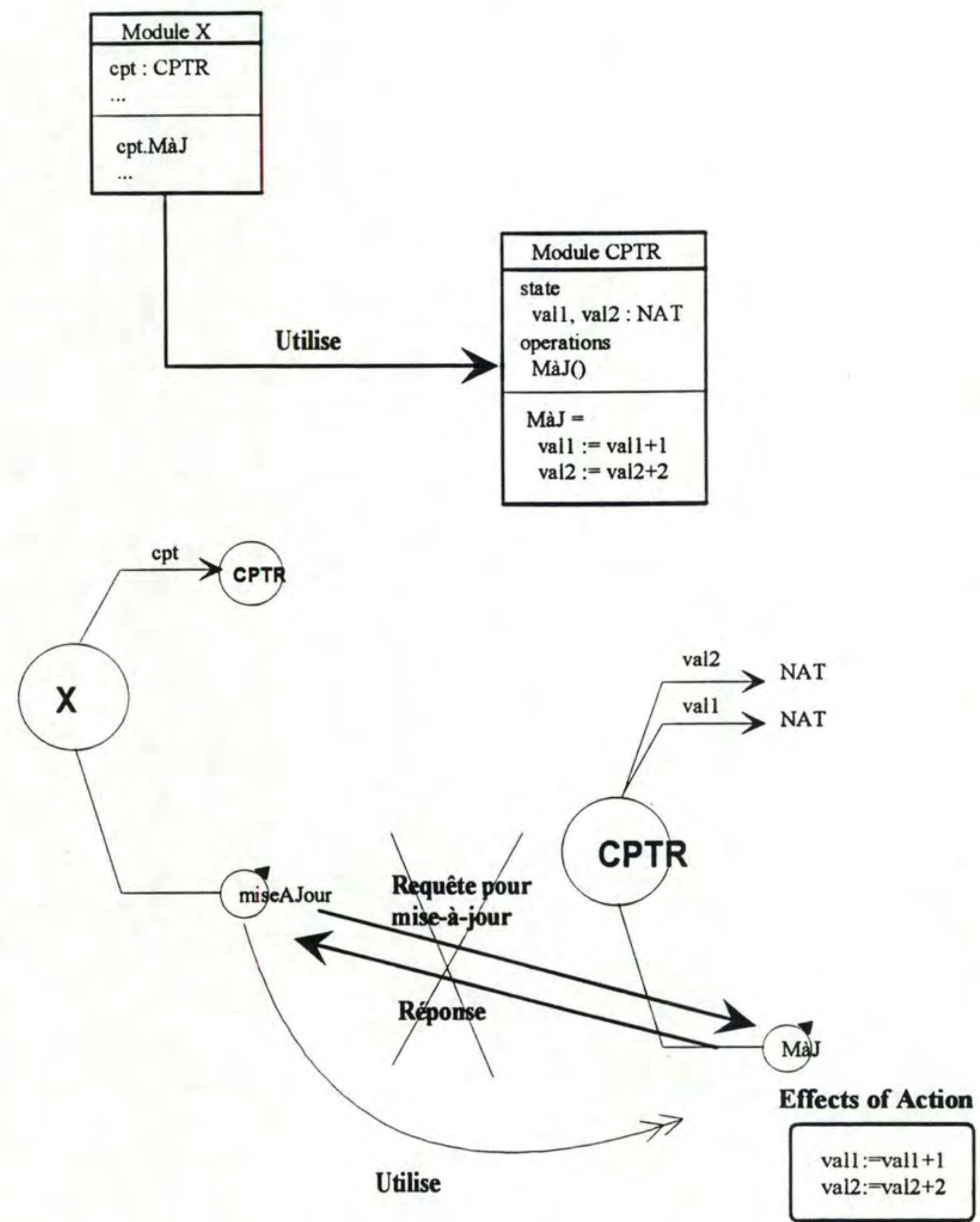


Figure 5 - Le mécanisme du call (en détail)

Une autre restriction du langage OBLOGlight est l'interdiction de déclencher plusieurs événements en parallèle au sein du même objet[1].

Afin de satisfaire aux exigences de la phase de conception statique, nous proposons de permettre ce type de parallélisme à condition qu'il n'y ait aucune dépendance de données, auquel cas le parallélisme n'a pas de sens.

Par la suite, il suffira au concepteur d'introduire un séquencement arbitraire afin d'être conforme au langage OBLOGlight utilisé dans la phase de conception dynamique. ▪

---

1 Par contre, plusieurs objets peuvent être "exécutés" en parallèle.

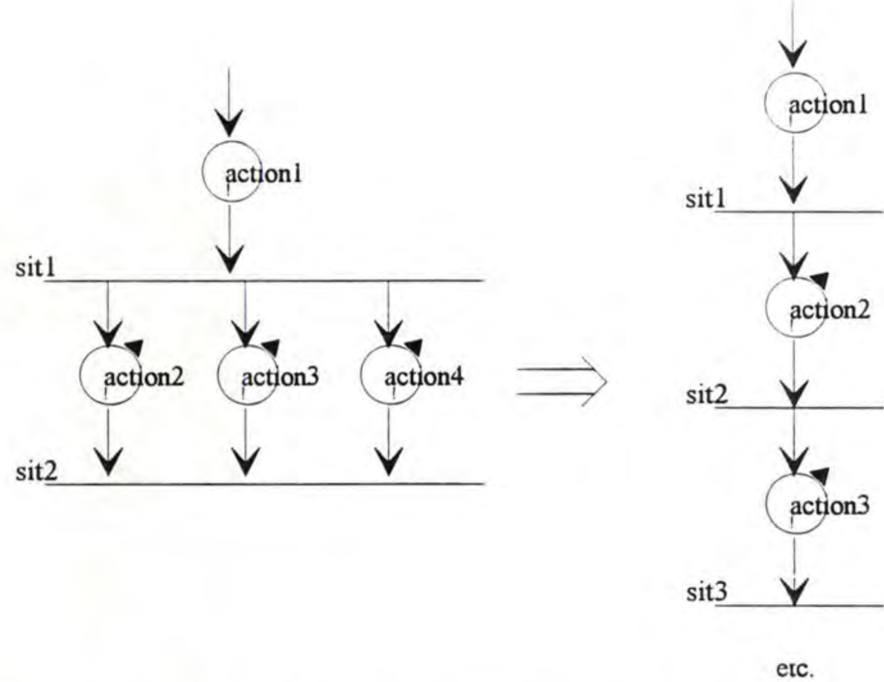


Figure 6 : Introduction d'un séquencement arbitraire

2. TRADUCTION

En ce qui concerne le raffinement des fonctions et l'introduction de fonctions intermédiaires, étant donné que le langage OBLOGstatic est utilisé pour la première fois dans le développement, le gros du travail est laissé au concepteur. Cependant, sa tâche sera facilitée par la présence de l'*éditeur de dialogue* pour la conception de l'interface et du concept de *type de données* prédéfini ou défini par l'utilisateur pour le format des données.

Le concepteur trouvera dans la notion d'*unité* une solution possible à la découpe en niveaux. Cependant, il devra gérer lui-même le bon emplacement des objets : un objet relatif à l'interface par exemple peut se trouver dans une quelconque unité sans que cela ne pose de problèmes au système OBLOG.

Etant donné que le langage OBLOGstatic est orienté-objet, le travail sera facilité si l'on adopte le critère de modularisation O-O mais il ne sera pas impossible si l'on adopte le critère fonctionnel : le langage OBLOGstatic supporte les deux approches.

3. AVANTAGES D'OBLOG-STATIC

Avant de continuer, on pourrait se demander pourquoi utiliser un autre langage que la logique des prédicats du premier ordre couplé aux types abstraits dans la phase de conception statique.

Voici les trois raisons principales :

- le langage OBLOGstatic permet de dépasser les limitations des types abstraits quant aux manipulations d'états : lorsqu'une fonction effectue des modifications successives sur un état, celui-ci doit être désigné par des termes tels que état', état", etc., qui représentent différentes instances de cet état. En OBLOGstatic, un état est représenté par un objet et ses différentes évolutions décrivent sa "vie". Il n'est pas nécessaire de le désigner par différents termes, ce qui allège considérablement l'écriture.
- la spécification d'une fonction peut se faire par un enchaînement d'événements et d'observations exprimés dans le diagramme comportemental de l'objet qui représente cette fonction. Cette façon d'expliciter le comportement interne clarifie la spécification en ce qu'elle trace une sorte d'arbre de décision plus facile à comprendre qu'une suite de clauses de prédicats.
- nous avons vu que le langage OBLOGlight se plie facilement aux exigences de la phase de conception statique. Or, avec ce seul langage et avec l'outil OBLOG-CASE, nous pouvons couvrir la suite des phases de développement du logiciel, ce qui est un avantage considérable.

Il existe certainement d'autres langages qui présentent les mêmes avantages, mais notre choix d'adopter OBLOG a été guidé par le fait que nous le connaissions bien suite à notre stage réalisé chez ESDI S.A. au Portugal.

IV. LE DÉVELOPPEMENT

1. LE CADRE

Le développement du cas MS-C entend être non seulement une illustration des possibilités du langage OBLOGlight/static mais aussi d'une façon d'utiliser les différents outils que sont
- l'*éditeur de dialogues* et
- l'*éditeur de rapports*.

Pour plus de détails sur ces outils, le lecteur est invité à consulter les sections II.5.3 (éditeur de dialogues) et III.5.1 (éditeur de rapports) du chapitre 4.

2. LES SPÉCIFICATIONS

Dans cette section, nous allons présenter quelques extraits des spécifications du cas de l'hôpital. Le développement complet se trouve en annexe.

Nous commencerons par présenter et expliquer l'architecture globale et les trois niveaux. Ensuite, nous exposerons la philosophie générale de chaque niveau que nous illustrerons par un exemple détaillé. Suivront alors quelques exemples qu'il nous a semblé intéressant de présenter.

2.1. ARCHITECTURE GLOBALE

2.1.1. Schéma

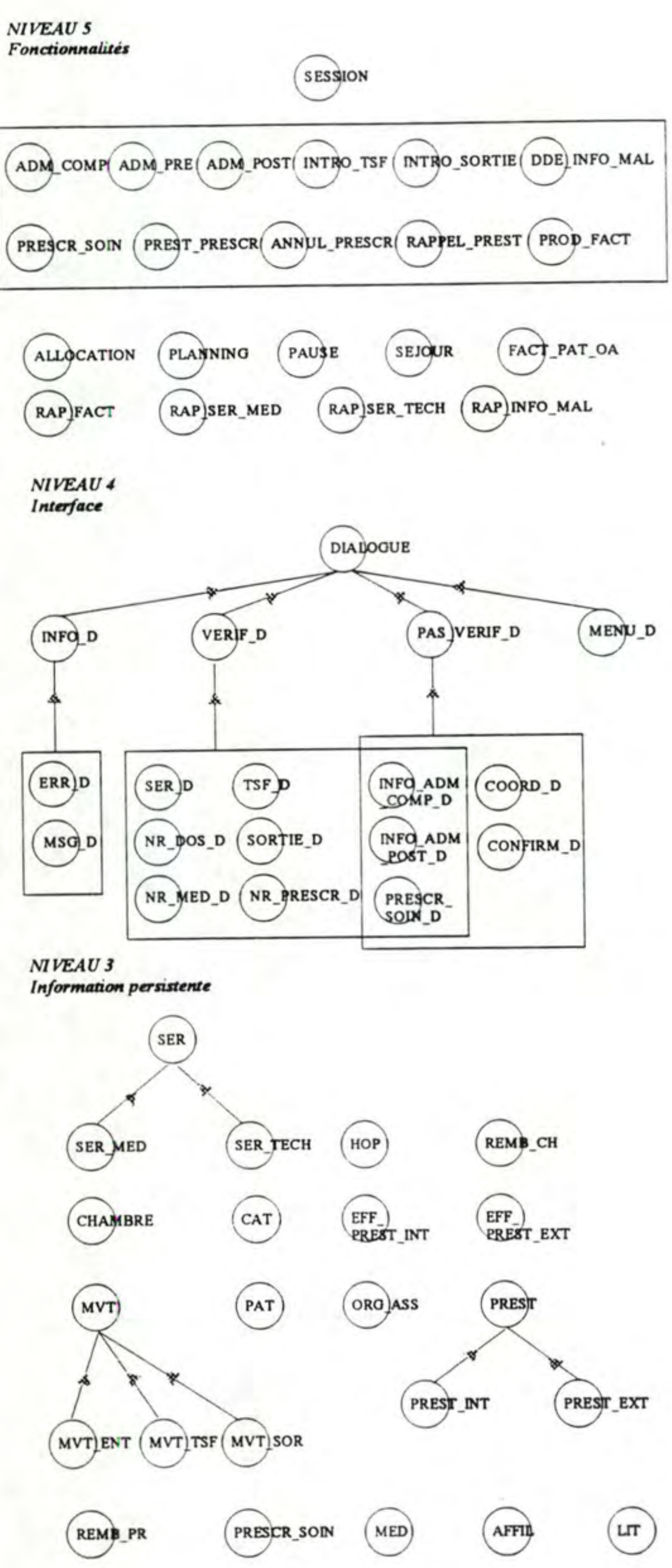


Figure 7 - Architecture globale

2.1.2. Commentaires

A la figure 7, on peut voir[2] que, pour notre architecture, nous avons adopté :

- une unité pour chaque niveau d'abstraction (niveaux 5, 4 et 3)
- au niveau 5, un objet de type *OBL* pour chaque fonction des spécifications fonctionnelles (`Admission_Complète`, `Production_Facture`, etc.), quelques objets qui servent de "support" à ces fonctions (`Planning`, `Pause`, `Allocation`, les objets de type rapport dont est préfixé par `RAP`, etc.) et un objet `Session Manager` qui gère l'exécution des différentes fonctions selon les souhaits des utilisateurs
- au niveau 4, un objet de type *DBX* (interface) pour chaque boîte de dialogue. Nous avons utilisé la relation d'héritage car la communauté des objets de ce niveau peut être classée en 4 catégories (non nécessairement exclusives) : les boîtes d'information qui affichent un message à l'utilisateur, les boîtes de dialogue qui ne vérifient pas l'existence des informations qu'elles saisissent (un numéro de téléphone, une adresse), celles qui vérifient cette information (un numéro de dossier, de lit, etc.) et les différents menus qui sont proposés en fonction du service qui l'utilise (l'accueil, la comptabilité, etc.).
- au niveau 3, un objet de type *TBL* (persistant) pour chaque type d'entité du modèle entités-associations présenté dans le cahier des charges et pour chaque type d'association qui possède des attributs.

Cette architecture est fortement influencée par notre stage au Portugal. En effet, lorsque nous avons tenté de spécifier le sous-système de l'hôpital composé des fonctionnalités admission-transfert-sortie, notre première idée de l'architecture était semblable à celle représentée à la figure 7. Cependant, le langage dont nous disposions à l'époque était une version réduite du langage OBLOGkernel tel que présenté au chapitre 4, ce qui nous amené à "dégrader" nos spécifications et à modifier notre architecture.

Les nouveaux concepts et mécanismes présents dans le langage OBLOGlight (la prochaine évolution du langage OBLOGkernel) nous permettent de conserver notre architecture intacte et de disposer de facilités de modélisation non négligeables : il s'agit principalement de l'*observation*, du concept de *métaclasse*, des *attributs dérivés*, des *clés*, de la séparation entre *interface* et *corps* d'une spécification, des nouveaux types d'objets *rapport* et *interface*, du concept d'*objet interne* et des relations de *spécialisation/généralisation* et d'*association*. Pour plus de détails à propos de ces concepts, consulter le chapitre 4.

[2] Le type d'objet n'a pas été indiqué afin de ne pas surcharger la figure.

2.2. LES NIVEAUX EN DÉTAIL

2.2.1. Le niveau 5

Schéma

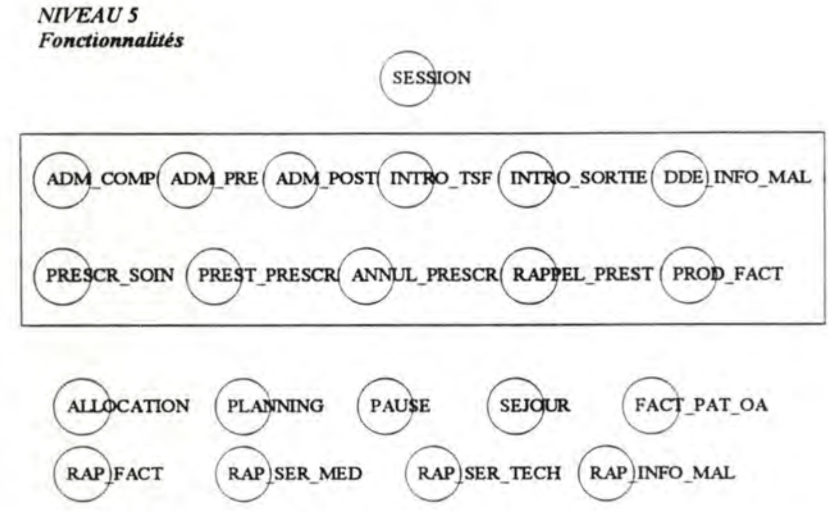


Figure 8 - Le niveau 5

Commentaires

Comme il a été dit plus haut, on peut trouver deux catégories d'objets dans ce niveau : les objets qui implémentent une fonctionnalité des spécifications fonctionnelles (représentés dans le rectangle) et les objets qui implémentent des services utilisés par la première catégorie d'objets (représentés en dehors du rectangle). Les objets de la deuxième catégorie peuvent à nouveau être classés en deux catégories : les objets OBLOG "classiques" (de type OBL - première couche) et les objets rapports (de type RPT - deuxième couche).

Dans le tableau suivant, nous donnons le critère de modularisation adopté par chaque objet.

Objet	Critère
ADM_COMP	Orienté-traitement
ADM_PRE	Orienté-traitement
ADM_POST	Orienté-traitement
INTRO_TSF	Orienté-traitement
INTRO_SORTIE	Orienté-traitement
DDE_INFO_MAL	Orienté-traitement
PRESCR_SOIN	Orienté-traitement
PREST_PRESCR	Orienté-traitement
ANNUL_PRESCR	Orienté-traitement
RAPPEL_PREST	Orienté-traitement
PROD_FACT	Orienté-traitement
ALLOCATION	Orienté-objet
PLANNING	Orienté-objet

PAUSE	Orienté-objet
SEJOUR	Orienté-objet
FACT_PAT_OA	Objet interne
RAP_FACT	Orienté-traitement
RAP_SER_MED	Orienté-traitement
RAP_SER_TECH	Orienté-traitement
RAP_INFO_MAL	Orienté-traitement

Tous les objets sont de cardinalité simple (une seule instance au maximum) sauf l'objet Planning qui est de cardinalité multiple.

Morceaux choisis

Nous allons illustrer l'implémentation que nous avons réalisée pour ce niveau au travers de deux extraits : le premier présente l'objet-fonctionnalité Introduction-transfert (dont l'abréviation est `Intro-tsf`) et le second l'objet de type rapport `Rap-ser-med` réalisant un rapport imprimé des prestations de soins à effectuer par tous les services médicaux.

OBJET INTRO-TSF

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - litOrig, litDest : LIT
- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")
- **Actions**
  - * start
  - ! nrDosD (pat : PAT)
    >> NR_DOS_D.open (pat)
    ⇒ pat := nrDosD.pat
  - ! tsfD (litOrig, litDest : LIT, med : MED)
    >> TSF_D.open
    ⇒ litOrig := tsfD.litOrig
    ⇒ litDest := tsfD.litDest
    ⇒ med := tsfD.med
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPasLitOrig
    IS putError (code)
    := code = TSF_NOT_IN_LIT_OR
  - !+ putErrorLitDestOcc
    IS putError (code)
    := code = TSF_LIT_DEST_OCC

- !+ putErrorPatPasMal
  IS putError (code)
  := code = TSF_PAT_NOT_MAL
- ! inLitOrig
- ! LitDestOcc
- ! inUrgence
- ! notInUrgence
- ! litDestLibre
- ! admPost
  >> ADM_POST.start (pat)
- ! creeMvtTsf
  >> mvt_tsf.birth (litOrig, litDest, pat, med, TODAY(), NOW())
  >> allocation.déplacerPrescr (litOrig.serDeLit, litDest.serDeLit, pat)
- !+ putMsg (code : MSG_CODE_DT)
  >> MSG_D.open (code)
  := code = TSF_OK

BEHAVIOUR

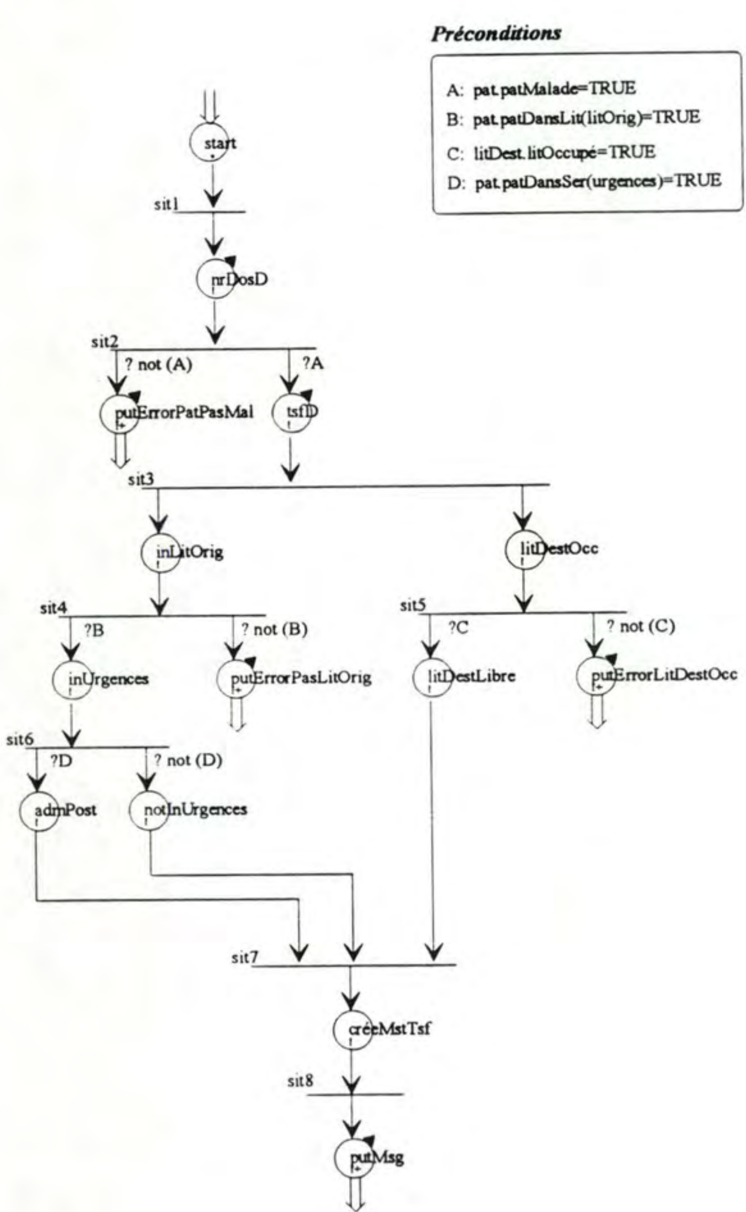
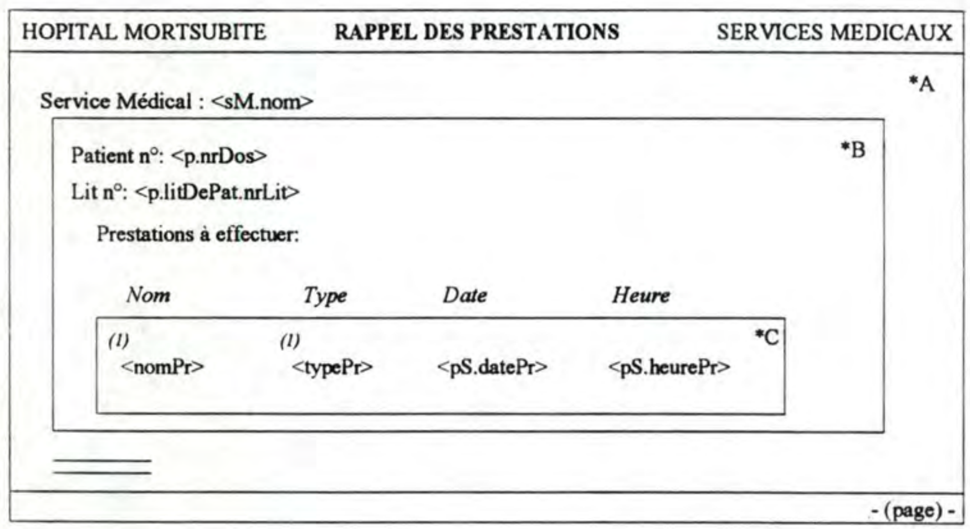


Figure 9 - Le diagramme de comportement de la classe d'objets INTRO-TSF

Remarquons simplement dans cette spécification la séparation (très utile) entre interface et corps d'un objet ainsi que la présence, dans le corps de l'objet, d'un attribut dérivé `Urgences` qui représente l'instance de l'objet `SER_MED` dont le nom identifiant est "`Urgences`" et dont la valeur est obtenue en invoquant la *clé* de l'objet `ser_med`.

Rappelons aussi que le symbole ">>" désigne un appel d'action, le symbole "⇒" une mise-à-jour d'attribut et le symbole ":=" une instanciation d'attribut d'événement.

OBJET RAP-SER-MED



A : ALL [sM : SER_MED]
B : ALL [p : PAT | p.patDansSer(sM)=TRUE]
C : ALL [pS : PRESCR_SOIN |
pS.subitePar=p AND
CLASS_OF(pS.pourPrest)="PREST_INT" AND
pS.enAttente=TRUE AND
pause.typePause(pS.datePr,pS.heurePr)=EN_COURS]

Figure 10 - Le rapport RAP-SER-MED

La logique de ce rapport est la suivante :

∀ service médical
∀ patient de ce service médical
∀ prescription de soin de ce patient
"en attente",
prévue pour la pause courante et
prescrivant une prestation interne
⇒ imprimer les nom, type, date et heure de cette prestation.

2.2.2. Le niveau 4

Schéma

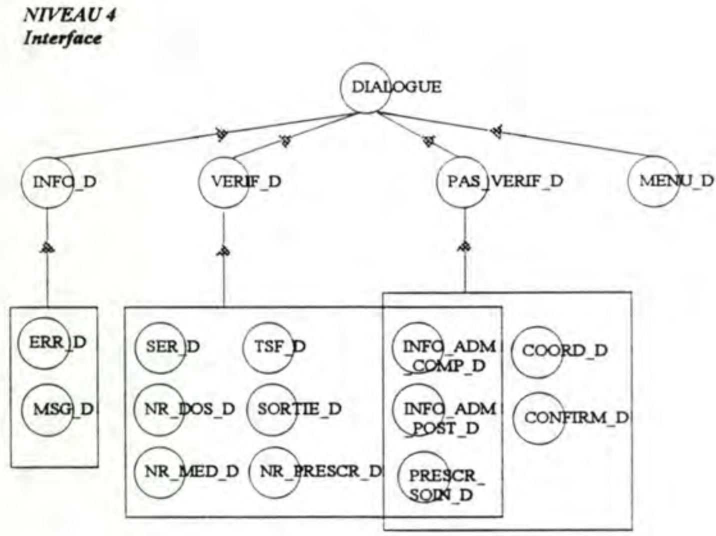


Figure 11 - Le niveau 4

Commentaires

Mise à part l'utilisation massive de l'outil éditeur de dialogue, ce niveau est caractérisé essentiellement par l'utilisation de la relation d'héritage (cfr. chapitre 4). En effet, comme on peut le constater sur la figure 11, l'ensemble des objets de type dialogue est issu d'un objet que nous avons (évidemment) nommé DIALOGUE. Cet objet définit les caractéristiques communes à tous les objets interface c'est-à-dire une naissance, une mort, un titre de fenêtre. Outre ces caractéristiques relativement "banales", nous aurions pu y ajouter une position à l'écran, une taille, une couleur, etc. Nous ne l'avons pas fait pour garantir une certaine lisibilité au cas.

Sur cet objet générique viennent se greffer d'autres catégories d'objets qui ajoutent leurs particularités telles que, par exemple, le fait qu'il s'agit d'un menu, d'une boîte de message ou d'une boîte de dialogue. Une fois toutes les catégories définies, on peut enfin définir les objets "réels". En effet, les catégories que nous avons définies auparavant (les objets en dehors des rectangles sur la figure 11) sont quelque peu artificielles en ce qu'elles ne font que clarifier l'architecture du niveau sans être absolument nécessaires.

Signalons que ce niveau adopte le critère de modularisation orienté-objet. On s'en convaincra en remarquant que, à l'intérieur d'un objet, se retrouvent par exemple toutes les fonctions qui doivent assurer des contrôles syntaxiques sur les données saisies.

En raison de l'utilisation importante de la relation d'héritage dans ce niveau, nous n'exposerons pas d'extrait : il faudrait présenter une ou deux spécifications avant d'avoir les éléments nécessaires pour comprendre une spécification "réelle". Le lecteur intéressé pourra consulter l'étude de cas en annexe pour un développement détaillé.

2.2.3. Le niveau 3

Schéma

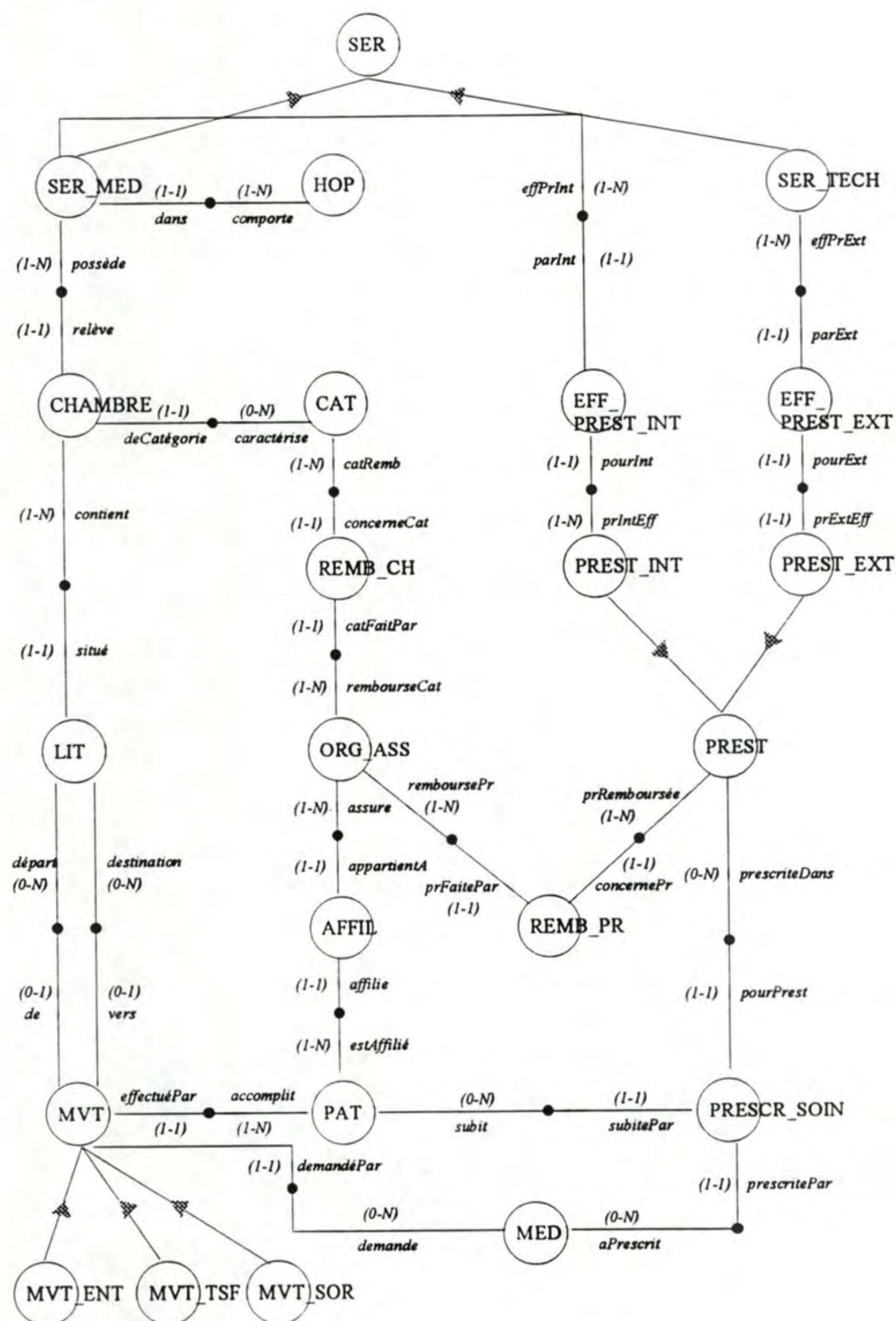


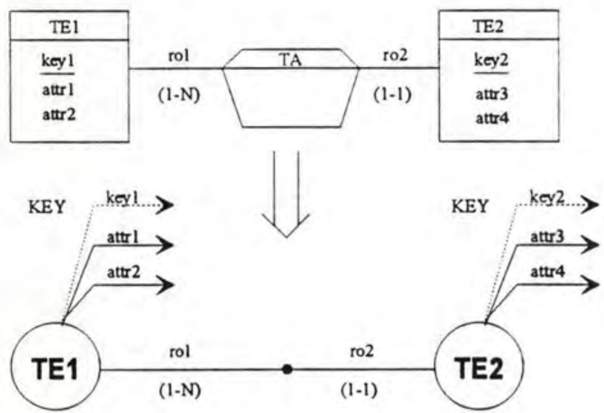
Figure 12 - Le niveau 3

Commentaires

Ce niveau est caractérisé par l'utilisation de la relation d'association (cfr. chapitre 4) entre classes d'objets ainsi que du mécanisme d'observation. La séparation entre interface et corps d'un objet trouve ici sa pleine signification : beaucoup de services offerts par un objet sont en fait définis par un "calcul" (dérivation) sur le schéma, ce calcul utilisant des valeurs d'attributs internes à l'objet ou d'autres observations et étant totalement caché pour les autres objets. Le concept de *clé identifiante* se révèle être d'un grand secours également.

Avant de présenter des extraits des spécifications, nous allons exposer brièvement la façon par laquelle nous avons traduit le schéma entités-associations en classes d'objets OBLOG.

Etant donné que le schéma E/A dont nous disposons au sortir de l'étape spécification fonctionnelle utilise des concepts relativement simples, nous nous contenterons d'un schéma pour expliquer la traduction que nous avons réalisée.



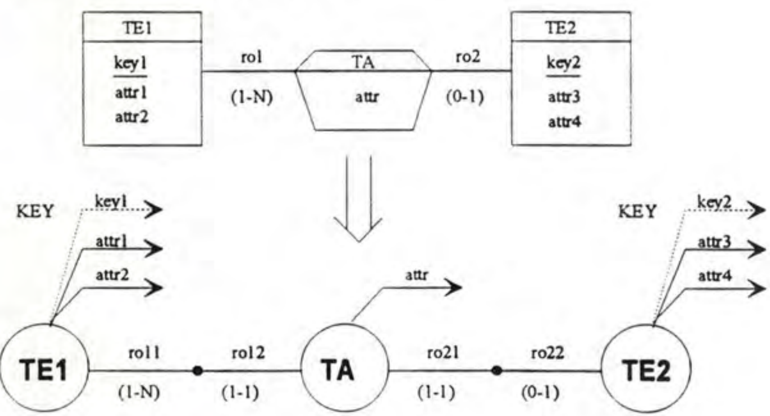


Figure 13 - Traduction E/A - OBLOG

On ne manquera pas de remarquer la similitude de cette traduction avec celle opérée à partir du modèle E/A vers le modèle MAG [Hai86].

Morceaux choisis

Nous avons choisi de présenter les objets `PATIENT` et `MOUVEMENT` (désignés respectivement par les abréviations `PAT` et `MVT`) car ils constituent un échantillon représentatif des classes d'objets spécifiées dans ce niveau 3.

OBJET MVT

INTERFACE

- **Observations**
  - nrMvt : NAT
  - dateMvt : DATE
  - heureMvt : TIME
  - typeMvt : TYPE_MVT_DT
  - *de, vers : LIT*
  - demandéPar : MED
  - effectuéPar : PAT
  - $mvtExistant (nrMvt : NAT) : BOOL

- **Keys**
  - $mouvement (nrMvt : NAT) : MVT
    ? $mvtExistant (nrMvt)
- **Events**
  - * birth (p : PAT, med : MED, date : DATE, heure : TIME)

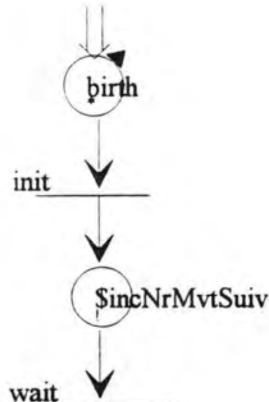
BODY

- **Attributes**
  - nrMvt, dateMvt, heureMvt, typeMvt, demandéPar, effectuéPar, *de, vers*
  - $nrMvtSuiv : NAT

- **Derivations**
  - $mouvement (nrMvt)
    := ONE[m:MVT | m.nrMvt = nrMvt]
  - $litExistant (nrMvt)
    := EXISTS[m:MVT | m.nrMvt = nrMvt]

- **Actions**
  - * birth (p, med, date, heure)
    ⇒ nrMvt := $nrMvtSuiv
    ⇒ dateMvt := birth.date
    ⇒ heureMvt := birth.heure
    ⇒ effectuéPar := birth.pat
    ⇒ demandéPar := birth.med
  - ! $incNrMvtSuiv
    ⇒ $NrMvtSuiv := $nrMvtSuiv+1

BEHAVIOUR



Il est intéressant de noter que cet objet est un sur-type des objets `MVT-ENT`, `MVT-SOR` et `MVT-TSF`. Seules quelques caractéristiques (telles que la participation aux associations *vers* et/ou *de* par exemple) sont propres à ces objets, ce qui motive l'emploi d'un objet générique contenant les caractéristiques communes à tous ces objets. On pourra trouver en annexe la spécification complète des classes d'objets `MVT-ENT`, `MVT-SOR` et `MVT-TSF`.

Remarquons l'emploi de précondition sur une observation (ici, une observation particulière : la clé identifiante) : il est nécessaire de satisfaire à cette condition si l'on veut obtenir le résultat de l'observation.

Notons aussi l'utilisation de l'attribut `$nrMvtSuiv` défini sur la *méta-classe* : le numéro du mouvement suivant (numéro identifiant) est en effet du ressort de la population et non pas d'une instance particulière de la classe. Le même raisonnement peut être appliqué à l'action `$incNrMvtSuiv`.

OBJET PAT

INTERFACE

- **Observations**
  - nrDos : NAT
  - coord : COORD_DT
  - nom, prenom : STRING
  - dateNaiss : DATE
  - adrPat, tél : STRING
  - etatCivil : ETAT_CIVIL_DT
  - sexe : SEX_DT
  - estAffilié : AFFIL
  - subit : LIST of PRESCR_SOIN
  - accomplit : LIST of MVT
  - $patExistantParCoord (coord : COORD_DT) : BOOL
  - $patExistantParNrDos (nrDos : NAT) : BOOL
  - patDansSer(serMed : SER_MED) : BOOL
  - oADePat : ORG_ASS
  - listeMvtDates (dateD, heureD, DateF, heureF) : CP[listesMvtDates : LIST of MVT, nbHosp : NAT]
  - listePrescrDates (dateD, heureD, DateF, heureF) : LIST of PRESCR_SOIN
  - patMalade : BOOL

- litDePat : LIT
- patDansLit (lit : LIT) : BOOL

- **Keys**
  - $patParCoord (coord : COORD_DT) : PAT
    ? $patExistantParCoord (coord)
  - $patParNrDos (nrDos : NAT) : PAT
    ? $patExistantParNrDos (nrDos)

- **Events**
  - * birth (coord : COORD_DT)
  - fixeAdr (adr : STRING)
  - fixeTel (tel : STRING)
  - fixeEtatCivil (etatCivil : ETAT_CIVIL_DT)
  - fixeSexe (sexe : SEX_DT)

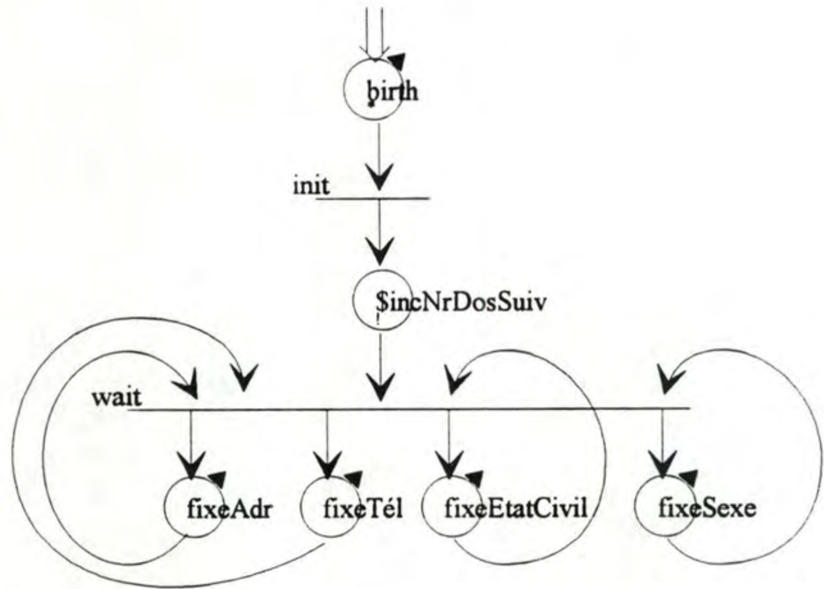
BODY

- **Attributes**
  - nrDos, coord, adr, etatCivil, sexe, tel, $nrDosSuiv : NAT

- **Derivations**
  - estAffilié
    := ONE[aff : AFFIL | aff.affilie=SELF]
  - subit
    := ALL[pS : PRESCR_SOIN | pS.subiePar = SELF]
  - accomplit
    := ALL[m : MVT | m.effectuéPar = SELF]
  - nom, prénom, dateNaiss
    := coord.nom, .prénom, .dateNaiss
  - $patExistantParCoord/NrDos (coord/nrDos)
    := EXISTS[p : PAT | p.coord/nrDos = coord/nrDos]
  - patDansSer (serMed)
    := SELF.litDePat.serDeLit = serMed
  - oADePat
    := SELF.estAffilié.appartientA
  - listeMvtDates (...)
    := listeMvtDates := ALL[m : MVT | m in SELF.accomplit AND dateF >= m.dateMvt > dateD AND heureF >= m.heureMvt > heureD] SORTED ON (dateMvt, heureMvt)
    := nbHosp := #(ALL[m : MVT | m in listeMvtDates AND (CLASS_OF (m) = "MVT_ENT" OR CLASS_OF (m) = "MVT_SOR")]) DIV 2
  - listePrescrDates (...)
    := listePrescrDates := ALL[pS : PRESCR_SOIN | pS in SELF.subit AND dateF >= pS.datePr > dateD AND heureF >= pS.heurePr > heureD AND pS.enAttente= FALSE] SORTED ON (datePr, heurePr)
  - patMalade
    := EXISTS[l : LIT | l.litOccupéPar = SELF]
  - litDePat
    := ONE[l : LIT | l.litOccupéPar = SELF]
  - patDansLit (lit)
    := lit.litOccupéPar = SELF
  - $patParCoord/NrDos (coord/nrDos)
    := ONE[p : PAT | p.coord/nrDos = coord/nrDos]

- **Actions**
  - * birth (coord)
    ⇒ coord := birth.coord
    ⇒ nrDos := $nrDosSuiv
  - $incNrDosSuiv
    ⇒ $nrDosSuiv := $nrDosSuiv + 1
  - fixeAdr (adr)
    ⇒ adr := fixeAdr.adr
  - fixeTel (tel)
    ⇒ tel := fixeTel.tel
  - fixeEtatCivil (EtatCivil)
    ⇒ EtatCivil := fixeEtatCivil.EtatCivil
  - fixeSexe (sexe)
    ⇒ Sexe := fixeSexe.Sexe

BEHAVIOUR



Nous finirons ce chapitre en demandant au lecteur d'observer la puissance d'expression du langage OBLOGstatic au travers de cette spécification. Il est remarquable de savoir que ce langage possède aussi une sémantique opérationnelle.

Chapitre 6
Conception dynamique

CONCEPTION DYNAMIQUE

I. INTRODUCTION

Nous allons développer ici la conception dynamique. Pour ce faire, nous allons d'abord exposer les objectifs de cette étape (section II). Nous verrons ensuite comment la concurrence est supportée par le langage OBLOGlight (section III). Nous terminerons par l'examen des modifications apportées au cas (section IV).

II. OBJECTIFS DE LA CONCEPTION DYNAMIQUE

Au fur et à mesure que nous développions le logiciel, nous avons posé une série d'hypothèses qui avaient pour objectif de nous faciliter la tâche de conception. Lors de la phase de conception statique, nous avions posé l'hypothèse d'un système mono-utilisateur et d'une description statique à l'aide du concept de fonction mathématique. Ceci nous avaient permis de ne pas tenir compte de l'aspect distribué et concurrent du logiciel pour nous concentrer sur l'architecture de celui-ci.

Il est maintenant temps de lever ces hypothèses. Par ailleurs, étant donné que la conception dynamique doit fournir une solution à tous les aspects du cahier des charges, des éléments de ce cahier que nous avions "oubliés" vont réapparaître. Nous pensons par exemple à l'aspect organisationnel que nous avons partiellement supprimé : les services vont devenir des unités de traitement de l'information, des fonctionnalités vont y être exécutées.

Le point crucial à gérer est l'aspect concurrent du logiciel. Pour le cas MortSubite, il pourrait être décrit comme suit :
- les services effectuent des fonctionnalités de manière concurrente : par exemple, au même moment, le service "accueil" exécute la fonctionnalité Admission, les services de soins "Cardiologie", "Chirurgie" et "Urgences" demandent chacun un transfert, la comptabilité lance la fonctionnalité Facturation et le service technique "Biologie" preste une prescription de soin.
- ces fonctionnalités accèdent à des ressources (lits, rapports d'impression, prescriptions de soins, etc.) de manière concurrente.

“Un des problèmes-clé de la conception d'un logiciel concurrent est d'assurer la cohérence des données [ressources] qui sont partagées par les modules concurrents.” (Traduit de [Ghe91]).

Afin d'illustrer ce problème, prenons l'exemple d'un lit. Supposons que, pour attribuer ce lit à un patient `pat`, la fonction `fixeLitOccupéPar(pat)` soit définie. Cette fonction est soumise à un test préalable qui assure que le lit soit libre (`litLibre?`).
Supposons encore que deux (ou plusieurs) fonctionnalités `F1` et `F2` demandent d'attribuer ce lit à leur patients respectifs (`pat1` et `pat2`). Nous pourrions spécifier ces fonctionnalités comme suit :

```
F1
IF lit.litLibre?
THEN lit.fixeLitOccupéPar(pat1)

F2
IF lit.litLibre?
THEN lit.fixeLitOccupéPar(pat2)
```

Malheureusement, cela ne suffit pas pour assurer que l'objet lit sera accédé de manière cohérente. En effet, imaginons que la fonctionnalité `F1` accède à la fonction `fixeLitOccupéPar`, la condition `litLibre?` est vraie et elle entre dans la branche `THEN`. Cependant, avant que `F1` ait eu le temps d'exécuter la suite, la fonctionnalité `F2` teste la condition `litLibre?`, la trouve vraie, elle aussi, et exécute directement la branche `THEN`.
Lorsque la fonctionnalité `F1` va tenter d'exécuter la branche `THEN`, l'objet lit aura atteint un *état illégal* : le lit est occupé et pourtant l'objet accepte de l'allouer à un autre patient.

Cet exemple illustre le besoin de *synchronisation* entre modules s'exécutant de manière concurrente. Deux modules peuvent s'exécuter en parallèle tant que leurs actions n'ont pas d'interférences. Mais s'ils doivent coopérer ou entrer en compétition pour l'accès à une ressource, alors ils ne peuvent pas simplement s'exécuter indépendamment l'un de l'autre et ils doivent synchroniser leurs actions.
Une façon de faire est d'assurer qu'une ressource partagée est accédée en *mutuelle exclusion* : lorsqu'une fonctionnalité `F1` accède à l'objet `Lit`, aucune autre fonctionnalité ne devrait être autorisée à accéder à cet objet.
Plus généralement, les opérations qui affectent l'état interne d'un objet partagé devraient toujours être exécutées en mutuelle exclusion de telle manière qu'elles laissent l'objet dans un état cohérent. Le même principe vaut pour les séquences d'opérations qui testent un attribut de l'objet et qui, en fonction du résultat, modifient ou non l'objet (problème du *test & set*).

Deux approches émergent de la littérature [Ghe91] :
- l'approche ***moniteurs*** : les ressources partagées sont représentées comme des entités passives et protégées par une entité appelée moniteur. Le moniteur exporte des opérations sur l'objet qu'il protège et assure aux clients que ces opérations seront exécutées en mutuelle exclusion. Les objets appellent l'opération du moniteur. Si l'opération est soumise à une condition (comme par exemple `litLibre?`), le moniteur vérifie cette condition. Si elle est vraie, l'opération est exécutée normalement mais en mutuelle exclusion. Si elle est fausse, l'objet qui a émis l'appel est suspendu et attend que la condition devienne vraie. Lorsque, à la suite d'une autre opération, la condition deviendra vraie, un des objets suspendus pourra alors reprendre son activité : tout se passera alors comme s'il venait d'appeler l'opération et que la condition était vraie.

- l'approche *gardes et rendez-vous* : dans cette approche, les gardes sont des objets actifs dont la seule tâche est de garantir un accès ordonné à un "secret caché" représentant une ressource encapsulée. Les gardes sont des tâches qui ne s'arrêtent jamais et qui attendent des requêtes pour exécuter une opération. Ces requêtes peuvent être acceptées ou non, en fonction d'une condition basée sur l'état interne de la ressource contrôlée par le garde. Les requêtes sont acceptées par le garde une à la fois.
  Un objet émettant une requête à un garde est suspendu jusqu'à ce que le garde accepte la requête et termine l'exécution de l'action demandée. Cette forme d'interaction entre un garde et un objet est appelée *rendez-vous*.

On remarquera que les deux approches offrent des solutions *non-déterministes* aux problèmes de concurrence : la politique de sélection d'une demande suspendue n'est pas explicite. On pourra alors la moduler en fonction du problème : certains choisiront la politique FIFO (*first-in, first-out*), d'autres LIFO (*last-in, first-out*) ou encore une politique HPFI (*highest-priority, first-out*). Remarquons aussi que, assez souvent, la politique FIFO est offerte "par défaut".

III. LA CONCURRENCE EN OBLOG

L'approche OBLOG est une approche qui supporte la concurrence. Le système opératoire[1] OBLOG est fait de telle manière que les objets que nous spécifions seront "exécutés" en parallèle. Cependant, au vu des problèmes que nous avons énoncés ci-dessus, on pourrait se demander si la cohérence des données partagées est respectée.

Afin de répondre à cette question, rappelons d'abord que l'accès à une ressource (un objet) se fait grâce aux deux mécanismes que sont l'*appel d'action* et l'*observation*.

Examinons d'abord le cas de l'appel d'action et pour l'illustrer, prenons un exemple (figure 1).

Supposons que deux instances (`X` et `Y`) d'une même classe d'objets (ou de deux classes d'objets différentes) veuillent accéder à la fonction `fixeLitOccupéPar` de l'instance `lit` de la classe d'objets `LIT`. Si `X` et `Y` veulent accéder au même moment, un seul des deux appels sera accepté par `lit` et ce d'une manière qui nous est transparente. L'autre appel sera suspendu jusqu'à ce que l'instance `lit` se trouve à nouveau dans la situation où elle peut accepter cet appel. On peut donc dire que la sémantique de l'appel d'action est telle qu'elle assure qu'aucune autre opération de l'instance ne peut avoir lieu pendant qu'elle en exécute une particulière. Ceci est vrai à condition que, dans la suite des situations-transitions qui constituent l'opération, on n'y ait pas mis une action qui serait le point d'entrée d'une autre opération (c'est-à-dire l'action qu'il faut appeler pour exécuter l'opération).

[1] Rappelons que OBLOG est un langage qui possède aussi une sémantique opérationnelle : à partir de spécifications, du code peut être généré pour donner, *in fine*, un programme exécutable. Une partie de ce programme sera composée d'un système opératoire (*operating system*) qui assure l'exécution, à tour de rôle, des objets présents dans la spécification. Du point de vue de l'utilisateur, ces objets sont exécutés en parallèle.

Si X accède d'abord à l'opération puis que, alors que l'instance `lit` est en train d'effectuer l'opération, Y tente d'y accéder aussi, vu que `lit` n'est pas dans une situation où il peut accepter cet appel, celui-ci sera suspendu.

On peut donc dire que la suite des situations-transitions qui constitue une opération est une *section critique* : nous sommes certains de pouvoir exécuter cette suite sans être interrompu. Comme un seul appel à une opération est traité à la fois, nous pouvons parler d'exécution en *mutuelle exclusion*.

Cependant, il en va tout autrement pour les observations. En effet, si un troisième objet Z décide d'observer le `lit`, rien ne l'empêche de le faire pendant que l'instance `lit` est en train d'effectuer un changement sur son état interne. L'état qui sera alors observé sera un état incohérent de l'objet et cela pourra mener à des erreurs.

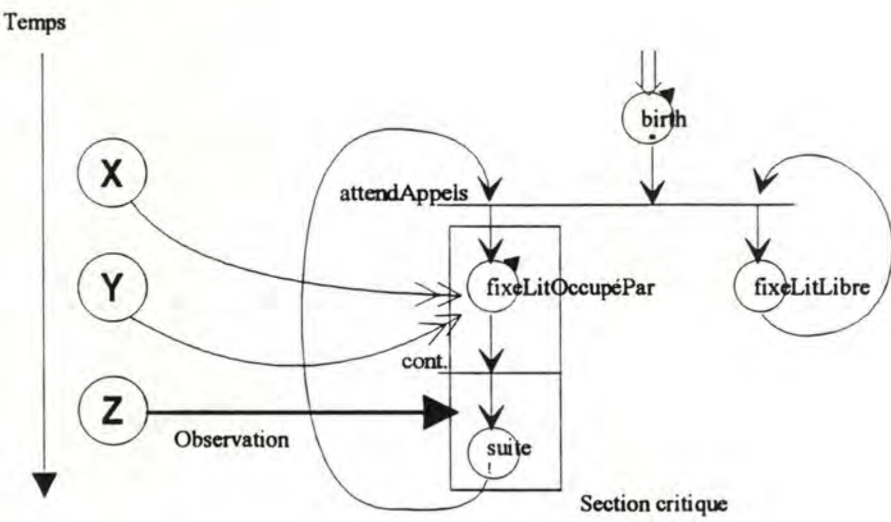


Figure 1 - Appels et observations concurrents

La solution à ce problème se trouve dans le concept de *précondition globale*[2] que l'on peut associer à une action ou à une observation.

Par définition, cette précondition empêche le déclenchement d'une action ou l'observation d'un attribut si la condition associée n'est pas vérifiée.

Il suffira donc d'associer à chaque objet susceptible d'accès concurrents un attribut artificiel `état` dont les valeurs sont soit '`libre`', soit '`occupé`'. Cet attribut indiquera si l'objet est dans un état cohérent ou s'il est occupé à traiter une demande d'opération. Chaque action ou suite d'actions qui change l'état interne de l'objet devra, en commençant, donner la valeur '`occupé`' à `état`, effectuer son traitement et, à la fin, donner la valeur '`libre`' à `état`.

Nous associerons ensuite à chaque observation de l'objet une précondition globale que nous énoncerons comme suit : `{? état=libre}`. Ainsi, aucun objet ne pourra observer un état incohérent de l'objet et cela, grâce au simple mécanisme de précondition globale. ▪

[2] pour plus détails, se reporter au chapitre traitant de l'approche OBLOG, section III.2.3.

Nous allons d'abord présenter les transformations que nous devons apporter à notre conception statique afin d'être conforme au langage OBLOGlight que nous devons utiliser dans cette phase ultime du développement. Nous verrons ensuite l'évolution de l'architecture logique et des spécifications face aux exigences de la conception dynamique. Suivra finalement la résolution des problèmes d'optimisation des demandes de lits et de prescriptions de soin.

1. TRANSFORMATIONS

1.1. PARALLÉLISME

Au chapitre précédent (chapitre 5 section III.1), nous avions posé l'hypothèse que, au sein d'un objet, plusieurs actions ou suites d'actions pouvaient se dérouler en parallèle. Or, pour être conforme au langage OBLOGlight, nous devons lever cette hypothèse. Nous introduirons donc ce que nous appelions un séquencement arbitraire (figure 2).

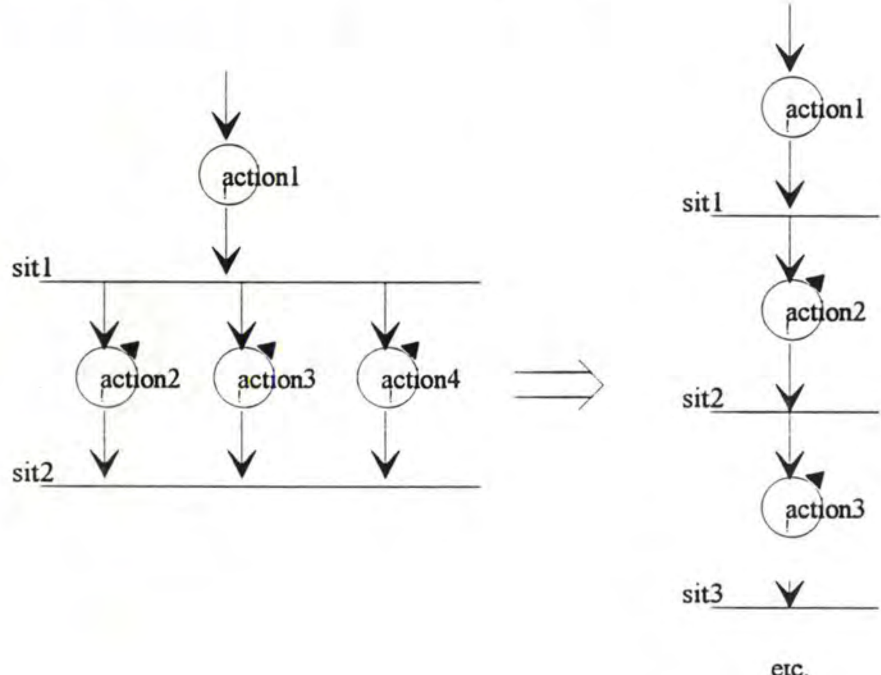


Figure 2 - Introduction d'un séquencement arbitraire

1.2. LE MÉCANISME DE STORED ACTION

Dans l'étape de conception statique, nous avions déjà évoqué ce mécanisme mais dans les grandes lignes, afin de ne pas surcharger l'exposé.
Cependant, comme nous devons être conforme au langage OBLOGlight, nous allons voir ce mécanisme en détail. La relative complexité de ce mécanisme nous a poussé à ne pas changer notre modélisation et à garder l'hypothèse que nous avions posée lors de la conception statique, à savoir, lorsqu'un objet en appelle un autre, il reçoit la réponse directement, sans devoir passer par le mécanisme de *stored action*.
Le lecteur qui disposerait de peu de temps peut, sans risques, passer à la section suivante.

1.2.1. Stored action

Dans la phase de conception statique, nous avions considéré que, lorsqu'une action en appelle une autre et qu'elles doivent s'échanger des paramètres, cet échange était "instantané". Or, ce n'est pas le cas. Il faut passer par le mécanisme de *stored action*. Illustrons par un exemple.

Supposons deux objets `X` et `Y`, `X` appelant et `Y` étant appelé. Supposons encore que l'action `appel` de `X` appelle l'action `reçoitAppel` de `Y` et lui passe comme un paramètre `param`. Sur base de cette valeur, l'objet `Y` va effectuer un traitement et calculer un résultat qu'il stockera dans son attribut `résultat`. Une fois ce calcul terminé, `Y` doit rappeler `X` et lui passer en paramètre la valeur de son attribut `résultat`. Mais comment sait-il quelle action il doit rappeler ? C'est ici que la *stored action* intervient.
L'objet `X`, lorsqu'il va appeler `reçoitAppel`, va lui donner un paramètre supplémentaire : l'action à laquelle l'objet `Y` doit le "recontacter" (l'action `retour`) lorsqu'il aura calculé le résultat. L'objet Y va d'abord stocker cette action dans un attribut `retourAction`, puis effectuer son traitement et, une fois terminé, il va appeler l'action qui se trouve dans son attribut et lui passer la valeur `résultat`.

Pour être encore plus précis dans l'explication, on peut ajouter que, dans un premier temps, il faut déclarer (dans le diagramme de déclaration de l'objet `X`) la *structure* de l'action `retour`, à savoir :

```
retour (result : NAT)
```

puis, déclarer l'action `appel` comme suit :

```
appel (param : STRING, actionRetour : retour)
>> Y.reçoitAppel
```

Ensuite, dans le diagramme de comportement de l'objet `X`, il faut associer à l'action `appel` une *instanciation de paramètre d'action*, c'est-à-dire qu'il faut donner la valeur réelle du paramètre `actionRetour` qui sera fournie à l'objet `Y`. Dans notre cas, cette valeur sera l'action `retour` mais rien n'empêcherait de donner comme valeur une action dont la structure est identique. Ainsi, nous pourrions avoir trois actions de même structure et, en fonction d'une certaine condition, nous donnerions comme action de retour l'une de ces trois actions.

Nous ajouterons que, du côté de l'objet `Y`, le rappel peut se spécifier comme suit (en supposant que l'action qui effectue le rappel est l'action `rappel`) :

```
Y.rappel
>> retourAction with
     retourAction.résultat = Y.résultat
```

La figure 3 illustre l'exemple que nous venons de développer. ▪

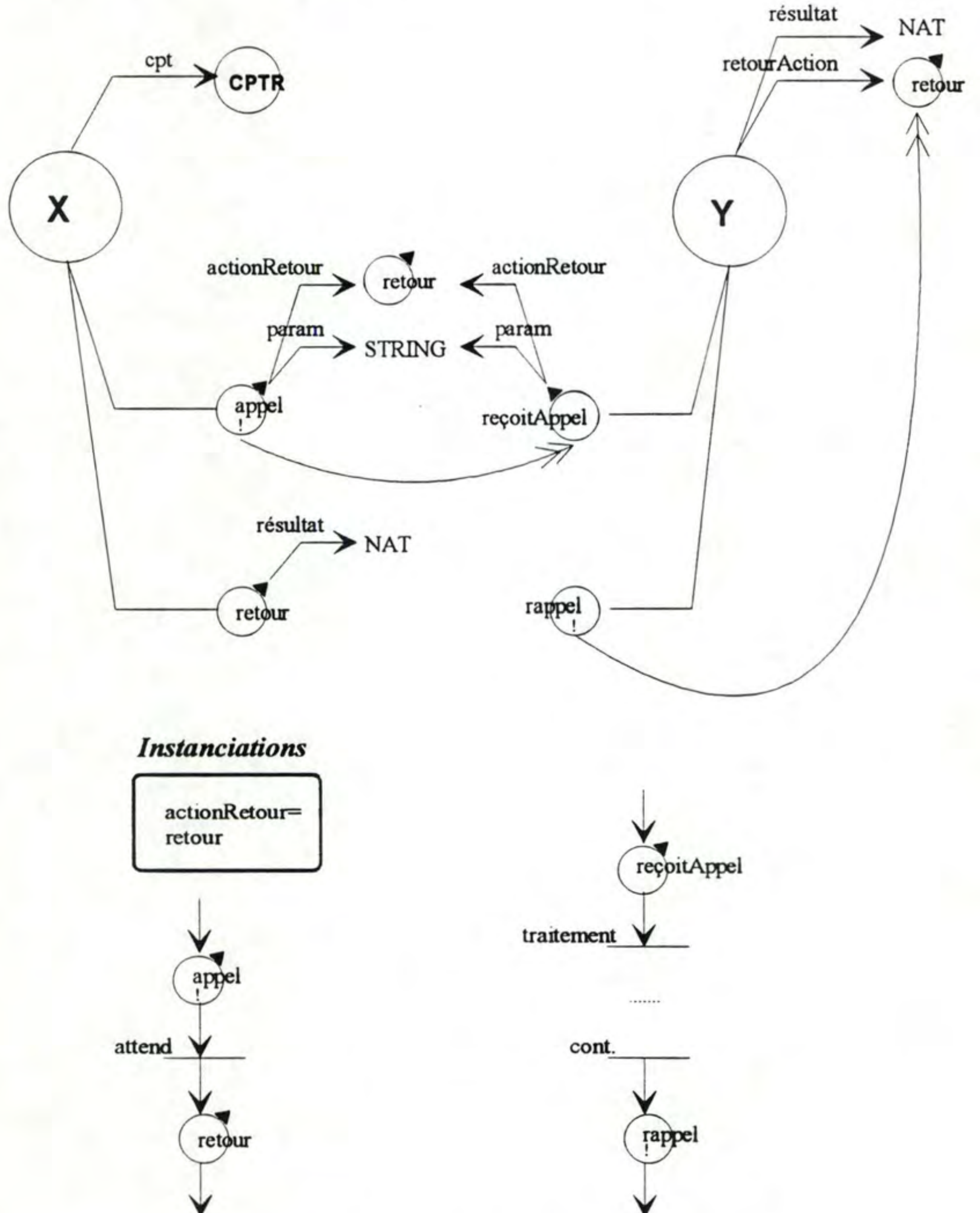


Figure 3 - Stored action (en détail)

On imagine facilement ce que deviendrait notre développement s'il nous fallait respecter ce mécanisme. Il suffirait, pour chaque appel, d'ajouter une action qui attend le résultat et qui effectue la mise-à-jour de la valeur reçue, cette action étant passée lors de l'appel et servant d'action de retour. Répétons une fois de plus que nous avons choisi de garder notre ancienne modélisation afin de ne pas "alourdir" le développement.

1.2.2. Applications

Architecture clients-serveur

Le mécanisme de *stored action* est à la base d'une architecture clients-serveur. En effet, il est facile d'imaginer que l'on peut avoir un objet serveur et plusieurs objets clients. Les clients appellent le serveur et lui passent en paramètre l'action où il doit rappeler une fois le traitement effectué. Pendant ce temps, les clients peuvent continuer leur "vie" sans se soucier du serveur. Ils se donnent juste un point de rendez-vous.

Interface utilisateur

Une autre application du mécanisme de *stored action* est le traitement de l'interaction avec utilisateur. En effet, dans la plupart des boîtes de dialogues, l'utilisateur a le choix entre un bouton `OK` qui valide l'entrée réalisée dans la boîte ou un bouton `Cancel` qui provoque l'annulation de la fonction en cours. En OBLOG, une boîte de dialogue est un objet et il peut donc agir comme un serveur. L'objet qui appelle la boîte devra donc fournir une action où à laquelle la boîte de dialogue doit rappeler si l'utilisateur sélectionne le bouton `OK` et une autre action s'il sélectionne le bouton `Cancel`. En fonction de l'action choisie, le traitement se poursuit ou il est annulé.

Dans le cas MortSubite, nous utilisons ce type d'interaction mais sans expliciter le mécanisme de *stored action*. Pour être conforme, il suffirait d'ajouter deux actions `OK` et `Cancel` qui attendent, l'une une réponse favorable de l'utilisateur ainsi que les valeurs saisies dans la boîte de dialogue, et l'autre une réponse défavorable qui mettra fin à la fonctionnalité. Ces deux actions seraient passées à la boîte de dialogue à l'appel.

2. EVOLUTION

Comme nous l'avions annoncé dans ce chapitre, notre architecture doit évoluer pour tenir compte de la dimension concurrente du logiciel. De nouveaux modules (objets) vont apparaître pour gérer cette concurrence et des modifications vont être faites dans la spécifications des modules existants.

2.1. PROTECTIONS

La modification majeure est celle qui, dans chaque objet accédé de manière concurrente par d'autres objets, introduit un mécanisme de "protection". Il s'agit du concept de *précondition globale* que nous avons exposé ci-dessus.
Comme tous les objets du niveau 3 (données persistantes) sont susceptibles d'une telle protection, nous leur avons ajouté un attribut "artificiel" `état` qui peut prendre les valeurs '`occupé`' et '`libre`'. Nous avons ajouté, à chaque observation de ces objets, la précondition globale `{? état=libre}` et nous nous sommes arrangé pour "allumer" et "éteindre" cet attribut lorsqu'une suite d'actions modifiait l'état interne de l'objet.

Voici ce que cela donne pour l'objet `LIT`.

OBJET LIT

INTERFACE

- **Observations**
  - catégorieLit : CAT
    {? état=libre}
  - situé : CHAMBRE
    {? état=libre}
  - relève : SER_MED
    {? état=libre}
  - nrLit : NAT
    {? état=libre}
  - départ, destination : LIST of MVT
    {? état=libre}
  - litOccupé : BOOL
    {? état=libre}
  - serDeLit : SER_MED
    {? état=libre}
  - litOccupéPar : PAT
    {? état=libre}
  - $litExistant (nrLit : NAT) : BOOL
    {? état=libre}

- **Keys**
  - $lit (nrLit : NAT) : LIT
    {? état=libre}
    {? $litExistant (nrLit)}

- **Events**
  - * birth (ch : CHAMBRE)
  - fixeLitOccupéPar (pat :PAT)
  - fixeLitLibre

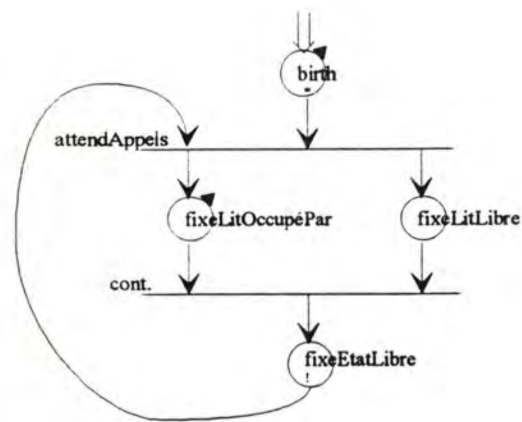
BODY

- **Attributes**
  - nrLit, situé, litOccupéPar
  - $nrLitSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - départ
    := ALL[m : MVT_TSF/SOR | m.de = SELF]
  - destination
    := ALL[m : MVT_TSF/ENT | m.vers = SELF]
  - litOccupé
    := litOccupéPar <> NULL
  - catégorieLit
    := SELF.situé.deCatégorie
  - serDeLit
    := SELF.situé.relève
  - $lit (nrLit)
    := ONE[l : LIT | l.nrLit = nrLit]
  - $litExistant (nrLit)
    := EXISTS[l : LIT | l.nrLit = nrLit]

- **Actions**
  - * birth (ch)
    ⇒ nrLit := $nrLitSuiv
    ⇒ situé := birth.ch
    ⇒ litOccupéPar := NULL
    ⇒ état=occupé
  - ! $incNrLitSuiv
    ⇒ $NrLitSuiv := $nrLitSuiv+1
    ⇒ état=libre
  - fixeLitOccupéPar(p)
    ⇒ litOccupéPar := fixeLitOccupéPar.p
    ⇒ état=occupé
  - fixeLitLibre
    ⇒ litOccupéPar := NULL
    ⇒ état=occupé
  - ! fixeEtatLibre
    ⇒ état=libre

BEHAVIOUR



2.2. "CARDINALITÉ" DES OBJETS

Dans notre architecture précédente, puisque nous avions posé l'hypothèse d'un système mono-utilisateur, l'ensemble des objets représentant des fonctionnalités avaient une "cardinalité" simple (symbole "1"), c'est-à-dire qu'il pouvait exister au plus une seule instance de la classe d'objets. En effet, une fonctionnalité tel que l'introduction d'un transfert ne pouvait être déclenchée que par une seule personne à la fois.

A présent, afin de refléter l'aspect distribué du logiciel, nous devons modifier ces cardinalités. En effet, si nous voulons représenter le fait qu'une fonctionnalité peut être déclenchée par plusieurs services à la fois, nous devons lui attribuer la cardinalité multiple (symbole "?"[3]). Cette fonctionnalité devient alors une classe d'objets et chaque service qui souhaite y faire appel doit créer une instance de cette classe. Ces instances vont se dérouler en parallèle et, une fois leur travail terminé, elles vont se "suicider". Il en va de même pour les objets du niveau 4 : un objet d'interface, qui était auparavant de cardinalité simple, va devenir une classe d'objets. Une fonctionnalité qui y fait appel va créer une instance de cette classe et l'utiliser.

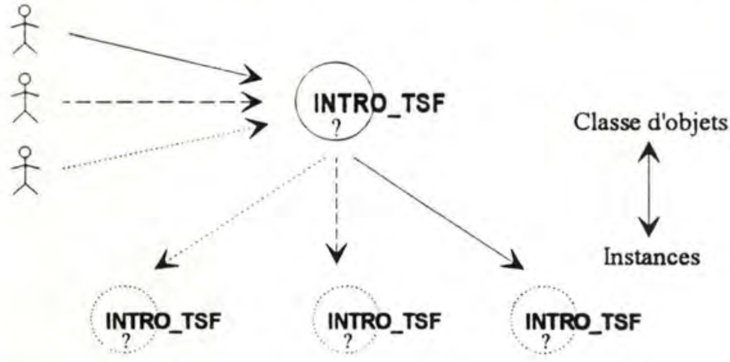


Figure 4 - Classe d'objets et instances de classe

[3] Bien que, dans le langage OBLOGlight, la cardinalité multiple d'un objet soit représentée par l'absence d'un symbole (par opposition au symbole "1" qui désigne une cardinalité simple), nous avons choisi de la désigner par le symbole "?". Ce symbole était utilisé dans la version précédente du langage.

Au niveau 4, six nouveaux objets ont été crées : il s'agit des objets représentant les menus que l'on associera aux différents services organisationnels (Accueil, comptabilité, services de soins etc.). Ces menus contiennent les fonctionnalités que chaque service peut déclencher.

Cependant, au niveau 5, l'objet `Allocation` garde la cardinalité `"1"`. En effet, déjà auparavant, cet objet assurait l'allocation de ressources telles qu'un lit, une prescription de soin, etc. Nous avions choisi de grouper toutes ces allocations dans un seul objet afin de ne pas disperser ces traitements. Et c'est à dessein que nous l'avions fait car, à présent, c'est cet objet qui va assurer la plus grosse partie de la gestion concurrentielle.

2.3. GESTION DE LA CONCURRENCE

La "protection" de l'état interne étant assurée par chaque objet, il nous reste à gérer le problème de l'allocation des lits et des prescriptions de soins.

Dans la phase de conception statique, une seule demande de lit ou de prescription de soin pouvait survenir à la fois. Il suffisait à la fonctionalité concernée de faire un appel à la fonction `allocLit` ou `allocPrescr` de l'objet `Allocation` pour obtenir un lit ou une prescription. A présent, puisque le cahier des charges nous impose de pouvoir exécuter plusieurs fonctionnalités en parallèle, nous devons imaginer un mécanisme qui permette de gérer les demandes multiples c'est-à-dire ce que l'on appelle traditionnellement une *file d'attente*. Ces demandes seront alors optimisées afin d'en accorder le plus possible, compte tenu de leurs priorités.

Les fonctionnalités doivent donc émettre des demandes de lits et de prescriptions de soins et c'est à l'objet `Allocation` de les optimiser. Ces demandes seront en fait des instances d'une classe d'objets que nous qualifierons de passive car son seul rôle sera de "signaler" que telle fonctionnalité souhaiterait soit un lit, soit une prescription de soin. Afin de tenir compte de cet état de fait, nous devons modifier nos spécifications pour qu'elles émettent ces demandes et attendent une réponse qui sera soit favorable, auquel cas elles continueront leur exécution, soit négative, auquel cas il faudra arrêter l'exécution et prévenir l'utilisateur.

Les fonctionnalités que nous devons changer sont les suivantes : Admission_Complète (ADM_COMP), Admission_Pré (ADM_PRE), Introduction_Transfert (INTRO_TSF), Introduction_Sortie (INTRO_SORTIE) et Prescription_Soin (PRESCR_SOIN).

Nous présenterons uniquement les modifications sur l'objet INTRO_TSF. Le lecteur intéressé pourra consulter les annexes pour un développement complet de ces modifications.

OBJET INTRO_TSF

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - litOrig, litDest : LIT

- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")

- **Actions**
  - * start
  - ! nrDosD (pat : PAT)
    >> NR_DOS_D.open (pat)
    ⇒ pat := nrDosD.pat
  - ! tsfD (litOrig, litDest : LIT, med : MED)
    >> TSF_D.open
    ⇒ litOrig := tsfD.litOrig
    ⇒ litDest := tsfD.litDest
    ⇒ med := tsfD.med
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorLitDestOcc
    IS putError (code)
    := code = TSF_LIT_DEST_OCC
  - !+ putErrorPatPasMal
    IS putError (code)
    := code = TSF_PAT_NOT_MAL
  - ! ddeTsf
    >> dde_mvt.birth (litOrig, litDest, pat, "T", litOrig.serDeLit.priorité, creeMvtTsf, putErrorLitDestOccupé)
    { création d'un instance de la classe d'objets DDE_MVT avec comme paramètres : le lit d'origine et de destination, le patient, le type de demande (E/T/S), les actions où rappeler en cas de réponse positive/négative }
  - ! creeMvtTsf
    >> mvt_tsf.birth (litOrig, litDest, pat, med, TODAY(), NOW())
    >> allocation.déplacerPrescr (litOrig.serDeLit, litDest.serDeLit, pat)
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = TSF_OK

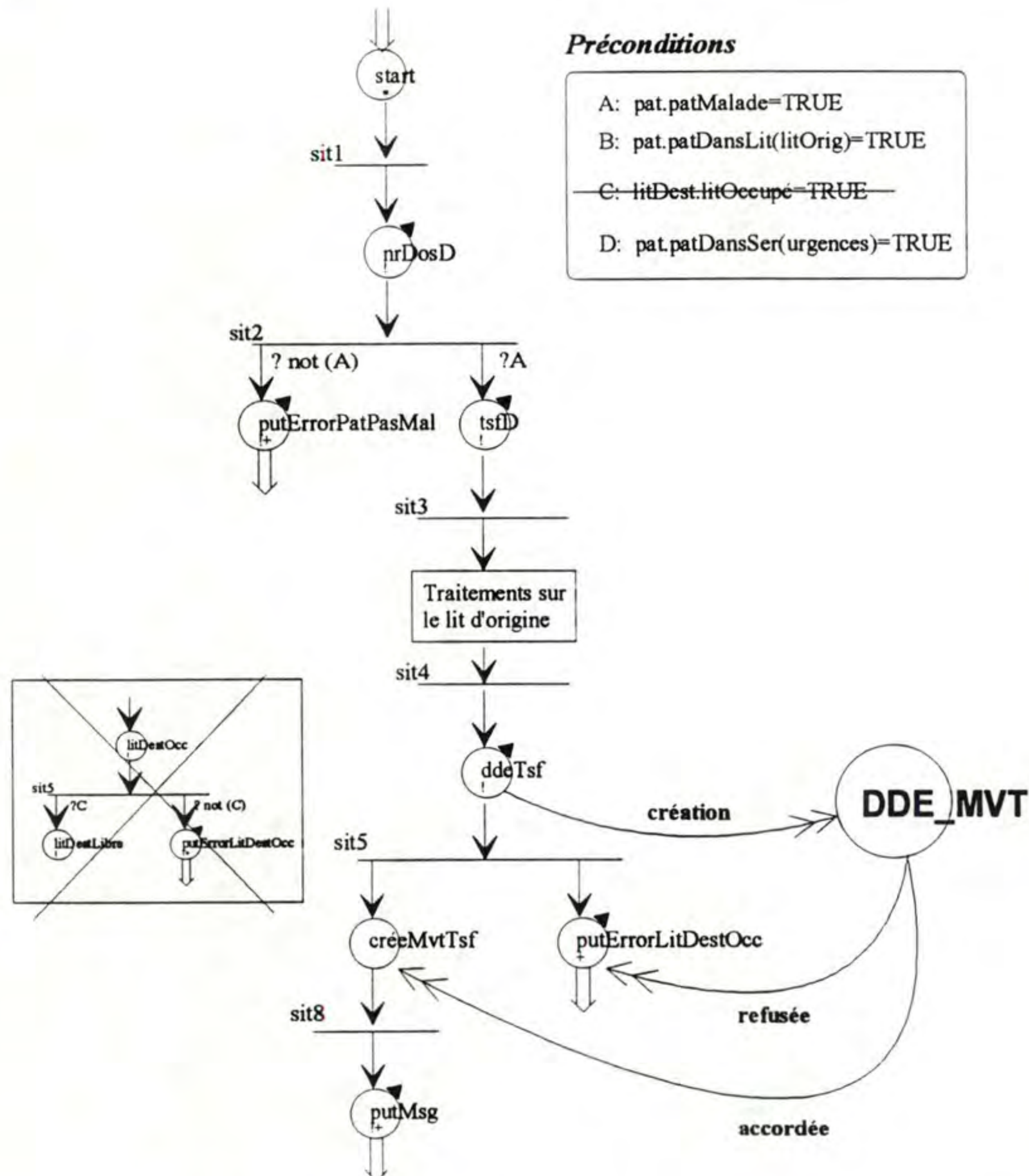


Figure 5 - Le diagramme de comportement de INTRO_TSF

Remarquons que nous avons supprimé le parallélisme : les traitements sur le lit d'origine (teste son existence, vérifie que le patient est bien dedans, etc.) se font avant ceux sur le lit de destination. Parmi ces tests sur le lit de destination, il y avait auparavant un test qui vérifiait que le lit était bien vide (comme le rappellent la précondition C barrée et l'encadré à gauche de la figure 5). Or, ce test n'a plus de sens car il se peut que, bien que le lit soit occupé, en raison de la priorité de la demande ou pour une autre raison, ce lit se libère à l'optimisation et qu'il soit attribué. Nous devons donc reporter ce test au moment de l'optimisation, c'est-à-dire le plus près possible de l'allocation. C'est d'ailleurs de cette façon que nous avons résolu les problèmes de *test & set* qui se présentaient à nous.

Le changement majeur dans cette spécification est le suivant : l'objet-fonctionnalité ne fait plus appel directement à la fonction `allocLit` mais il émet une demande. Il crée une instance de la classe d'objets `DDE_MVT` et lui passe en paramètres le lit d'origine et de destination, le patient concerné par ce transfert, la priorité de la demande ainsi que les actions auxquelles l'instance de `DDE_MVT` doit le recontacter en cas de réponse positive ou négative.

Avant d'examiner la solution que nous avons retenue pour optimiser l'allocation des lits et des prescriptions de soins, nous tenons à faire une remarque : si notre architecture et nos spécifications tentent de tenir compte de l'aspect distribué du logiciel, elles ne le font pas jusqu'au bout. En effet, nous n'avons pas trouvé en OBLOG de mécanisme qui permet de spécifier que tel menu doit être affiché sur tel terminal par exemple. C'est ce qui nous pousse à dire que notre logiciel, tel qu'il est actuellement, fera du "distribué virtuel".

3. OPTIMISATIONS

3.1. ALLOCATION D'UN LIT

3.1.1. Le problème

Les fonctionnalités Admission_Complète, Admission_Pré, Introduction_Transfert et Introduction_Sortie ont émis des demandes de mouvement de trois types :
- Ep (Entrée) : demande d'un lit d'une catégorie particulière (c) dans un service particulier (s), accompagnée d'une priorité (p). Cette demande est émise par les admissions.
- Tp (Transfert) : demande d'un transfert d'un lit (l1) vers un lit (l2), accompagnée d'une priorité (p). Cette demande est émise par l'introduction d'un transfert.
- Sp (Sortie) : demande d'une sortie d'un lit (l) vers la sortie, accompagnée d'une priorité (p). Cette demande est émise par l'introduction d'une sortie.

Il convient de répondre à ces demandes comme il a été exigé dans le cahier des charges (chapitre 2, section II.7) :
- les services ont une priorité de traitement :
  - "cas spéciaux" 0
  - urgences........................... 1
  - soins intensifs 2
  - cardiologie 3
  - chirurgie........................... 3
  - médecine interne 3
  - maternité.......................... 3
  - gériatrie 3

  les "cas spéciaux" étant des demandes ponctuelles qui ne peuvent souffrir de retard et qu'il convient donc d'exécuter immédiatement.
- pour l'attribution des lits, on tentera de maximiser les demandes :
  - en tenant compte des priorités. La priorité étant celle du service demandeur dans le cas d'un transfert ou celle du récepteur dans le cas d'une admission
  - sans se limiter à l'examen des demandes sur un seul lit mais en étendant aux "précédents" (sur plusieurs lits)
  - une demande de priorité 0 doit être servie impérativement

Les réponses possibles aux différentes demandes sont :
- pour les Ep :
  - soit acceptée avec un lit attribué de la catégorie souhaitée (c) ou non
  - soit refusée
- pour les Tp :
  - soit accordée (les lits (l1, l2) sont déjà déterminés)
  - soit refusée
- pour les Sp :
  - toujours accordées car elles libèrent un lit

3.1.2. Une solution

Nous allons présenter le pseudo-algorithme de la solution que nous avons choisie. Un pseudo-algorithme plus proche du langage OBLOGlight est développé en annexe.

L'idée générale est la suivante : sur base de la liste des demandes émises au moment où l'objet commence son traitement et sur base de la liste des lits disponibles dans les services concernés par les demandes, satisfaire le plus possible de demandes qui ne nécessitent pas d'optimisation. Ensuite construire le graphe des demandes, repérer les composantes "particulières" et effectuer le traitement adéquat. Finalement, optimiser les composantes restantes. Par optimisation, nous entendons la satisfaction d'un ensemble de demandes telles que la somme des poids de ces demandes est maximum. Le poids d'une demande est inversément proportionnel à sa priorité : une demande de priorité 0 se verra attribuer un poids "∞", une demande de priorité 1 un poids de 6, une demande de priorité 2 un poids de 3 et une demande priorité 3 un poids de 1.

1. Dresser une liste avec toutes les demandes que l'on doit traiter. Ces demandes sont réparties selon leur type dans une des trois listes suivantes : Lde (Liste des demandes d'entrée) pour les demandes de type Ep, Ldt (Liste des demandes de transfert) pour les demandes de type Tp et Lds (Liste des demandes de sortie) pour les demandes de type Sp.
2. Filtrer toutes les Sp, les exécuter et ajouter les lits libérés par ces sorties à la liste des lits vides Lv
3. Construire la liste des services concernés par les demandes Ep. (Lse)
4. Pour chaque service (ser) de Lse, sélectionner tous les lits vides de ser et les ajouter à la liste des lits vides Lv.
5. Construire la liste des lits "vides vides" (Lvv) c'est-à-dire des lits appartenant à la liste des lits vides Lv et qui ne sont la destination d'aucune demande de transfert Tp.
6. i:=1

   Tant que NOT EMPTY (Lvv) et que NOT END-OF-LIST (Lde)

   dde := Lde(i)

   Si il existe un lit (l) dans Lvv, de la catégorie et du service spécifié par dde

   Alors allouer le lit l (exécuter la demande)

   Sinon

   Si il existe un lit (l) dans Lvv, du service spécifié par dde

   Alors allouer le lit

   Sinon passer à la demande suivante (i := i + 1)

   {Servir un maximum de demandes Ep dans l'ordre des priorités}
7. Si EMPTY(Lde) et NOT EMPTY(Lvv)

   Alors Lv := Lv - Lvv

   {S'il n'y a plus de demandes Ep à satisfaire et qu'il reste des lits "vides vides", alors il faut les éliminer de la liste des lits vides mais qui sont destination d'un transfert.}

8. Comme seules demandes, il reste des Ep (éventuellement) et des Tp. On peut représenter la situation comme un graphe dont les noeuds sont des lits et les arcs sont les demandes de transfert. On a aussi des arcs sans origine et sans destination : ce sont les Ep. En effet, il est impossible de les attacher à un lit particulier. La figure 6 illustre ce que nous appellerons le graphe des demandes. Remarquons que ce graphe possède des propriétés particulières : un noeud peut avoir plusieurs pères mais il n'a au plus qu'un seul fils. Le graphe est divisé en plusieurs composantes : une composante étant un ensemble de noeuds tels qu'ils sont reliés entre eux par des arcs. On peut identifier différents types de composantes : les composantes *cycliques*, *impossibles*, *linéaires* et *à optimiser*. Une composante est impossible si le noeud final (le dernier fils) est un lit occupé. Une composante est linéaire si chaque noeud possède un et un seul père et fils, sauf éventuellement les noeuds racine et final. Une composante cyclique est une composante dont un sous-ensemble des noeuds forme un cycle.

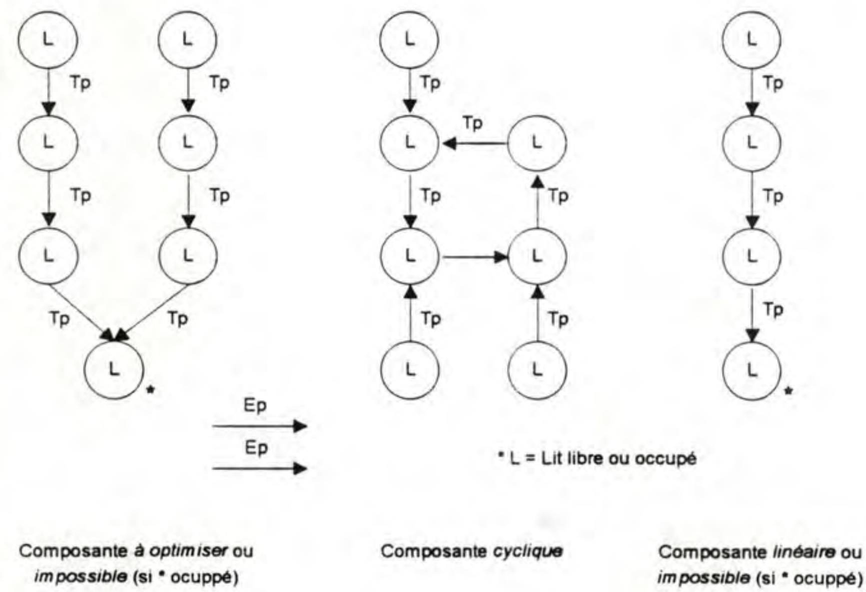


Figure 6 - Le graphe des demandes

9. Eliminer les cycles, c'est-à-dire accepter les demandes qui sont au coeur du cycle et refuser toutes les demandes qui sont des pères de ces noeuds. En effet, quelles que soient les priorités des demandes qui sont hors du cycle, la seule solution est d'exécuter le cycle.

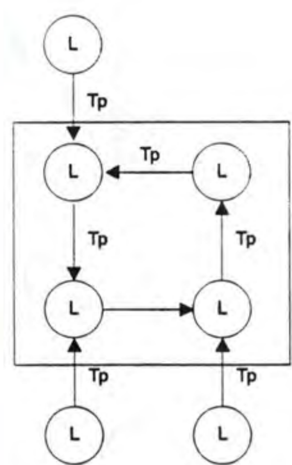


Figure 7 - Les demandes à accepter

10. Eliminer les composantes impossibles. Ces composantes se terminant par un lit occupé, il est impossible d'accepter ses pères.
11. Exécuter les composantes linéaires. Vu que le noeud terminal est un lit vide et qu'aucune demande n'entre en compétition avec d'autres, il convient d'exécuter les demandes de cette composante.
12. Le graphe se résume alors à une série de composantes "non-linéaires", sans cycles et avec un noeud terminal qui est un lit vide. Chacune des composantes est sujette à une optimisation. Remarquons que, pour chaque composante, quelle que soit la suite de demandes que l'on décidera d'accepter, l'exécution de celles-ci se traduira par la libération d'un lit (l). Ce lit sera un noeud racine de la composante. Il sera alors possible d'accepter une demande de type Ep si le service du lit l est le même que le service demandé par Ep.

Une solution pour obtenir l'optimum global pourrait être la suivante : tester toutes les possibilités (combiner) et choisir celle qui a le poids le plus grand.
Supposons trois composantes C1, C2 et C3, trois demandes d'admission E1 (priorité 0, service S1, catégorie souhaitée C1), E2 (3, S2, C2) et E3 (1, S4, C1) et une série de demandes de transfert telle que représentée à la figure 8.

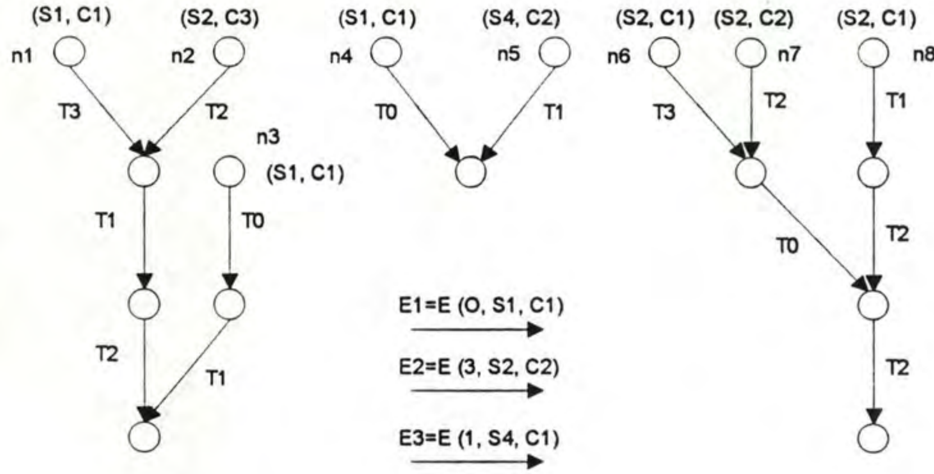


Figure 8 - Un exemple de graphe à optimiser

Il faudrait alors évaluer le poids total si l'on considère que
- (1) on libère le lit n1 (donc que l'on exécute les demandes qui en découlent) et on attribue le lit n1 à la demande E1
- (2) on libère le lit n5 et on l'attribue à E3
- (3) on libère le lit n6 et on l'attribue à E2

puis évaluer le poids en changeant l'hypothèse (3) pour qu'elle devienne : libérer n7 et l'attribuer à E2. Idem pour n8, etc pour chaque combinaison possible. La combinaison qui maximise le poids total étant celle que nous choisirons d'exécuter.

Cette optimisation globale risque d'être très lourde à implémenter. Pour cette raison et au vu du temps qu'il nous reste, nous avons choisi de ne réaliser qu'un optimum local c'est-à-dire un optimum pour chaque composante prise une à une. Il est fort probable que la somme des optima locaux sera proche de l'optimum global.

Dans cette perspective, nous allons décrire un algorithme qui, étant donné une composante C, optimise les demandes comme il a été exigé dans le cahier des charges.

13. Recherche de l'optimum local. Soit C la composante à optimiser.

Soient N = liste des noeuds d'entrée de C (figure 8 : n1-n2, n4-n5, n6-n7-n8)
n = un élément de N
En = liste des Ep tels que le service de Ep est le même que le service de n
Ch = chemin de poids maximum
Max = poids de Ch
PoidsT = poids du chemin courant (temporaire)
ChT = chemin courant (temporaire)

```
Tant que NOT END-OF-LIST (N)
      Prendre n dans N tel qu'il n'a pas été encore testé
      Construire En
      Calculer PoidsT et stocker le chemin dans ChT
      Si PoidsT > Max
      Alors  Ch := ChT
             Max := PoidsT
      Pour tout élément e de En
            Calculer le poids de ChT + e (le poids du chemin plus celui de la demande e)
            (si la catégorie de n = catégorie souhaitée dans e alors poids := poids + 2)
            Si ce poids > Max
            Alors Ch := ChT + e et Max := poids de (ChT+e)
      Fin-Pour-tout
Fin-Tant-Que
```

3.2. ALLOCATION D'UNE PRESCRIPTION

3.2.1. Le problème

La fonctionnalité Prescription_Soin émet des demandes pour allouer une prescription de soin. Il convient de traiter ces demandes comme il a été exigé dans le cahier des charges (chapitre 2, section II.7) :

- pour la gestion des prestations, on tentera d'atteindre le "plein emploi" des ressources :
  - en tenant compte des priorités
  - en tenant compte de la capacité disponible pour cette prestation : les demandes refusées ou "éjectées" (accordées précédemment mais cédant leurs places) seront reportées à la pause suivante avec une plus grande chance d'être servies
  - seules les demandes de priorité 0 sont autorisées pour la pause en cours (p). Elles seront exécutées sans attendre mais sans éjecter aucune demande prévue pour cette pause : elle seront "casées" parmi les demandes prévues
  - la gestion du planning est donc à faire sur les pauses p+1 et suivantes.

3.2.2. Une solution

Ce problème est beaucoup plus facile à traiter que le précédent. En effet, nous avons déjà une partie de la solution que nous avons développée lors de la phase de conception statique. Il s'agit de la "fonction" AllocPrescription. Cette fonction a pour objectif de placer une prescription de soin dans le planning des prescriptions prévues. Elle peut être réutilisée sans changements.

Exprimée dans le même formalisme que l'algorithme précédent, une solution pourrait être la suivante :

1. Construire une liste des demandes à traiter (Ldps)
   {Liste des demandes de prescriptions de soins}
2. Trier cette liste sur base des priorités
3. Pour tout élément (e) de la liste Ldps
   exécuter allocationPrescription
   Fin-Pour-tout

Chapitre 7
Conclusion

CONCLUSION

Pour conclure, nous proposons de résumer le travail réalisé (section I), de l'évaluer (section II) puis d'examiner les perspectives d'avenir pour ce mémoire (section III).

I. TRAVAIL RÉALISÉ

Tout au long de ce mémoire, nous nous sommes efforcés d'appliquer ce que nous avons appelé l'approche transformationnelle au développement d'un cas de taille réelle et présentant une dimension concurrente et distribuée : le cas de l'hôpital MortSubite.
Ce travail était pour nous l'occasion d'appliquer et d'évaluer un certain nombre d'idées qui nous avaient été communiquées par le promoteur de ce mémoire.

Ainsi, après avoir examiné l'étude des besoins et les spécifications fonctionnelles, nous nous sommes intéressés de près à la conception du logiciel.
Afin de réaliser au mieux cette étape, nous l'avons scindée en deux phases : la conception statique et la conception dynamique.

Cette découpe originale nous a permis, dans un premier temps (conception statique), de ne pas tenir compte du caractère distribué et concurrent du logiciel pour nous concentrer sur l'architecture de celui-ci. Pour ce faire, nous devons poser les hypothèses suivantes : le système est mono-utilisateur et nous réalisons une description statique du logiciel à l'aide du concept de fonction mathématique (fonctions instantanées).

Pour construire l'architecture, nous avons fait appel à trois principes : la structuration, l'abstraction et la modularisation. Ces principes nous ont aidés à structurer nos spécifications en un tout cohérent, compréhensible et aisément maintenable.

Dans un second temps (conception dynamique), nous avons levés les hypothèses posées afin de fournir *in fine* une solution complète au problème posé dans le cahier des charges. Des éléments du cahier des charges que nous avions temporairement "oubliés" réapparaissent et il nous faut alors imaginer des stratégies pour gérer la concurrence et intégrer l'aspect distribué du logiciel dans nos spécifications.

Pour réaliser la conception du logiciel, nous avons utilisé le langage OBLOG. Ce langage formel et orienté-objet permet de modéliser un système d'information comme une communauté d'objets concurrents et qui interagissent entre eux.

Nous avons dû modifier le langage OBLOG afin qu'il supporte les hypothèses de la conception statique. Le changement principal concernait la sémantique de l'appel d'action : nous avons distingué une partie statique (la définition d'une fonction) et une partie dynamique (le moyen que l'on se donne pour exécuter cette fonction).

Lors de la conception dynamique, nous avons vu comment, à l'aide du simple mécanisme de précondition globale, la cohérence des accès à une ressource partagée pouvait être garantie. Nous avons aussi développé, avec moins de facilité, un module optimisant les demandes d'allocations de lits et de prescriptions de soins.

II. ÉVALUATION

Comme nous le sous-entendions ci-dessus, les cas sur lesquels a été appliquée l'approche transformationnelle ne représentent pas des cas de taille réelle. Ce mémoire prétend donc remédier à cette lacune. Nous pouvons affirmer que cette approche peut être recommandée pour le développement de logiciels qui appartiennent à la même catégorie que le cas de l'hôpital MortSubite.

Mais nous tenons surtout à communiquer au lecteur la facilité avec laquelle nous avons pu réaliser la conception de notre logiciel.

L'idée originale qui consiste à scinder la conception en une phase statique et une phase dynamique en est une des principales raisons. Ce mémoire confirme donc l'observation sous-jacente à cette idée : dans la catégorie de systèmes d'informations que nous traitons, la dimension concurrente et distribuée n'a pas une influence décisive sur l'architecture du logiciel. Pour s'en convaincre, il suffira de remarquer que les changements dans notre architecture se réduisent à l'addition d'un seul module et à la modification de la "cardinalité" des objets.

La seconde raison est, sans aucun doute, la puissance d'expression et de modélisation dont fait preuve le langage OBLOG. Cependant il ne nous a pas été possible d'exprimer pleinement l'aspect distribué du logiciel à l'aide des concepts d'OBLOG. Il n'est en effet pas possible de spécifier par exemple qu'une boîte de dialogue doit être affichée sur un tel terminal ou qu'un tel processus sera effectué sur une machine particulière. Cette lacune nous pousse à dire que, bien que ses capacités de modélisation soient indiscutables, le langage OBLOG n'offre pas encore de mécanismes suffisamment proches de l'implémentation.

III. PERSPECTIVES D'AVENIR

Nous sommes conscient que ce travail est loin d'être terminé et, à l'heure où il nous faut le clôturer, une foule d'idées se présentent à nous. Ces idées ne sont que des pistes que nous souhaiterions investiguer si nous disposions de plus de temps.

Pour commencer, nous corrigerions les petites erreurs, les omissions ou les sous-spécifications qui sont certainement encore présentes dans l'étude de cas.

Ensuite, nous changerions la façon dont nous avons conçu la solution au problème de l'allocation des lits. Cette solution souffre d'une lourdeur due à une utilisation massive du concept de liste. Il nous semble qu'une meilleure modélisation serait de représenter la structure du graphe par une collection d'objets, chacun ayant une "connaissance" de son suivant, de son précédent, etc.

Un prolongement fort intéressant serait aussi de valider nos spécifications en les entrant dans l'atelier logiciel OBLOG-CASE. Nous pourrions ainsi suggérer des modifications à apporter au langage OBLOGlight afin de supporter quelques unes des primitives que nous avons introduites pour nous simplifier la tâche de conception. A ce sujet, rappelons que nous n'avons fait qu'ajouter des primitives qui sont présentes dans la plupart des langages du même niveau qu'OBLOGlight.

Sur le plan méthodologique, nous souhaiterions aussi découvrir d'autres principes pour aider le concepteur à construire une *bonne* architecture ou du moins, expliciter quelques principes existant afin de les rendre compréhensibles est applicables par n'importe quel concepteur.

Comme le lecteur a pu s'en rendre compte, l'implémentation d'une ressource partagée en OBLOGlight se fait de manière simple. Etant donné que ce langage se veut un langage concurrent, il nous semblerait intéressant de prolonger la recherche de concepts aidant à modéliser ce genre d'objets. Nous pensons par exemple à introduire un objet du type ressource partagée offrant, de manière transparente, la possibilité de définir des sections critiques ou d'autres mécanismes ayant pour but de protéger l'état interne d'un objet.

Finalement, nous souhaiterions nous lancer dans une activité beaucoup plus complexe : développer une architecture "générique", une démarche semi-automatique qui assisterait le concepteur dans sa tâche. En effet, nous pensons qu'il est possible de trouver dans notre architecture un sous-ensemble commun à tous les logiciels de la catégorie de MortSubite. Il suffirait alors au concepteur "d'instancier" l'architecture générique pour qu'elle réponde aux caractéristiques particulières de son logiciel. Il nous semble aussi pertinent d'automatiser (si possible) un certain nombre de principes qui peuvent aider à construire une bonne architecture.

A l'heure où de plus en plus d'ateliers logiciels sont disponibles sur le marché, la recherche de telles méthodologies assistées nous semble être une activité prometteuse et utile.

Bibliographie

BIBLIOGRAPHIE

OUVRAGES OBLOG

[ESDI92a] ESDI SA, "Pragmatic introduction to the OBLOG approach in system design", notes du séminaire présenté dans le cadre du cours de Méthodologie de développement de logiciels, m.a., FUNDP Namur, Lisbon, April 1992

[ESDI92b] ESDI SA, "OBLOGkernel - Language Summary and Workbench Presentation", Document, Lisbon, September 1992

[ESDI92c] ESDI SA, "OBLOG", Folder, Lisbon, September 1992

[ESDI92d] ESDI SA, "OBLOG-CASE v1.0 - Software Product Specification", Lisbon, October 1992

[ESDI92e] ESDI SA, "Towards OBLOGlight - Draft Language Proposal", Lisbon, October 1992

[ESDI92f] ESDI SA, "Towards OBLOGlight - Draft Language Reference", Lisbon, November 1992

[ESDI93] ESDI SA, "OBLOG-CASE v1.0 - User's Guide", Lisbon, April 1993

[Ser91a] A. Sernadas, C. Sernadas, P. Gouveia, P. Resende and J. Gouveia, "OBLOG - An informal introduction", Computer Science Group INESC, Lisbon, January 1991

[Ser91b] A. Sernadas, H.-D. Ehrich, "What is an object after all ?", Object-oriented databases : analysis, design and construction, R. Meersman, W. Kent, S. Khosla (eds), North Holland, 1991, pp. 36-69

[Zei92] J.-M. Zeippen, P. Barqueira, "A simple way to integrate technologies into the OBLOG approach", ESDI SA, Lisbon, 1992

AUTRES OUVRAGES

[Bjø78] D. Bjørner and C.B. Jones, "The Vienna Development Method. The metalanguage", vol. 61 of LNCS, Springer-Verlag, 1978.

[Bod89] F. Bodard et Y. Pigneur, "Conception assistée des systèmes d'information — 1. Méthode - Modèles - Outils", coll. MIPS, Masson, Paris, 1989

[Boe76] B. Boehm, "Software Engineering", IEEE Transactions on Computers, vol. C-25, n° 12, December 1976

[Boe84] B. Boehm, "Prototyping Versus Specifying : A Multiproject Experiment", IEEE Transactions on Software Engineering, vol. SE-10, n° 3, pp. 290-302, May 1984

[Che76] P.P. Chen, "An Entity/Relationship Model - Towards a unified view of data", ACM TODS, vol.1.1., 1976

[Dub92] E. Dubois, "Méthodologie de développement de logiciels", notes de cours, Institut d'Informatique, FUNDP, Namur, 1992.

[Dub93] E. Dubois, "Génie logiciel : état de l'art et perspectives", in Journal de Réflexion sur l'Informatique, n° 26, Namur, Avril 1993.

[Eri87] R.W. Erickson and D.R. Musser, "The AFFIRM Theorem Prover : Proof Forest and Management of Large Proofs", 5th Conference on Automated Deduction, Editors W. Bibel & R. Kowalski, published by Springer-Verlag, LNCS 1987

[Fai85] R. Fairley, "Software Engineering Concepts", Mc Graw Hill, 1985

[Fol92] Projet FOLON...

[Ghe91] C. Ghezzi, M. Jazayeri, D. Mandrioli, "Fundamentals of software engineering", Prentice-Hall Internationnal Editions, 1991

[Goo79] D.I. Good, Richard M. Cohen, J. Keeton-Williams, "Principles of proving concurrent programs in GYPSY", Conferences Record of Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 42-52, San Antonio, Texas, January 29-31 1979

[Hab93] N. Habra and E. Dubois, "Putting into practice advanced software engineering techniques through students projects", à paraître in 7th Conference on Software Education, San Antonio, Texas, January 1994.

[Hai86] J.-L. Hainaut, "Conception assistée des systèmes d'information — 2. Conception de la base de données", coll. MIPS, Masson, Paris, 1986

[Krö87] F. Kröger, "Abstract modules : combining algebraic and temporal logic specification means", in Technique et Science Informatiques, vol.6-n° 6, 1987

[Mey80] B. Meyer, "Sur le formalisme des spécifications", Globule, Bulletin du Groupe de Travail "Génie Logiciel" de l'AFCET, 1980

[Par72a] D.L. Parnas, "On the criteria to be used in in decomposing systems into modules", in Communications of the ACM, 15(12), 1972

[Par72b] D.L. Parnas, "A technique for software module specification with examples", in Communications of the ACM, 15(5), 1972

[Par79] D.L. Parnas, "Designing Software for Ease of Extension and Contraction", IEEE Trans. Software Engineering, SE 5(2), pp. 128-137, 1979

[Par83] H. Partsch and R. Steinbruggen, "Program Transformation Systems", Computing Surveys, vol. 15, n° 3, pp. 199-236, September 1983.

[Rai92] Raise Language Group, The RAISE Specification Language, BCS Practitioner Series, Prentice-Hall, 1992.

[Spi89] J.M. Spivey, "The Z notation - a reference manual", Prentice-Hall International, 1989.

[Wie91] R.J. Wieringa, "Object-Oriented Analysis, Structured Analysis, and Jackson System Development", Proc. of the working conference on the object-oriented approach to Information Systems, Quebec City, October 1991

[Win85a] J.V. Guttag, J.J. Horning and J.M. Wing, "Larch in Five Easy Pieces", Technical Report 5, DEC Systems Research Center, July 1985

[Win85b] J.V. Guttag, J.J. Horning and J.M. Wing, "The Larch Family of Specification Languages", IEEE Software, 2(5), pp. 24-36, September 1985

[Win86] M.P. Herlity and J.M. Wing, "Axioms for concurrent objects", Department of Computer Science Carnegie-Mellon University, Pittsburgh, 11/3/1986

Facultés Universitaires Notre-Dame de la Paix à Namur
Institut d'Informatique

UTILISATION D'OBLOG DANS LE DÉVELOPPEMENT TRANSFORMATIONNEL D'UN SYSTÈME D'INFORMATION CONCURRENT

—

ANNEXE

Pierre PELLEZ

Promoteur : Eric Dubois

Mémoire présenté en vue de l'obtention
du diplôme de Licencié et Maître en Informatique

Année académique 1992-1993

Annexe

Etude de cas

TABLE DES MATIÈRES

ETUDE DES BESOINS 1
I. ETUDE D'OPPORTUNITÉ 1
1. Cadre général 1
2. Organisation générale 1
3. Accueil 2
4. Hospitalisation 2
4.1. Mouvements internes et externes 2
4.2. Prestations 2
5. Facturation 2
6. Statistiques 3
7. Politique d'allocation 3
II. MODÈLE DE STRUCTURATION DES TRAITEMENTS 3
1. Niveau projet 3
2. Niveau application 4
3. Niveau phase 4
III. MODÈLE DE STRUCTURATION DES DONNÉES 6
1. Entités 6
2. Associations 8

SPÉCIFICATIONS FONCTIONNELLES 11
I. FONCTIONS 11
1. Description des fonctions 11
1.1. Fonction ADMISSION_COMPLETE 11
1.2. Fonction ADMISSION_PRE 12
1.3. Fonction ADMISSION_POST 13
1.4. Fonction INTRODUCTION_TRANSFERT 13
1.5. Fonction INTRODUCTION_SORTIE 13
1.6. Fonction PRESCRIPTION_SOIN 14
1.7. Fonction PRESTATION_D'UNE_PRESCRIPTION 14
1.8. Fonction ANNULATION_PRESCRIPTION 15
1.9. Fonction RAPPEL_PRESTATION 15
1.10. Fonction DEMANDE_INFO_MALADE 15
1.11. Fonction PRODUCTION_FACTURES 16
1.12. Fonction INTERROGATION_FACTURE 17
1.13. Fonction STATISTIQUES_ON_LINE 17
1.14. Fonction STATISTIQUES_PERIODIQUES 18
2. Remarques 19
II. SCHÉMA CONCEPTUEL 19
1. Schéma Entités-associations 19
2. Contraintes 20

CONCEPTION STATIQUE 21
I. LE NIVEAU 5 21
1. Architecture globale 21
2. Objet ADM_COMP 21
3. Objet ADM_PRE 23
4. Objet ADM_POST 24
5. Objet INTRO_TSF 25
6. Objet INTRO_SORTIE 26
7. Objet PRESCR_SOIN 28
8. Objet PREST_PRESCR 29
9. Objet ANNUL_PRESCR 30
10. Objet RAPPEL_PREST 31
11. Objet DDE_INFO_MAL 33
12. Objet PROD_FACT 35
13. Objet FACT_PAT_OA 36
14. Objet ALLOCATION 39
15. Objet PLANNING 41
16. Objet SEJOUR 43
17. Objet PAUSE 44
18. Objet SESSION 45
II. LE NIVEAU 4 46
1. Architecture globale 46
2. Objet DIALOGUE 46
3. Objet INFO_D 47
4. Objet ERR_D 48
5. Objet MSG_D 48
6. Objet VERIF_D 48
7. Objet SER_D 50
8. Objet NR_DOS_D 50
9. Objet NR_MED_D 51
10. Objet TSF_D 51

11. Objet SORTIE_D 52
12. Objet NR_PRESCR_D 52
13. Objet PAS_VERIF_D 53
14. Objet COORD_D 54
15. Objet CONFIRM_D 54
16. Objet INFO_ADM_FULL_D 54
17. Objet INFO_ADM_POST_D 55
18. Objet PRESCR_SOIN_D 55
19. Objet MENU_D 56
III. LE NIVEAU 3 57
1. Architecture globale 57
2. Objet HOP 57
3. Objet SER 58
4. Objet SER_MED 58
5. Objet CHAMBRE 59
6. Objet CAT 60
7. Objet LIT 60
8. Objet MVT 61
9. Objet MVT_ENT 62
10. Objet MVT_TSF 63
11. Objet MVT_SOR 64
12. Objet REMB_CH 65
13. Objet REMB_PR 65
14. Objet ORG_ASS 66
15. Objet SER_TECH 66
16. Objet AFFIL 68
17. Objet PAT 68
18. Objet MED 70
19. Objet PRESCR_SOIN 71
20. Objet PREST 72
21. Objet PREST_EXT 73
22. Objet PREST_INT 74
23. Objet EFF_PREST_INT 74
24. Objet EFF_PREST_EXT 75
CONCEPTION DYNAMIQUE 76
I. LE NIVEAU 5 76
1. Architecture globale 76
2. Objet ADM_COMP 76
3. Objet ADM_PRE 78
4. Objet ADM_POST 79
5. Objet INTRO_TSF 80
6. Objet INTRO_SORTIE 81
7. Objet PRESCR_SOIN 83
8. Objet PREST_PRESCR 84
9. Objet ANNUL_PRESCR 85
10. Objet RAPPEL_PREST 86
11. Objet DDE_INFO_MAL 88
12. Objet PROD_FACT 90
13. Objet FACT_PAT_OA 91
14. Objet ALLOCATION 94
15. Objet PLANNING 95
16. Objet SEJOUR 97
17. Objet PAUSE 98
18. Optimisation 100
18.1. Allocation-lit 100
18.2. Allocation-prescription-soin 104
19. Objet SESSION 105
II. LE NIVEAU 4 107
1. Architecture globale 107
2. Objet DIALOGUE 107
3. Objet INFO_D 108
4. Objet ERR_D 109
5. Objet MSG_D 109
6. Objet VERIF_D 109
7. Objet SER_D 111
8. Objet NR_DOS_D 111
9. Objet NR_MED_D 112
10. Objet TSF_D 112
11. Objet SORTIE_D 113
12. Objet NR_PRESCR_D 113
13. Objet PAS_VERIF_D 114
14. Objet COORD_D 115
15. Objet CONFIRM_D 115
16. Objet INFO_ADM_FULL_D 115
17. Objet INFO_ADM_POST_D 116

18. Objet PRESCR_SOIN_D 116
19. Objet MENU_ACCUEIL_D 117
20. Objet MENU_DIRECTION_D 117
21. Objet MENU_COMPTA_D 117
22. Objet MENU_URGENCES_D 117
23. Objet MENU_SV_SOIN_D 118
24. Objet MENU_SV_TECH_D 118
III. LE NIVEAU 3 119
1. Architecture globale 119
2. Objet HOP 119
3. Objet SER 120
4. Objet SER_MED 120
5. Objet CHAMBRE 121
6. Objet CAT 122
7. Objet LIT 123
8. Objet MVT 124
9. Objet MVT_ENT 125
10. Objet MVT_TSF 126
11. Objet MVT_SOR 127
12. Objet REMB_CH 128
13. Objet REMB_PR 128
14. Objet ORG_ASS 129
15. Objet SER_TECH 130
16. Objet AFFIL 131
17. Objet PAT 131
18. Objet MED 133
19. Objet PRESCR_SOIN 133
20. Objet PREST 134
21. Objet PREST_EXT 135
22. Objet PREST_INT 135
23. Objet EFF_PREST_INT 136
24. Objet EFF_PREST_EXT 137

ETUDE DES BESOINS

I. ETUDE D'OPPORTUNITÉ

1. CADRE GÉNÉRAL

Une société de service informatique veut développer un logiciel de gestion hospitalière répondant aux besoins de l'hôpital MortSubite.

La société envisage plusieurs autres contrats dans le domaine de la gestion hospitalière; des négociations sont déjà engagées avec d'autres hôpitaux de tailles et de besoins différents de ceux de l'hôpital MortSubite.

Des qualités essentielles du logiciel à développer sont donc:
- généralité : la plus grande indépendance possible par rapport aux données particulières de MortSubite.
- flexibilité : adaptation facile à des besoins spécifiques d'un autre hôpital.
- extensibilité : possibilité d'automatisation de nouvelles fonctions en relation avec celles qui sont réalisées, sans devoir changer ces dernières.

2. ORGANISATION GÉNÉRALE

L'hôpital MortSubite est subdivisé en 4 unités qui sont : les services de soins, les services administratifs, le service des urgences et les services techniques comme le montre le schéma suivant :

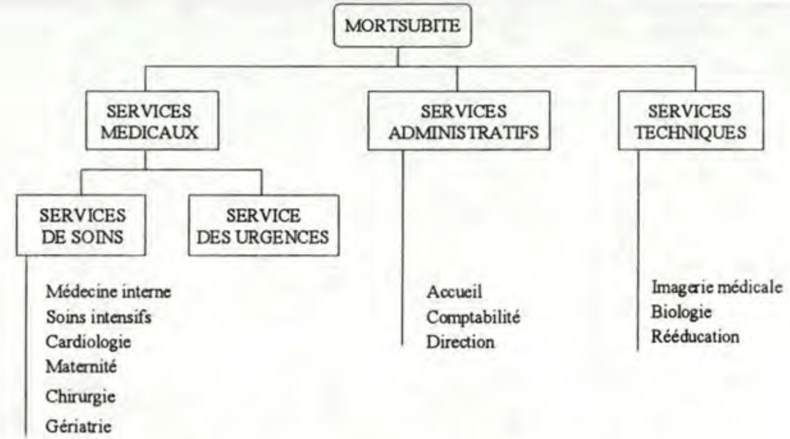


Figure 1 - Organisation générale de MortSubite

L'hôpital comprend 200 lits d'hospitalisation partagés entre les 7 services médicaux (6 services de soins et 1 service d'urgences). Le nombre de lits dans chaque service médical varie entre 8 et 24. Chaque service médical contient un certain nombre de chambres de différents types. Ainsi nous avons les types suivants:
- chambre à quatre lits,
- chambre à deux lits,
- chambre particulière.

Nous considérons que, pour un hôpital donné, la configuration des chambres et des lits est fixe. Dans chaque service médical un poste d'infirmières assure les soins quotidiens.

Le personnel de l'hôpital travaille en "feux continus" : trois "pauses" de 8 heures (6h-14h, 14h-22h, 22h-6h).

3. ACCUEIL

L'admission passe par le service d'accueil qui reçoit les informations nécessaires concernant le malade. Ce service attribue au patient un lit dans le service de soins correspondant. Le type de la chambre (1,2 ou 4 lits) correspondra au souhait du patient dans la mesure du possible. Il n'y a donc refus d'admission que dans le cas où le service demandé est complètement occupé.

Cependant, dans des cas d'urgence, l'admission se fera en partie au service des urgences où l'on demandera les informations minimales et elle sera complétée au service d'accueil lorsque le patient quittera les urgences pour entrer dans un service de soins ou lorsqu'il sortira directement de l'hôpital.

4. HOSPITALISATION

4.1. MOUVEMENTS INTERNES ET EXTERNES

Pendant son hospitalisation, le malade peut changer de service médical et/ou de lit une ou plusieurs fois (transfert en chirurgie, retour au service, changement de chambre dans le même service, etc). Il n'y a en aucun cas un transfert en direction des urgences.

Un malade peut effectuer plusieurs mouvements sur une même journée.

4.2. PRESTATIONS

Pendant son hospitalisation, le malade reçoit un certain nombre de soins et un certain nombre de médicaments [Prestations internes].
De même, il peut être envoyé une ou plusieurs fois aux services techniques (une heure de rééducation par jour, un scanner, etc) [Prestations externes].

Ces prestations font toujours l'objet d'une prescription d'un des médecins de l'hôpital.
Les infirmières du service concerné (médical ou technique) assurent le suivi de ces prescriptions.

Une prestation peut être exécutée (par une unité exécutrice) au plus un certain nombre de fois par pause. Par exemple, un scanner accepte au plus 32 patients par pause, une infirmière peut faire jusqu'à 80 piqûres par pause, une heure de rééducation "accepte" jusqu'à 8 patients par pause.

Afin d'aider les infirmières à planifier leur travail, on souhaite disposer, en début de pause, d'une liste des prestations à effectuer pour la pause en cours. Ainsi, pour les services médicaux, on leur fournira une liste des prestations internes et externes concernant leurs patients tandis que les services techniques disposeront d'une liste des prestations externes concernant l'ensemble des patients qu'ils vont recevoir.

Après une hospitalisation d'une durée quelconque, le malade quitte l'hôpital. Pour simplifier les choses, nous supposerons qu'un malade peut sortir n'importe quel jour de la semaine avec accord du médecin.

5. FACTURATION

La comptabilité facture toutes les prestations ainsi que tous les jours d'hospitalisation. Le prix d'une prestation dépend de son type, tandis que le prix d'une journée d'hospitalisation dépend lui du type de la chambre. Pour simplifier les choses, le type sera celui de la chambre que le patient occupe à 16h. Rappelons, en effet, qu'un patient peut effectuer plusieurs mouvements au cours d'une journée.

Pour chaque élément facturé (journée d'hospitalisation ou prestation) une partie est supportée par l'organisme assureur, et le reste par le patient.

Le service de comptabilité prépare mensuellement les factures concernant les malades sortant durant le mois écoulé et envoie ces factures aux patients et aux organismes assureurs. On ne facture donc que les hospitalisations terminées.

L'hôpital dispose de ses propres tarifs qui donnent :
- le prix plein unitaire pour chaque type de chambre,
- le prix plein unitaire pour chaque type de prestation,

où prix plein = montant à payer par le patient
+ montant supporté par l'organisme assureur.

On dispose aussi, et pour chaque organisme assureur connu, de l'adresse de cet organisme, ainsi que des prix unitaires remboursés par cet organisme pour chaque type de chambre ainsi que pour chaque type de prestation. Tous les tarifs ainsi que les parties supportées par les organismes assureurs peuvent changer.

6. STATISTIQUES

La direction de l'hôpital souhaite disposer de certaines statistiques comme :
- le taux d'occupation des lits de l'hôpital,
- le taux d'occupation des lits d'un service donné,
- le taux d'occupation des lits d'un type de chambre donné.

Elle souhaite aussi pouvoir calculer périodiquement les statistiques suivantes:
- le taux d'occupation de tout l'hôpital réparti sur les mois de l'année
- les taux d'occupation par service et par catégorie répartis sur les mois de l'année

Afin de simplifier le calcul des statistiques périodiques, on considérera l'état d'occupation de l'hôpital à une certaine date comme étant l'état d'occupation de l'hôpital à cette date et à une heure fixée.

7. POLITIQUE D'ALLOCATION

La direction demande aussi d'envisager la possibilité de traiter les demandes des services en parallèle. Afin de gérer la concurrence qui en résultera, elle propose les règles suivantes :

- les services ont une priorité de traitement :
  - "cas spéciaux" 0
  - urgences 1
  - soins intensifs 2
  - cardiologie 3
  - chirurgie 3
  - médecine interne 3
  - maternité 3
  - gériatrie 3

  les "cas spéciaux" étant des demandes ponctuelles qui ne peuvent souffrir de retard et qu'il convient donc d'exécuter immédiatement.

- pour l'attribution des lits, on tentera de maximiser les demandes :
  - en tenant compte des priorités. La priorité sera celle du service demandeur dans le cas d'un transfert ou celle du récepteur dans le cas d'une admission
  - sans se limiter à l'examen des demandes sur un seul lit mais en étendant aux "précédents" (sur plusieurs lits)
  - une demande de priorité 0 doit être accordée impérativement

- pour la gestion des prestations, on tentera d'atteindre le "plein emploi" des ressources :
  - en tenant compte des priorités
  - en tenant compte de la capacité disponible pour cette prestation : les demandes refusées ou "éjectées" (accordées précédemment mais cédant leurs places) seront reportées à la pause suivante avec une plus grande chance d'être accordées
  - seules les demandes de priorité 0 sont autorisées pour la pause en cours (p). Elles seront exécutées sans attendre mais sans éjecter aucune demande prévue pour cette pause : elle seront "casées" parmi les demandes prévues
  - la gestion du planning est donc à faire sur les pauses p+1 et suivantes

II. MODÈLE DE STRUCTURATION DES TRAITEMENTS

1. NIVEAU PROJET

L'objectif étant d'informatiser le fonctionnement de l'hôpital tel qu'il est décrit dans le cadre général (section II de ce chapitre), nous avons choisi de prendre la gestion de l'hôpital comme projet.

2. NIVEAU APPLICATION

Nous avons opté pour une découpe en application qui reflète la réalité organisationnelle. Nous aurons donc une application pour chaque service : l'accueil, la comptabilité, la direction, les urgences, les soins intensifs, la cardiologie, la chirurgie, la médecine interne, la maternité, la gériatrie, l'imagerie médicale, la biologie et la rééducation.

Notre choix est justifié par le fait que les services fonctionnent de façon plus ou moins autonome sans aucun échange d'information ou alors seulement via la mémoire du système.

L'objectif de chacune des applications est de gérer le fonctionnement du service concerné.

3. NIVEAU PHASE

Sur base des critères fournis dans la définition, nous avons identifié les phases de chaque application dégagée au point précédent. Nous donnerons le nom ainsi que l'objectif de chacune d'entre elles.

Application Accueil

Phases :
- admission-complète : introduire et/ou mettre à jour les informations concernant un patient lors de son admission pour une hospitalisation
- renseignements-malade : consulter la mémoire du système pour connaître des informations concernant un patient malade donné

Application Urgences

Phases :
- pré-admission-patient : introduire et/ou mettre à jour les informations minimales concernant un patient lors de son admission en urgences
- transfert-patient : transférer un patient des urgences vers un autre service.
- prescription-soin-patient : introduction d'une prescription dictée par un médecin pour un soin à administrer à un patient malade
- annuler-prescription-patient : annuler une prescription de soin
- information-patient : obtenir des informations sur un patient malade
- sortie-patient : transférer un patient vers la sortie
- exécuter-prescription-patient : exécuter une prescription de soin

Application Imagerie médicale

Phases :
- renseignements-patient : obtenir des renseignements sur un patient
- opérer-prescription-patient : opérer une prescription de soin

Application Soins intensifs & Médecine interne

Phases :
- information-malade : présenter des informations au sujet d'un malade
- prescrire-soin : prescrire un soin dicté par un médecin
- annuler-prescription : annuler une prescription
- exécuter-prescription : exécuter une prescription de soin
- introduction-sortie : introduire un mouvement de sortie d'un patient
- introduction-transfert : introduire un mouvement de transfert d'un patient vers un autre service.

Application Comptabilité

Phase :
- production-factures : facturer périodiquement les frais de séjours et de prestations
- interrogation-facture : connaître, à tout moment, ce que doit un patient (facture instantanée)

Application Cardiologie & Chirurgie

Phases :
- sortie : introduire la sortie d'un patient
- supprimer-prescription : annuler une prescription de soin pour un patient
- transfert : faire passer un patient du service de cardiologie vers un autre service.
- prescription-soin : prescrire un soin à administrer à un patient malade
- compte-rendu-malade : obtenir des renseignements sur un patient malade
- prestation-soin : prester une prescription de soin

Application Rééducation

Phases :
- information-patient : donner des information sur un patient
- exécuter-prescription-patient : exécuter une prescription de soin

Application Direction

Phases :
- rappel-prestations : production d'une liste de rappel de prestations par service. Cette liste reprend les prestations prévues pour la pause en cours à la date du jour
- demande-information-malade : obtenir des renseignements sur un malade
- statistiques-on-line : calculer le taux d'occupation instantané des lits de l'hôpital (taux d'occupation global, pour un service ou pour une catégorie de chambre)
- statistiques-périodiques : calculer périodiquement le taux d'occupation des lits par mois pour tout l'hôpital, par service ou par catégorie de chambre.

Application Maternité & Gériatrie

Phases :
- transfert-patient : déplacer un patient vers un autre service.
- prescription-soin-patient : introduire une prescription pour un soin à administrer à un patient
- annulation-prescription-patient : supprimer une prescription de soin
- information-patient : acquérir des information sur un patient
- sortie-patient : transférer un patient vers la sortie
- prestation-prescription-patient : exécuter une prescription de soin

Application Biologie

Phases :
- obtenir-info-malade : obtenir des information sur un patient malade
- accomplir-prescription-soin : accomplir une prescription de soin

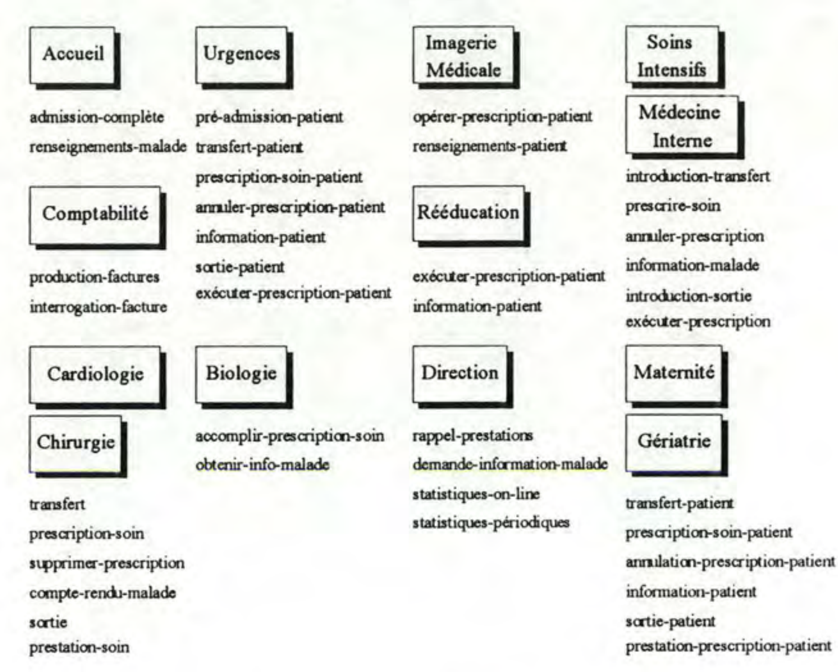


Figure 2 - Schéma récapitulatif

1. ENTITÉS

1.1. Entité HOPITAL

Définition:

Organisme assurant des soins médicaux à des malades lors d'une ou de plusieurs hospitalisations au sein de cet hôpital.

Identifiant(s):

nom_hop : le nom de l'hôpital.

Attributs:

date_der_fact : la date de la dernière facturation effectuée.
date_der_stat : la date de la dernière fois qu'un calcul de statistiques périodiques a été fait.

1.2. Entité PATIENT

Définition:

Toute personne hospitalisée (ou ayant été hospitalisée) dans l'hôpital MortSubite.

Identifiant(s):

nr_dos : le numéro de dossier du patient, ce numéro est un identifiant attribué définitivement à la première hospitalisation de la personne
(id1)
nom_pat : le nom du patient
pre_pat : son prénom (id2)
dnais : sa date de naissance

Attributs:

adr_pat : l'adresse du patient
phone : son numéro de téléphone
sexe : m ou f.
etat_civ : son état civil (célibataire, marié, veuf, ...)

1.3. Entité PRESTATION

Définition :

Type de soin ou de médicament pouvant être donné à un malade et donnant lieu à une facturation.

Identifiant(s):

code_prest : le code de la prestation.

Attributs:

nom_prest : le nom de la prestation.
type_prest : type de la prestation (interne ou externe).
cap_maxi : nombre maximum de fois que cette prestation peut être exécutée par une unité exécutrice sur l'espace d'une pause.
prix_plein_prest : le prix plein unitaire de la prestation.

Contraintes:

Si type_prest = interne alors est_effectuée_par_ext(prest) n'existe pas.
Si type_prest = externe alors est_effectuée_par_int(prest) n'existe pas.

1.4. Entité MEDECIN

Définition :

Toute personne ayant droit de prescrire une prestation ou un mouvement.

Identifiant(s):

nr_med : le numéro du médecin.

Attributs:

nom_med : le nom du médecin;
pre_med : le prénom du médecin;
adr_med : l'adresse du médecin.

1.5. Entité LIT

Définition :

Tout lit d'hospitalisation pouvant être attribué à une personne pendant toute (ou une partie de) son hospitalisation dans MortSubite.

Identifiant(s):

nr_lit : un numéro identifiant le lit dans l'hôpital.

1.6. Entité CHAMBRE

Définition :

Partie de l'hôpital contenant des lits d'hospitalisation et faisant partie d'un service médical.

Identifiant(s):

nr_ch : un numéro identifiant la chambre au sein de l'hôpital.

1.7. Entité CATEGORIE_DE_CHAMBRE

Définition :

Classe de chambres ayant toutes le même prix unitaire de séjour pour un lit.

Identifiant(s):

cat : le nom de cette catégorie (1L, 2L, etc).

Attributs:

prix_plein_lit_j : le prix plein d'un séjour d'une journée d'hospitalisation dans un lit d'une chambre de cette catégorie.

1.8. Entité SERVICE_MEDICAL

Définition :

Unité fonctionnelle de MortSubite comprenant des chambres qui contiennent des lits d'hospitalisation. Regroupe les services de soins et le service d'urgence.

Identifiant(s):

ser : le nom du service

1.9. Entité SERVICE_TECHNIQUE

Définition :

Unité fonctionnelle de MortSubite ne comprenant pas de chambres. Propose des prestations (radio, scanner) aux services médicaux.

Identifiant(s):

ser_tech : le nom du service

1.10. Entité MOUVEMENT

Définition :

Changement d'attribution de lit à une personne; un mouvement peut donc être d'un des 3 types:
- entrée: mouvement dont l'origine est extérieure à l'hôpital, et la destination est un lit d'hospitalisation.
- sortie : mouvement dont l'origine est un lit d'hospitalisation et la destination est extérieure à l'hôpital.
- transfert: mouvement dont l'origine et la destination sont des lits d'hospitalisation.

Identifiant(s):

nr_mv : un numéro identifiant le mouvement.

Attributs:

date_mv : la date effective du mouvement.
heure_mv : l'heure effective du mouvement
type_mv : (entrée, sortie, ou transfert).

Contraintes :

- Un mouvement de type entrée est associé à un seul "lit" par "destination".
- Un mouvement de type sortie est associé à un seul "lit" par "origine".
- Un mouvement de type transfert est associé à deux "lits" différents par "origine" et "destination".

1.11. Entité ORGANISME_ASSUREUR

Définition :

Association qui rembourse une partie des frais d'hospitalisation des personnes affiliées.

Identifiant(s):

nr_oa : un numéro identifiant l'organisme assureur.

Attributs:

nom_oa : le nom de l'organisme assureur.
adr_oa : l'adresse de l'organisme assureur.

1.12. Entité PRESCRIPTION_SOIN

Définition :

Exprime la prescription par un médecin d'un soin ou d'un médicament à administrer à un malade.

Attributs:

nr_pr : numéro d'ordre identifiant une prescription et attribué lors de la création de cette entité.
date_pr : la date prévue pour l'exécution de la prestation.
heure_pr : l'heure prévue pour l'exécution de la prestation.
état_pr : l'état actuel de la prescription (en attente ou exécutée).

2. ASSOCIATIONS

2.1. Association PR_PRESCRITE

Définition:

Associe une prescription de soins à la prestation concernée.

2.2. Association : PR_PRESCRITE_POUR

Définition:

Associe une prescription de soins au patient qui va la subir.

2.3. Association PR_PRESCRITE_PAR

Définition:

Associe une prescription de soins au médecin qui l'a prescrite.

2.4. Association AFFILIATION

Définition :

Associe un patient à l'organisme assureur auquel il est affilié.

Attributs:

nr_tit : le numéro de matricule du titulaire de l'assurance.
type_aff : le type de l'affiliation (vipo, salarié, indépendant...).

2.5. Association DEMANDE_MOUVEMENT

Définition :

Associe un mouvement au médecin demandeur de ce mouvement.

2.6. Association EFFECTUE_MVT

Définition :

Associe un mouvement au patient qui l'a effectué.

2.7. Association DESTINATION

Définition :

Associe un mouvement (du type entrée ou transfert) au lit de sa destination c-à-d. le lit où le patient se trouvera après le mouvement.

2.8. Association ORIGINE

Définition :

Associe un mouvement (du type transfert ou sortie) au lit de son origine c-à-d. le lit où le patient se trouvait avant le mouvement.

2.9. Association APPARTIENT

Définition :

Associe un lit à la chambre où il se trouve.

2.10. Association PARTIE_DE

Définition :

Associe une chambre au service de soins dont elle fait partie.

2.11. Association COMPORTE

Définition :

Associe un service de soins à l'hôpital dont il fait partie.

2.12. Association REMBOURSEMENT_CHAMBRE

Définition :

Associe une catégorie de chambres à un organisme assureur qui rembourse une partie du prix unitaire de séjour dans cette catégorie de chambre.

Attributs:

prix_remb_lit_j : le prix remboursé par l'organisme assureur pour un séjour d'une journée dans un lit d'une chambre de cette catégorie.

2.13. Association REMBOURSEMENT_PRESTATION

Définition :

Associe une prestation à un organisme assureur qui rembourse une partie du prix unitaire de cette prestation.

Attributs:

prix_remb_prest : le prix remboursé par l'organisme assureur pour cette prestation.

2.14. Association EST_DE_CATEGORIE

Définition :

Associe une chambre à la classe de chambres dont elle fait partie.

2.15. Association EFFECTUE_PREST_INT

Définition :

Associe une prestation interne à un service médical.

Attributs:

nb_unités_exéc : nombre d'unités exécutrices pour cette prestation dans ce service.

2.16. Association EFFECTUE_PREST_EXT

Définition :

Associe une prestation externe au seul service technique qui la propose.

Attributs:

nb_unités_exéc : nombre d'unités exécutrices pour cette prestation dans ce service.

SPÉCIFICATIONS FONCTIONNELLES

I. FONCTIONS

1. DESCRIPTION DES FONCTIONS

1.1. FONCTION ADMISSION_COMPLETE

Objectifs

Introduire et/ou mettre à jour les informations concernant un patient lors de son admission pour une hospitalisation.

Messages d'entrée

- nom_pat : le nom de la personne qui demande l'admission
- pre_pat : son prénom
- dnais : sa date de naissance
- ser : le service de soins demandé
- cat : la catégorie de chambre souhaitée
- nr_med : le numéro du médecin qui en cas d'admission sera responsable du mouvement d'entrée
- adr_pat : l'adresse du patient
- phone : son numéro de téléphone
- sexe : (M ou F).
- état_civ : son état civil
- nr_oa : identifiant de son organisme assureur.
- nr_tit : le numéro du titulaire de l'assurance.
- type_aff : son type d'affiliation à l'organisme assureur.

Les données "ser", "cat" et "nr_med" identifient des entités de l'hôpital.

Messages de sortie

Message1: "le service est plein".
Message2: "le patient est déjà malade dans MortSubite".
Message3: <nr_dos, nr_lit>.

Actions sur le S.I.

E1: Création d'une occurrence de "patient".
E2: Mise-à-jour des attributs d'une occurrence de "patient" .
E3: Création des entités et associations: "affiliation", "effectue_mvt", "demande_mouvement", "mouvement" (du type entrée), et "destination".

Règles de traitement

Dénotons par C(i) et E(i) les Conditions et Effets suivants:

C1: L'identifiant introduit correspond à un patient connu (c-à-d nr_dos existant).
C2: L'identifiant introduit correspond à un patient actuellement malade.
C3: Le service désiré est complètement plein.
C4: Les chambres du type souhaité dans le service "ser" sont toutes pleines.
E4: Production du message1.
E5: Production du message2.
E6: Attribution d'un lit dans une chambre du type souhaité.
E7: Attribution d'un lit dans une chambre d'un type autre que le type souhaité.
E8: Production du message3 .

Les règles de traitement sont illustrées par la table suivante:

	R1	R2	R3	R4	R5	R6	R7	R8
C1	V	F	-	-	V	V	F	F
C2	V	V	F	F	F	F	F	F
C3	-	-	V	V	F	F	F	F
C4	-	-	V	F	F	V	F	V
E1							X	X
E2					X	X		
E3					X	X	X	X
E4			X					
E5	X							
E6					X		X	
E7						X		X
E8					X	X	X	X
??		X		X				

1.2. FONCTION ADMISSION_PRE

Objectifs

Introduire les informations minimales concernant un patient lors de son admission en urgence.

Messages d'entrée

- nom_pat : le nom de la personne qui demande l'admission
- pre_pat : son prénom
- dnais : sa date de naissance
- nr_med : le numéro du médecin qui en cas d'admission sera responsable du mouvement d'entrée

Le "nr_med" identifie l'entité Médecin de l'hôpital.

Messages de sortie

Message1: "le service des urgences est plein".
Message2: "le patient est déjà malade dans MortSubite".
Message3: <nr_dos, nr_lit>.

Actions sur le S.I.

E1: Création d'une occurrence de "patient".
E2: Création des entités et associations: "effectue_mvt", "demande_mouvement", "mouvement" (du type entrée), et "destination".

Règles de traitement

Dénotons par C(i) et E(i) les Conditions et Effets suivants:

C1: L'identifiant introduit correspond à un patient connu (c-à-d nr_dos existant).
C2: L'identifiant introduit correspond à un patient actuellement malade.
C3: Le service des urgences est complètement plein.
E3: Production du message1.
E4: Production du message2.
E5: Attribution d'un lit dans le service des urgences.
E6: Production du message3.
??: impossible.

Les règles de traitement sont illustrées par la table suivante:

	R1	R2	R3	R4	R5
C1	V	F	-	V	F
C2	V	V	F	F	F
C3	-	-	V	F	F
E1					X
E2				X	X
E3			X		
E4	X				
E5				X	X
E6				X	X
??		X			

1.3. FONCTION ADMISSION_POST

Objectifs

Compléter les informations concernant un patient malade lorsqu'il quitte le service d'urgence pour sortir de l'hôpital ou être transféré dans un service de soins.

Messages d'entrée

- nr_dos : le numéro de dossier du patient
- adr_pat : l'adresse du patient
- phone : son numéro de téléphone
- sexe : (M ou F).
- état_civ : son état civil
- nr_oa : identifiant de son organisme assureur.
- nr_tit : le numéro du titulaire de l'assurance.
- type_aff : son type d'affiliation à l'organisme assureur.

Nr_dos identifie un patient malade dans le service des urgences.

Messages de sortie

/

Actions sur le S.I.

E1: Mise-à-jour des attributs d'une occurrence de "patient" .
E2: Création de l'association "affiliation".

Règles de traitement

Exécution de E1 et E2.

1.4. FONCTION INTRODUCTION_TRANSFERT

Objectifs

Introduction (si c'est possible) d'un mouvement du type transfert pour un patient malade à l'hôpital.

Messages d'entrée

- nr_dos : un numéro de dossier d'un patient malade.
- nr_med : le numéro du médecin demandeur du transfert.
- nr_lit_or : le numéro du lit d'origine, où le patient est sensé être avant le transfert.
- nr_lit_des : le numéro du lit de la destination voulue.

Les données "nr_dos", "nr_med", "nr_lit_or" et "nr_lit_des" identifient des entités de l'hôpital.

Messages de sortie

Message1: "le patient n'est pas dans le lit d'origine".
Message2: "le lit de destination est occupé".

Actions sur le S.I.

Création des occurrences de: "mouvement" de type transfert, "demande-mouvement", "effectue_mvt", "origine" et "destination".

Règles de traitement

Si le patient se trouve dans le lit d'origine, et si la destination est un lit non-occupé, alors il faut mettre le S.I à jour pour ajouter le mouvement demandé. Si, en plus, le patient est actuellement en urgence, alors il faut déclencher la fonction admission_post; sinon, il faut produire le message adéquat.

1.5. FONCTION INTRODUCTION_SORTIE

Objectifs

Introduction (si c'est possible) d'un mouvement du type sortie pour un patient malade à l'hôpital.

Messages d'entrée

nr_dos : le numéro de dossier d'un patient malade.
nr_med : le numéro du médecin demandeur du transfert.
nr_lit_or : le numéro du lit d'origine, où le patient est sensé être.

Les données "nr_dos", "nr_med" et "nr_lit_or" identifient des entités de l'hôpital.

Messages de sortie

"le patient n'est pas dans le lit d'origine".

Actions sur le S.I.

Création des occurrences de: "mouvement" de type sortie, "demande-mouvement", "effectue_mvt", et "origine".

Règles de traitement

Si le patient se trouve dans le lit d'origine, alors il faut mettre le S.I à jour pour ajouter le mouvement demandé. Si, en plus, le patient est actuellement en urgence, alors il faut déclencher la fonction admission_post; sinon, il faut produire le message adéquat.

1.6. FONCTION PRESCRIPTION_SOIN

Objectifs

Introduction d'une prescription dictée par un médecin pour un soin (au sens large : prestation interne ou externe) à administrer à un patient malade.

Messages d'entrée

nr_dos : le numéro du dossier du malade concerné.
nr_med : le numéro du médecin.
code_prest : le code de la prestation prescrite.
date_pr : la date d'exécution prévue.
heure_pr : l'heure d'exécution prévue.
priorité_0 : indique si oui ou non la demande porte la priorité 0.

Les données "nr_dos", "nr_med" et "code_prest" identifient des entités de l'hôpital.

La date et l'heure prévue représentent une pause qui est au moins supérieure à la pause en cours sauf si la demande est de priorité 0.

Actions sur le S.I.

Création d'une occurrence de "prescription_soins", "pr_prescrite_pour", "pr_prescrite_par", "pr_prescrite"

Règles de traitement

L'état de la prescription créée est "en_attente".

1.7. FONCTION PRESTATION_D'UNE_PRESCRIPTION

Objectifs

Confirmer l'exécution d'une prescription d'un soin.

Messages d'entrée

nr_pr : le numéro de la prescription. "nr_pr" 'identifie une prescription de soins en attente prévue pour la date du jour.

Actions sur le S.I.

Changement de l'attribut "état" de la prescription concernée.

1.8. FONCTION ANNULATION_PRESCRIPTION

Objectifs

Supprimer une prescription d'une prestation non exécutée (par exemple à cause d'un changement de traitement, de la sortie du malade,....)

Messages d'entrée

nr_pr : le numéro de la prescription concernée. "nr_pr" identifie une prescription de soins qui est en attente.

Actions sur le S.I.

Suppression de l'occurrence concernée de "prescription_soin", "pr_prescrite_pour", "pr_prescrite_par", "pr_prescrite"

1.9. FONCTION RAPPEL_PRESTATION

Objectifs

Production d'une liste de rappel de prestations par service (médical ou technique). Cette liste reprend les prestations prévues pour la pause en cours à la date du jour et non encore exécutées.
Chaque service médical recevra une liste de rappel des prestations internes et externes à effectuer sur ses patients. Chaque service technique recevra une liste des prestations externes ("commandées" par les services médicaux) à effectuer.

Messages de sortie

Services médicaux

La sortie est un ensemble de listes de rappels (une par service). Chaque élément de cet ensemble est composé du nom de service concerné et d'une liste des rappels le concernant.
Cette liste est un ensemble de rappels où chaque rappel est un tuple de la forme: <nr_dos, nr_pr, date_pr, heure_pr, code_prest, nom_prest>.

Services techniques

La sortie est un ensemble de listes de rappels (une par service). Chaque élément de cet ensemble est composé du nom de service concerné et d'une liste des rappels concernant ce service.
Cette liste est un ensemble de rappels où chaque rappel est un tuple de la forme: <nr_dos, ser, nr_pr, date_pr, heure_pr, code_prest, nom_prest>.

Actions sur le S.I.

\

Règles de traitement

Pour tout service (médical ou technique) de l'hôpital il faut produire une liste de rappels. Cette liste contient les prescriptions à rappeler concernant les malades séjournant dans ce service. Une prescription est à rappeler si sa date et son heure prévue est inférieure ou égale à la pause en cours à la date du jour et si elle est en attente.

1.10. FONCTION DEMANDE_INFO_MALADE

Objectifs

Consulter la base de données pour connaître des informations concernant un patient malade donné.

Messages d'entrée

nom_pat : le nom du malade
pre_pat : son prénom
dnais : sa date de naissance

Messages de sortie

- "Message1": Les informations suivantes sur le malade même : "Message1A"

- nr_dos : le numéro du dossier du malade cherché.

"Message1B"
- adr_pat : son adresse
- phone : son numéro de téléphone
- état_civ : son état civil
- sexe : (M ou F).

- "Message2" qui contient les informations suivantes concernant l'organisme assureur du malade:
  - nr_oa : le numéro de l'organisme assureur
  - adr_oa : l'adresse de l'organisme assureur
  - nom_oa : le nom de l'organisme assureur
  - nr_tit : le numéro du titulaire de l'assurance
  - type_aff : son type d'affiliation (VIPO, ...)

- "Message3" qui contient des informations concernant les mouvements effectués par le malade lors de l'hospitalisation courante. "Message3" est une liste de mouvements dans laquelle chaque ligne représente un mouvement effectué par le malade concerné; une ligne est un tuple de la forme: <date_mv, heure_mv, type_mv, nr_med, nr-lit_or, nr_lit_des>.

- "Message4" est une liste de prestations dans laquelle chaque ligne représente une prestation effectuée pour le malade concerné; une ligne est un tuple de la forme:
  <nr_pr, date_pr, code_prest, nr_med>.

Actions sur le S.I.

\

Règles de traitement

Les informations fournies (message1A, 1B, 2, 3 et 4) en sortie concernent le patient malade identifié par le triplet fourni en entrée et ne se trouvant pas dans le service d'urgence.
Seuls les messages 1B, 3 et 4 seront fournis si le patient malade est en urgence.

1.11. FONCTION PRODUCTION_FACTURES

Objectifs

Facturer périodiquement les frais de séjour et de prestations concernant les patients sortis durant la période écoulée depuis la dernière facturation. Deux ensembles de factures seront produits :
- les factures des patients, chacune de ces factures reprend les frais à charge d'un patient à facturer
- les factures des organismes assureurs, chacune de ces factures reprend les frais à charge d'un organisme pour un patient affilié

Messages de sortie

- Un ensemble de factures destinées aux patients. Une facture de cet ensemble est composée des parties suivantes:
  - L'en-tête : un tuple de la forme <nr_dos, nom_pat, adr_pat, nom_oa, adr_oa>.
  - La partie "frais de séjour": une suite de lignes ayant chacune la forme:
    <cat, prix_plein_lit_j, prix_remb_lit_j, prix_pat_lit_j, date_debut_sej, date_fin_sej,
    duree_sej, cout_sejour_oa, cout_sejour_pat>.
  - La partie "frais de prestations": une suite de lignes ayant chacune la forme:
    <code_prest, nom_prest, date_pr, heure_prest, prix_plein_prest, prix_remb_prest, prix_pat_prest>.
  - Le total dû par le patient.
- Un ensemble de factures destinées aux organismes assureurs des patients à facturer. Une facture de cet ensemble a la même structure et le même contenu qu'une facture destinée au patient, sauf en ce qui concerne la dernière partie qui contient - dans ce cas - le total dû par l'organisme assureur.

Actions sur le S.I.

Mise à jour de l'attribut "date_der_fact" de l'hôpital.

Règles de traitement

- Toute "hospitalisation" terminée entre la date de la dernière facturation et la date du jour est une "hospitalisation à facturer". Le patient concerné est le "patient à facturer".

- A chaque "hospitalisation à facturer" correspond une et une seule facture dans l'ensemble des factures adressées aux patients et une et une seule facture dans l'ensemble des factures adressées aux organismes assureurs; ces deux ensembles ne contiennent que des factures relatives à des "hospitalisations à facturer". Les 2 factures correspondant à une hospitalisation d'un patient sont identiques sauf pour la partie "total".

- Toute paire de mouvements successifs et facturables de l' "hospitalisation à facturer" donne lieu à une et une seule ligne dans la partie "frais de séjour" de chacune des 2 factures concernant cette hospitalisation (séjour facturable).

- Toute prescription de soins concernant un "patient à facturer", avec un état "exécutée" et une date comprise entre le début et la fin de l' "hospitalisation à facturer" donne lieu à une et une seule ligne dans la partie "frais de prestations" de chacune des 2 factures concernant cette hospitalisation.

- Dans une facture:
  "prix_pat_lit_j" = "prix_plein_lit_j" - "prix_remb_lit_j"
  "duree_sej" = "date_fin_sej" - "date_debut_sej".
  "cout_sej_oa" = "duree_sej" * "prix_remb_lit_j"
  "cout_sejour_pat" = "duree_sej" * "prix_pat_lit_j"
  "prix_pat_prest" = "prix_plein_prest" - "prix_remb_prest".

Le total dû par le patient est la somme de tous les "cout_sej_pat" + la somme de tous les "prix_pat_prest".
Le total dû par l'organisme assureur est la somme de tous les "cout_sej_oa" + la somme de tous les "prix_remb_prest".

Rappelons qu'un patient peut effectuer plusieurs mouvements sur une journée. L'état d'occupation de l'hôpital à une certaine date sera donc l'état d'occupation de cet hôpital à la date et à une heure fixée : les sorties étant généralement avant 14h, on peut convenir que cette heure est 16h.

1.12. FONCTION INTERROGATION_FACTURE

Objectifs

Cette fonction permet de savoir à chaque instant ce que doit un malade donné jusqu'à la date de la demande (abstraction faite de ce que doit son organisme assureur). Ceci revient à produire une facture concernant un seul patient et reprenant les frais de son hospitalisation comme si cette hospitalisation se termine le jour même de la demande.

Messages d'entrée

nr_dos : le numéro du dossier du patient malade.

Messages de sortie

- La structure du message est identique à la structure d'une facture destinée à un patient décrite dans la fonction PRODUCTION FACTURES ci-dessus.

Actions sur le S.I.

\

Règles de traitement

En considérant le malade identifié par "nr_dos" comme un "patient à facturer", et considérant son hospitalisation courante comme une hospitalisation qui se termine la date du jour; la facture est parfaitement décrite par les règles de la fonction PRODUCTION FACTURES ci-dessus.

1.13. FONCTION STATISTIQUES_ON_LINE

Objectifs

Calcul on-line de différents taux d'occupation de lits dans l'hôpital MortSubite.

Messages d'entrée

- glob : indique que l'on désire le taux d'occupation global ou
- ser : le nom du service pour lequel on souhaite calculer le taux ou
- cat : le nom de la catégorie pour laquelle on souhaite calculer le taux

Messages de sortie

- Taux_glob: le taux d'occupation global de l'hôpital et pour tout service de l'hôpital
- Taux_ser: le taux d'occupation de ce service et pour toute catégorie de chambre
- Taux_cat: le taux d'occupation de cette catégorie de chambre.

Actions sur le S.I.

\

Règles de traitement

- Taux-glob = (100 * nbroc_hop) / lithop
  où:
  - nbroc_hop = le nombre de lits occupés dans MortSubite.
  - lithop = le nombre total de lits dans MortSubite.
- Taux-ser = (100 * nbroc_ser) / lit_ser
  où
  - nbroc_ser = le nombre de lits occupés du service.
  - lit_ser = le nombre total de lits du service.
- Taux-cat = (100 * nbroc_cat) / lit_cat
  où:
  - nbroc_cat = le nombre de lits occupés appartenant à des chambres de la catégorie ,
  - lit_cat = le nombre total de lits appartenant à des chambres de la catégorie.

1.14. FONCTION STATISTIQUES_PERIODIQUES

Objectifs

Calculer périodiquement quelques statistiques intéressantes concernant les séjours dans MortSubite. Nous envisageons actuellement le taux d'occupation par mois pour tout l'hôpital, le taux d'occupation par mois par service, et le taux d'occupation par mois par catégorie de chambre.

Messages de sortie

- Les taux d'occupation de l'hôpital répartis par mois, une suite de tuples :
  <mois, Taux-glob-pm, liste-taux-ser, liste-taux cat>
  où:
  liste-taux-ser est une suite de couple <ser, Taux-ser-pm>
  liste-taux cat est une suite de couple <cat, Taux cat-pm>

Actions sur le S.I.

Mise à jour de l'attribut "date_der_stat".

Règles de traitement

A chaque mois écoulé depuis la date du dernier calcul de statistiques "date_der_stat" correspond un et un seul tuple <mois, Taux-glob-pm, liste-taux-ser, liste-taux cat> dans lequel
- mois est le mois concerné
- Taux-glob-pm = (100 * nuitées_hop) / (jourmois * lithop)
  où:
  - nuitées = la somme, pour chaque lit de l'hôpital, du nombre de journées d'hospitalisation passées dans ce lit pendant ce mois.
  - jourmois = le nombre de jours du mois.
  - lithop = le nombre de lits dans MortSubite.
- A chaque service de l'hôpital correspond un et un seul couple <ser, Taux-ser-pm> de la suite liste-taux-ser dans lequel
  - ser est le service concerné
  - Taux-ser-pm= (100 * nuitées_ser) / (jourmois * lit_ser)
  où:

a) nuitées_ser = la somme, pour chaque lit du service concerné, du nombre de journées d'hospitalisation passées dans ce lit pendant ce mois.
b) journois = le nombre de jours du mois
c) lit_ser = le nombre total de lits du service.

- A chaque catégorie de chambre de l'hôpital correspond un et un seul couple <cat,Taux-cat-pm> de la suite liste-taux-cat dans lequel
  - cat est la catégorie concernée
  - Taux-cat-pm = (100 * nuitées_cat) / (jourmois * lit_cat)

  où:

  a) nuitées_cat = la somme, pour chaque lit de la catégorie concernée, du nombre de journées d'hospitalisation passées dans ce lit pendant ce mois.
  b) jourmois = le nombre de jours du mois
  c) lit_cat = le nombre total de lits appartenant à des chambres de la catégorie.

Rappelons qu'un patient peut effectuer plusieurs mouvements sur une journée. L'état de l'hôpital à une certaine date sera donc l'état de cet hôpital à la date et à une heure fixée : 16h.

2. REMARQUES

- L'aspect syntaxique a été omis intentionnellement dans les spécifications précédentes. Le schéma conceptuel ainsi que les messages de fonctions ne décrivent qu'une structure abstraite des données. La définition de formats adéquats pour les données permanentes ainsi que pour les messages externes est laissée pour la suite.

- Les spécifications précédentes couvrent la fonctionnalité et les données, l'interface utilisateur n'a pas été spécifiée. Néanmoins, on demande que l'interface soit "amicale" et "ergonomique". Ainsi, les modules qui couvrent cette interface sont à définir puis à réaliser. Ces modules assureraient aussi la validation syntaxique par rapport aux formats choisis dans le paragraphe précédent.

II. SCHÉMA CONCEPTUEL

1. SCHÉMA ENTITÉS-ASSOCIATIONS

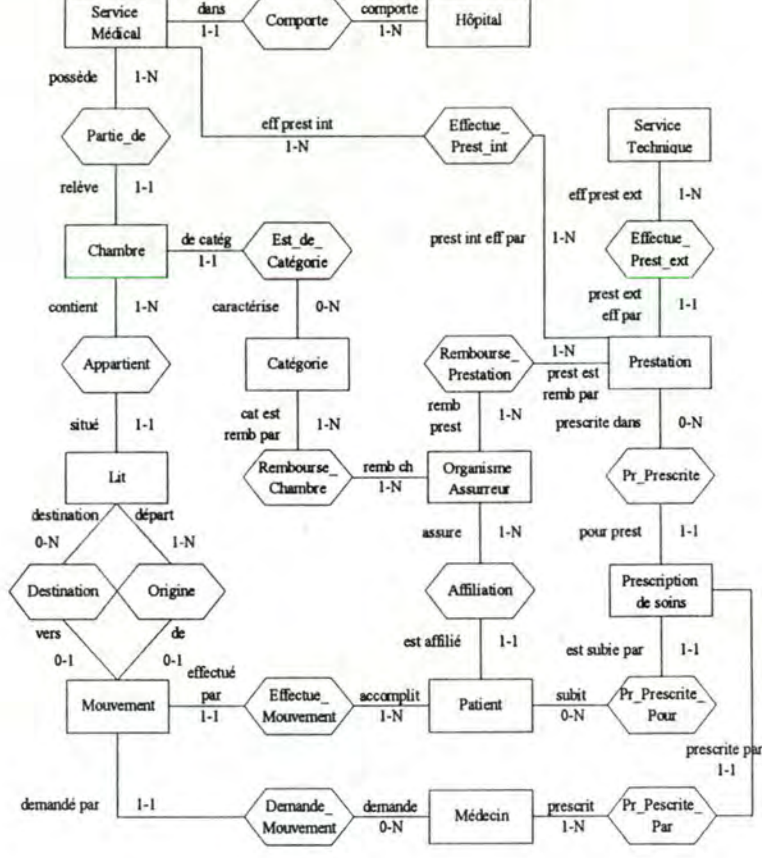


Figure 1 - Schéma E-A de l'hôpital

2. CONTRAINTES

- Un patient est toute personne admise (au moins une fois) à l'hôpital MortSubite, et ayant donc un numéro de dossier dans cet hôpital.

- Un patient malade est tout patient ayant actuellement un lit attribué.

- Un ancien malade est tout patient n'ayant pas actuellement un lit attribué.

- Un patient malade P occupe un lit L si le mouvement le plus récent effectué par ce malade a le lit L comme destination. On dit donc que le lit L est occupé par le patient P.

- Un malade se trouve dans la chambre C si ce malade occupe un lit appartenant à cette chambre.

- Un malade se trouve dans le service S si ce malade se trouve dans une chambre C faisant partie du service S.

- Une chambre est pleine si tous les lits appartenant à cette chambre sont occupés.

- Un service est plein si toutes les chambres appartenant à ce service sont pleines.

- Deux mouvements MV1 et MV2 effectués par un patient P sont des mouvements successifs et facturables si :
  - MV1 est le mouvement le plus proche effectué par le patient P avant une heure fixée
  - et MV2 est le mouvement le plus proche effectué par le patient P après la date et l'heure de MV1 + 24h

- Un séjour facturable du patient P dans le lit L est la période écoulée entre deux mouvements successifs et facturables MV1 et MV2 de ce patient:
  - MV1 ayant L comme destination,
  - et s'il existe un mouvement MV3 (non nécessairement différent de MV2) tel que :
    a) date/heure(MV1)<date/heure(MV3)<=date/heure(MV2) et,
    b) origine(MV3)=L
  - la date du mouvement MV1 est le début du séjour
  - et la date du mouvement MV2 est la fin du séjour.

- Deux séjours du patient P sont successifs si le début du deuxième séjour est égal à la fin du premier séjour.
- Une hospitalisation d'un patient P est la période écoulée entre D1 et D2 (D1 < D2) dont:
  - D1 est la date/heure d'un mouvement "entrée" effectué par le patient P;
  - D2 est
    a) soit la date/heure du mouvement "sortie" le plus proche de D1, on parle alors d'une hospitalisation terminée,
    b) soit la date/heure du jour. S'il n'existe aucun mouvement "sortie" postérieure à D1, on parle alors de l'hospitalisation courante.

CONCEPTION STATIQUE

I. LE NIVEAU 5

1. ARCHITECTURE GLOBALE

NIVEAU 5
Fonctionnalités

SESSION

ADM_COMP ADM_PRE ADM_POST INTRO_TSF INTRO_SORTIE DDE_INFO_MAL

PRESCR_SOIN PREST_PRESCR ANNUL_PRESCR RAPPEL_PREST PROD_FACT

ALLOCATION PLANNING PAUSE SEJOUR FACT_PAT_OA

RAP_FACT RAP_SER_MED RAP_SER_TECH RAP_INFO_MAL

2. OBJET ADM_COMP

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

BODY

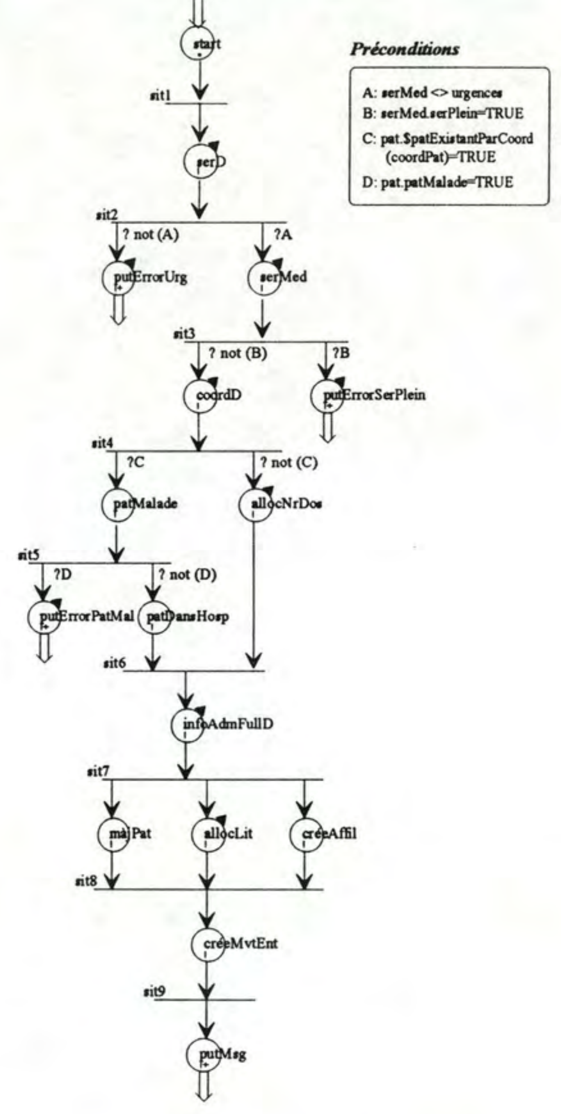
- **Attributes**
  - serMed : SER_MED
  - coordPat : COORD_DT
  - cat : CAT
  - med : MED
  - adrPat : ADR_DT
  - tel : PHONE_DT
  - sexe : SEX_DT
  - etatCiv : ETAT_CIV_DT
  - oA : ORG_ASS
  - nrTit : NR_DT
  - typeAff : TYPE_AFF_DT
  - lit : LIT
  - pat : PAT

- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")

- **Actions**
  - * start
  - ! serD (serM : SER_MED)
    >> SER_D.open
    ⇒ serMed := serD.serM
  - ! serMed
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorUrg
    IS putError (code)

:= code = ADM_COMP_URG
- !+ putErrorSerPlein
  IS putError (code)
  := code = ADM_SER_PLEIN
- !+ putErrorPatMal
  IS putError (code)
  := code = ADM_PAT_MAL
- ! coordD (coordP : COORD_DT)
  >> COORD_D.open (coordP)
  ⇒ coordPat := coordD.coordP
- ! urgences
- ! patMalade
  ⇒ pat := pat.$patParCoord(coordPat)
- ! patDansHosp
- ! allocNrDos (p : PAT)
  >> allocation.allocNrDos (coordPat, p)
  ⇒ pat := allocNrDos.p
- ! infoAdmCompD (cat : CAT, med : MED, adr : ADR_DT, tel : PHONE_DT, sexe : SEX_DT, etatCiv : ETAT_CIV_DT, oA : ORG_ASS, nrTit : NR_DT)
  >> INFO_ADM_COMP_D.open (cat, med, adr, tel, sexe, etatCiv, oA, nrTit)
  ⇒ cat := infoAdmCompD.cat
  ⇒ ...
  ⇒ nrTit := infoAdmCompD.nrTit
- ! majPat
  >> pat.fixeAdr (adrPat)
  >> pat.fixeTel (tel)
  >> pat.fixeSexe (sexe)
  >> pat.fixeEtatCiv (etatCiv)
- ! allocLit (lit : LIT)
  >> allocation.allocLit (cat, serMed, lit)
  ⇒ lit := allocLit.lit
- ! creeMvtEntree
  >> mvt_ent.birth (lit, pat, med, TODAY(), NOW())
- ! creeAffil
  >> affil.birth (pat, oA, nrTit, typeAff)
- !+ putMsg (code : MSG_CODE_DT)
  >> MSG_D.open (code)
  := code = ADM_COMP_OK

BEHAVIOUR



3. OBJET ADM_PRE

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

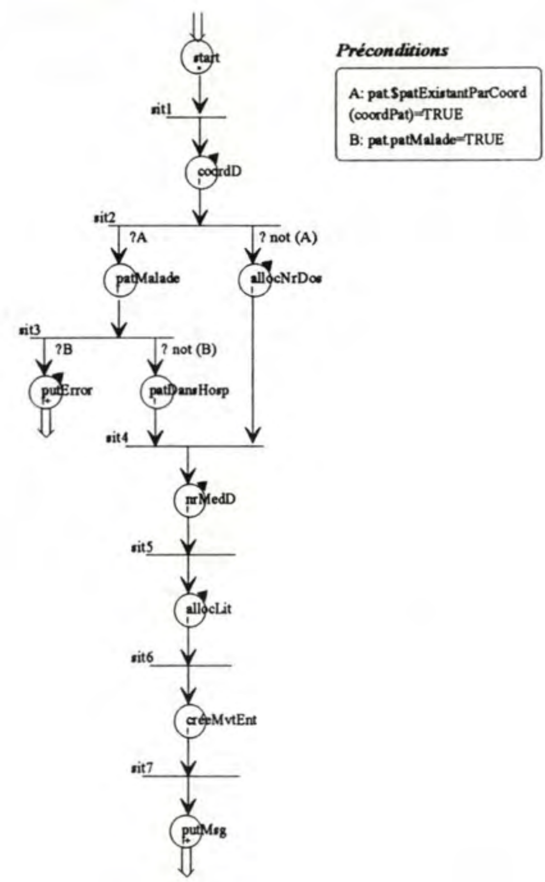
BODY

- **Attributes**
  - coordPat : COORD_DT
  - med : MED
  - lit : LIT
  - pat : PAT

- **Derivations**
  - urgences : SER_MED
    := SER_MED.$serviceMedical("Urgences")
  - cat : CAT
    := cat.$catégorie("1Lit")

- **Actions**
  - * start
  - ! coordD (coordP : COORD_DT)
    >> COORD_D.open (coordP)
    ⇒ coordPat := coordD.coordP
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
    := code = ADM_PAT_MAL
  - ! patMalade
    ⇒ pat := pat.$patParCoord(coordPat)
  - ! patDansHosp
  - ! allocNrDos (p : PAT)
    >> allocation.allocNrDos (coordPat, p)
    ⇒ pat := allocNrDos.p
  - ! nrMedD (med : MED)
    >> NR_MED_D.open (med)
    ⇒ med := nrMedD.med
  - ! allocLit (lit : LIT)
    >> allocation.allocLit (cat, urgences, lit)
    ⇒ lit := allocLit.lit
  - ! creeMvtEntree
    >> mvt_ent.birth (lit, pat, med, TODAY(), NOW())
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = ADM_PRE_OK

BEHAVIOUR



4. OBJET ADM_POST

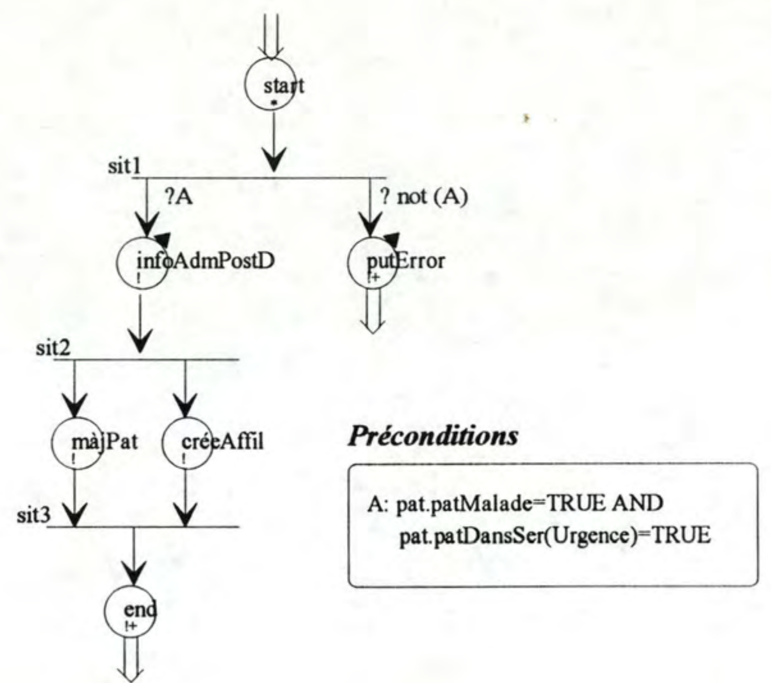
INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start (pat : PAT)

BODY

- **Attributes**
  - adrPat : ADR_DT
  - tel : PHONE_DT
  - sexe : SEX_DT
  - etatCiv : ETAT_CIV_DT
  - oA : ORG_ASS
  - nrTit : NR_DT
  - typeAff : TYPE_AFF_DT
  - pat : PAT
- **Derivations**
  - Urgence : SER_MED
    := ser_med.$serviceMedical("Urgence")
- **Actions**
  - * start (pat)
    ⇒ pat := start.pat
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
    := code = ADM_POST_NOT_IN_URG
  - ! infoAdmPostD (adr : ADR_DT, tel : PHONE_DT, sexe : SEX_DT, etatCiv : ETAT_CIV_DT, oA : ORG_ASS, nrTit : NR_DT)
    >> INFO_ADM_POST_D.open (adr, tel, sexe, etatCiv, oA, nrTit)
    ⇒ adr := infoAdmPostD.adr
    ⇒ ...
    ⇒ nrTit := infoAdmPostD.nrTit
  - ! majPat
    >> pat.fixeAdr (adrPat)
    >> pat.fixeTel (tel)
    >> pat.fixeEtatCiv (etatCiv)
  - ! creeAffil
    >> affil.birth (pat, oA, nrTit, typeAff)
  - !+ end

BEHAVIOUR



5. OBJET INTRO_TSF

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

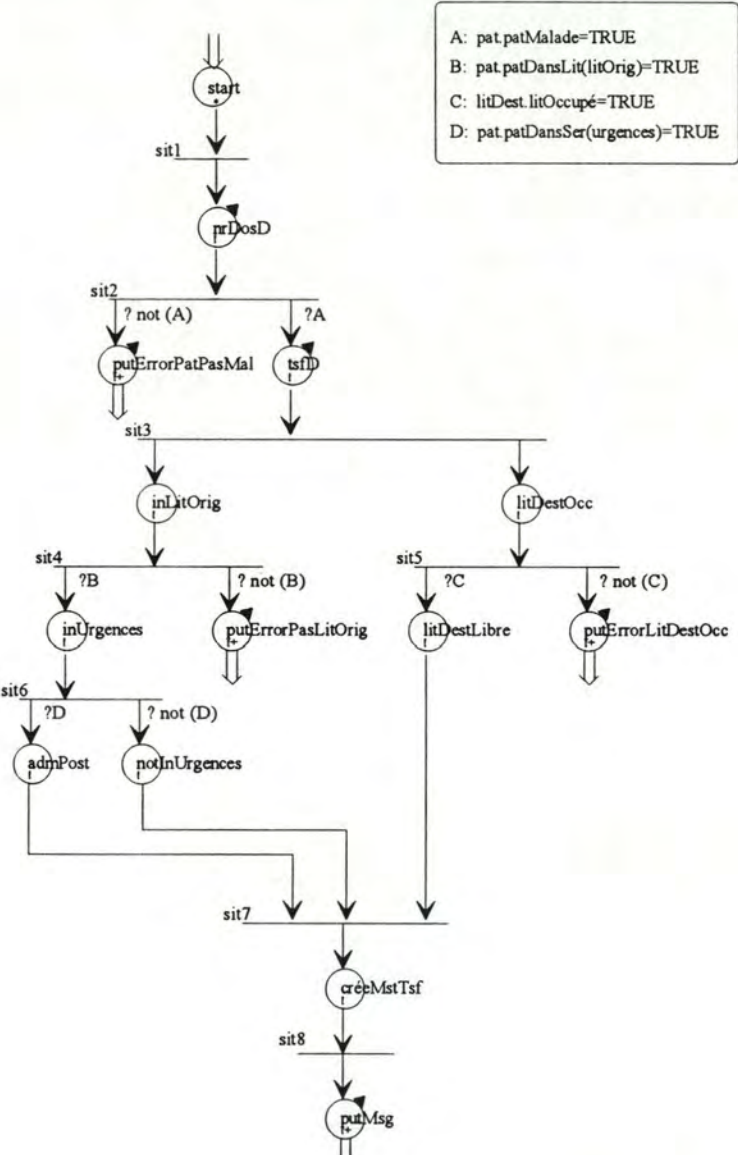
BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - litOrig, litDest : LIT

- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")

- **Actions**
  - * start
  - ! nrDosD (pat : PAT)
    >> NR_DOS_D.open (pat)
    ⇒ pat := nrDosD.pat
  - ! tsfD (litOrig, litDest : LIT, med : MED)
    >> TSF_D.open
    ⇒ litOrig := tsfD.litOrig
    ⇒ litDest := tsfD.litDest
    ⇒ med := tsfD.med
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPasLitOrig
    IS putError (code)
    := code = TSF_NOT_IN_LIT_OR
  - !+ putErrorLitDestOcc
    IS putError (code)
    := code = TSF_LIT_DEST_OCC
  - !+ putErrorPatPasMal
    IS putError (code)
    := code = TSF_PAT_NOT_MAL
  - ! inLitOrig
  - ! LitDestOcc
  - ! inUrgence
  - ! notInUrgence
  - ! litDestLibre
  - ! admPost
    >> ADM_POST.start (pat)
  - ! creeMvtTsf
    >> mvt_tsf.birth (litOrig, litDest, pat, med, TODAY(), NOW())
    >> allocation.déplacerPrescr (litOrig.serDeLit, litDest.serDeLit, pat)
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = TSF_OK

BEHAVIOUR



6. OBJET INTRO_SORTIE

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

BODY

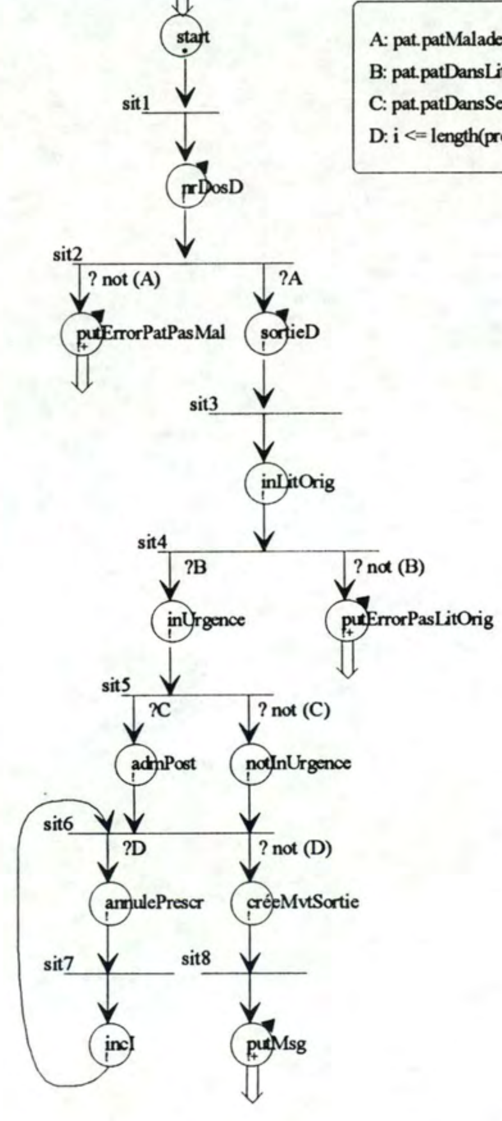
- **Attributes**
  - pat : PAT
  - med : MED
  - litOrig : LIT
  - i : NAT
  - prescr's : LIST of PRESCR_SOIN

- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")

- **Actions**
  - * start
  - ! nrDosD (pat : PAT)
    >> NR_DOS_D.open (pat)
    ⇒ pat := nrDosD.pat
  - ! sortieD (litOrig : LIT, med : MED)
    >> SORTIE_D.open
    ⇒ litOrig := sortieD.litOrig
    ⇒ med := sortieD.med
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPasLitOrig
    IS putError (code)
    := code = OUT_NOT_IN_LIT_OR
  - !+ putErrorPatPasMal
    IS putError (code)
    := code = OUT_PAT_NOT_MAL
  - ! inLitOrig

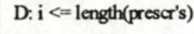
- ! inUrgence
  ⇒ i := 1
  ⇒ prescr's := ALL [pS : PRESCR_SOIN | pS in pat.subit AND pS.enAttente=TRUE]
- ! notInUrgence
- ! admPost
  >> ADM_POST.start (pat)
- ! annulePrescr
  >> allocation.prescrAnnulée(prescr's[i])
- ! incI
  ⇒ i := i + 1
- ! creeMvtSortie
  >> mvt_sor.birth (litOrig, pat, med, TODAY(), NOW())
- !+ putMsg (code : MSG_CODE_DT)
  >> MSG_D.open (code)
  := code = OUT_OK

BEHAVIOUR



Préconditions

A: pat.patMalade=TRUE
B: pat.patDansLit(litOrig)=TRUE
C: pat.patDansSer(urgence)=TRUE
D: i <= length(prescr's)



7. OBJET PRESCR_SOIN

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

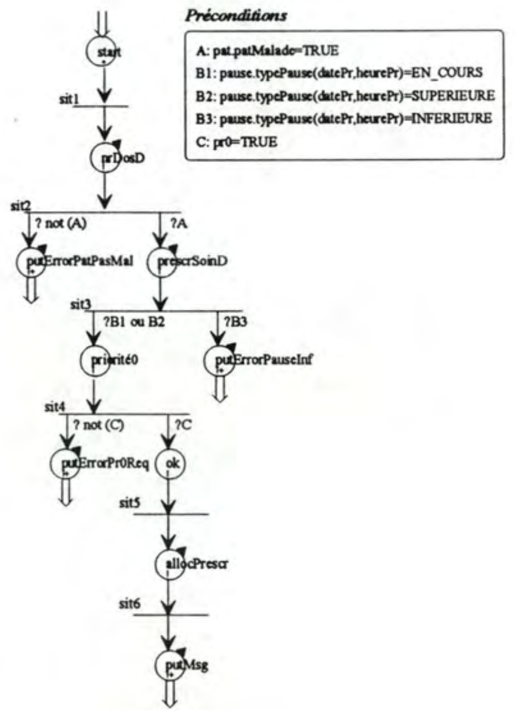
BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - prest : PREST (= PREST_INT ∪ PREST_EXT)
  - datePr : DATE
  - heurePr : TIME
  - pr0 : BOOL

- **Derivations**

- **Actions**
  - * start
  - ! nrDosD (pat : PAT)
    >> NR_DOS_D.open (pat)
    ⇒ pat := nrDosD.pat
  - ! prescrSoinD (med, prest, date, time)
    >> PRESCR_SOIN_D.open
    ⇒ med := prescrSoinD.med
    ⇒ prest := prescrSoinD.prest
    ⇒ datePr := prescrSoinD.date
    ⇒ heurePr := prescrSoinD.time
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPauseInf
    IS putError (code)
    := code = PR_S_PAUSE_INF
  - !+ putErrorPr0Req
    IS putError (code)
    := code = PR_S_PR0_REQ
  - !+ putErrorPatPasMal
    IS putError (code)
    := code = PR_S_PAT_NOT_MAL
  - ! allocPrescr
    >> allocation.allocPrescr (pat, med, prest, datePr, heurePr, pr0)
  - ! priorité0
  - !+ ok
  - ! notInUrgence
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = PR_S_OK

BEHAVIOUR



8. OBJET PREST_PRESCR

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

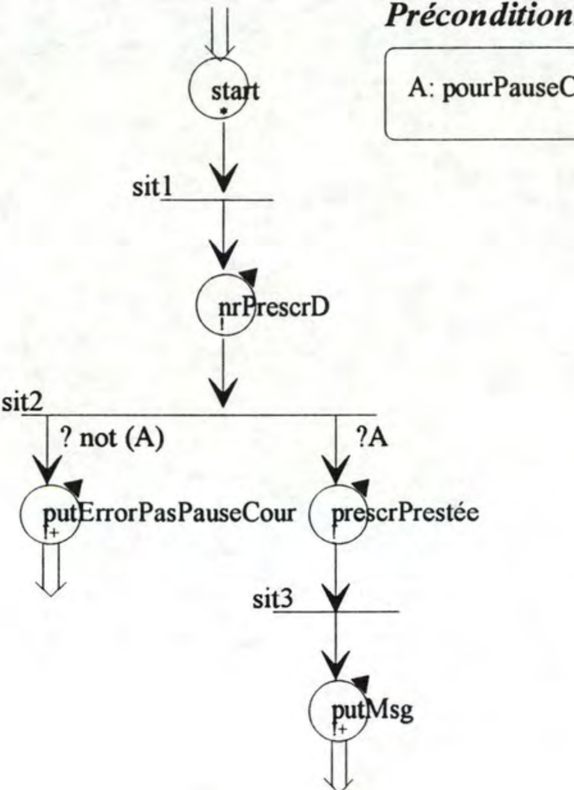
BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - prescr : PRESCR_SOIN
  - pourPauseCour : BOOL

- **Derivations**

- **Actions**
  - * start
  - ! nrPrescrD (pres : PRESCR_SOIN)
    >> NR_PRESCR_D.open (pres)
    ⇒ prescr := nrPrescrD.pres
  - ! pauseCour
    ⇒ pouPauseCour :=
    pause.typePause(prescr.date,prescr.heure)=EN_COURS
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPasPauseCour
    IS putError (code)
    := code = PREST_PR_NOT_PAUSE_COUR
  - ! prescrPrestée
    >> allocation.prescrPrestée (prescr)
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = PREST_PR_OK

BEHAVIOUR



Préconditions

A: pourPauseCour=TRUE

9. OBJET ANNUL_PRESCR

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

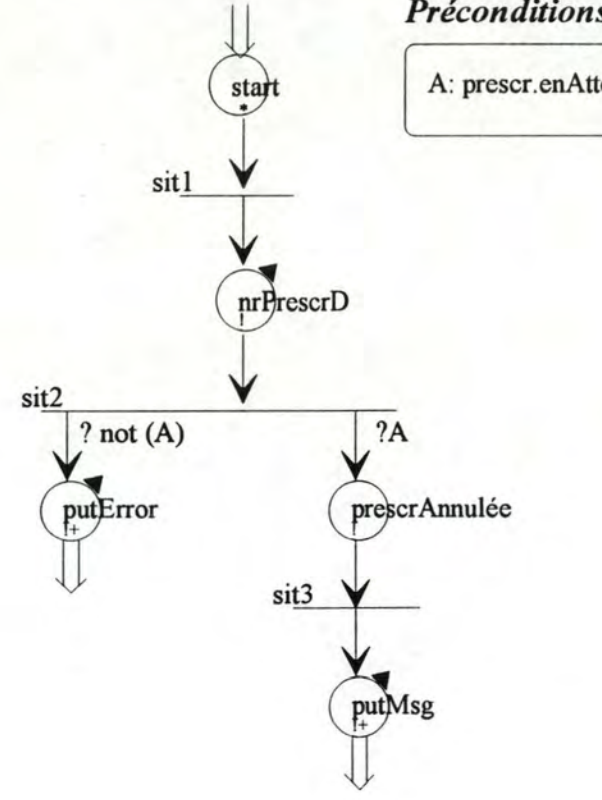
BODY

- **Attributes**
  - prescr : PRESCR_SOIN

- **Derivations**

- **Actions**
  - * start
  - ! nrPrescrD (pres : PRESCR_SOIN)
    >> NR_PRESCR_D.open (pres)
    ⇒ prescr := nrPrescrD.pres
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
    := code = ANNUL_PRESCR_NOT_WAITING
  - ! prescrAnnulée
    >> allocation.prescrAnnulée (prescr)
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = ANNUL_PRESCR_OK

BEHAVIOUR



Préconditions

A: prescr.enAttente=TRUE

INTERFACE

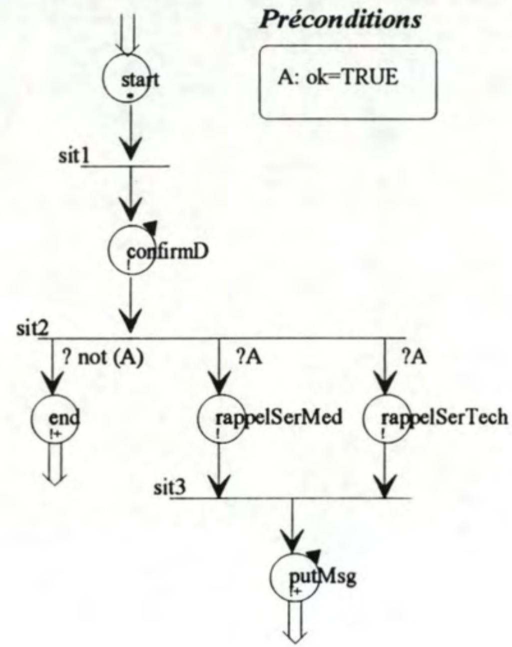
- **Observations**
- **Keys**
- **Events**
  - * start

BODY

- **Attributes**
  - ok : BOOL

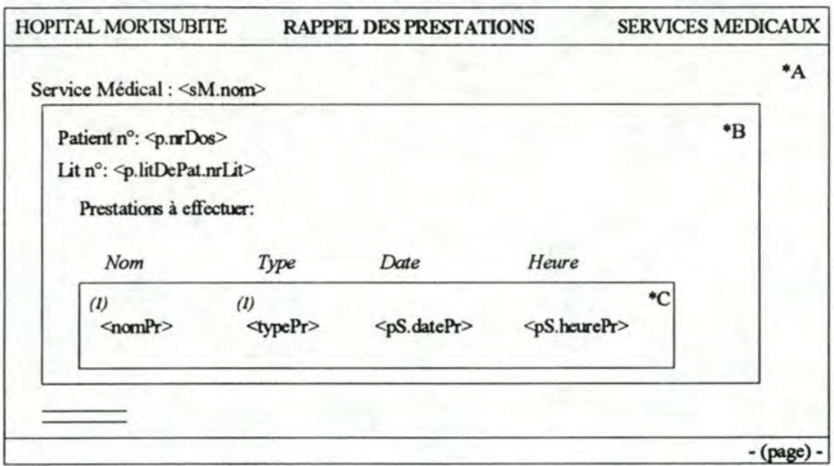
- **Derivations**

- **Actions**
  - * start
  - ! confirmD (ok :BOOL)
    >> CONFIRM_D.open (ok)
    ⇒ ok := confirmD.ok
  - ! rappelSerMed
    >> RAP_SER_MED.start
  - ! rappelSerTech
    >> RAP_SER_TECH.start
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = RAPPEL_PREST_OK
  - !+ end



objets de type rapport réalisée à l'aide de l'éditeur de rapport.

RAPPORT RAP_SER_MED

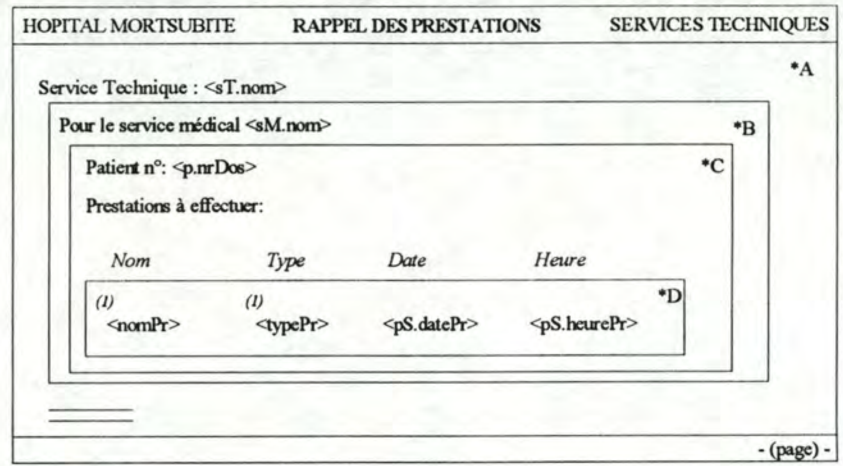


(1): pS.pourPrest.+...

A : ALL [sM : SER_MED]
B : ALL [p : PAT | p.patDansSer(sM)=TRUE]
C : ALL [pS : PRESCR_SOIN |
    pS.subitePar=p AND
    CLASS_OF(pS.pourPrest)="PREST_INT" AND
    pS.enAttente=TRUE AND
    pause.typePause(pS.datePr,pS.heurePr)=EN_COURS]

La logique de ce rapport est la suivante :

```
∀ service médical
  ∀ patient de ce service médical
        ∀ prescription de soin de ce patient
           "en attente",
           prévue pour la pause courante et
           prescrivant une prestation interne
⇒ imprimer les nom, type, date et heure de cette
prestation.
```



(1): pS.pourPrest.+...

A : ALL [sT : SER_TECH]
B : ALL [sM : SER_MED]
C : ALL [p : PAT | p.patDansSer(sM)=TRUE]
D : ALL [pS : PRESCR_SOIN |
    pS.subitePar=p AND
    CLASS_OF(pS.pourPrest)="PREST_EXT" AND
    pS.enAttente=TRUE AND
    pause.typePause(pS.datePr,pS.heurePr)=EN_COURS]

La logique de ce rapport est la suivante :

```
∀ service technique
  ∀ service médical
        ∀ patient de ce service médical
           ∀ prescription de soin de ce patient
              "en attente",
              prévue pour la pause courante et
              prescrivant une prestation externe
              effectuée par ce service technique
⇒ imprimer les nom, type, date et heure de cette
prestation.
```

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

BODY

- **Attributes**
  - coordPat : COORD_DT
  - nrDos : NR_DT
  - pat : PAT
  - listeMvtEnt, listeMvt : LIST of MVT
  - listePrescr : LIST of PRESCR_SOIN
  - m : MVT
  - dateD, dateF : DATE
  - heureD, heureF : TIME

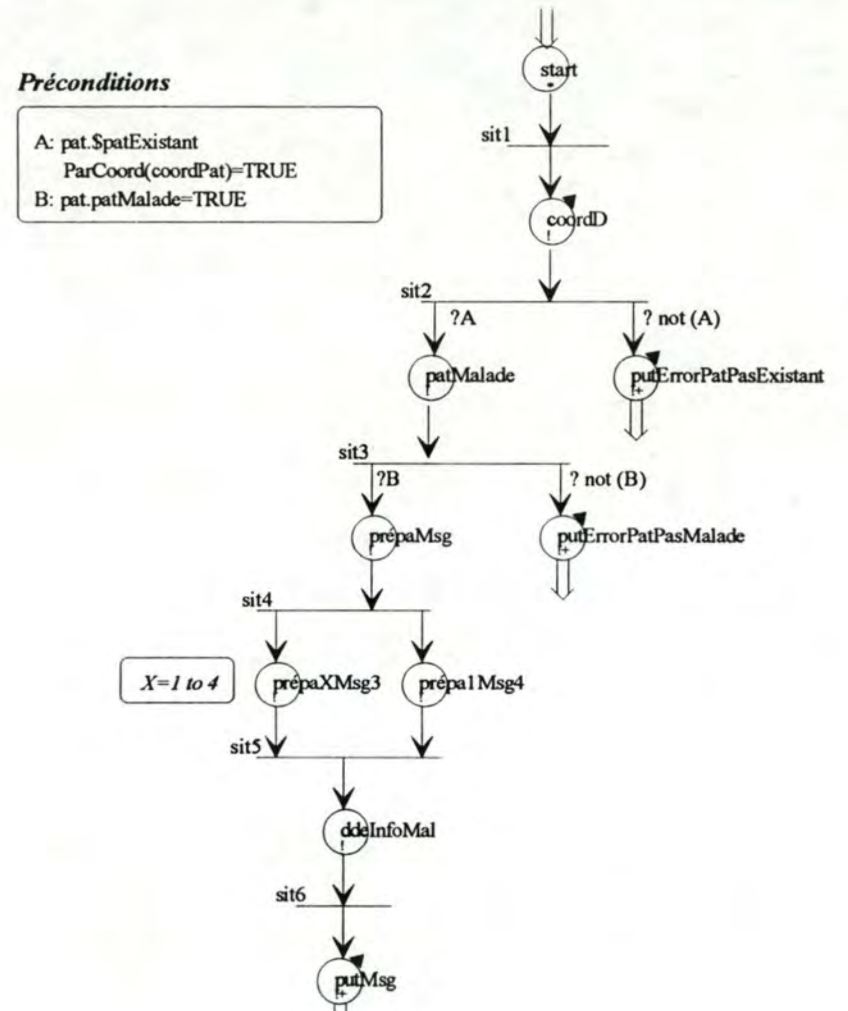
- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")

- **Actions**
  - * start
  - ! coordD (coordP : COORD_DT)
    >> COORD_D.open (coordP)
    ⇒ coordPat := coordD.coordP
  - ! prepaMsg
  - ! prepa1Msg3
    := listeMvtEnt = ALL [m : MVT |
       m in pat.accomplit AND
       CLASS_OF(m) = "MVT_ENT"]
  - ! prepa2Msg3
    := m = LAST (listeMvtEnt SORTED ON (dateMvt, heureMvt))
    := dateD = m.dateMvt
    := dateF = TODAY()
    := heureD = m.heureMvt
    := heureF = NOW()
  - ! prepa4Msg3
    := listeMvt = pat.listeMvtDates[1] (dateD, heureD, dateF, heureF)
  - ! prepa1Msg4
    := listePrescr = pat.listePrescrDates[2] (dateD, heureD, dateF, heureF)
  - ! patMalade
    ⇒ pat := pat.$patParCoord(coordPat)
  - ! ddeInfoMal
    >> RAP_INFO_MAL.start (pat, listeMvt, listePrescr)
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPatPasMal
    IS putError (code)
    := code = DDE_INFO_PAT_NOT_MAL
  - !+ putErrorPatPasExistant
    IS putError (code)
    := code = DDE_INFO_PAT_NOT_EXIST
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = DDE_INFO_OK

---

1 Observation de l'objet PAT qui renvoie la liste des mouvements que ce patient a effectués entre les deux dates/heures.

2 Observation de l'objet PAT qui renvoie la liste des prescriptions de soin effectuées que ce patient a subies entre les deux dates/heures.



A : pat.patDansSer(Urgence)=FALSE

B : ALL [m : MVT | m in listeMvt]

C : CLASS_OF(m) <> MVT_ENT

D : CLASS_OF(m) <> MVT_SOR

E : ALL [pS : PRESCR_SOIN | pS in listePrescr]

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

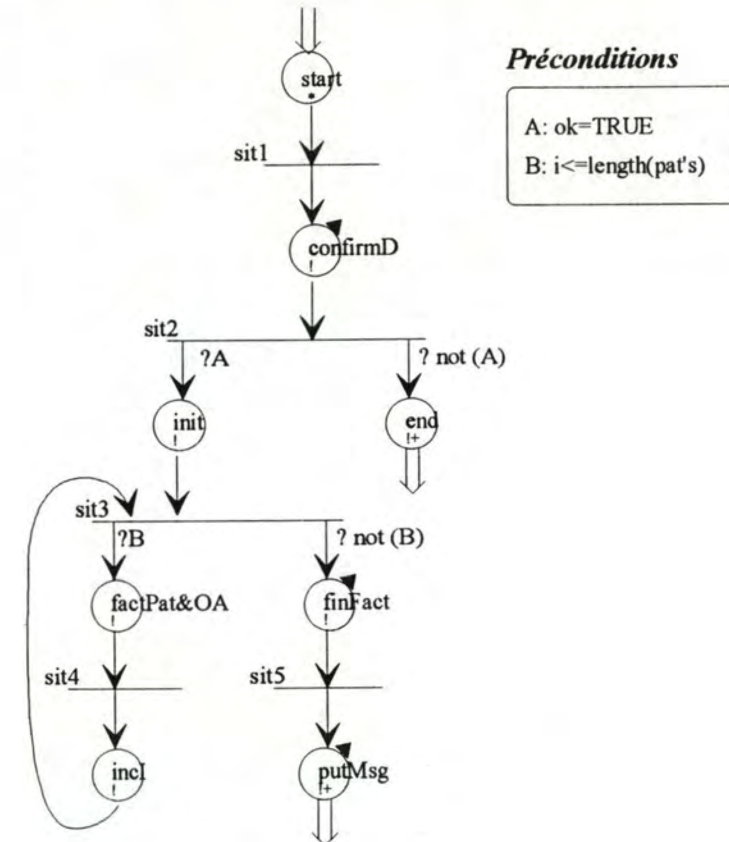
BODY

- **Attributes**
  - pat : PAT
  - pat's : LIST of PAT
  - dateD, dateF : DATE
  - heureD, heureF : TIME
  - i : INTEGER
  - ok : BOOL

- **Derivations**
  - hop : HOP
    hop := hop.$hôpital("MortSubite")

- **Actions**
  - * start
  - ! confirmD (ok :BOOL)
    >> CONFIRM_D.open (ok)
    ⇒ ok := confirmD.ok
  - ! init
    := dateD = HOP.dateDerFact
    := dateF = TODAY()
    := heureD, heureF = NOW()
    := pat's = ALL[p:PAT]
    := i = 1
  - ! factPat&OA
    := pat = pat's[i]
    >> FACT_PAT_OA.start(dateD,heureD,dateF,heureF,pat)
  - ! incI
    := i = i + 1
  - !+ end
    >> hop.fixeDateDerFact(TODAY())
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = PROD_FACT_OK

BEHAVIOUR



INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start (dateD,dateF : DATE, heureD,heureF : TIME, pat : PAT)

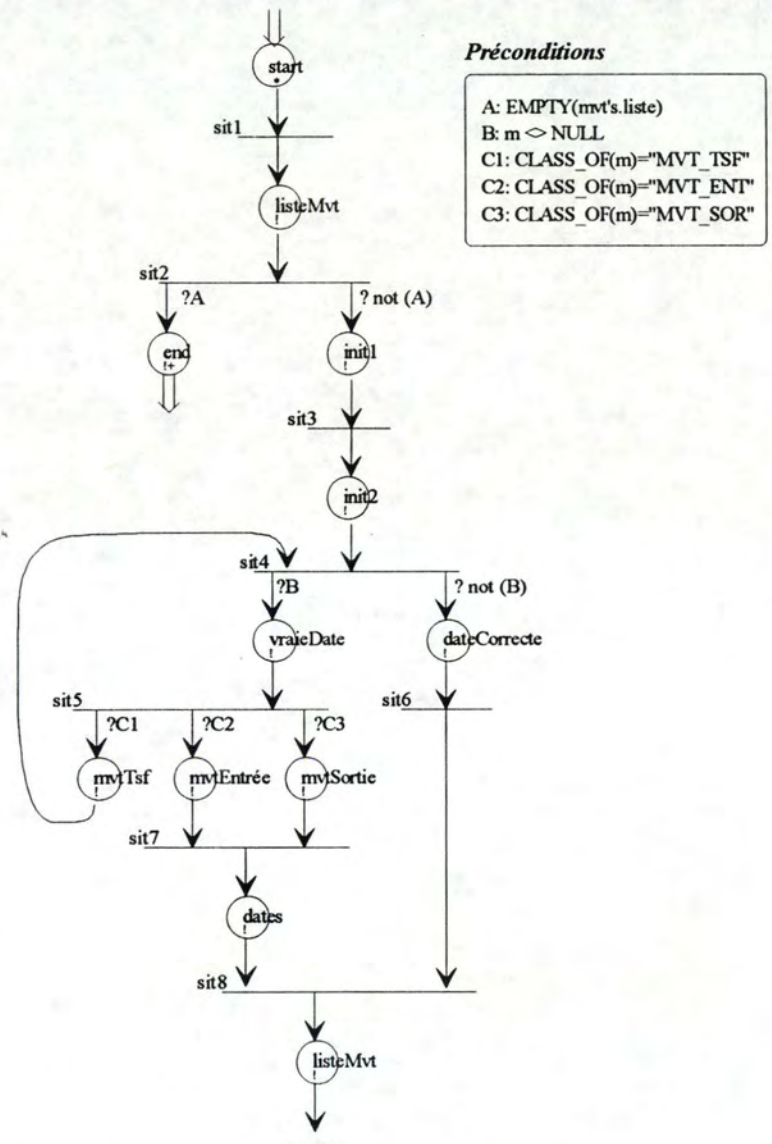
BODY

- **Attributes**
  - pat : PAT
  - mvt's : CP[liste : LIST of MVT, nbHosp : INT]
  - dateD, dateF : DATE
  - heureD, heureF : TIME
  - m : MVT
  - sej's : LIST of CP[mv1,mv2 : MVT]
  - prescr's : LIST of PRESCR_SOIN
  - dest : DESTINATAIRE_DT
  - hosp : LIST of MVT
  - i, j : INT

- **Derivations**

- **Actions**
  - * start (dateD,heureD,dateF,heureF,pat)
    ⇒ dateD := start.dateD
    ⇒ ...
    ⇒ pat := start.pat
  - ! listeMvt
    := mvt's = pat.listeMvtDates(dateD,heureD,dateF,heureF)
  - ! init1
    := m = FIRST(mvt's.liste)
  - ! init2
    := m = m.mvtPréc
  - ! vraieDate
  - ! dateCorrecte
  - ! mvtEntrée
    := m = FIRST(mvt's.liste)
  - ! mvtTsf
    := m = m.mvtPréc
  - ! dates
    := dateD = m.dateMvt
    := heureD = m.heureMvt
  - ! incI
    := i = i + 1
  - !+ end
  - ! factHosp's
    := j = 1
  - ! factHosp
    := m = mvt's.liste[j]
  - ! incJ
    := j = j + 1
  - ! mvtSuivant
    := m = mvt's.liste[j]
  - ! extraitHosp
    := hosp = EXTRACT(mvt's.liste[1..j]) {copie puis efface}
  - ! séjours
    >> SEJOUR.sejour(hosp, sejs)
    ⇒ sej's := sejour.sejs
  - ! prescr's
    := pat.listePrescrDates(FIRST(sej's).mv1.dateMvt/heureMvt, LAST(sej's).mv2.dateMvt/heureMvt)
  - ! fact
    >> RAP_FACT.start(pat,sej's,prescr's,dest)
  - ! factPat
    IS RAP_FACT
    := dest = PATIENT
  - ! factOA
    IS RAP_FACT
    := dest = OA

[3] Cet objet peut être considéré comme un Inner Object.

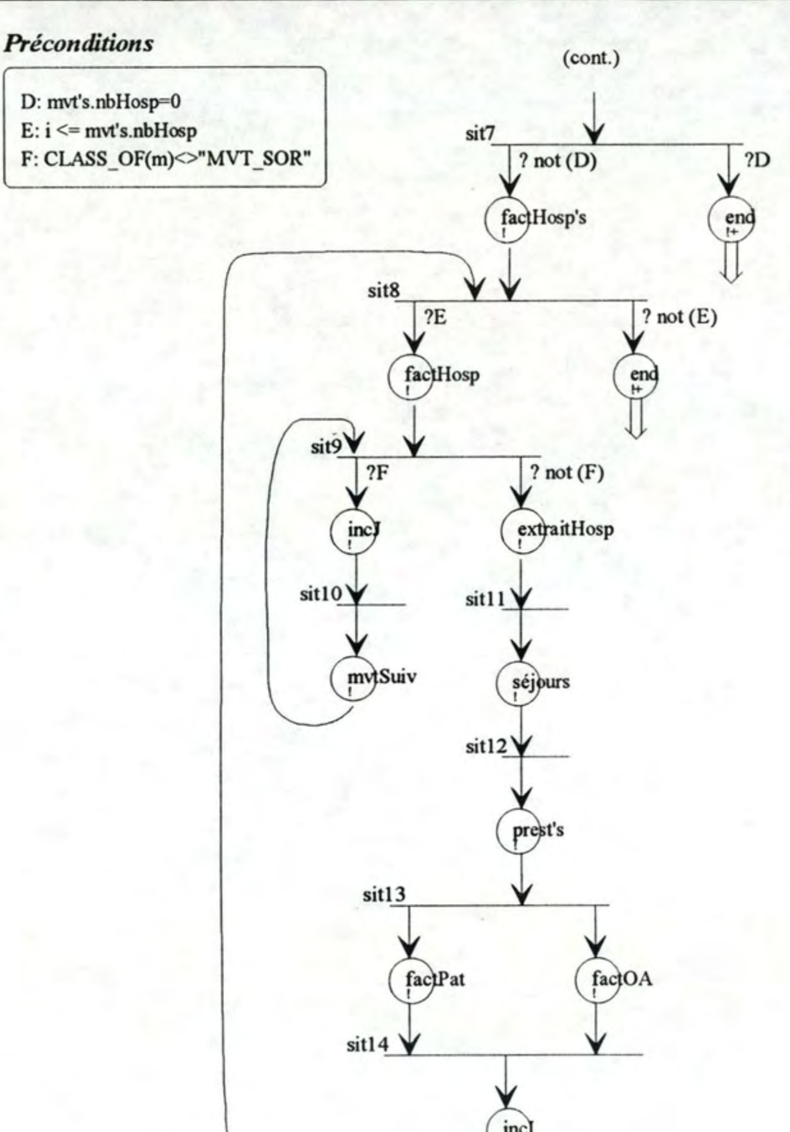


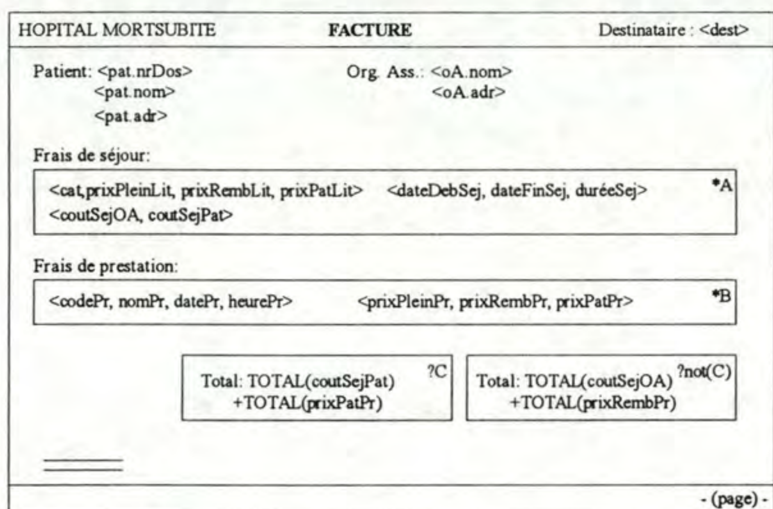
Préconditions

A: EMPTY(mvt's.liste)
B: m <> NULL
C1: CLASS_OF(m)="MVT_TSF"
C2: CLASS_OF(m)="MVT_ENT"
C3: CLASS_OF(m)="MVT_SOR"

Préconditions

D: mvt's.nbHosp=0
E: i <= mvt's.nbHosp
F: CLASS_OF(m)<>"MVT_SOR"





A : ALL [sej : CP[mv1,mv2:MVT] | sej in sej's]
B : ALL [pS : PRESCR_SOIN | pS in prescr's]
C : dest = PATIENT

En posant...

```
oA = pat.oADePat
l = sej.mv1.de (lit de référence pour la facturation)
cat = l.catégorieLit

prixPleinLit = cat.prixPlein
rembCh = remb_ch.$remboursementChambre(cat,oA)
prixRembLit = rembCh.prixRemb
prixPatLit = prixPleinLit - prixRembLit
dateDebSej = sej.mv1.dateMvt
dateFinSej = sej.mv2.dateMvt
duréeSej = dateFinSej - dateDebSej
coutSejOA = prixRembLit * duréeSej
coutSejPat = prixPatLit * duréeSej

prest = pS.pourPrest
codePr = prest.codePr
nomPr = prest.nomPr
datePr/heurePr = pS.datePr/heurePr
```

```
prixPleinPr = prest.prixPleinPr
prixRembPr = rembPr.prixRemb
prixPatPr = prixPleinPr - prixRembPr
```

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - allocNrDos (coord : COORD_DT)
  - allocLit (ser : SER_MED, cat : CAT)
  - allocPrescr (pat : PAT, med : MED, prest : PREST, date : DATE, heure : TIME, pr0 : BOOL)
  - deplacerPrescr (ser1, ser2 : SER_MED, pat)
  - prescrPrestée (prescr : PRESCR_SOIN)
  - prescrAnnulée (prescr : PRESCR_SOIN)

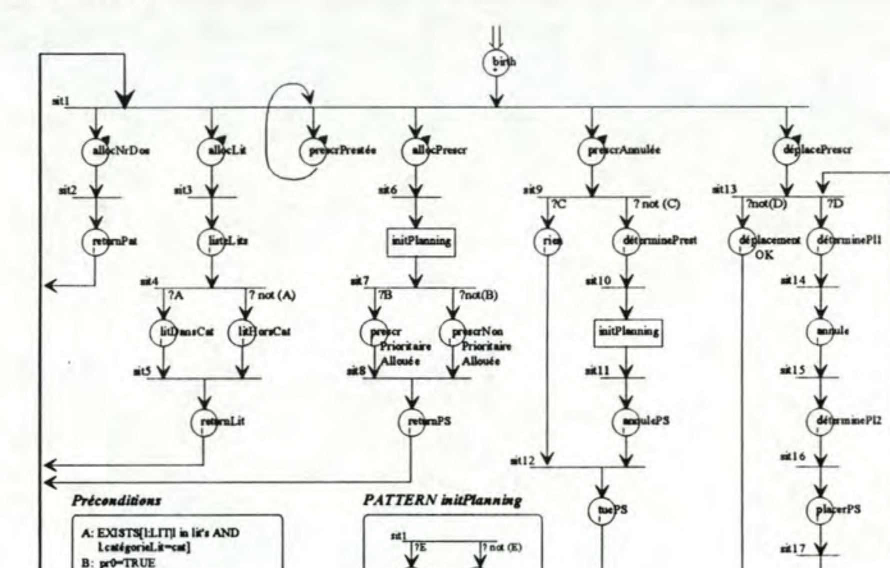
BODY

- **Attributes**
  - ser, ser1, ser2 : SER_MED
  - cat : CAT
  - pat : PAT
  - med : MED
  - prest : PREST
  - date, heure : DATE/TIME
  - pr0 : BOOL
  - lit : LIT
  - lit's : LIST of LIT
  - planning, pl1, pl2 : PLANNING
  - pS : PRESCR_SOIN
  - prescr's : LIST of PRESCR_SOIN
  - i : NAT

- **Derivations**
  - priorité (ser : SER) : NAT
    := {attribue la priorité déterminée}

- **Actions**
  - !* birth
  - allocNrDos (coord)
    ⇒ coord := allocNrDos.coord
    {>> rappelle où on l'a appelé avec la référence de pat grâce au mécanisme de stored action : cf. problème en début de chapitre}
  - allocLit (ser, cat)
    ⇒ ser := allocLit.ser
    ⇒ cat := allocLit.cat
  - ! listeLits
    ⇒ lit's := ALL [l : LIT | l.serDeLit = ser AND l.litOccupé = FALSE]
  - ! litDansCat
    ⇒ lit := ONE [l : LIT | l in lit's AND l.catégorieLit=cat]
  - ! litHorsCat
    ⇒ lit := ONE [l : LIT | l in lit's]
  - returnLit
    {>> rappelle où on l'a appelé avec la référence de lit}
  - allocPrescr (pat, med, prest, date, heure, pr0)
    ⇒ pat := allocPrescr.pat
    ⇒ med := allocPrescr.med
    ⇒ date := allocPrescr.date
    ⇒ heure := allocPrescr.heure
    ⇒ pr0 := allocPrescr.pr0
    >> pS.birth (pat, med, prest, date, heure) {juste pour avoir la référence}
  - ! pourSerMed
    ⇒ ser := ONE [sM : SER_MED | pat.patDansSer(sM)=TRUE]
  - ! pourSerTech
    ⇒ ser := prest.serDePrest
  - ! initPlanning
    ⇒ planning.$planning (ser, prest)
  - ! prescrPrioritaireAllouée
    >> planning.prescrPriorAllouée (pS)
  - ! prescrNonPrioritaireAllouée
    >> planning.placerPS (pS, priorité(ser), pause.nrPause (date, heure))
  - returnPS
    {>> rappelle où on l'a appelé avec la référence de pS}
  - deplacerPrescr (ser1, ser2, pat)
    ⇒ ser1, ser2 := deplacerPrescr.ser1, ser2
    ⇒ pat := deplacerPrescr.pat
    ⇒ prescr's := ALL [pS : PRESCR_SOIN | pS in pat.subit AND CLASS_OF (pS.pourPrest) = "PREST_INT"]
    ⇒ i := 1
  - ! déterminerPl1
    ⇒ pl1 := planning.$planning (prescr's[i].pourPrest, ser1)

- ⇒ pl2 := planning.$planning (prescr's[i].pourPrest, ser2)
- ! annule
  >> planning.annulePrescr (prescr's[i], pause.nrPause(prescr's[i].date, prescr's[i].heure))
- ! placerPS
  >> planning.placerPS (prescr[i], priorité(ser2), pause.nrPause(prescr's[i].date, prescr's[i].heure))
- ! incI
  ⇒ i := i + 1
- prescrPrestée (prescr : PRESCR_SOIN)
  >> prescr.fixeEtat (PRESTEE)
- prescrAnnulée (prescr : PRESCR_SOIN)
  ⇒ pS := prescrAnnulée.prescr
- ! rien
- ! déplacementOK
- ! déterminePrest
  ⇒ prest := pS.pourPrest
  ⇒ pat := pS.prescritePour
- ! annulePS
  >> planning.annulePrescr (pS, pause.nrPause(pS.date, pS.heure))
- ! tuePS
  >> pS.death



Rem : kind-of TBL
Rem : PL_PAUSE_DT =
CP [nbMax : NAT,
nbEffectif : NAT,
nbSurcharge : NAT,
listePS : LIST of CP [pS : PRESCR_SOIN,
prior : NAT]
]

INTERFACE

- **Observations**
- **Keys**
  - $planning (ser : SER, prest : PREST)
- **Events**
  - prescrPriorAllouée (pS : PRESCR_SOIN)
  - placerPS (pS : PRESCR_SOIN, priorité : NAT, nrPause : NAT) ? > 0 ? > 1
  - chgmtPause
  - * birth (ser : SER, prest : PREST, nbPauses : NAT)
  - annulePrescr (pS : PRESCR_SOIN, nrPause : NAT)

BODY

- **Attributes**
  - ser : SER_MED
  - prest : PREST
  - listePauses : PL_PAUSE_DT
  - nbPauses : NAT
  - nrPause : NAT
  - priorité, prTemp : NAT
  - pS, pSTemp : PRESCR_SOIN
  - i : NAT
  - eff : EFF_PREST_INT ∪ EFF_PREST_EXT

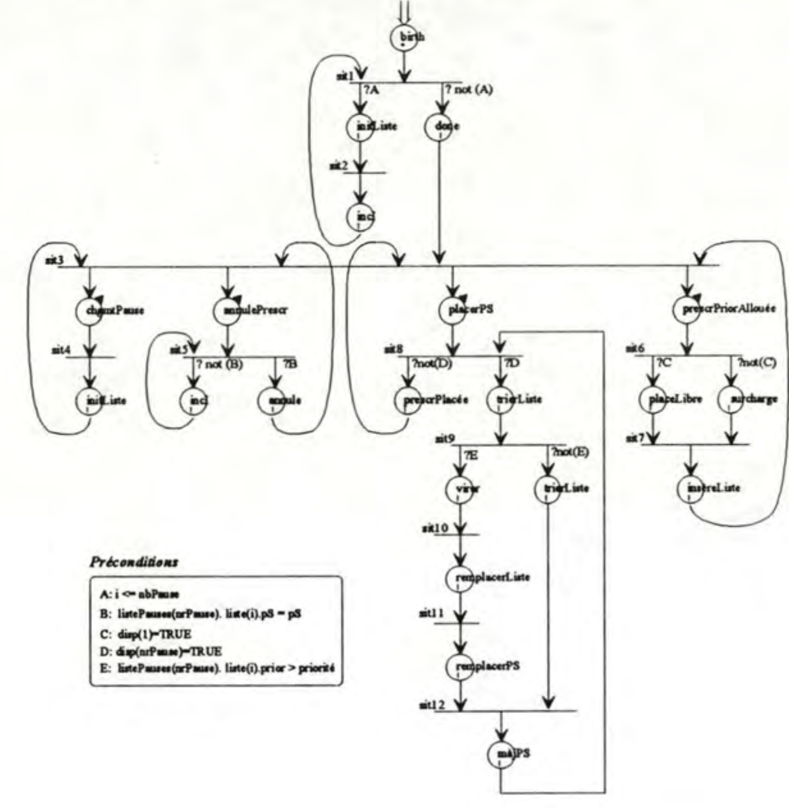
- **Derivations**

:= listePauses[i].nbSurcharge = 0
  - $planning (ser, prest)
    := ONE[pl : PLANNING | pl.ser = ser AND pl.prest = prest]

- **Actions**
  - * birth (ser, prest, nbPauses)
    ⇒ nbPauses := birth.nbPauses
    ⇒ ser := birth.ser
    ⇒ prest := birth.prest
    {? CLASS_OF (prest) = "PREST_INT"} ⇒ eff := eff_prest_int.$effectuePrestationInterne(ser, prest)
    {? CLASS_OF (prest) = "PREST_EXT"} ⇒ eff := prest.prExtEff
    ⇒ i := 1
  - ! initListe
    ⇒ APPEND (listePauses, prest.capMaxi*eff.nbUnitésExec ∪ 0 ∪ 0 ∪ EMPTY())
  - ! incl
    ⇒ i := i + 1
  - ! done
  - prescrPriorAllouée (pS)
    ⇒ pS := prescrPriorAllouée.pS
  - ! placeLibre
    ⇒ listePauses[1].nbEffectif := listePauses[1].nbEffectif + 1
  - ! surcharge
    ⇒ listePauses[1].nbSurcharge := listePauses[1].nbSurcharge + 1
  - ! insèreListe
    ⇒ APPEND (listePauses[1].liste, pS ∪ 0)
  - placerPS (pS, priorité, nrPause)
    ⇒ pS := placerPS.pS
    ⇒ priorité := placerPS.priorité
    ⇒ nrPause := placerPS.nrPause
  - ! prescrPlacée
    ⇒ APPEND (listePauses[nrPause].liste, pS ∪ priorité)
    ⇒ listePauses[nrPause].nbEffectif := listePauses[nrPause].nbEffectif + 1
  - ! trierListe
    ⇒ listePauses[nrPause].liste := listePauses[nrPause].liste SORTED ON prior
    ⇒ i := length (listePauses[nrPause].liste)

⇒ pSTemp := listePauses[nrPause].liste[i].pS
⇒ prTemp := listePauses[nrPause].liste[i].prior

- ! remplacerListe
  ⇒ listePauses[nrPause].liste[i].pS := pS
  ⇒ listePauses[nrPause].liste[i].prior := priorité
- ! remplacerPS
  ⇒ pS := pSTemp
  ⇒ priorité := prTemp - 1
- ! màjPS
  >> pS.fixeDateHeure (pause.pausePlusUn(pS.datePr, pS.heurePr))
  ⇒ nrPause := nrPause + 1
- ! pauseSuiv
- annulePrescr (pS, nrPause)
  ⇒ pS := annulePrescr.pS
  ⇒ nrPause := annulePrescr.nrPause
  ⇒ i := 1
- ! annule
  ⇒ ERASE (listePauses[nrPause].liste[i]
  {? listePauses[nrPause].nbSurcharge > 0} ⇒
  listePauses[nrPause].nbSurcharge := listePauses[nrPause].nbSurcharge -
  1
  {? listePauses[nrPause].nbSurcharge = 0} ⇒
  listePauses[nrPause].nbEffectif := listePauses[nrPause].nbEffectif - 1
- ! chgmtPause
  ⇒ ERASE (listePauses[1])



INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * séjour (hosp : LIST of MVT)

BODY

- **Attributes**
  - sej's : LIST of CP[mv1, mv2 : MVT]
  - hosp : LIST of MVT
  - i, j : NAT
  - trouvé : BOOL
  - date, heure : DATE/TIME

- **Derivations**

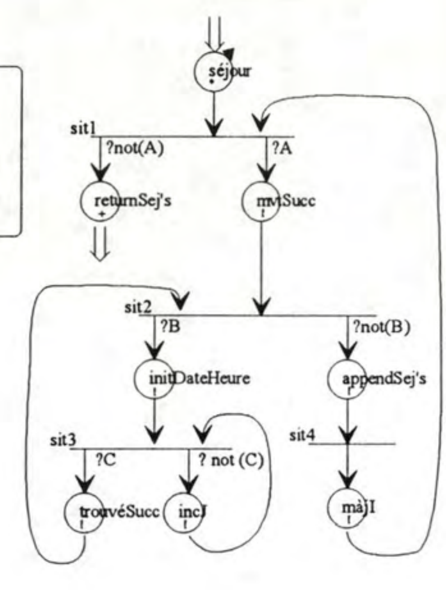
- **Actions**
  - * séjour (hosp)
    ⇒ hosp := séjour.hosp
    ⇒ i := 1
    ⇒ j := 1
  - ! mvtSucc
    ⇒ trouvé := FALSE
    ⇒ j := j + 1
  - ! initDateHeure
    ⇒ date := hosp[i].heureMvt + 1 {+24h}
    ⇒ heure := hosp[i].dateMvt
  - trouvéSucc
    ⇒ trouvé := TRUE
  - ! incJ
    ⇒ j := j + 1
  - ! appendSej's
    ⇒ APPEND (sej's, hosp[i] ∪ hosp[j])
  - ! màjI
    ⇒ i := j

{>> rappelle où on l'a appelé avec sej's}

BEHAVIOUR

Préconditions

A: j < length(hosp)
B: trouvé=FALSE
C: (hosp(i).dateMvt >= date AND hosp(i).heureMvt >= heure) OR j = length(hosp)



Rem : objet *single*

INTERFACE

- **Observations**
  - typePause (date : DATE, heure : TIME) : TYPE_PAUSE_DT
  - nrPause (date : DATE, heure : TIME) : NAT
    ? typePause (date, heure) <> INFERIEURE
  - pausePlusUn (date : DATE, heure : TIME) : CP[date : DATE, heure : TIME]

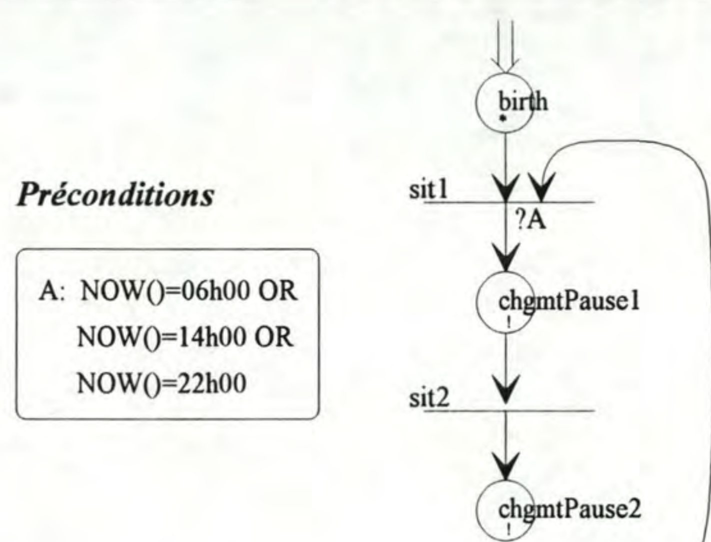
- **Keys**
- **Events**

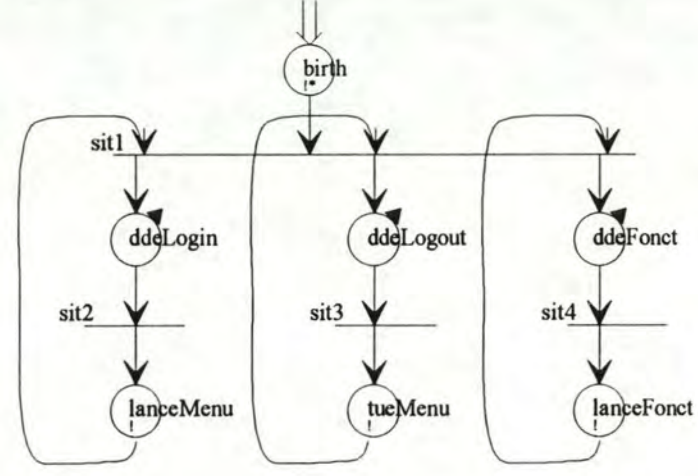
BODY

- **Attributes**
  - debPauseCour, finPauseCour : CP[dateP : DATE, heureP : TIME]
  - plannings : LIST of PLANNING

- **Derivations**
  - typePause (date, heure)
    {? (date < debPauseCour.dateP) and (heure < debPauseCour.heureP)}
    ⇒ INFERIEURE
    {? (date > debPauseCour.dateP) and (heure > debPauseCour.heureP)}
    ⇒ SUPERIEURE
    {? [(date ≥ debPauseCour.dateP) and (heure ≥ debPauseCour.heureP)]
    and [(date ≤ debPauseCour.dateP) and (heure ≤ debPauseCour.heureP)]}
    ⇒ EN_COURS
  - nrPause (date, heure)
    ⇒ [(date - finPauseCour.dateP)*24 + (heure - finPauseCour.heureP)] + 1
  - pausePlusUn (date, heure)
    {? heure > 16h00}
    ⇒ heure := heure - 16   {= heure + 8 - 24}
    ⇒ date := date + 1
    {? heure ≤ 16h00}
    ⇒ heure := heure + 8
    ⇒ date := date

- **Actions**
  - ! birth
    {? (06h00 ≤ NOW() < 14h00}
    ⇒ debPauseCour.dateP := TODAY()
    ⇒ debPauseCour.heureP := 06h00
    ⇒ finPauseCour.dateP := TODAY()
    ⇒ finPauseCour.heureP := 14h00
    {? (14h00 ≤ NOW() < 22h00}
    ⇒ debPauseCour.dateP := TODAY()
    ⇒ debPauseCour.heureP := 14h00
    ⇒ finPauseCour.dateP := TODAY()
    ⇒ finPauseCour.heureP := 22h00
    {? (22h00 ≤ NOW() < 00h00}
    ⇒ debPauseCour.dateP := TODAY()
    ⇒ debPauseCour.heureP := 22h00
    ⇒ finPauseCour.dateP := TODAY()+1
    ⇒ finPauseCour.heureP := 06h00
    {? (0h00 ≤ NOW() < 06h00}
    ⇒ debPauseCour.dateP := TODAY()-1
    ⇒ debPauseCour.heureP := 22h00
    ⇒ finPauseCour.dateP := TODAY()
    ⇒ finPauseCour.heureP := 06h00
  - ! chgmtPause1
    ⇒ debPauseCour.dateP := pausePlusUn (debPauseCour.dateP, .heureP)
    ⇒ plannings := ALL[pl : PLANNING | TRUE]
  - ! chgmtPause2
    ⇒ finPauseCour.dateP := pausePlusUn (debPauseCour.dateP, .heureP)
    >> plannings.chgmtPause   {*multicast*}





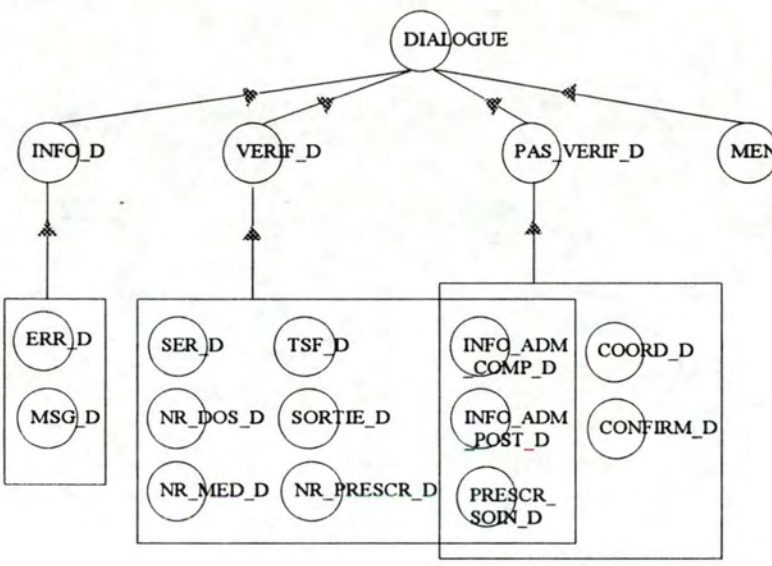
18. OBJET SESSION

Pour cet objet, nous nous contenterons d'une description sommaire.

Cet objet dispose de trois fonctions qui sont `ddeLogin`, `ddeLogout` et `ddeFonct`. La fonction `ddeLogin` est activée lorsque, depuis son terminal, l'utilisateur demande à entrer dans le système. Le menu complet est alors affiché (action `lanceMenu`). La fonction `ddeLogout` est activée lorsque, depuis son terminal, l'utilisateur demande à sortir du système. Le menu est alors effacé de son terminal (action `tueMenu`). La fonction `ddeFonct` est activée lorsque, à partir de son menu, l'utilisateur sélectionne une fonctionnalité. La fonctionnalité concernée est alors déclenchée (action `lanceFonct`).

1. ARCHITECTURE GLOBALE

NIVEAU 4
Interface



2. OBJET DIALOGUE

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - ***** birth
  - + death

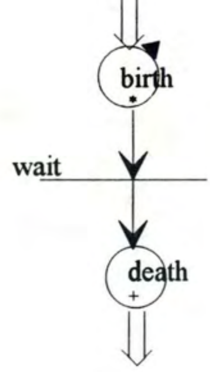
BODY

- **Attributes**

- **Derivations**
  - titre : STRING := "Dialogue"

- **Actions**
  - ***** birth
  - + death

BEHAVIOUR



3. OBJET INFO_D

Rem : IS-A DIALOGUE

INTERFACE

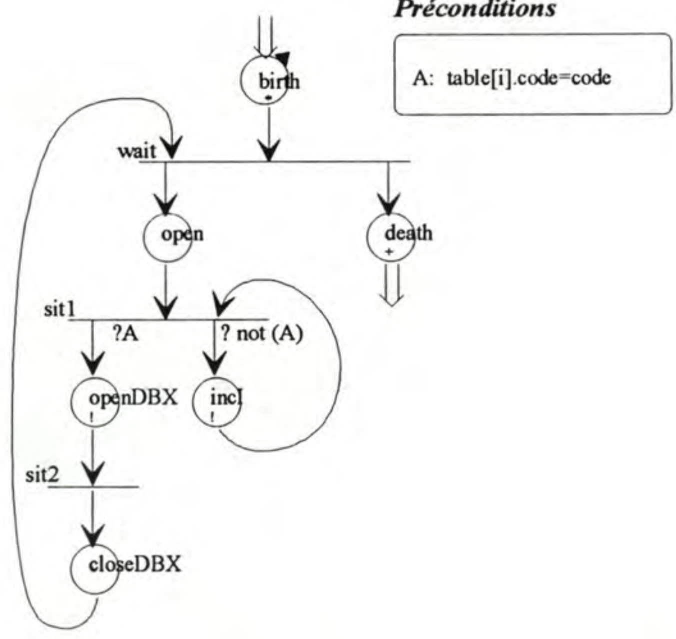
- **Observations**
- **Keys**
- **Events**
  - * birth
  - open (code : INFO_CODE_DT)
    {cette spécif est incorrecte car il n'existe pas de notion de sur-type d'un type de donnée. L'idéal serait de pouvoir déclarer INFO_CODE_DT comme le sur-type de ERR_CODE_DT et de MSG_CODE_DT. }
  - + death

BODY

- **Attributes**
  - table : ARRAY of CP[code : INFO_CODE_DT, chaîne : STRING]
  - code : INFO_CODE_DT
  - i : NAT
- **Derivations**
  - titre : STRING
    := "Information"
- **Actions**
  - * birth
    ⇒ table[1].code := ...
    ⇒ table[1].chaîne := "..."
    ⇒ etc.
  - open (code)
    ⇒ code := open.code
    ⇒ i := 1
  - ! incI
    ⇒ i := i + 1
  - ! openDBX
    CALLS to EXT with (table[i].chaîne, titre)
    {ouvre la boîte}
  - closeDBX
    CALLED from EXT
    {l'utilisateur a cliqué sur le seul bouton OK}
    CALLS to EXT
    {ferme la boîte}
  - + death

Je vais, pour l'instant, me limiter à spécifier le sur-type de chaque classe sans entrer dans les détails des différents objets spécialisés. Seules une image de la boîte et, si nécessaire, une liste des primitives spécifiques à un objet spécialisé seront exposées. Je ne crois pas que j'irais plus loin dans la suite.

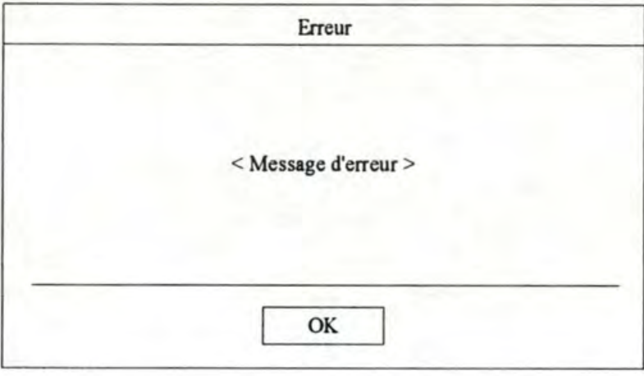
BEHAVIOUR



4. OBJET ERR_D

Rem : IS-A INFO_D

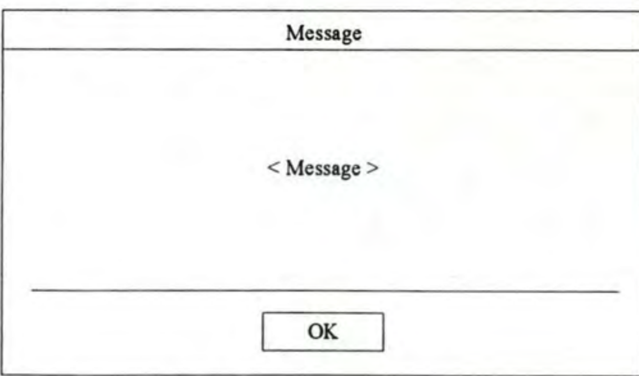
APERÇU DE LA BOÎTE



5. OBJET MSG_D

Rem : IS-A INFO_D

APERÇU DE LA BOÎTE



6. OBJET VERIF_D

Rem : IS-A DIALOGUE

But : saisir un (plusieurs) identifiant(s) relatif(s) à un (plusieurs) objet(s). Vérifier leur existence et en obtenir la référence interne. Dans les sous-types de cet objet (ainsi que des autres), il sera fait mention des observations utilisées.

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * birth
  - open *(okEv, cancelEv : EVENT)*
    {cette spécif est doublement incorrecte car (1) il n'existe pas de notion de sur-type d'un type événement. L'idéal serait de pouvoir déclarer un type EVENT représentant un événement quelconque : cela allégerait le mécanisme lourd du stored action. (2) Par ailleurs, ces paramètres ne devraient pas être spécifiés à cette phase : à discuter (cf. problème soulevé au début du document}
  - + death

BODY

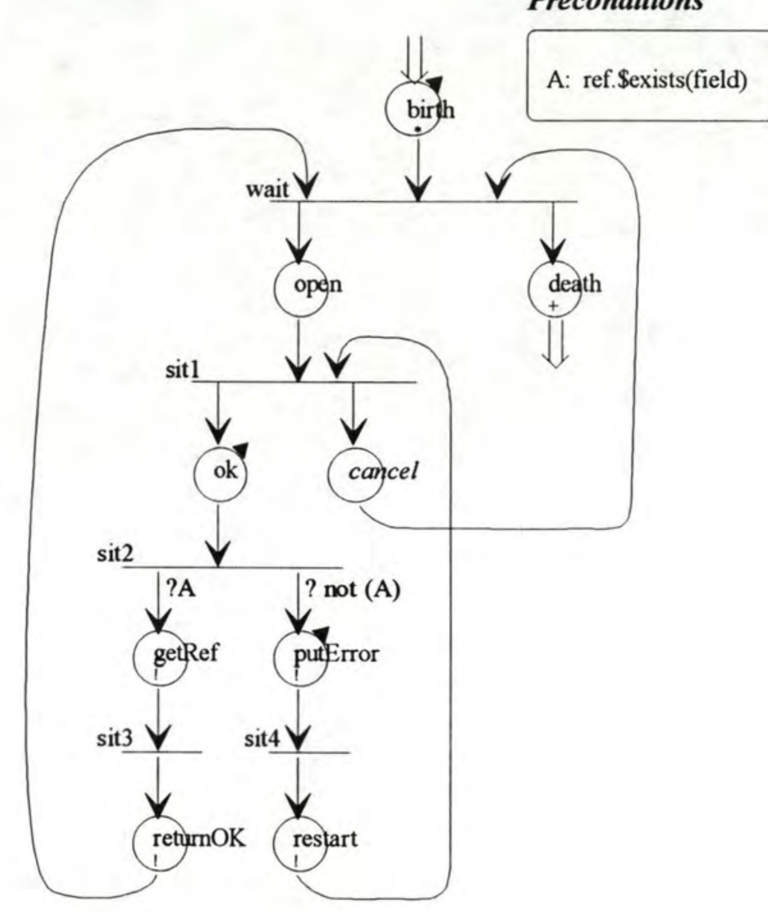
- **Attributes**
  - field : STRING
  - ref : OBJECT *{n'importe quel objet : à redéfinir dans la spécialisation}*
  - *okEv, cancelEv : EVENT*

- **Derivations**
  - titre : STRING
    := "Vérification"

- **Actions**
  - * birth
  - open *(okEv, cancelEv)*
    ⇒ *okEv, cancelEv := open.okEv, open.cancelEv*
    CALLS to EXT with (titre) *{ouvre la boîte}*

- ok (field : STRING)
  CALLED from EXT
  ⇒ field := ok.field
- *cancel*
  CALLED from EXT
  >> cancelEv
  CALLS to EXT {ferme la boîte}
- ! getRef
  ⇒ ref := ref.$key (field) *{obtient la référence de l'objet sur base de son identifiant : à redéfinir dans la spécialisation}*
- ! putError (code : ERR_CODE_DT)
  := code = ...
  >> ERR_D.open (code)
- ! restart
  CALLS to EXT *{redémarrer la boîte : la réactiver}*
- *! returnOk*
  CALLS to EXT {ferme la boîte}
  >> okEv
  := okEv.ref := ref {rappelle où on l'a appelé et passe ref dans le paramètre de l'événement okEv}
- + death

BEHAVIOUR



7. OBJET SER_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE

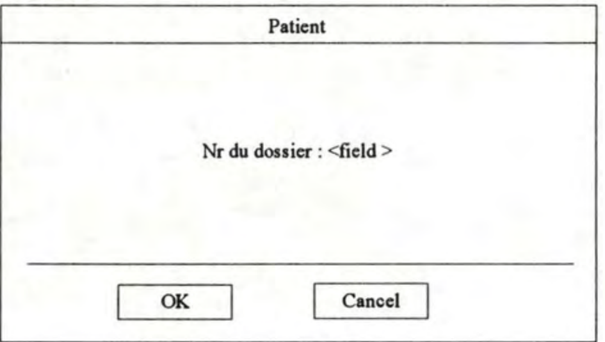
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : SER_MED
- **Primitives**
  - $serMedExistant(nom)
  - $serviceMédical(nom)
- **Code d'erreur** : SER_MED_NOT_EXISTS

8. OBJET NR_DOS_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE



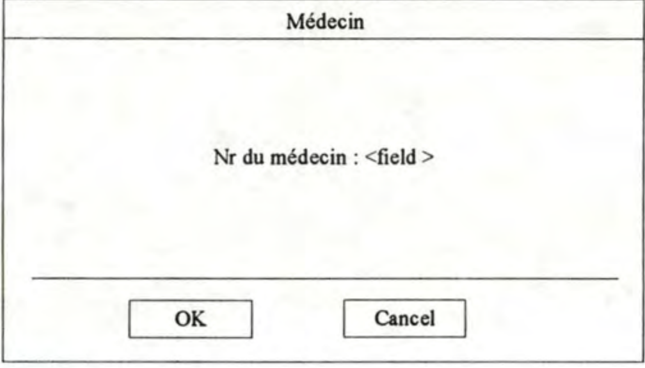
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : PAT
- **Primitives**
  - $patExistantParNrDos(nrDos)
  - $patParNrDos(nrDos)
- **Code d'erreur** : PAT_NOT_EXISTS

9. OBJET NR_MED_D

Rem : IS-A VERIF_D

Aperçu de la boîte



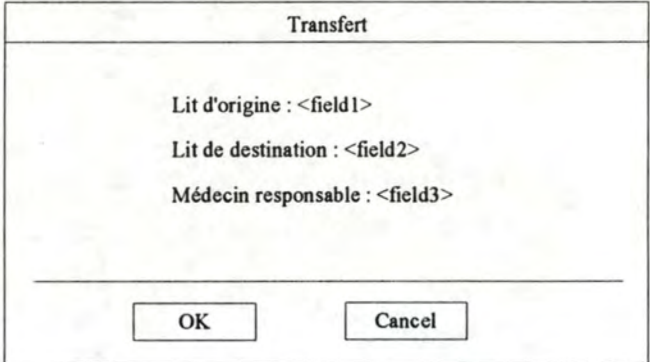
Objet et Primitives de l'objet

- **Objet** : MED
- **Primitives**
  - $medExistant(nrMed)
  - $médecin(nrMed)
- **Code d'erreur** : MED_NOT_EXISTS

10. OBJET TSF_D

Rem : IS-A VERIF_D

Aperçu de la boîte



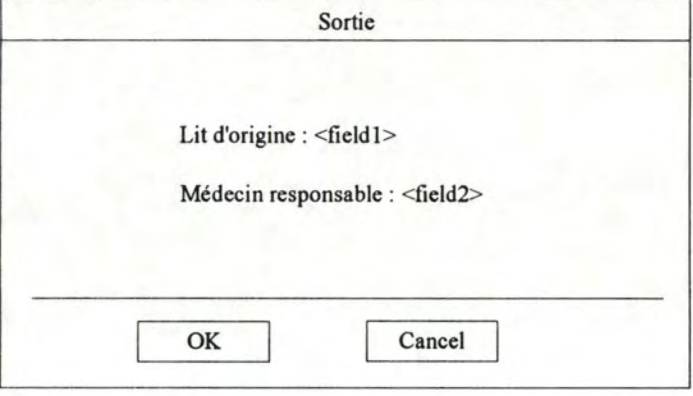
Objet et Primitives de l'objet

- **Objet** : MED, LIT
- **Primitives**
  - $litExistant(nrLit)
  - $lit(nrLit)
  - $medExistant(nrMed)
  - $médecin(nrMed)
- **Code d'erreur** : MED_NOT_EXISTS, LIT_NOT_EXISTS

11. OBJET SORTIE_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE



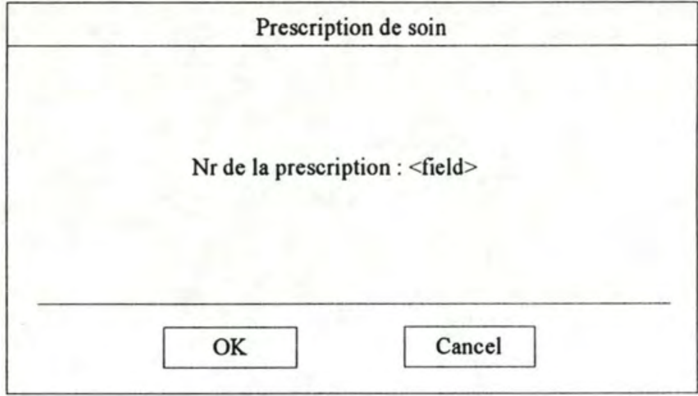
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : MED, LIT
- **Primitives**
  - $litExistant(nrLit)
  - $lit(nrLit)
  - $medExistant(nrMed)
  - $médecin(nrMed)
- **Code d'erreur** : MED_NOT_EXISTS, LIT_NOT_EXISTS

12. OBJET NR_PRESCR_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE



OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : PRESCR_SOIN
- **Primitives**
  - $prescrSoinExistante(nrPrescr)
  - $prescriptionSoin(nrPrescr)
- **Code d'erreur** : PRESCR_SOIN_NOT_EXISTS

13. OBJET PAS_VERIF_D

Rem : IS-A DIALOGUE

But : saisir un ou plusieurs renseignement(s) ne nécessitant pas de vérification d'existence (adresse, téléphone, etc.).

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * birth
  - open *(okEv, cancelEv : EVENT)*
    {cette spécif est doublement incorrecte : cf. VERIF_D}
  - \+ death

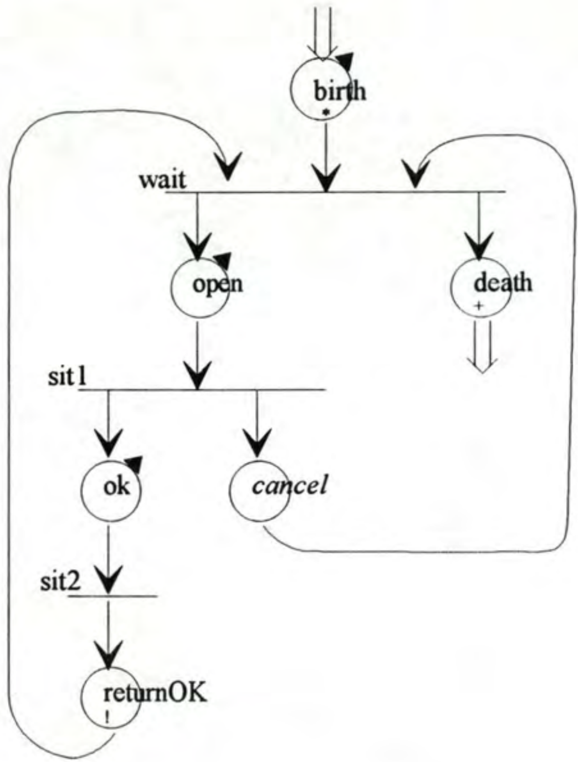
BODY

- **Attributes**
  - field : STRING
  - *okEv, cancelEv : EVENT*

- **Derivations**
  - titre : STRING
    := "Sans Vérification"

- **Actions**
  - * birth
  - open *(okEv, cancelEv)*
    ⇒ *okEv, cancelEv := open.okEv, open.cancelEv*
    CALLS to EXT with (titre) *{ouvre la boîte}*
  - ok (field : STRING)
    CALLED from EXT
    ⇒ field := ok.field
  - *cancel*
    CALLED from EXT
    >> cancelEv
    CALLS to EXT {ferme la boîte}
  - *! returnOk*
    CALLS to EXT {ferme la boîte}
    >> okEv
    := okEv.field := field {rappelle où on l'a appelé et passe field dans le paramètre de l'événement okEv}
  - \+ death

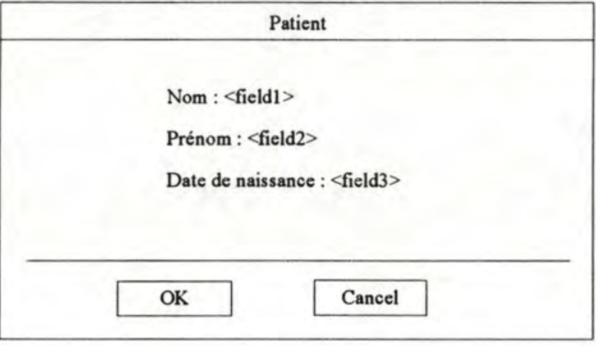
BEHAVIOUR



14. OBJET COORD_D

Rem : IS-A PAS_VERIF_D

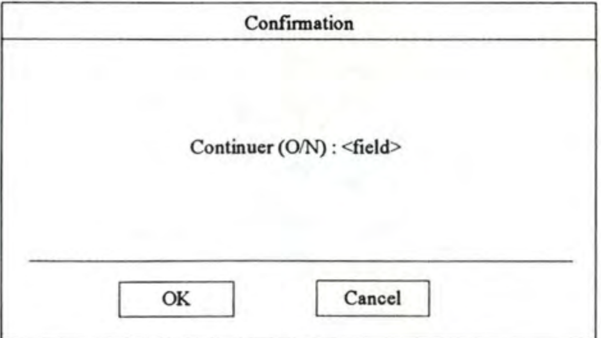
APERÇU DE LA BOÎTE



15. OBJET CONFIRM_D

Rem : IS-A PAS_VERIF_D

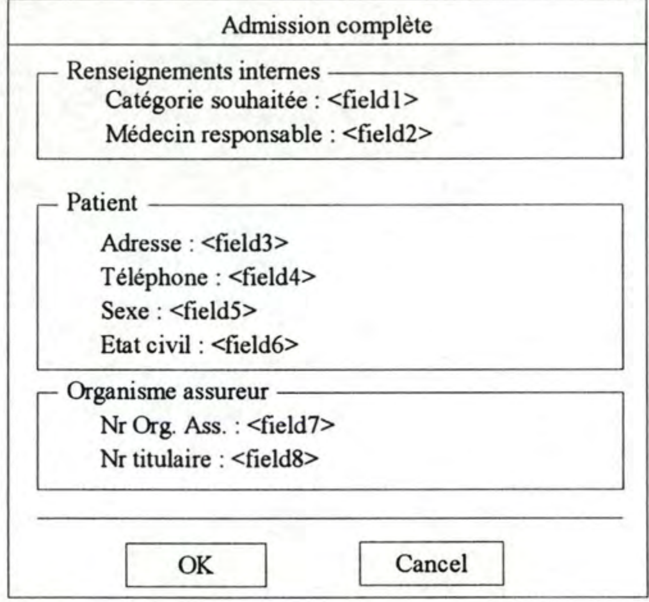
APERÇU DE LA BOÎTE



16. OBJET INFO_ADM_FULL_D

Rem : IS-A PAS_VERIF_D AND VERIF_D
{les boîtes qui suivent mélangent les services des deux types proposés jusqu'ici : certaines informations sont vérifiées et d'autres pas. Le problème est que OBLOGlight ne permet pas d'héritage multiple !}

APERÇU DE LA BOÎTE



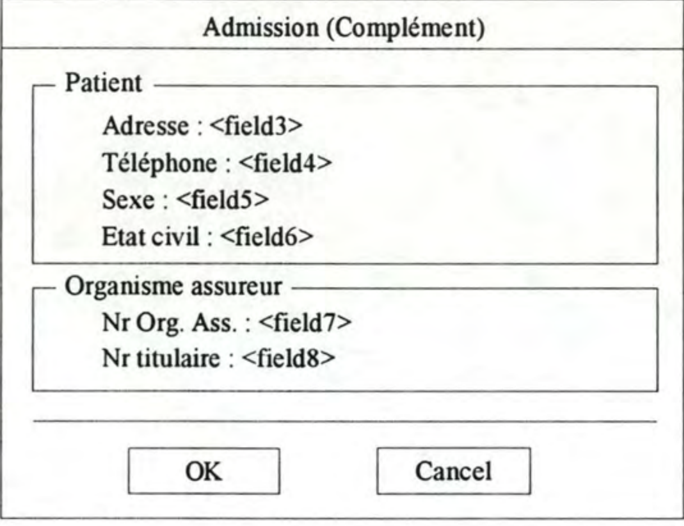
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : CAT, MED (cf. supra), ORG_ASS
- **Primitives**
  - $catExistante
  - $catégorie
  - $oAExistant
  - $organismeAssureur
- **Code d'erreur** : CAT_NOT_EXISTS, OA_NOT_EXISTS

17. OBJET INFO_ADM_POST_D

Rem : IS-A PAS_VERIF_D AND VERIF_D

APERÇU DE LA BOÎTE



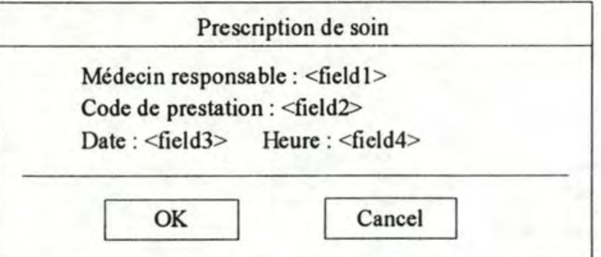
OBJET ET PRIMITIVES DE L'OBJET

déjà défini.

18. OBJET PRESCR_SOIN_D

Rem : IS-A PAS_VERIF_D AND VERIF_D

APERÇU DE LA BOÎTE



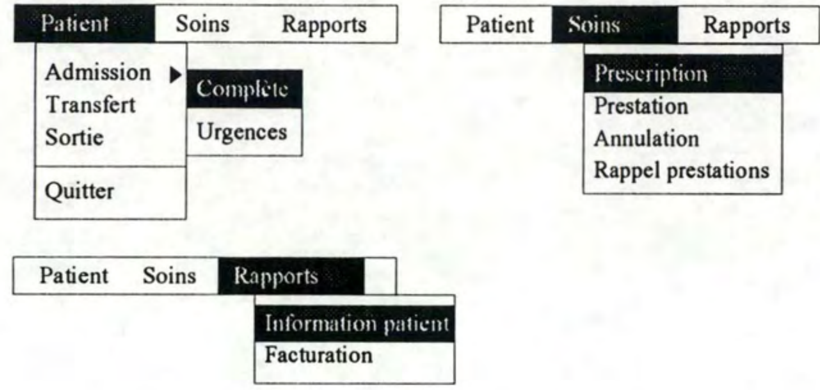
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : PREST
- **Primitives**
  - $prestExistante
  - $prestation
- **Code d'erreur** : PREST_NOT_EXISTS

19. OBJET MENU_D

Rem : IS-A DIALOGUE

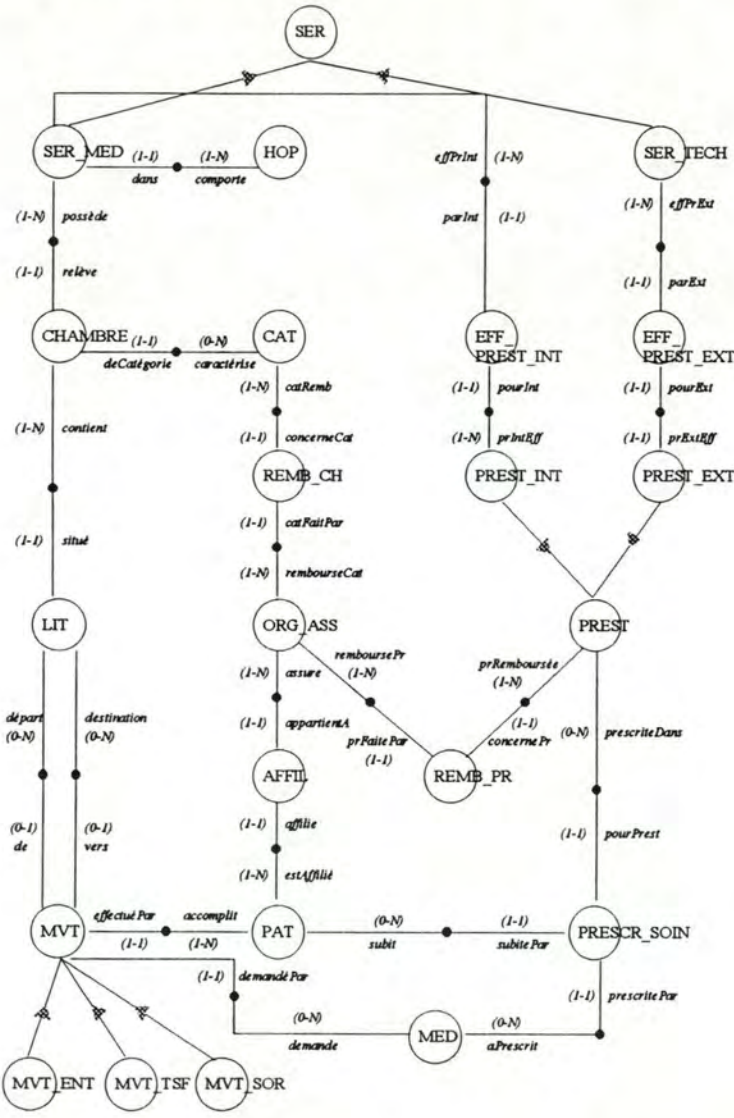
APERÇU DU MENU



Le fait de sélectionner une fonctionnalité entraîne l'appel de la fonction `ddeFonct` de l'objet `Session`. L'objet `Menu` est crée et détruit par ce même objet.

III. LE NIVEAU 3

1. ARCHITECTURE GLOBALE



2. OBJET HOP

Rem : tous les objets sont de types tables (TBL) et la base de données est supposée initialisée.

INTERFACE

- **Observations**
  - dateDerFact : DATE
  - dateDerStat : DATE
  - comporte : LIST of SER_MED
  - nom : STRING

- **Keys**
  - **$hôpital (nom : STRING) : HOP**
    ? $hopExistant (nom)

- **Events**
  - *** birth (nom : STRING, dateDerFact, dateDerStat : DATE)**
  - **fixeDateDerFact (date : DATE)**

BODY

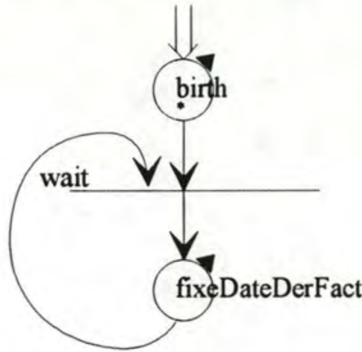
- **Attributes**
  - dateDerFact : DATE
  - dateDerStat : DATE
  - nom : STRING

- **Derivations**
  - comporte
    := ALL[sM : SER_MED | sM.dans = SELF]
  - $hôpital (nom)
    := ONE[h:HOP | h.nom = nom]

- **Actions**
  - *** birth (nom, dateDerFact, dateDerStat)**
    ⇒ nom := birth.nom
    ⇒ dateDerFact := birth.dateDerFact
    ⇒ dateDerStat := birth.dateDerSat

- fixeDateDerFact (date)
  ⇒ dateDerFact := fixeDateDerFact.date

BEHAVIOUR



3. OBJET SER

INTERFACE

- **Observations**
  - nom : STRING
  - $serExistant (nom : STRING) : BOOL

- **Keys**
  - $service (nom : STRING) : SER_MED
    ? $serExistant (nom)

- **Events**
  - * birth (nom : STRING)

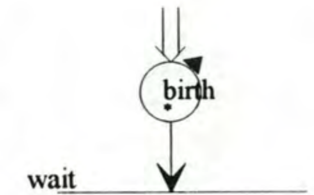
BODY

- **Attributes**
  - nom : STRING

- **Derivations**
  - $service (nom)
    := ONE[s : SER | s.nom = nom]
  - $serExistant (nom)
    := EXISTS[s : SER | s.nom = nom]

- **Actions**
  - * birth (nom)
    ⇒ nom := birth.nom

BEHAVIOUR



4. OBJET SER_MED

Rem : IS-A SER

INTERFACE

- **Observations**
  - dans : HOP
  - possède : LIST of CHAMBRE
  - serPlein : BOOL
  - nom : STRING
  - effPrInt : LIST of EFF_PREST_INT
  - $serMedExistant (nom : STRING) : BOOL

- **Keys**
  - $serviceMédical (nom : STRING) : SER_MED
    ? $serMedExistant (nom)

- **Events**
  - * birth (nom : STRING, hop : HOP)

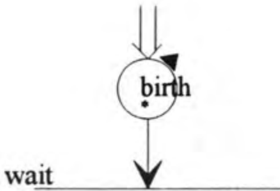
Body

- **Attributes**
  - dans : HOP
  - nom : STRING

- **Derivations**
  - possède
    := ALL[ch : CHAMBRE | ch.relève = SELF]
  - $serviceMédical (nom)
    := ONE[sM : SER_MED | sM.nom = nom]
  - serPlein
    := NOT EXISTS[l : LIT | l.serDeLit = SELF AND l.litOccupé=FALSE]
  - $serMedExistant (nom)
    := EXISTS[sM : SER_MED | sM.nom = nom]
  - effPrInt
    := ALL[eff : EFF_PREST_INT | eff.parInt = SELF]

- **Actions**
  - * birth (nom, hop)
    ⇒ nom := birth.nom
    ⇒ dans := birth.hop

Behaviour



5. OBJET CHAMBRE

Interface

- **Observations**
  - deCatégorie : CAT
  - contient : LIST of LIT
  - relève : SER_MED
  - nrCh : NAT
  - $chambreExistante (nrCh : NAT) : BOOL

- **Keys**
  - $chambre (nrCh : NAT) : CHAMBRE
    ? $chambreExistante (nrCh)

- **Events**
  - * birth (serMed : SER_MED, cat : CAT)

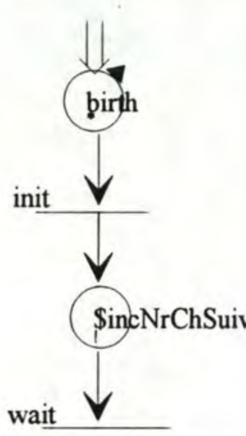
Body

- **Attributes**
  - nrCh, deCatégorie, relève
  - $nrChSuiv : NAT

- **Derivations**
  - contient
    := ALL[l : LIT | l.situé = SELF]
  - $chambre (nrCh)
    := ONE[ch : CHAMBRE | ch.nrCh = nrCh]
  - $chambreExistante (nrCh)
    := EXISTS[ch : CHAMBRE | ch.nrCh = nrCh]

- **Actions**
  - * birth (serMed, cat)
    ⇒ nrCh := $nrChSuiv
    ⇒ deCatégorie := birth.cat
    ⇒ relève := birth.serMed
  - ! $incNrChSuiv
    ⇒ $NrChSuiv := $nrChSuiv+1

BEHAVIOUR



6. OBJET CAT

INTERFACE

- **Observations**
  - cat : STRING
  - caractérise : LIST of CHAMBRE
  - prixPlein : NAT
  - catRemb : LIST of REMB_CH
  - $catExistante (cat : STRING) : BOOL

- **Keys**
  - $catégorie (cat : STRING) : CAT
    ? $catExistante (cat)

- **Events**
  - * birth (cat : STRING, prixPlein : NAT)

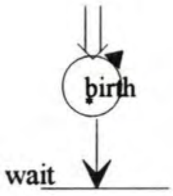
BODY

- **Attributes**
  - cat, prixPlein

- **Derivations**
  - caractérise
    := ALL[ch : CHAMBRE | ch.deCatégorie = SELF]
  - $catégorie (cat)
    := ONE[c : CAT | c.cat = cat]
  - catRemb
    := ALL[rCh : REMB_CH | rCh.concerneCat = SELF]
  - $catExistante (cat)
    := EXISTS[c : CAT | c.cat = cat]

- **Actions**
  - * birth (cat, prixPlein)
    ⇒ cat := birth.cat
    ⇒ prixPlein := birth.prixPlein

BEHAVIOUR



7. OBJET LIT

INTERFACE

- **Observations**
  - catégorieLit : CAT
  - situé : CHAMBRE
  - relève : SER_MED
  - nrLit : NAT
  - départ, destination : LIST of MVT
  - litOccupé : BOOL
  - serDeLit : SER_MED
  - litOccupéPar : PAT

- $litExistant (nrLit : NAT) : BOOL

- **Keys**
  - $lit (nrLit : NAT) : LIT
    ? $litExistant (nrLit)

- **Events**
  - * birth (ch : CHAMBRE)
  - fixeLitOccupéPar (pat :PAT)
  - fixeLitLibre

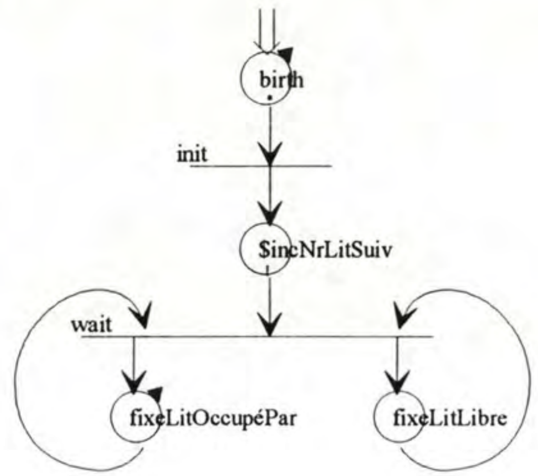
BODY

- **Attributes**
  - nrLit, situé, litOccupéPar
  - $nrLitSuiv : NAT

- **Derivations**
  - départ
    := ALL[m : MVT_TSF/SOR | m.de = SELF]
  - destination
    := ALL[m : MVT_TSF/ENT | m.vers = SELF]
  - litOccupé
    := litOccupéPar <> NULL
  - catégorieLit
    := SELF.situé.deCatégorie
  - serDeLit
    := SELF.situé.relève
  - $lit (nrLit)
    := ONE[l : LIT | l.nrLit = nrLit]
  - $litExistant (nrLit)
    := EXISTS[l : LIT | l.nrLit = nrLit]

- **Actions**
  - * birth (ch)
    ⇒ nrLit := $nrLitSuiv
    ⇒ situé := birth.ch
    ⇒ litOccupéPar := NULL
  - ! $incNrLitSuiv
    ⇒ $NrLitSuiv := $nrLitSuiv+1
  - fixeLitOccupéPar(p)
    ⇒ litOccupéPar := fixeLitOccupéPar.p
  - fixeLitLibre
    ⇒ litOccupéPar := NULL

BEHAVIOUR



8. OBJET MVT

INTERFACE

- **Observations**
  - nrMvt : NAT
  - dateMvt : DATE
  - heureMvt : TIME
  - typeMvt : TYPE_MVT_DT
  - *de, vers : LIT*
  - demandéPar : MED
  - effectuéPar : PAT
  - $mvtExistant (nrMvt : NAT) : BOOL

- **Keys**
  - $mouvement (nrMvt : NAT) : MVT
    ? $mvtExistant (nrMvt)

- **Events**
  - * birth (p : PAT, med : MED, date : DATE, heure : TIME)

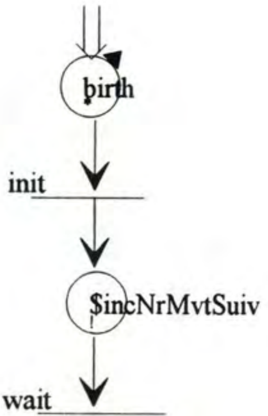
BODY

- **Attributes**
  - nrMvt, dateMvt, heureMvt, typeMvt, demandéPar, effectuéPar, *de, vers*
  - $nrMvtSuiv : NAT

- **Derivations**
  - $mouvement (nrMvt)
    := ONE[m:MVT | m.nrMvt = nrMvt]
  - $litExistant (nrMvt)
    := EXISTS[m:MVT | m.nrMvt = nrMvt]

- **Actions**
  - * birth (p, med, date, heure)
    ⇒ nrMvt := $nrMvtSuiv
    ⇒ dateMvt := birth.date
    ⇒ heureMvt := birth.heure
    ⇒ effectuéPar := birth.pat
    ⇒ demandéPar := birth.med
  - ! $incNrMvtSuiv
    ⇒ $NrMvtSuiv := $nrMvtSuiv+1

BEHAVIOUR



9. OBJET MVT_ENT

Rem : IS-A MVT

INTERFACE

- **Observations**
  - nrMvt : NAT
  - dateMvt : DATE
  - heureMvt : TIME
  - typeMvt : TYPE_MVT_DT
  - vers : LIT
  - demandéPar : MED
  - effectuéPar : PAT
  - $mvtExistant (nrMvt : NAT) : BOOL

- **Keys**
  - $mouvement (nrMvt : NAT) : MVT
    ? $mvtExistant (nrMvt)
- **Events**
  - * birth (litDest : LIT, p : PAT, med : MED, date : DATE, heure : TIME)

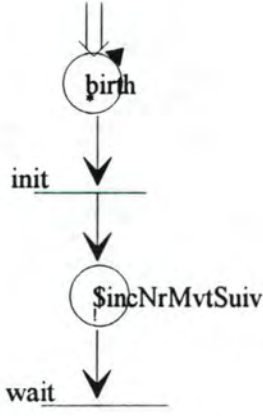
BODY

- **Attributes**
  - nrMvt, dateMvt, heureMvt, typeMvt, demandéPar, effectuéPar, vers
  - $nrMvtSuiv : NAT

- **Derivations**
  - $mouvement (nrMvt)
    := ONE[m:MVT | m.nrMvt = nrMvt]
  - $litExistant (nrMvt)
    := EXISTS[m:MVT | m.nrMvt = nrMvt]

- **Actions**
  - * birth (p, med, date, heure)
    ⇒ nrMvt := $nrMvtSuiv
    ⇒ dateMvt := birth.date
    ⇒ heureMvt := birth.heure
    ⇒ effectuéPar := birth.pat
    ⇒ demandéPar := birth.med
    ⇒ vers := birth.litDest
    >> birth.litDest.fixeLitOccupéPar(p)
    ⇒ typeMvt = ENTREE
  - ! $incNrMvtSuiv
    ⇒ $NrMvtSuiv := $nrMvtSuiv+1

BEHAVIOUR



10. OBJET MVT_TSF

Rem : IS-A MVT

INTERFACE

- **Observations**
  - nrMvt : NAT
  - dateMvt : DATE
  - heureMvt : TIME
  - typeMvt : TYPE_MVT_DT
  - de, vers : LIT
  - demandéPar : MED
  - effectuéPar : PAT
  - $mvtExistant (nrMvt : NAT) : BOOL

- **Keys**
  - $mouvement (nrMvt : NAT) : MVT
    ? $mvtExistant (nrMvt)
- **Events**
  - * birth (litOr, litDest : LIT, p : PAT, med : MED, date : DATE, heure : TIME)

BODY

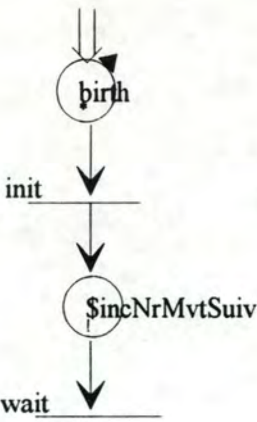
- **Attributes**
  - nrMvt, dateMvt, heureMvt, typeMvt, demandéPar, effectuéPar, de, vers
  - $nrMvtSuiv : NAT

- **Derivations**
  - $mouvement (nrMvt)
    := ONE[m:MVT | m.nrMvt = nrMvt]
  - $litExistant (nrMvt)
    := EXISTS[m:MVT | m.nrMvt = nrMvt]

- **Actions**
  - * birth (p, med, date, heure)
    ⇒ nrMvt := $nrMvtSuiv
    ⇒ dateMvt := birth.date
    ⇒ heureMvt := birth.heure
    ⇒ effectuéPar := birth.pat
    ⇒ demandéPar := birth.med
    ⇒ vers := birth.litDest
    ⇒ de := birth.litOr
    >> birth.litDest.fixeLitOccupéPar(p)
    >> birth.litOr.fixeLitLibre
    ⇒ typeMvt = TRANSFERT

- ! $incNrMvtSuiv
  ⇒ $NrMvtSuiv := $nrMvtSuiv+1

BEHAVIOUR



11. OBJET MVT_SOR

Rem : IS-A MVT

INTERFACE

- **Observations**
  - nrMvt : NAT
  - dateMvt : DATE
  - heureMvt : TIME
  - typeMvt : TYPE_MVT_DT
  - de : LIT
  - demandéPar : MED
  - effectuéPar : PAT
  - $mvtExistant (nrMvt : NAT) : BOOL

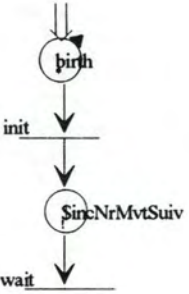
- **Keys**
  - $mouvement (nrMvt : NAT) : MVT
    ? $mvtExistant (nrMvt)

- **Events**
  - * birth (litOr : LIT, p : PAT, med : MED, date : DATE, heure : TIME)

BODY

- **Attributes**
  - nrMvt, dateMvt, heureMvt, typeMvt, demandéPar, effectuéPar, de
  - $nrMvtSuiv : NAT
- **Derivations**
  - $mouvement (nrMvt)
    := ONE[m:MVT | m.nrMvt = nrMvt]
  - $litExistant (nrMvt)
    := EXISTS[m:MVT | m.nrMvt = nrMvt]
- **Actions**
  - * birth (litOr, p, med, date, heure)
    ⇒ nrMvt := $nrMvtSuiv
    ⇒ dateMvt := birth.date
    ⇒ heureMvt := birth.heure
    ⇒ effectuéPar := birth.pat
    ⇒ demandéPar := birth.med
    ⇒ de := birth.litOr
    >> birth.litOr.fixeLitLibre
    ⇒ typeMvt = SORTIE
  - ! $incNrMvtSuiv
    ⇒ $NrMvtSuiv := $nrMvtSuiv+1

BEHAVIOUR



12. OBJET REMB_CH

INTERFACE

- **Observations**
  - concerneCat : CAT
  - catFaitPar : ORG_ASS
  - prixRemb : NAT
  - $rembChExistant (cat : CAT, oA : ORG_ASS) : BOOL

- **Keys**
  - $remboursementChambre (cat : CAT, oA : ORG_ASS) : REMB_CH
    ? $rembChExistant (cat, oA)
- **Events**
  - * birth (cat : CAT, oA : ORG_ASS, prix : NAT)

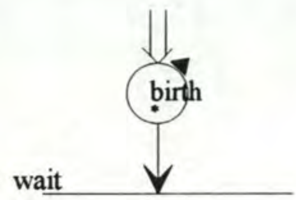
BODY

- **Attributes**
  - concerneCat, catFaitPar, prixRemb

- **Derivations**
  - $remboursementChambre (cat, oA)
    := ONE[rCh : REMB_CH | rCh.concerneCat = cat AND rCh.catFaitPar = oA]
  - $rembChExistant (cat, oA)
    := EXISTS[rCh : REMB_CH | rCh.concerneCat = cat AND rCh.catFaitPar = oA]

- **Actions**
  - * birth (cat, oA, prix)
    ⇒ concerneCat := birth.cat
    ⇒ catFaitpar := birth.oA
    ⇒ prixRemb := birth.prix

BEHAVIOUR



13. OBJET REMB_PR

INTERFACE

- **Observations**
  - concernePr : PREST
  - prFaitPar : ORG_ASS
  - prixRemb : NAT
  - $rembPrExistant (prest : PREST, oA : ORG_ASS) : BOOL

- **Keys**
  - $remboursementPrestation (prest : PREST, oA : ORG_ASS) : REMB_CH
    ? $rembPrExistant (prest, oA)
- **Events**
  - * birth (prest : PREST, oA : ORG_ASS, prix : NAT)

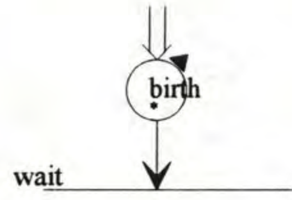
BODY

- **Attributes**
  - concernePr, prFaitPar, prixRemb

- **Derivations**
  - $remboursementPrestation (prest, oA)
    := ONE[rPr : REMB_PR | rPr.concernePr = prest AND rPr.prFaitPar = oA]
  - $rembPrExistant (prest, oA)
    := EXISTS[rPr : REMB_PR | rPr.concernePr = prest AND rPr.prFaitPar = oA]

- **Actions**
  - * birth (prest, oA, prix)
    ⇒ concernePr := birth.prest
    ⇒ prFaitpar := birth.oA
    ⇒ prixRemb := birth.prix

BEHAVIOUR



14. OBJET ORG_ASS

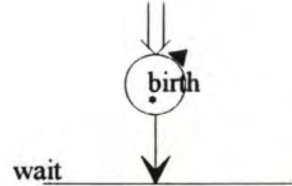
INTERFACE

- **Observations**
  - nrOA : NAT
  - adrOA : STRING
  - nomOA : STRING
  - rembourseCat : LIST of REMB_CH
  - assure : LIST of AFFIL
  - remboursePr : LIST of PREST
  - $oAExistant (nrOA : NAT) : BOOL
- **Keys**
  - $organismeAssureur (nrOA : NAT) : ORG_ASS
    ? $oAExistant (nrOA)
- **Events**
  - * birth (nrOA : NAT, adrOA : STRING, nomOA : STRING)

BODY

- **Attributes**
  - nrOA, adrOA, nomOA
- **Derivations**
  - $organismeAssureur(nrOA)
    := ONE[oA : ORG_ASS | oA.nrOA = nrOA]
  - $oAExistant (nrOA)
    := EXISTS[oA : ORG_ASS | oA.nrOA = nrOA]
  - rembourseCat
    := ALL[rCh : REMB_CH | rCh.catFaitPar=SELF]
  - assure
    := ALL[aff : AFFIL | aff.appartientA=SELF]
  - remboursePr
    := ALL[rPr : REMB_PR | rPr.prFaitPar=SELF]
- **Actions**
  - * birth (nrOA, adrOA, nomOA)
    ⇒ nrOA := birth.nrOA
    ⇒ adrOA := birth.adrOA
    ⇒ nomOA := birth.nomOA

BEHAVIOUR



15. OBJET SER_TECH

Rem : IS-A SER

INTERFACE

- **Observations**
  - nom : STRING
  - $serTechExistant (nom : STRING) : BOOL
  - effPrExt : LIST of EFF_PREST_EXT

- **Keys**
  - $serviceTechnique (nom : STRING) : SER_TECH
    ? $serTechExistant (nom)

- **Events**
  - * birth (nom : STRING)

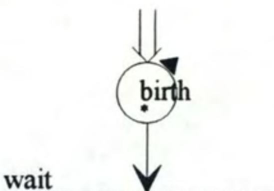
BODY

- **Attributes**
  - nom : STRING

- **Derivations**
  - $serviceTechnique (nom)
    := ONE[sT : SER_TECH | sT.nom = nom]
  - $serTechExistant (nom)
    := EXISTS[sT : SER_TECH | sT.nom = nom]
  - effPrExt
    := ALL[eff : EFF_PREST_EXT | eff.parExt = SELF]

- **Actions**
  - * birth (nom)
    ⇒ nom := birth.nom

BEHAVIOUR



16. OBJET AFFIL

INTERFACE

- **Observations**
  - appartientA : ORG_ASS
    {? état=libre}
  - affilie : PAT
    {? état=libre}
  - nrTit : NAT
    {? état=libre}
  - typeAffil : TYPE_AFFIL_DT
    {? état=libre}
  - $affilExistante (oA, pat) : BOOL
    {? état=libre}

- **Keys**
  - $affiliation (oA, pat) : AFFIL
    {? état=libre}
    {? $affilExistante (oA, pat)}

- **Events**
  - * birth (pat, oA, nrTit, typeAffil)

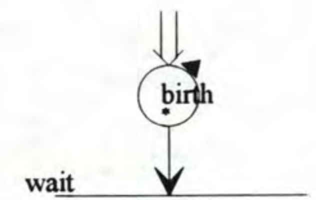
BODY

- **Attributes**
  - appartientA, affilie, nrTit, typeAffil
  - état : STATE_DT

- **Derivations**
  - $affiliation (oA, pat)
    := ONE[aff : AFFIL | aff.appartientA=oA AND aff.affilie=pat]
  - $affilExistante (oA, pat)
    := EXISTS[aff : AFFIL | aff.appartientA=oA AND aff.affilie=pat]

- **Actions**
  - * birth (pat, oA, nrTit, typeAffil)
    ⇒ affilie := birth.pat
    ⇒ appartientA := birth.oA
    ⇒ nrTit := birth.nrTit
    ⇒ typeAffil := birth.typeAffil
    ⇒ état := libre

BEHAVIOUR



17. OBJET PAT

INTERFACE

- **Observations**
  - nrDos : NAT
    {? état=libre}
  - coord : COORD_DT
    {? état=libre}
  - nom, prenom : STRING
    {? état=libre}
  - dateNaiss : DATE
    {? état=libre}
  - adrPat, tél : STRING
    {? état=libre}
  - etatCivil : ETAT_CIVIL_DT
    {? état=libre}
  - sexe : SEX_DT
    {? état=libre}
  - estAffilié : AFFIL
    {? état=libre}
  - subit : LIST of PRESCR_SOIN

{? état=libre}
- accomplit : LIST of MVT
  {? état=libre}
- $patExistantParCoord (coord : COORD_DT) : BOOL
  {? état=libre}
- $patExistantParNrDos (nrDos : NAT) : BOOL
  {? état=libre}
- patDansSer(serMed : SER_MED) : BOOL
  {? état=libre}
- oADePat : ORG_ASS
  {? état=libre}
- listeMvtDates (dateD, heureD, DateF, heureF) :
  CP[listesMvtDates : LIST of MVT, nbHosp : NAT]
  {? état=libre}
- listePrescrDates (dateD, heureD, DateF, heureF) : LIST of PRESCR_SOIN
  {? état=libre}
- patMalade : BOOL
  {? état=libre}
- litDePat : LIT
  {? état=libre}
- patDansLit (lit : LIT) : BOOL
  {? état=libre}

- **Keys**
  - $patParCoord (coord : COORD_DT) : PAT
    {? état=libre}
    {? $patExistantParCoord (coord)}
  - $patParNrDos (nrDos : NAT) : PAT
    {? état=libre}
    {? $patExistantParNrDos (nrDos)}

- **Events**
  - * birth (coord : COORD_DT)
  - fixeAdr (adr : STRING)
  - fixeTel (tel : STRING)
  - fixeEtatCivil (etatCivil : ETAT_CIVIL_DT)
  - fixeSexe (sexe : SEX_DT)

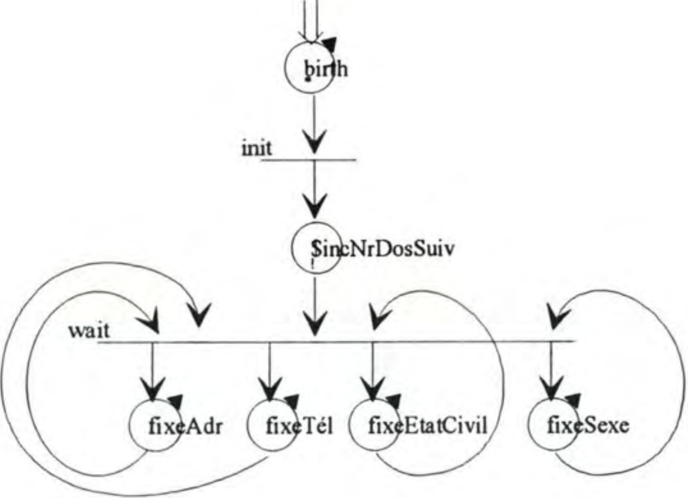
BODY

- **Attributes**
  - nrDos, coord, adr, etatCivil, sexe, tel, $nrDosSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - estAffilié
    := ONE[aff : AFFIL | aff.affilie=SELF]
  - subit
    := ALL[pS : PRESCR_SOIN | pS.subiePar = SELF]
  - accomplit
    := ALL[m : MVT | m.effectuéPar = SELF]
  - nom, prénom, dateNaiss
    := coord.nom, .prénom, .dateNaiss
  - $patExistantParCoord/NrDos (coord/nrDos)
    := EXISTS[p : PAT | p.coord/nrDos = coord/nrDos]
  - patDansSer (serMed)
    := SELF.litDePat.serDeLit = serMed
  - oADePat
    := SELF.estAffilié.appartientA
  - listeMvtDates (...)
    := listeMvtDates := ALL[m : MVT | m in SELF.accomplit AND dateF >= m.dateMvt > dateD AND heureF >= m.heureMvt > heureD] SORTED ON (dateMvt, heureMvt)
    := nbHosp := #(ALL[m : MVT | m in listeMvtDates AND (CLASS_OF (m) = "MVT_ENT" OR CLASS_OF (m) = "MVT_SOR")]) DIV 2
  - listePrescrDates (...)
    := listePrescrDates := ALL[pS : PRESCR_SOIN | pS in SELF.subit AND dateF >= pS.datePr > dateD AND heureF >= pS.heurePr > heureD AND pS.enAttente= FALSE] SORTED ON (datePr, heurePr)
  - patMalade
    := EXISTS[l : LIT | l.litOccupéPar = SELF]
  - litDePat
    := ONE[l : LIT | l.litOccupéPar = SELF]
  - patDansLit (lit)
    := lit.litOccupéPar = SELF
  - $patParCoord/NrDos (coord/nrDos)
    := ONE[p : PAT | p.coord/nrDos = coord/nrDos]

- **Actions**
  - * birth (coord)
    ⇒ coord := birth.coord
    ⇒ nrDos := $nrDosSuiv
    ⇒ état := occupé
  - $incNrDosSuiv
    ⇒ $nrDosSuiv := $nrDosSuiv + 1
    ⇒ état := libre
  - fixeAdr (adr)
    ⇒ adr := fixeAdr.adr
  - fixeTel (tel)
    ⇒ tel := fixeTel.tel
  - fixeEtatCivil (EtatCivil)
    ⇒ EtatCivil := fixeEtatCivil.EtatCivil
  - fixeSexe (sexe)
    ⇒ Sexe := fixeSexe.Sexe

BEHAVIOUR



18. OBJET MED

INTERFACE

- **Observations**
  - nrMed : NAT
    {? état=libre}
  - nom, prenom : STRING
    {? état=libre}
  - adr : STRING
    {? état=libre}
  - aPrescrit : LIST of PRESCR_SOIN
    {? état=libre}
  - demande : LIST of MVT
    {? état=libre}
  - $medExistant (nrMed : NAT) : BOOL
    {? état=libre}

- **Keys**
  - $médecin (nrMed : NAT) : MED
    {? état=libre}
    {? $medExistant (nrMed)}

- **Events**
  - * birth (nom, prenom adr : STRING)

BODY

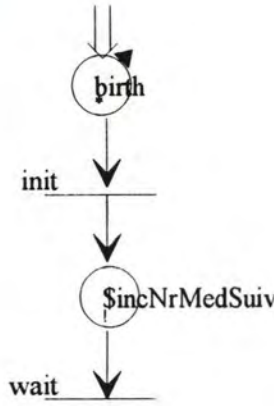
- **Attributes**
  - nrMed, nom, prenom, adr,$nrMedSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - aPrescrit
    := ALL[pS : PRESCR_SOIN | pS.prescritePar = SELF]
  - demande
    := ALL[m : MVT | m.demandéPar = SELF]
  - $medExistant(nrMed)
    := EXISTS [med : MED | med.nrMed = nrMed]

- $médecin (nrMed)
  := ONE[med : MED | med.nrMed = nrMed]

- **Actions**
  - * birth (...)
    ⇒ nom; prenom, adr := birth.nom, prenom, adr
    ⇒ nrMed := $nrMedSuiv
    ⇒ état := occupé
  - $incNrMedSuiv
    ⇒ $nrMedSuiv := $nrMedSuiv + 1
    ⇒ état := libre

BEHAVIOUR



19. OBJET PRESCR_SOIN

INTERFACE

- **Observations**
  - nrPr : NAT
    {? état=libre}
  - datePr/heurePr : DATE/TIME
    {? état=libre}
  - enAttente : BOOL
    {? état=libre}
  - subitePar : PAT
    {? état=libre}
  - prescritePar : MED
    {? état=libre}
  - pourPrest : PREST
    {? état=libre}
  - $prescrExistante (nrPr : NAT) : BOOL
    {? état=libre}

- **Keys**
  - $prescriptionSoin (nrPr : NAT) : PRESCR_SOIN
    {? état=libre}
    {? $prescrExistante (nrPr)}

- **Events**
  - * birth (pat, med, prest, date, heure)
  - fixeEtat (etat)
  - + death

BODY

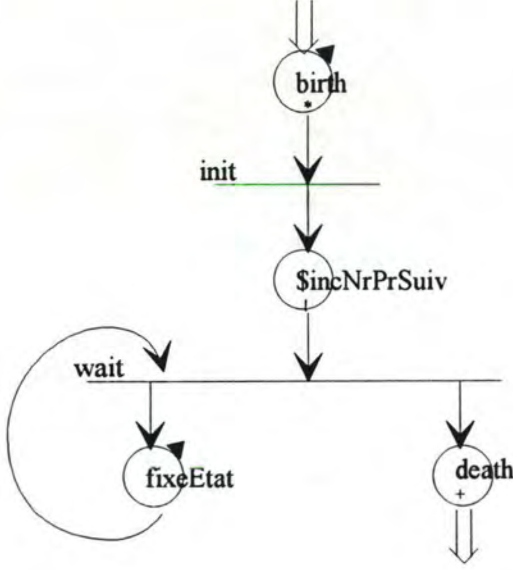
- **Attributes**
  - nrPr, datePr, heurePr, etatPr : ETAT_PRESCR_DT, subitePar, prescritePar, pourPrest, $nrPrSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - $prescrExistante(nrPr)
    := EXISTS [pS : PRESCR_SOIN | pS.nrPrescr = nrPr]
  - $prescriptionSoin (nrPr)
    := ONE[pS : PRESCR_SOIN | pS.nrPrescr = nrPr]

- **Actions**
  - * birth (...)
    ⇒ etc. := birth.etc.
    ⇒ nrPr := $nrPrSuiv
    ⇒ état := occupé

- $incNrPrSuiv
  ⇒ $nrPrSuiv := $nrPrSuiv + 1
  ⇒ état := libre
- fixeEtat (etat)
  ⇒ etatPr := fixeEtat.etat
- death

BEHAVIOUR



20. OBJET PREST

INTERFACE

- **Observations**
  - codePrest : STRING
    {? état=libre}
  - nomPr : STRING
    {? état=libre}
  - typePrest : TYPE_PREST_DT
    {? état=libre}
  - capMaxi, prixPleinPr : NAT
    {? état=libre}
  - prRemboursée : LIST of REMB_PR
    {? état=libre}
  - prescriteDans : LIST of PRESCR_SOIN
    {? état=libre}
  - $prestExistante (code) : BOOL
    {? état=libre}

- **Keys**
  - $prestation (code) : PREST
    {? état=libre}
    {? $prestExistante (code)}

- **Events**
  - * birth (code, nom, type, capMaxi, prix)

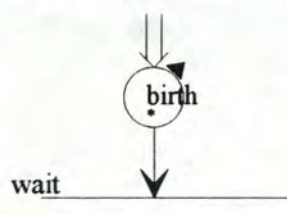
BODY

- **Attributes**
  - codePrest, nomPrest, typeprest, capMaxi, prixPlein
  - état : STATE_DT

- **Derivations**
  - $prestExistante(code)
    := EXISTS [prest : PREST | prest.codePrest = code]
  - $prestation (code)
    := ONE[prest : PREST | prest.codePrest = code]
  - prRemboursée
    := ALL[rPr : REMB_PR | rPr.concernePr=SELF]
  - prescriteDans
    := ALL[pS : PRESCR_SOIN | pS.pourPrest=SELF]

- **Actions**
  - * birth (...)
    ⇒ etc. := birth.etc
    ⇒ état := libre

Behaviour



21. OBJET PREST_EXT

Interface

- **Observations**
  - codePrest : STRING
    {? état=libre}
  - nomPr : STRING
    {? état=libre}
  - typePrest : TYPE_PREST_DT
    {? état=libre}
  - capMaxi, prixPleinPr : NAT
    {? état=libre}
  - prRemboursée : LIST of REMB_PR
    {? état=libre}
  - prescriteDans : LIST of PRESCR_SOIN
    {? état=libre}
  - $prestExistante (code) : BOOL
    {? état=libre}
  - prExtEff : EFF_PREST_EXT
    {? état=libre}
  - serDePrest : SER_MED
    {? état=libre}

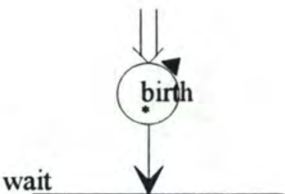
- **Keys**
  - $prestation (code) : PREST
    {? état=libre}
    {? $prestExistante (code)}
- **Events**
  - * birth (code, nom, type, capMaxi, prix)

Body
- **Attributes**
  - codePrest, nomPrest, typeprest, capMaxi, prixPlein
  - état : STATE_DT

- **Derivations**
  - $prestExistante(code)
    := EXISTS [prest : PREST | prest.codePrest = code]
  - $prestation (code)
    := ONE[prest : PREST | prest.codePrest = code]
  - prRemboursée
    := ALL[rPr : REMB_PR | rPr.concernePr=SELF]
  - prescriteDans
    := ALL[pS : PRESCR_SOIN | pS.pourPrest=SELF]
  - prExtEff
    := ONE [eff : EFF_PREST_EXT | pourExt=SELF]
  - serDePrest
    := SELF.prExtEff.parExt

- **Actions**
  - * birth (...)
    ⇒ etc. := birth.etc.
    ⇒ état := libre

Behaviour



22. OBJET PREST_INT

INTERFACE

- **Observations**
  - codePrest : STRING
    {? état=libre}
  - nomPr : STRING
    {? état=libre}
  - typePrest : TYPE_PREST_DT
    {? état=libre}
  - capMaxi, prixPleinPr : NAT
    {? état=libre}
  - prRemboursée : LIST of REMB_PR
    {? état=libre}
  - prescriteDans : LIST of PRESCR_SOIN
    {? état=libre}
  - $prestExistante (code) : BOOL
    {? état=libre}
  - prIntEff : LIST of EFF_PREST_INT
    {? état=libre}

- **Keys**
  - $prestation (code) : PREST
    {? état=libre}
    {? $prestExistante (code)}

- **Events**
  - * birth (code, nom, type, capMaxi, prix)

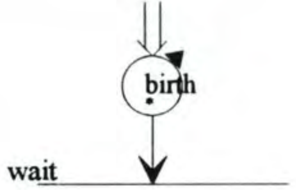
BODY

- **Attributes**
  - codePrest, nomPrest, typeprest, capMaxi, prixPlein
  - état : STATE_DT

- **Derivations**
  - $prestExistante(code)
    := EXISTS [prest : PREST | prest.codePrest = code]
  - $prestation (code)
    := ONE[prest : PREST | prest.codePrest = code]
  - prRemboursée
    := ALL[rPr : REMB_PR | rPr.concernePr=SELF]
  - prescriteDans
    := ALL[pS : PRESCR_SOIN | pS.pourPrest=SELF]
  - prExtEff
    := ALL [eff : EFF_PREST_INT | pourInt=SELF]

- **Actions**
  - * birth (...)
    ⇒ etc. := birth.etc.
    ⇒ état := libre

BEHAVIOUR



23. OBJET EFF_PREST_INT

INTERFACE

- **Observations**
  - pourInt : PREST_INT
    {? état=libre}
  - parInt : SER_MED
    {? état=libre}
  - nbUnitesExec : NAT
    {? état=libre}

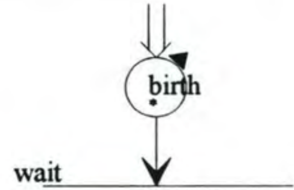
- **Keys**
- **Events**
  - * birth (serMed : SER_MED, prestInt : PREST_INT, nbExec : NAT)

BODY

- **Attributes**
  - pourInt, parInt, nbUnitesExec
  - état : STATE_DT

- **Actions**
  - * birth (serMed, prestInt, nbExec)
    ⇒ parInt := birth.serMed
    ⇒ pourInt := birth.prestInt
    ⇒ nbUnitesExec := birth.nbExec
    ⇒ état := libre

BEHAVIOUR



24. OBJET EFF_PREST_EXT

INTERFACE

- **Observations**
  - pourExt : PREST_EXT
    {? état=libre}
  - parExt : SER_TECH
    {? état=libre}
  - nbUnitesExec : NAT
    {? état=libre}

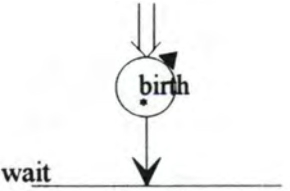
- **Keys**
- **Events**
  - * birth (serTech : SER_TECH, prestExt : PREST_EXT, nbExec : NAT)

BODY

- **Attributes**
  - pourExt, parExt, nbUnitesExec
  - état : STATE_DT

- **Actions**
  - * birth (serTech, prestExt, nbExec)
    ⇒ parExt := birth.serTech
    ⇒ pourExt := birth.prestExt
    ⇒ nbUnitesExec := birth.nbExec
    ⇒ état := libre

BEHAVIOUR

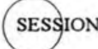


CONCEPTION DYNAMIQUE

I. LE NIVEAU 5

1. ARCHITECTURE GLOBALE

NIVEAU 5
Fonctionnalités



ADM_COMP ADM_PRE ADM_POST INTRO_TSF INTRO_SORTIE DDE_INFO_MAL

PRESCR_SOIN PREST_PRESCR ANNUL_PRESCR RAPPEL_PREST PROD_FACT

ALLOCATION PLANNING PAUSE SEJOUR FACT_PAT_OA

OPTIMISATION RAP_FACT RAP_SER_MED RAP_SER_TECH RAP_INFO_MAL

2. OBJET ADM_COMP

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

BODY

- **Attributes**
  - serMed : SER_MED
  - coordPat : COORD_DT
  - cat : CAT
  - med : MED
  - adrPat : ADR_DT
  - tel : PHONE_DT
  - sexe : SEX_DT
  - etatCiv : ETAT_CIV_DT
  - oA : ORG_ASS
  - nrTit : NR_DT
  - typeAff : TYPE_AFF_DT
  - lit : LIT
  - pat : PAT

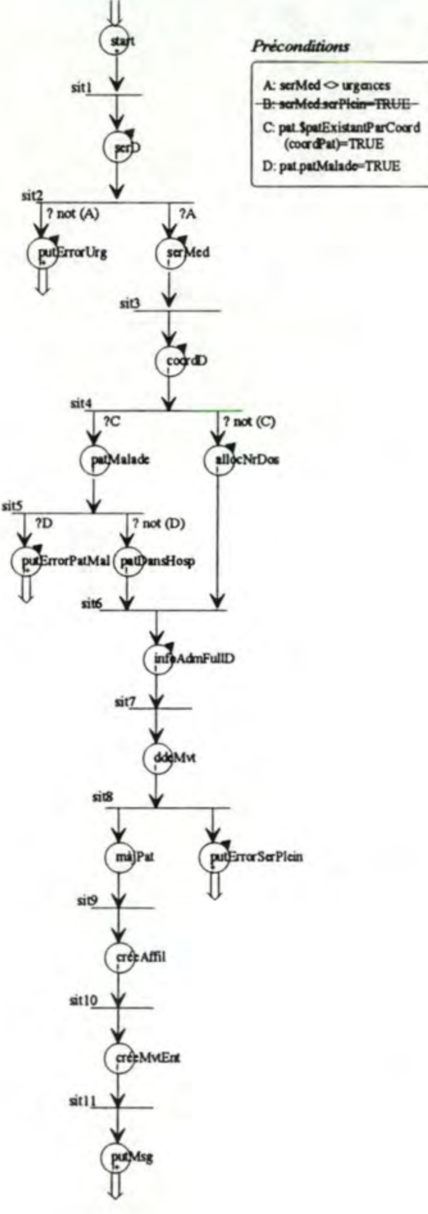
- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")

- **Actions**
  - * start
  - ! serD (serM : SER_MED)
    >> SER_D.open
    ⇒ serMed := serD.serM
  - ! serMed
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorUrg
    IS putError (code)

:= code = ADM_COMP_URG

- + putErrorSerPlein
  IS putError (code)
  := code = ADM_SER_PLEIN
- !+ putErrorPatMal
  IS putError (code)
  := code = ADM_PAT_MAL
- ! coordD (coordP : COORD_DT)
  >> COORD_D.open (coordP)
  ⇒ coordPat := coordD.coordP
- ! urgences
- ! patMalade
  ⇒ pat := pat.$patParCoord(coordPat)
- ! patDansHosp
- ! allocNrDos (p : PAT)
  >> allocation.allocNrDos (coordPat, p)
  ⇒ pat := allocNrDos.p
- ! infoAdmCompD (cat : CAT, med : MED, adr : ADR_DT, tel : PHONE_DT, sexe : SEX_DT, etatCiv : ETAT_CIV_DT, oA : ORG_ASS, nrTit : NR_DT)
  >> INFO_ADM_COMP_D.open (cat, med, adr, tel, sexe, etatCiv, oA, nrTit)
  ⇒ cat := infoAdmCompD.cat
  ⇒ ...
  ⇒ nrTit := infoAdmCompD.nrTit
- majPat (lit : LIT)
  ⇒ lit := maj.lit
  >> pat.fixeAdr (adrPat)
  >> pat.fixeTel (tel)
  >> pat.fixeSexe (sexe)
  >> pat.fixeEtatCiv (etatCiv)
- ! ddeMvt
  >> dde_mvt.birth (pat, serMed.priorité, cat, serMed, "E", putErrorSerPlein, majPat)
- ! creeMvtEntree
  >> mvt_ent.birth (lit, pat, med, TODAY(), NOW())
- ! creeAffil
  >> affil.birth (pat, oA, nrTit, typeAff)
- !+ putMsg (code : MSG_CODE_DT)
  >> MSG_D.open (code)
  := code = ADM_COMP_OK

BEHAVIOUR



3. OBJET ADM_PRE

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

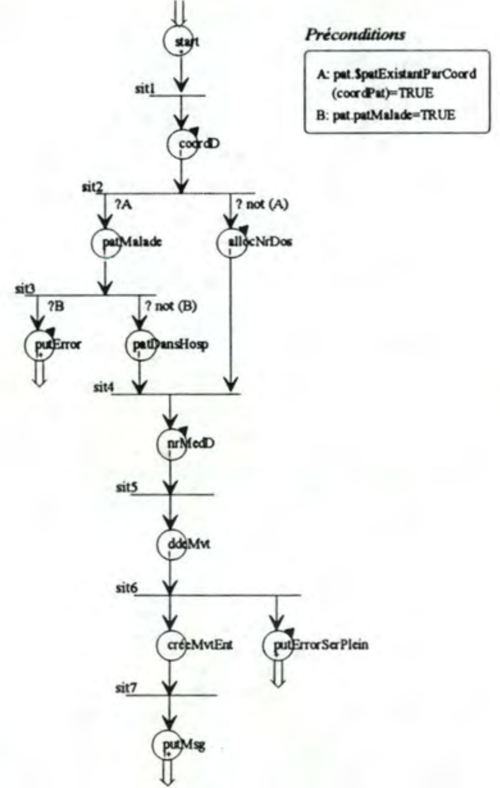
BODY

- **Attributes**
  - coordPat : COORD_DT
  - med : MED
  - lit : LIT
  - pat : PAT

- **Derivations**
  - urgences : SER_MED
    := SER_MED.$serviceMedical("Urgences")
  - cat : CAT
    := cat.$catégorie("1Lit")

- **Actions**
  - * start
  - ! coordD (coordP : COORD_DT)
    >> COORD_D.open (coordP)
    ⇒ coordPat := coordD.coordP
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
    := code = ADM_PAT_MAL
  - + putErrorSerPlein
    IS putError (code)
    := code = ADM_SER_PLEIN
  - ! patMalade
    ⇒ pat := pat.$patParCoord(coordPat)
  - ! patDansHosp
  - ! allocNrDos (p : PAT)
    >> allocation.allocNrDos (coordPat, p)
    ⇒ pat := allocNrDos.p
  - ! nrMedD (med : MED)
    >> NR_MED_D.open (med)
    ⇒ med := nrMedD.med
  - ! ddeMvt
    >> dde_mvt.birth (pat, urgences.priorité, cat, urgences, "E", putErrorSerPlein, creeMvtEnt)
  - creeMvtEnt (lit : LIT)
    ⇒ lit := creeMvtEnt.lit
    >> mvt_ent.birth (lit, pat, med, TODAY(), NOW())
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = ADM_PRE_OK

BEHAVIOUR



4. OBJET ADM_POST

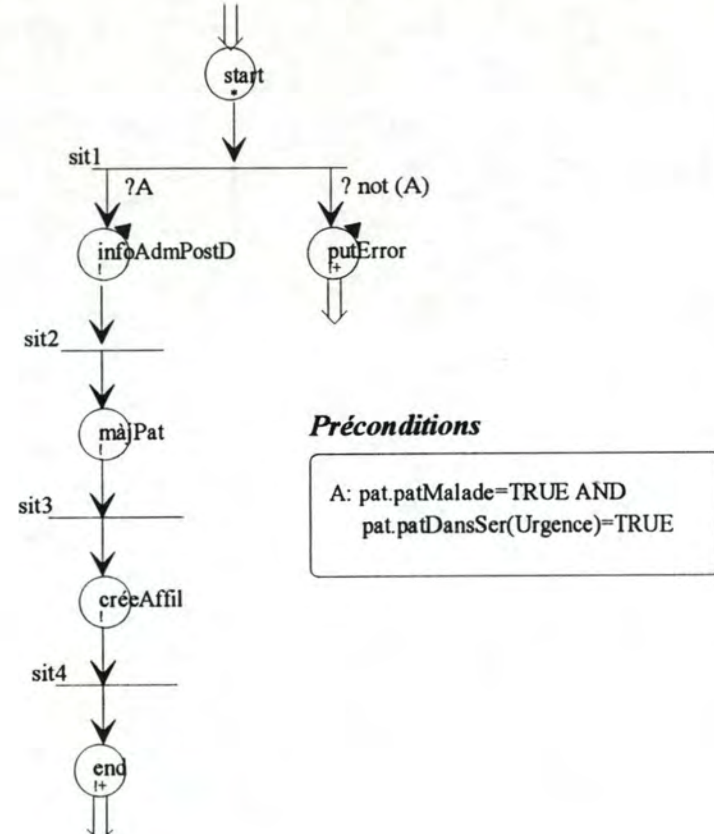
INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start (pat : PAT)

BODY

- **Attributes**
  - adrPat : ADR_DT
  - tel : PHONE_DT
  - sexe : SEX_DT
  - etatCiv : ETAT_CIV_DT
  - oA : ORG_ASS
  - nrTit : NR_DT
  - typeAff : TYPE_AFF_DT
  - pat : PAT
- **Derivations**
  - Urgence : SER_MED
    := ser_med.$serviceMedical("Urgence")
- **Actions**
  - * start (pat)
    ⇒ pat := start.pat
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
    := code = ADM_POST_NOT_IN_URG
  - ! infoAdmPostD (adr : ADR_DT, tel : PHONE_DT, sexe : SEX_DT, etatCiv : ETAT_CIV_DT, oA : ORG_ASS, nrTit : NR_DT)
    >> INFO_ADM_POST_D.open (adr, tel, sexe, etatCiv, oA, nrTit)
    ⇒ adr := infoAdmPostD.adr
    ⇒ ...
    ⇒ nrTit := infoAdmPostD.nrTit
  - ! majPat
    >> pat.fixeAdr (adrPat)
    >> pat.fixeTel (tel)
    >> pat.fixeEtatCiv (etatCiv)
  - ! creeAffil
    >> affil.birth (pat, oA, nrTit, typeAff)
  - !+ end

BEHAVIOUR



5. OBJET INTRO_TSF

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - litOrig, litDest : LIT

- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")

- **Actions**
  - * start
  - ! nrDosD (pat : PAT)
    >> NR_DOS_D.open (pat)
    ⇒ pat := nrDosD.pat
  - ! tsfD (litOrig, litDest : LIT, med : MED)
    >> TSF_D.open
    ⇒ litOrig := tsfD.litOrig
    ⇒ litDest := tsfD.litDest
    ⇒ med := tsfD.med
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPasLitOrig
    IS putError (code)
    := code = TSF_NOT_IN_LIT_OR
  - + putErrorLitDestOcc
    IS putError (code)
    := code = TSF_LIT_DEST_OCC
  - !+ putErrorPatPasMal
    IS putError (code)
    := code = TSF_PAT_NOT_MAL
  - ! inLitOrig
  - ! inUrgence
  - ! notInUrgence
  - ! admPost
    >> ADM_POST.start (pat)
  - ! ddeMvt
    >> dde_mvt.birth (pat, litDest, litOrig, litDest.serDeLit.priorité, "T", putErrorLitDestOcc, creeMvtTsf)
  - creeMvtTsf
    >> mvt_tsf.birth (litOrig, litDest, pat, med, TODAY(), NOW())
    >> allocation.déplacerPrescr (litOrig.serDeLit, litDest.serDeLit, pat)
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = TSF_OK

BEHAVIOUR

Préconditions

A: pat.patMalade=TRUE
B: pat.patDansLit(litOrig)=TRUE
~~C: litDest.litOccupé=TRUE~~
D: pat.patDansSer(urgences)=TRUE

6. OBJET INTRO_SORTIE

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - litOrig : LIT
  - i : NAT
  - prescr's : LIST of PRESCR_SOIN

- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")

- **Actions**
  - * start
  - ! nrDosD (pat : PAT)
    >> NR_DOS_D.open (pat)
    ⇒ pat := nrDosD.pat
  - ! sortieD (litOrig : LIT, med : MED)
    >> SORTIE_D.open
    ⇒ litOrig := sortieD.litOrig
    ⇒ med := sortieD.med
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPasLitOrig
    IS putError (code)
    := code = OUT_NOT_IN_LIT_OR
  - !+ putErrorPatPasMal
    IS putError (code)
    := code = OUT_PAT_NOT_MAL
  - ! inLitOrig

- ! inUrgence
  ⇒ i := 1
  ⇒ prescr's := ALL [pS : PRESCR_SOIN | pS in pat.subit AND pS.enAttente=TRUE]
- ! notInUrgence
- ! admPost
  >> ADM_POST.start (pat)
- ! annulePrescr
  >> allocation.prescrAnnulée(prescr's[i])
- ! incI
  ⇒ i := i + 1
- ! ddeMvt
  >> dde_mvt.birth (pat, litOrig, litOrig.serDeLit.priorité, "S", creeMvtSortie)
- creeMvtSortie
  >> mvt_sor.birth (litOrig, pat, med, TODAY(), NOW())
- !+ putMsg (code : MSG_CODE_DT)
  >> MSG_D.open (code)
  := code = OUT_OK

BEHAVIOUR

Préconditions

A: pat.patMalade=TRUE
B: pat.patDansLit(litOrig)=TRUE
C: pat.patDansSer(urgence)=TRUE
D: i <= length(prescr's)

7. OBJET PRESCR_SOIN

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

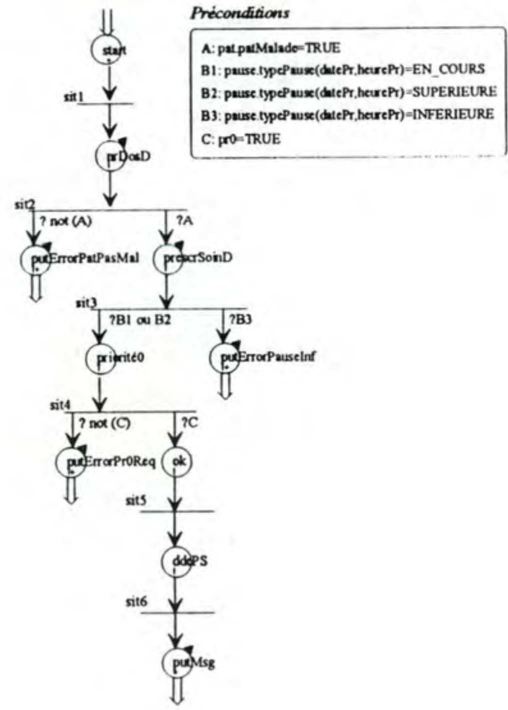
BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - prest : PREST (= PREST_INT ∪ PREST_EXT)
  - datePr : DATE
  - heurePr : TIME
  - pr0 : BOOL

- **Derivations**

- **Actions**
  - * start
  - ! nrDosD (pat : PAT)
    >> NR_DOS_D.open (pat)
    ⇒ pat := nrDosD.pat
  - ! prescrSoinD (med, prest, date, time)
    >> PRESCR_SOIN_D.open
    ⇒ med := prescrSoinD.med
    ⇒ prest := prescrSoinD.prest
    ⇒ datePr := prescrSoinD.date
    ⇒ heurePr := prescrSoinD.time
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPauseInf
    IS putError (code)
    := code = PR_S_PAUSE_INF
  - !+ putErrorPr0Req
    IS putError (code)
    := code = PR_S_PR0_REQ
  - !+ putErrorPatPasMal
    IS putError (code)
    := code = PR_S_PAT_NOT_MAL
  - ! ddePS
    >> dde_ps.birth (pat, med, prest, datePr, heurePr, priorité, putMsg)
  - ! priorité0
  - ! ok
  - ! notInUrgence
  - + putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = PR_S_OK

BEHAVIOUR



8. OBJET PREST_PRESCR

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

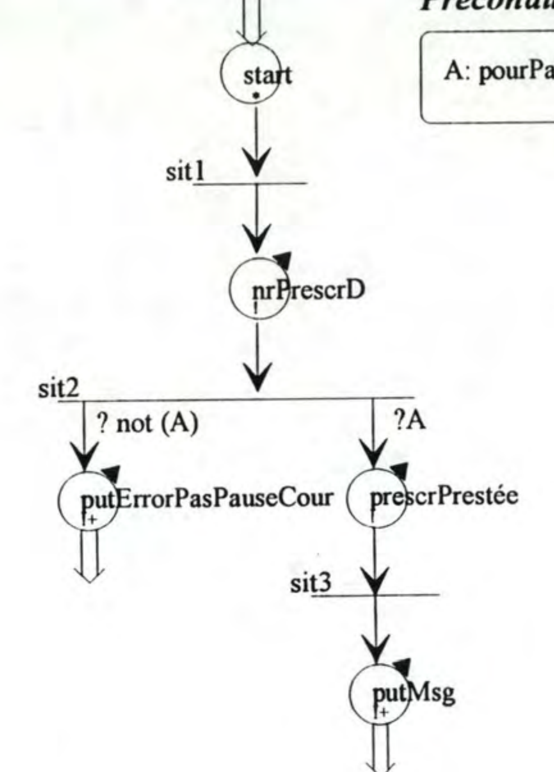
BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - prescr : PRESCR_SOIN
  - pourPauseCour : BOOL

- **Derivations**

- **Actions**
  - * start
  - ! nrPrescrD (pres : PRESCR_SOIN)
    >> NR_PRESCR_D.open (pres)
    ⇒ prescr := nrPrescrD.pres
  - ! pauseCour
    ⇒ pouPauseCour :=
    pause.typePause(prescr.date,prescr.heure)=EN_COURS
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPasPauseCour
    IS putError (code)
    := code = PREST_PR_NOT_PAUSE_COUR
  - ! prescrPrestée
    >> allocation.prescrPrestée (prescr)
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = PREST_PR_OK

BEHAVIOUR



Préconditions

A: pourPauseCour=TRUE

9. OBJET ANNUL_PRESCR

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

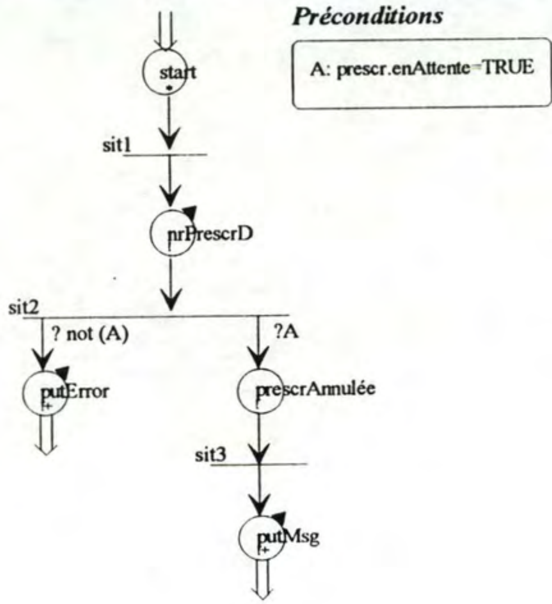
BODY

- **Attributes**
  - prescr : PRESCR_SOIN

- **Derivations**

- **Actions**
  - * start
  - ! nrPrescrD (pres : PRESCR_SOIN)
    >> NR_PRESCR_D.open (pres)
    ⇒ prescr := nrPrescrD.pres
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
    := code = ANNUL_PRESCR_NOT_WAITING
  - ! prescrAnnulée
    >> allocation.prescrAnnulée (prescr)
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = ANNUL_PRESCR_OK

BEHAVIOUR



10. OBJET RAPPEL_PREST

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

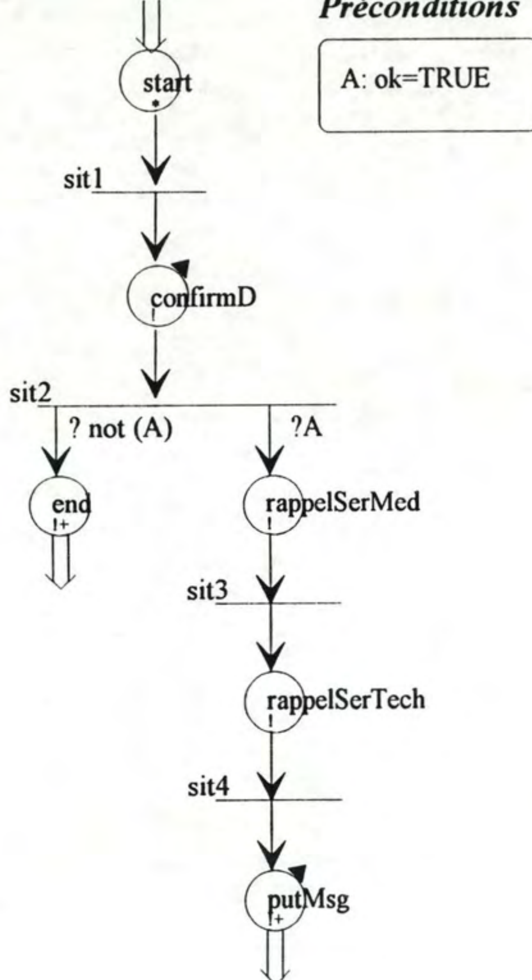
BODY

- **Attributes**
  - ok : BOOL

- **Derivations**

- **Actions**
  - * start
  - ! confirmD (ok :BOOL)
    >> CONFIRM_D.open (ok)
    ⇒ ok := confirmD.ok
  - ! rappelSerMed
    >> RAP_SER_MED.start
  - ! rappelSerTech
    >> RAP_SER_TECH.start
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = RAPPEL_PREST_OK
  - !+ end

BEHAVIOUR

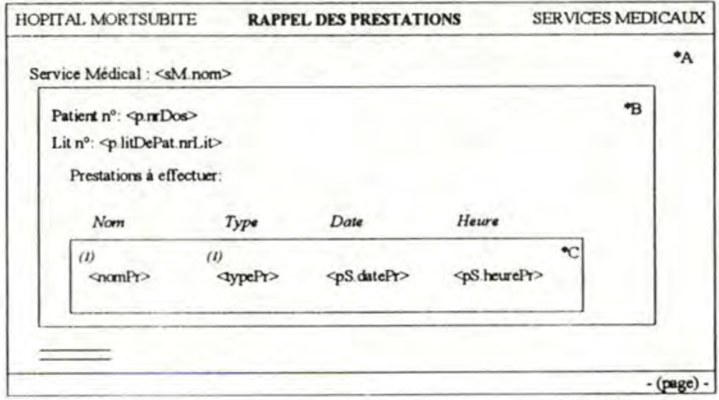


Préconditions

A: ok=TRUE

Afin de gagner du temps, nous allons vous exposer ici la spécification des deux objets de type rapport réalisée à l'aide de l'éditeur de rapport.

RAPPORT RAP_SER_MED



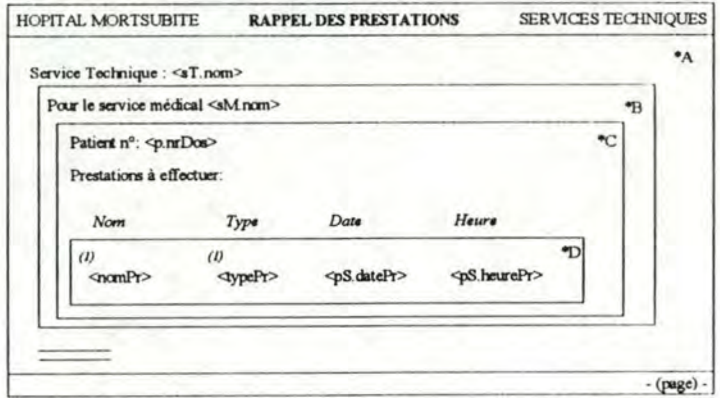
(1): pS.pourPrest +...

A : ALL [sM : SER_MED]
B : ALL [p : PAT | p.patDansSer(sM)=TRUE]
C : ALL [pS : PRESCR_SOIN |
  pS.subitePar=p AND
  CLASS_OF(pS.pourPrest)="PREST_INT" AND
  pS.enAttente=TRUE AND
  pause.typePause(pS.datePr,pS.heurePr)=EN_COURS]

La logique de ce rapport est la suivante :

```
∀ service médical
  ∀ patient de ce service médical
       ∀ prescription de soin de ce patient
          "en attente",
          prévue pour la pause courante et
          prescrivant une prestation interne
⇒ imprimer les nom, type, date et heure de cette
prestation.
```

RAPPORT RAP_SER_TECH



(1): pS.pourPrest +...

A : ALL [sT : SER_TECH]
B : ALL [sM : SER_MED]
C : ALL [p : PAT | p.patDansSer(sM)=TRUE]
D : ALL [pS : PRESCR_SOIN |
  pS.subitePar=p AND
  CLASS_OF(pS.pourPrest)="PREST_EXT" AND
  pS.enAttente=TRUE AND
  pause.typePause(pS.datePr,pS.heurePr)=EN_COURS]

La logique de ce rapport est la suivante :

```
∀ service technique
  ∀ service médical
       ∀ patient de ce service médical
          ∀ prescription de soin de ce patient
             "en attente",
             prévue pour la pause courante et
             prescrivant une prestation externe
             effectuée par ce service technique
⇒ imprimer les nom, type, date et heure de cette
prestation.
```

11. OBJET DDE_INFO_MAL

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

BODY

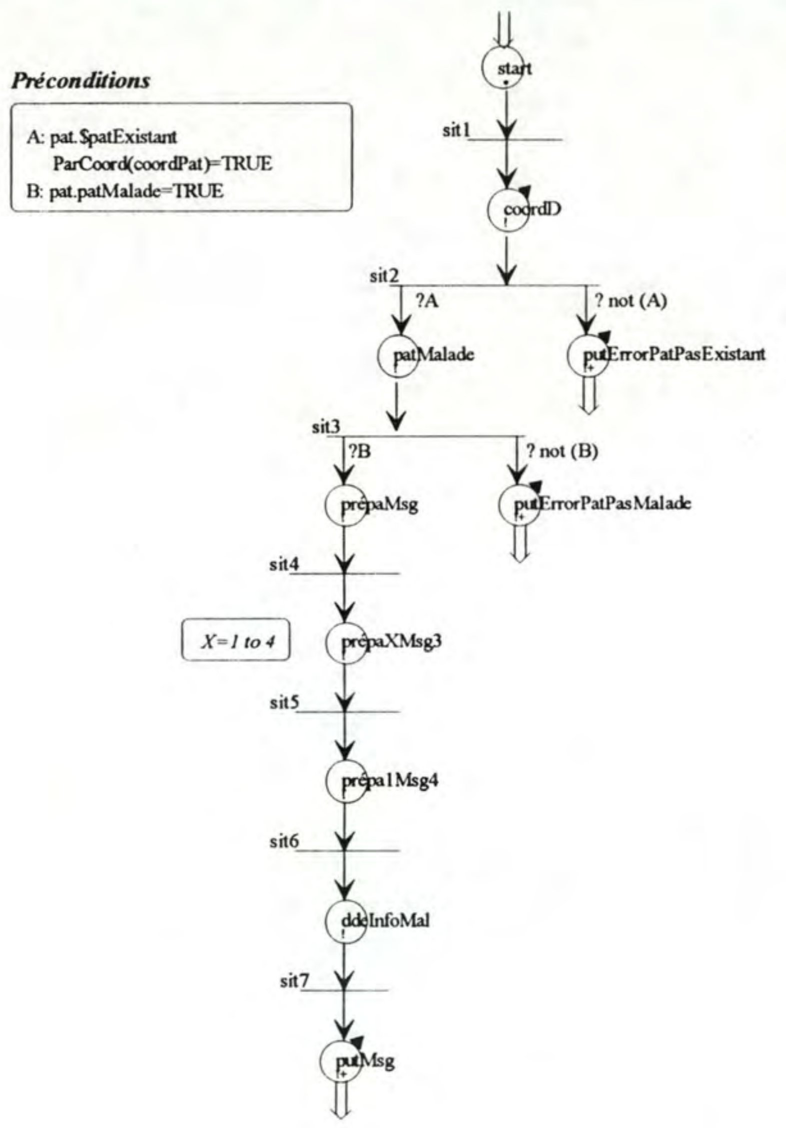
- **Attributes**
  - coordPat : COORD_DT
  - nrDos : NR_DT
  - pat : PAT
  - listeMvtEnt, listeMvt : LIST of MVT
  - listePrescr : LIST of PRESCR_SOIN
  - m : MVT
  - dateD, dateF : DATE
  - heureD, heureF : TIME

- **Derivations**
  - urgences : SER_MED
    := ser_med.$serviceMedical("Urgences")

- **Actions**
  - * start
  - ! coordD (coordP : COORD_DT)
    >> COORD_D.open (coordP)
    ⇒ coordPat := coordD.coordP
  - ! prepaMsg
  - ! prepa1Msg3
    := listeMvtEnt = ALL [m : MVT |
    m in pat.accomplit AND
    CLASS_OF(m) = "MVT_ENT"]
  - ! prepa2Msg3
    := m = LAST (listeMvtEnt SORTED ON (dateMvt, heureMvt))
  - ! prepa3Msg3
    := dateD = m.dateMvt
    := dateF = TODAY()
    := heureD = m.heureMvt
    := heureF = NOW()
  - ! prepa4Msg3
    := listeMvt = pat.listeMvtDates[1] (dateD, heureD, dateF, heureF)
  - ! prepa1Msg4
    := listePrescr = pat.listePrescrDates[2] (dateD, heureD, dateF, heureF)
  - ! patMalade
    ⇒ pat := pat.$patParCoord(coordPat)
  - ! ddeInfoMal
    >> RAP_INFO_MAL.start (pat, listeMvt, listePrescr)
  - !+ putError (code : ERR_CODE_DT)
    >> ERR_D.open (code)
  - !+ putErrorPatPasMal
    IS putError (code)
    := code = DDE_INFO_PAT_NOT_MAL
  - !+ putErrorPatPasExistant
    IS putError (code)
    := code = DDE_INFO_PAT_NOT_EXIST
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = DDE_INFO_OK

1 Observation de l'objet PAT qui renvoie la liste des mouvements que ce patient a effectués entre les deux dates/heures.

2 Observation de l'objet PAT qui renvoie la liste des prescriptions de soin effectuées que ce patient a subies entre les deux dates/heures.



RAPPORT RAP_INFO_MAL

HOPITAL MORTSUBITE   **INFORMATION MALADE**   Nr dossier:<pat.nrDos>

Nom : <pat.nom> Prénom : <pat.prenom> Date naiss. : <pat.dateNaiss>

Adresse : <pat.adr> Téléphone : <pat.tel> Etat civil : <pat.etatCivil> Sexe : <pat.sexe> ?A

Organisme Assureur :

Nr OA : <pat.oADePat.nrOA> Nr titulaire : <pat.estAffilié.nrTit>

Adresse OA : <pat.oADePat.adrOA> Type affil : <pat.estAffilié.typeAffil>

Nom OA : <pat.oADePat.nomOA>

Liste de mouvements:

Date *Heure* *Type*

Médecin *Lit d'origine* *Lit de destination*

<m.dateMvt> <m.heureMvt> <m.typeMvt> *B

<m.demandéPar.nrMed> <m.de.nrLit> ?C <m.vers.nrLit> ?D

Liste de prestations:

Date *Heure* *Numéro*

Code *Médecin*

<pS.datePr> <pS.heurePr> <pS.nrPr> *E

<pS.pourPrest.codePrest> <pS.prescritePar.nrMed>

- (page) -

A : pat.patDansSer(Urgence)=FALSE

B : ALL [m : MVT | m in listeMvt]

C : CLASS_OF(m) <> MVT_ENT

D : CLASS_OF(m) <> MVT_SOR

E : ALL [pS : PRESCR_SOIN | pS in listePrescr]

12. OBJET PROD_FACT

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start

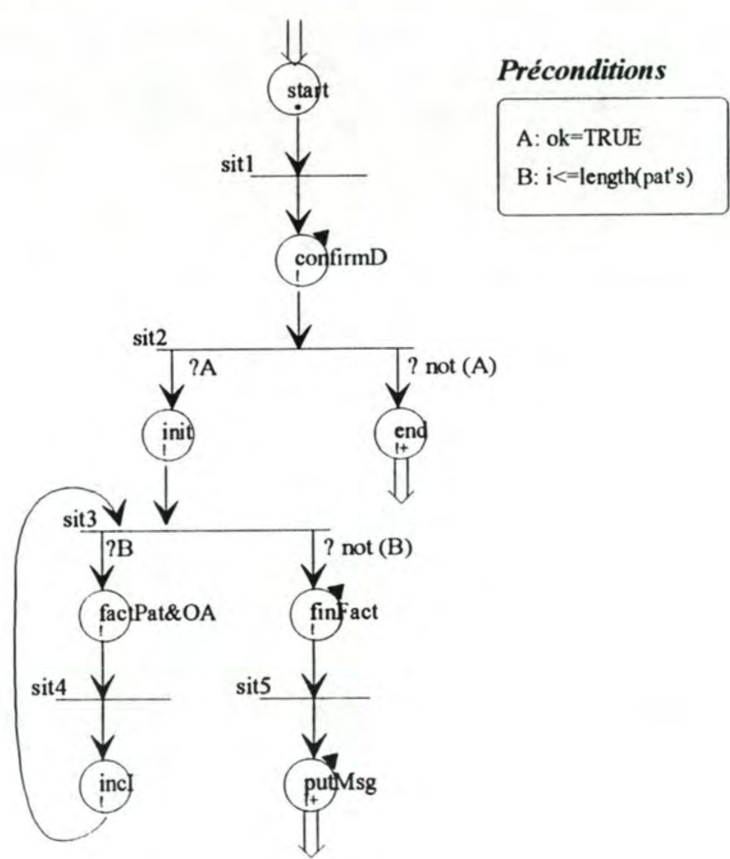
BODY

- **Attributes**
  - pat : PAT
  - pat's : LIST of PAT
  - dateD, dateF : DATE
  - heureD, heureF : TIME
  - i : INTEGER
  - ok : BOOL

- **Derivations**
  - hop : HOP
    hop := hop.$hôpital("MortSubite")

- **Actions**
  - * start
  - ! confirmD (ok :BOOL)
    >> CONFIRM_D.open (ok)
    ⇒ ok := confirmD.ok
  - ! init
    := dateD = HOP.dateDerFact
    := dateF = TODAY()
    := heureD, heureF = NOW()
    := pat's = ALL[p:PAT]
    := i = 1
  - ! factPat&OA
    := pat = pat's[i]
    >> FACT_PAT_OA.start(dateD,heureD,dateF,heureF,pat)
  - ! incI
    := i = i + 1
  - !+ end
  - ! finFact
    >> hop.fixeDateDerFact(TODAY())
  - !+ putMsg (code : MSG_CODE_DT)
    >> MSG_D.open (code)
    := code = PROD_FACT_OK

BEHAVIOUR



13. OBJET FACT_PAT_OA[3]

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * start (dateD,dateF : DATE, heureD,heureF : TIME, pat : PAT)

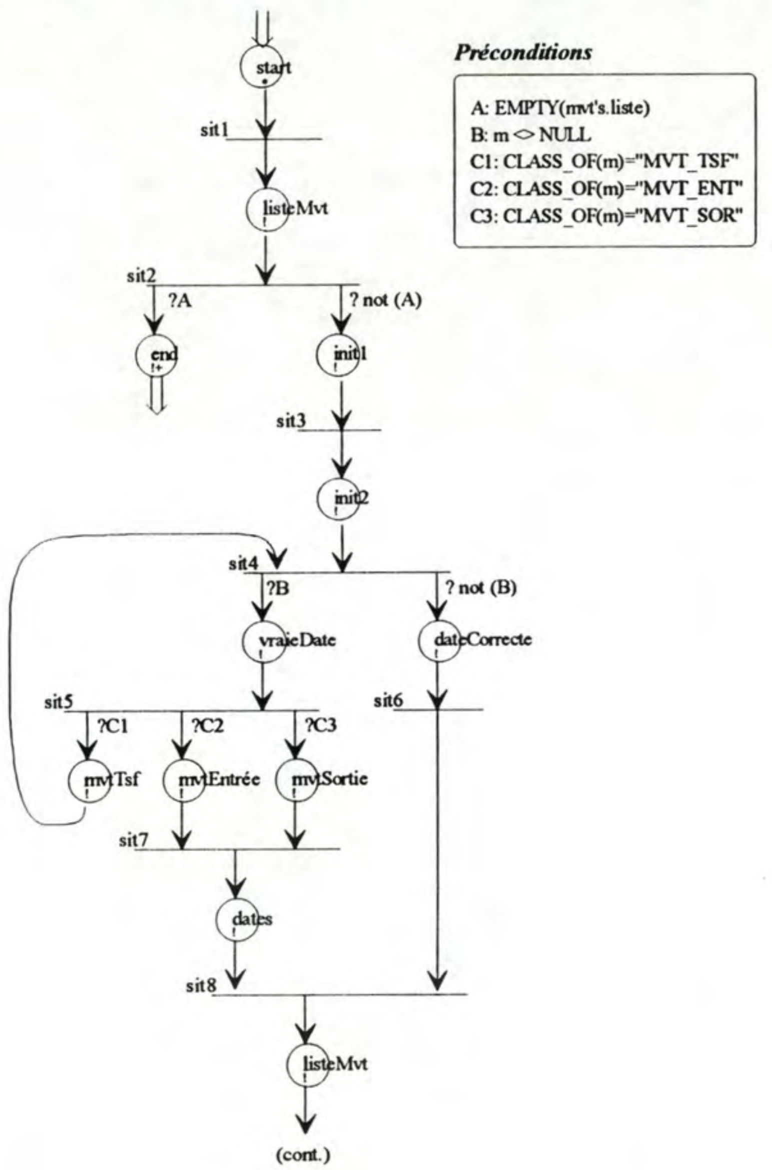
BODY

- **Attributes**
  - pat : PAT
  - mvt's : CP[liste : LIST of MVT, nbHosp : INT]
  - dateD, dateF : DATE
  - heureD, heureF : TIME
  - m : MVT
  - sej's : LIST of CP[mv1,mv2 : MVT]
  - prescr's : LIST of PRESCR_SOIN
  - dest : DESTINATAIRE_DT
  - hosp : LIST of MVT
  - i, j : INT

- **Derivations**

- **Actions**
  - * start (dateD,heureD,dateF,heureF,pat)
    ⇒ dateD := start.dateD
    ⇒ ...
    ⇒ pat := start.pat
  - ! listeMvt
    := mvt's = pat.listeMvtDates(dateD,heureD,dateF,heureF)
  - ! init1
    := m = FIRST(mvt's.liste)
  - ! init2
    := m = m.mvtPréc
  - ! vraieDate
  - ! dateCorrecte
  - ! mvtEntrée
  - ! mvtSortie
    := m = FIRST(mvt's.liste)
  - ! mvtTsf
    := m = m.mvtPréc
  - ! dates
    := dateD = m.dateMvt
    := heureD = m.heureMvt
  - ! incI
    := i = i + 1
  - !+ end
  - ! factHosp's
    := j = 1
  - ! factHosp
    := m = mvt's.liste[j]
  - ! incJ
    := j = j + 1
  - ! mvtSuivant
    := m = mvt's.liste[j]
  - ! extraitHosp
    := hosp = EXTRACT(mvt's.liste[1..j]) {copie puis efface}
  - ! séjours
    >> SEJOUR.sejour(hosp, sejs)
    ⇒ sej's := sejour.sejs
  - ! prescr's
    := pat.listePrescrDates(FIRST(sej's).mv1.dateMvt/heureMvt, LAST(sej's).mv2.dateMvt/heureMvt)
  - ! fact
    >> RAP_FACT.start(pat,sej's,prescr's,dest)
  - ! factPat
    IS RAP_FACT
    := dest = PATIENT
  - ! factOA
    IS RAP_FACT
    := dest = OA

[3] Cet objet peut être considéré comme un Inner Object.

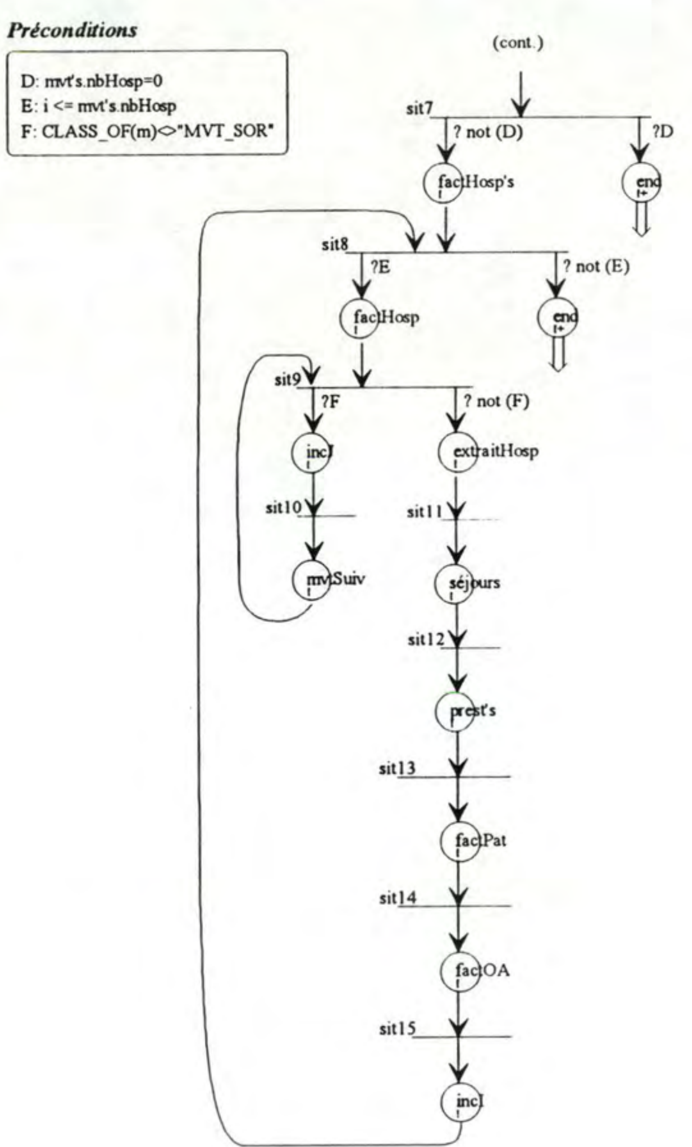


Préconditions

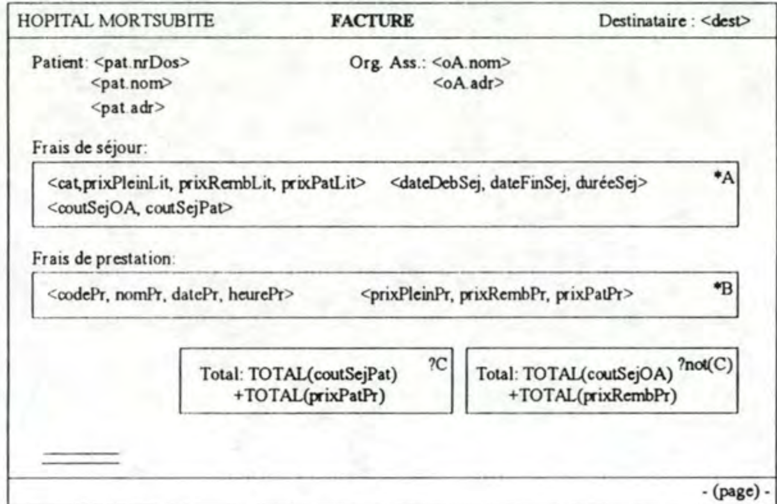
A: EMPTY(mvt's.liste)
B: m <> NULL
C1: CLASS_OF(m)="MVT_TSF"
C2: CLASS_OF(m)="MVT_ENT"
C3: CLASS_OF(m)="MVT_SOR"

Préconditions

D: mvt's.nbHosp=0
E: i <= mvt's.nbHosp
F: CLASS_OF(m)<>"MVT_SOR"



Rapport RAP_FACT



A : ALL [sej : CP[mv1,mv2:MVT] | sej in sej's]
B : ALL [pS : PRESCR_SOIN | pS in prescr's]
C : dest = PATIENT

En posant...

```
oA = pat.oADePat
l = sej.mv1.de (lit de référence pour la facturation)
cat = l.catégorieLit

prixPleinLit = cat.prixPlein
rembCh = remb_ch.$remboursementChambre(cat,oA)
prixRembLit = rembCh.prixRemb
prixPatLit = prixPleinLit - prixRembLit
dateDebSej = sej.mv1.dateMvt
dateFinSej = sej.mv2.dateMvt
duréeSej = dateFinSej - dateDebSej
coutSejOA = prixRembLit * duréeSej
coutSejPat = prixPatLit * duréeSej

prest = pS.pourPrest
```

```
codePr = prest.codePr
nomPr = prest.nomPr
datePr/heurePr = pS.datePr/heurePr
rembPr = remb_pr.$remboursementPrestation(prest,oA)
prixPleinPr = prest.prixPleinPr
prixRembPr = rembPr.prixRemb
prixPatPr = prixPleinPr - prixRembPr
```

14. OBJET ALLOCATION

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - allocNrDos (coord : COORD_DT)
  - allocPrescr (pat : PAT, med : MED, prest : PREST, date : DATE, heure : TIME, pr0 : BOOL)
  - deplacerPrescr (ser1, ser2 : SER_MED, pat)
  - prescrPrestée (prescr : PRESCR_SOIN)
  - prescrAnnulée (prescr : PRESCR_SOIN)

BODY

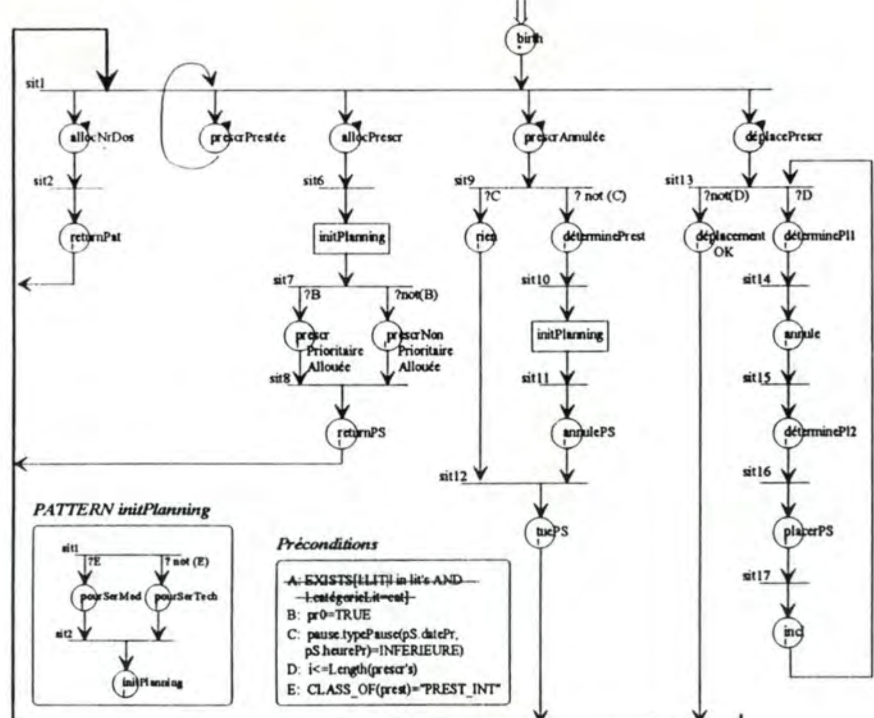
- **Attributes**
  - ser, ser1, ser2 : SER_MED
  - pat : PAT
  - med : MED
  - prest : PREST
  - date, heure : DATE/TIME
  - pr0 : BOOL
  - planning, pl1, pl2 : PLANNING
  - pS : PRESCR_SOIN
  - prescr's : LIST of PRESCR_SOIN
  - i : NAT

- **Derivations**
  - priorité (ser : SER) : NAT
    := {attribue la priorité déterminée}

- **Actions**
  - !* birth
  - allocNrDos (coord)
    ⇒ coord := allocNrDos.coord
  - ! returnPat
    {>> rappelle où on l'a appelé avec la référence de pat grâce au mécanisme de stored action : cf. problème en début de chapitre}
  - allocPrescr (pat, med, prest, date, heure, pr0)
    ⇒ pat := allocPrescr.pat
    ⇒ med := allocPrescr.med
    ⇒ date := allocPrescr.date
    ⇒ heure := allocPrescr.heure
    ⇒ pr0 := allocPrescr.pr0
    >> pS.birth (pat, med, prest, date, heure) {juste pour avoir la référence}
  - ! pourSerMed
    ⇒ ser := ONE [sM : SER_MED | pat.patDansSer(sM)=TRUE]
  - ! pourSerTech
    ⇒ ser := prest.serDePrest
  - ! initPlanning
    ⇒ planning.$planning (ser, prest)
  - ! prescrPrioritaireAllouée
    >> planning.prescrPriorAllouée (pS)
  - ! prescrNonPrioritaireAllouée
    >> planning.placerPS (pS, priorité(ser), pause.nrPause (date, heure))
  - returnPS
    {>> rappelle où on l'a appelé avec la référence de pS}
  - deplacerPrescr (ser1, ser2, pat)
    ⇒ ser1, ser2 := deplacerPrescr.ser1, ser2
    ⇒ pat := deplacerPrescr.pat
    ⇒ prescr's := ALL [pS : PRESCR_SOIN | pS in pat.subit AND CLASS_OF (pS.pourPrest) = "PREST_INT"]
    ⇒ i := 1
  - ! déterminerPl1
    ⇒ pl1 := planning.$planning (prescr's[i].pourPrest, ser1)
  - ! déterminerPl2
    ⇒ pl2 := planning.$planning (prescr's[i].pourPrest, ser2)
  - ! annule
    >> planning.annulePrescr (prescr's[i], pause.nrPause(prescr's[i].date, prescr's[i].heure))
  - ! placerPS
    >> planning.placerPS (prescr[i], priorité(ser2), pause.nrPause(prescr's[i].date, prescr's[i].heure))
  - ! incI
    ⇒ i := i + 1
  - prescrPrestée (prescr : PRESCR_SOIN)
    >> prescr.fixeEtat (PRESTEE)

- prescrAnnulée (prescr : PRESCR_SOIN)
  ⇒ pS := prescrAnnulée.prescr
- ! rien
- ! déplacementOK
- ! déterminePrest
  ⇒ prest := pS.pourPrest
  ⇒ pat := pS.prescritePour
- ! annulePS
  >> planning.annulePrescr (pS, pause.nrPause(pS.date, pS.heure))
- ! tuePS
  >> pS.death

BEHAVIOUR



15. OBJET PLANNING

Rem : kind-of TBL
Rem : PL_PAUSE_DT =
CP [nbMax : NAT,
nbEffectif : NAT,
nbSurcharge : NAT,
listePS : LIST of CP [pS : PRESCR_SOIN,
prior : NAT]
]

INTERFACE

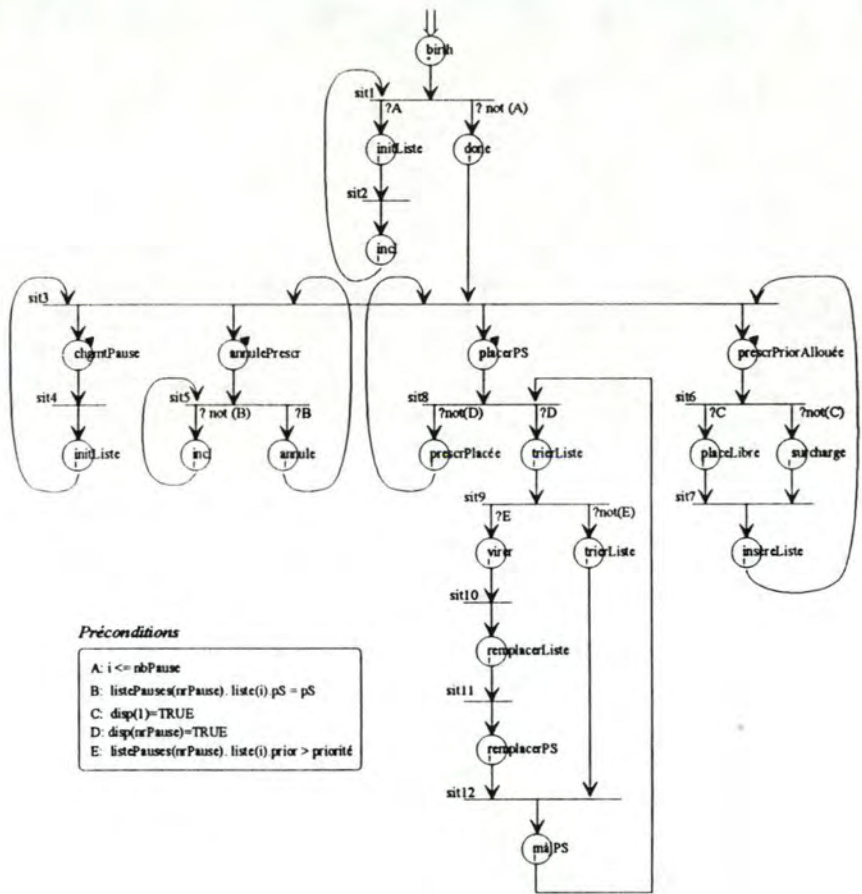
- **Observations**
- **Keys**
  - **$planning (ser : SER, prest : PREST)**
- **Events**
  - prescrPriorAllouée (pS : PRESCR_SOIN)
  - placerPS (pS : PRESCR_SOIN, priorité : NAT, nrPause : NAT)
    ? > 0
    ? > 1
  - chgmtPause
  - * birth (ser : SER, prest : PREST, nbPauses : NAT)
  - annulePrescr (pS : PRESCR_SOIN, nrPause : NAT)

BODY

- **Attributes**
  - ser : SER_MED
  - prest : PREST
  - listePauses : PL_PAUSE_DT
  - nbPauses : NAT
  - nrPause : NAT
  - priorité, prTemp : NAT
  - pS, pSTemp : PRESCR_SOIN
  - i : NAT
  - eff : EFF_PREST_INT ∪ EFF_PREST_EXT

- **Derivations**
  - disp (i : NAT) : BOOL
    := listePauses[i].nbSurcharge = 0
  - $planning (ser, prest)
    := ONE[pl : PLANNING | pl.ser = ser AND pl.prest = prest]

- **Actions**
  - * birth (ser, prest, nbPauses)
    ⇒ nbPauses := birth.nbPauses
    ⇒ ser := birth.ser
    ⇒ prest := birth.prest
    {? CLASS_OF (prest) = "PREST_INT"} ⇒ eff :=
    eff_prest_int.$effectuePrestationInterne(ser, prest)
    {? CLASS_OF (prest) = "PREST_EXT"} ⇒ eff := prest.prExtEff
    ⇒ i := 1
  - ! initListe
    ⇒ APPEND (listePauses, prest.capMaxi*eff.nbUnitésExec ∪ 0 ∪ 0 ∪
    EMPTY())
  - ! incl
    ⇒ i := i + 1
  - ! done
  - prescrPriorAllouée (pS)
    ⇒ pS := prescrPriorAllouée.pS
  - ! placeLibre
    ⇒ listePauses[1].nbEffectif := listePauses[1].nbEffectif + 1
  - ! surcharge
    ⇒ listePauses[1].nbSurcharge := listePauses[1].nbSurcharge + 1
  - ! insèreListe
    ⇒ APPEND (listePauses[1].liste, pS ∪ 0)
  - placerPS (pS, priorité, nrPause)
    ⇒ pS := placerPS.pS
    ⇒ priorité := placerPS.priorité
    ⇒ nrPause := placerPS.nrPause
  - ! prescrPlacée
    ⇒ APPEND (listePauses[nrPause].liste, pS ∪ priorité)
    ⇒ listePauses[nrPause].nbEffectif := listePauses[nrPause].nbEffectif + 1
  - ! trierListe
    ⇒ listePauses[nrPause].liste := listePauses[nrPause].liste SORTED ON
    prior
    ⇒ i := length (listePauses[nrPause].liste)
  - ! virer
    ⇒ pSTemp := listePauses[nrPause].liste[i].pS
    ⇒ prTemp := listePauses[nrPause].liste[i].prior
  - ! remplacerListe
    ⇒ listePauses[nrPause].liste[i].pS := pS
    ⇒ listePauses[nrPause].liste[i].prior := priorité
  - ! remplacerPS
    ⇒ pS := pSTemp
    ⇒ priorité := prTemp - 1
  - ! màjPS
    >> pS.fixeDateHeure (pause.pausePlusUn(pS.datePr, pS.heurePr))
    ⇒ nrPause := nrPause + 1
  - ! pauseSuiv
  - annulePrescr (pS, nrPause)
    ⇒ pS := annulePrescr.pS
    ⇒ nrPause := annulePrescr.nrPause
    ⇒ i := 1
  - ! annule
    ⇒ ERASE (listePauses[nrPause].liste[i]
    {? listePauses[nrPause].nbSurcharge > 0} ⇒
    listePauses[nrPause].nbSurcharge := listePauses[nrPause].nbSurcharge - 1
    {? listePauses[nrPause].nbSurcharge = 0} ⇒
    listePauses[nrPause].nbEffectif := listePauses[nrPause].nbEffectif - 1
  - ! chgmtPause
    ⇒ ERASE (listePauses[1])



16. OBJET SEJOUR

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * séjour (hosp : LIST of MVT)

BODY

- **Attributes**
  - sej's : LIST of CP[mv1, mv2 : MVT]
  - hosp : LIST of MVT
  - i, j : NAT
  - trouvé : BOOL
  - date, heure : DATE/TIME

- **Derivations**

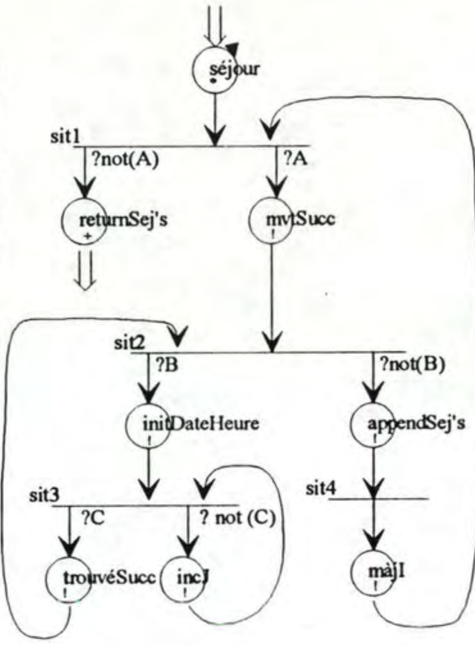
- **Actions**
  - * séjour (hosp)
    ⇒ hosp := séjour.hosp
    ⇒ i := 1
    ⇒ j := 1
  - ! mvtSucc
    ⇒ trouvé := FALSE
    ⇒ j := j + 1
  - ! initDateHeure
    ⇒ date := hosp[i].heureMvt + 1 {+24h}
    ⇒ heure := hosp[i].dateMvt
  - trouvéSucc
    ⇒ trouvé := TRUE
  - ! incJ
    ⇒ j := j + 1
  - ! appendSej's
    ⇒ APPEND (sej's, hosp[i] ∪ hosp[j])
  - ! màjI
    ⇒ i := j

- !+ returnSej's
  {>> rappelle où on l'a appelé avec sej's}

BEHAVIOUR

Préconditions

A: j < length(hosp)
B: trouvé=FALSE
C: (hosp(i).dateMvt >= date AND
hosp(i).heureMvt >= heure) OR
j = length(hosp)



17. OBJET PAUSE

Rem : objet *single*

INTERFACE

- **Observations**
  - typePause (date : DATE, heure : TIME) : TYPE_PAUSE_DT
  - nrPause (date : DATE, heure : TIME) : NAT
    ? typePause (date, heure) <> INFERIEURE
  - pausePlusUn (date : DATE, heure : TIME) : CP[date : DATE, heure : TIME]

- **Keys**
- **Events**

BODY

- **Attributes**
  - debPauseCour, finPauseCour : CP[dateP : DATE, heureP : TIME]
  - plannings : LIST of PLANNING

- **Derivations**
  - typePause (date, heure)
    {? (date < debPauseCour.dateP) and (heure < debPauseCour.heureP)}
    ⇒ INFERIEURE
    {? (date > debPauseCour.dateP) and (heure > debPauseCour.heureP)}
    ⇒ SUPERIEURE
    {? [(date ≥ debPauseCour.dateP) and (heure ≥ debPauseCour.heureP)] and [(date ≤ debPauseCour.dateP) and (heure ≤ debPauseCour.heureP)]}
    ⇒ EN_COURS
  - nrPause (date, heure)
    ⇒ [(date - finPauseCour.dateP)*24 + (heure - finPauseCour.heureP)] + 1

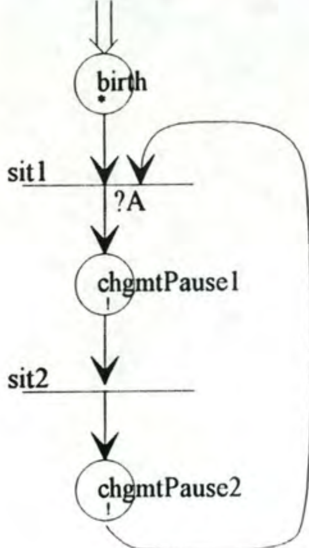
- pausePlusUn (date, heure)
  {? heure > 16h00}
  ⇒ heure := heure - 16   {= heure + 8 - 24}
  ⇒ date := date + 1
  {? heure ≤ 16h00}
  ⇒ heure := heure + 8
  ⇒ date := date

- **Actions**
  - ! birth
    {? (06h00 ≤ NOW() < 14h00}
    ⇒ debPauseCour.dateP := TODAY()
    ⇒ debPauseCour.heureP := 06h00
    ⇒ finPauseCour.dateP := TODAY()
    ⇒ finPauseCour.heureP := 14h00
    {? (14h00 ≤ NOW() < 22h00}
    ⇒ debPauseCour.dateP := TODAY()
    ⇒ debPauseCour.heureP := 14h00
    ⇒ finPauseCour.dateP := TODAY()
    ⇒ finPauseCour.heureP := 22h00
    {? (22h00 ≤ NOW() < 00h00}
    ⇒ debPauseCour.dateP := TODAY()
    ⇒ debPauseCour.heureP := 22h00
    ⇒ finPauseCour.dateP := TODAY()+1
    ⇒ finPauseCour.heureP := 06h00
    {? (0h00 ≤ NOW() < 06h00}
    ⇒ debPauseCour.dateP := TODAY()-1
    ⇒ debPauseCour.heureP := 22h00
    ⇒ finPauseCour.dateP := TODAY()
    ⇒ finPauseCour.heureP := 06h00
  - ! chgmtPause1
    ⇒ debPauseCour.dateP := pausePlusUn (debPauseCour.dateP, .heureP)
    ⇒ plannings := ALL[pl : PLANNING | TRUE]
  - ! chgmtPause2
    ⇒ finPauseCour.dateP := pausePlusUn (debPauseCour.dateP, .heureP)
    >> plannings.chgmtPause   {*multicast*}

BEHAVIOUR

Préconditions

A: NOW()=06h00 OR
NOW()=14h00 OR
NOW()=22h00



18. OPTIMISATION

18.1. ALLOCATION-LIT

18.1.1. La classe d'objets DDE-MVT

La classe d'objets DDE-MVT contient les demandes des différentes fonctionnalités qui souhaitent effectuer un mouvement du type Entrée, Transfert ou Sortie. Une instance de la classe sera créée pour chaque demande émise et sera détruite parce qu'elle aura été soit satisfaite, soit refusée.

La spécification de la classe d'objet est la suivante :

OBJET DDE-MVT

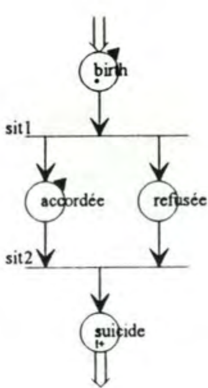
INTERFACE

- **Observations**
  - pat : PAT
  - litOrig, litDest : LIT
  - priorité : NAT
  - type : MVT-DT
  - cat : CAT
  - ser : SER_MED
  - poids : NAT
- **Keys**
- **Events**
  - * birth (pat : PAT, litOrig, litDest : LIT, prior : NAT, type : MVT-DT, acceptEv, refusEv : EVENT)

BODY

- **Attributes**
  - pat : PAT
  - priorité : NAT
  - litOrig, litDest : LIT
  - type : MVT_DT
  - cat : CAT
  - ser : SER_MED
  - acceptEv, refusEv : EVENT
- **Derivations**
  - poids : NAT
    {? priorité = 0}
    ⇒ ∞
    {? priorité = 1}
    ⇒ 6
    {? priorité = 2}
    ⇒ 3
    {? priorité = 3}
    ⇒ 1
- **Actions**
  - * birth (pat : PAT, litOrig, litDest : LIT, prior : NAT, type : MVT-DT, acceptEv, refusEv : EVENT)
    ⇒ pat := birth.pat
    ⇒ ...
  - accordée (litDest : LIT)
    ⇒ litDest := accordée.litDest
    >> acceptEv with litDest
  - refusée
    >> refusEv
  - !+ suicide

BEHAVIOUR



18.1.2. Implémentation

Nous allons exposer l'implémentation en OBLOGlight de la solution au problème d'optimisation de l'allocation des lits. Pour des raisons de lisibilité et pour des raisons techniques, cette implémentation n'aura pas la forme habituelle. Elle sera composée d'une suite d'instructions aussi proches que possible du langage OBLOG mais sans intégrer tout dans un seul diagramme de comportement auquel serait associée la série des déclarations d'actions.

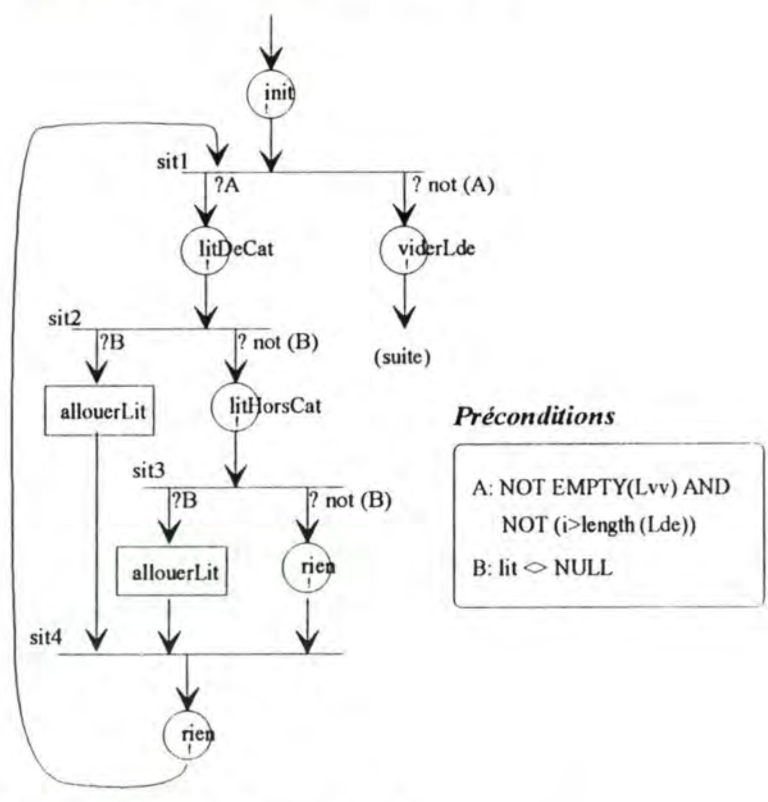
Cette suite d'instructions sera parfois complétée de quelques diagrammes comportementaux pour clarifier l'exposé. Nous avons fait en sorte que ces instructions soient aisément traduisibles en OBLOGlight.

Nous observerons une séquence numérotée qui respecte celle exposée dans la solution : ce que nous annoncions au point 3 (par exemple) de la solution est implémenté au point 3. Si nécessaire, un commentaire sera ajouté entre accolades "{}".

1. Lde := ALL [d : DD_MVT | d.type = "E"] SORTED ON d.priorité
   Ldt := ALL [d : DD_MVT | d.type = "T"] SORTED ON d.priorité
   Lds := ALL [d : DD_MVT | d.type = "S"] SORTED ON d.priorité
2. Lv := Lds.litOrig
   {Raccourci pour écrire ALL [d.litOrig FROM d:DD_MVT | d in Lds]}
   >> Lds.accordée ()
   >> Lds.litOrig.fixeLitLibre
   {mécanisme de *broadcast* : appeler une action pour chaque instance d'une liste}
3. Lse := ALL UNIQUE [ser : SER_MED|ser in Lde]
4. Lv := EMPTY
   i := 1
   <u>While</u> NOT (i>Length(Lse))
   Lv := Lv + ALL[l:LIT | l.serDeLit = Lse(i) AND l.litOccupé=FALSE]
   i := i+1
   <u>End-While</u>
5. Lvv := Lv - ALL [l:LIT | l in Lv AND l in Ldt.litDest]
6. a) <u>Pattern AllouerLit</u>
   >> lit.fixeLitOccupéPar (Lde(i).pat)
   >> Lde(i).accordée with Lde(i).accordée.litDest = SELF.lit
   Lt := Lt + Lde(i)
   Lvv := Lvv - lit
   Lv := Lv - lit
   {Un pattern OBLOG peut être comparé à une procédure}

   b) <u>Actions</u>
   - init : i := 1; Ltemp := EMPTY
   - litDeCat : lit := ONE [l:LIT | l in Lvv AND l.serDelit = Lde(i).ser AND l.catégorieLit = Lde(i).cat]
   - litHorsCat : lit := ONE [l:LIT | l in Lvv AND l.serDelit = Lde(i).ser]
   - incl : i := i + 1
   - rien :
   - viderLde : Lde := Lde - Ltemp

   c) <u>Behaviour</u>



7. IF Lde=EMPTY AND Lvv<>EMPTY

```
THEN Lv := Lv-Lvv
```

8. a) Type de données et variables

```
MVT_DT : CP [lit : LIT, dde : DD_MVT]
ARC_DT : CP [lit : LIT, marque : STRING, suiv : MVT_DT, prec : LIST of MVT_DT]
LA_DT : LIST of ARC_DT
Ln : LIST of LIT {Liste de noeuds : temporaire}
La : LIST of ARC_DT {Liste de noeuds et d'arcs}
```

b) Sélection des noeuds

```
Ln := ALL UNIQUE [(d.litOrig AND d.litDest) FROM d:DD_MVT | d in Ldt]
```

c) Construction du graphe

```
i := 1
While not (i>length (Ln))
        La(i).lit := Ln(i)
        dde := ONE [d:DDE_MVT|d.litOrig=Ln(i) AND d in Ldt]
        IF not (dde=NULL)
        THEN  La(i).suiv.lit := dde.litDest
              La(i).suiv.dde := dde
        ELSE  La(i).suiv.lit := NULL
              La(i).suiv.dde := NULL
        listeDde := ALL[d:DDE_MVT|d.litDest = Ln(i) AND d iin Ldt]
        IF not listeDde=EMPTY
        THEN La(i).prec := (listeDde.litOrig, listeDde)
        ELSE La(i).prec := EMPTY
        i := i + 1
End-While
```

d) Pattern Composante

```
{But : identifier tous les noeuds d'une composante et les marquer}
Variables : Laf, Lf, Ltemp : LIST of ARC_DT
{Listes de noeuds à faire, faits et temporaire}
{Le pattern est appelé à partir de Lr(i) : cfr point f) )
Lf := Lr(i); Laf := Lr(POS[1](Lr(i).prec),Lr)
While not (Laf=EMPTY)
        j := 1; long := length(Laf); Ltemp := EMPTY
        While not (j>long)
                Ltemp := Ltemp + Lr(POS(Laf(j).prec))
                Lf := Lf + Laf(j);
                j := j + 1
        End-while
        Laf := Ltemp
End-while
For j := 1 to length(Lf)
        Lf(j).marque := marque {marque est une variable reçue}
End-for
Composante := Lf {renvoie la variable composante}
```

e) Pattern Cycle

```
{But : déterminer et marquer le coeur d'un cycle puis identifier et marquer}
{tous les noeuds de la composante qui contient ce cycle}
Variables : Lcycle : LIST of NAT {liste des indices du cycle}
While Lr(i).marque <> "Coeur_cycle"
        Lr(i).marque := "Coeur_cycle"
        Lcycle := Lcycle + i
        i := POS (Lr(i).suiv, Lr)
End-while
{Exécuter pattern Composante pour chaque noeud de Lcycle}
k := 1
While not (k>length(Lcycle))
        i := Lcycle(k)
        pattern Composante {Rem : marque = "Cycle"}
        compTemp := compTemp + composante
End-while
Composante := compTemp
```

f) Construction des composantes

```
Variables :     Lc : LIST of (Lac : LA_DT) {Liste des composantes}
                Lr : LA_DT {Liste des arcs restant à analyser}

Lr := La
While Lr <> EMPTY
        Lr(1).marque := "Composante"
        i := 1
        (*) {étiquette}
        IF Lr(i).suiv.lit <> NULL
        THEN  i := POS (Lr(i).suiv, Lr)
              IF Lr(i).marque = "Composante"
              THEN  pattern Cycle with marque = "Cycle"
                    Lc := Lc + composante
                    Lr := Lr - composante
```

[1] Afin de ne pas alourdir le développement, nous considérons que l'opérateur POS, recevant en entrée un ou plusieurs élément(s) d'une liste L et l'identificateur d'un liste L', fournit le ou les indices de cet (ces) élément(s) dans la liste L'.

```
                ELSE   Lr(i).marque := "Composante"
                       Goto (*)
          ELSE  Lr(i).marque := "composante"
                IF Lr(i).lit.litOccupé = TRUE
                THEN marque := "Impossible"
                ELSE marque := "A optimiser"
                pattern Composante
                Lc := Lc + composante
                Lr := Lr - composante
  End-while
```

9-10. {Elimination des cycles et des composantes impossibles}
Variables : Lj : LIST of NAT {Liste des indices des composantes "à jeter"}
Lcoeur : LIST of NAT {Liste des indices du coeur du cycle}

```
  long := length(Lc)
  For i := 1 to Long
        IF Lc(i).Lac(1).marque = "Cycle" OR ...="Coeur_cycle"
        THEN
              For j := 1 TO length(Lc(i).Lac)
                    IF Lc(i).Lac(j).marque = "Cycle"
                    THEN
                          IF Lc(i).Lac(j).suiv.dde <> NULL
                          THEN >>Lc(i).Lac(j).suiv.dde.refusée
                    ELSE {Je suis au coeur du cycle}
                          Lcoeur := Lcoeur + j
              End-For
              For j := 1 To Length (Lcoeur)
                    >> Lc(i).Lac(Lcoeur(j)).lit.fixeLitLibre
              End-For
              For j := 1 To Length (Lcoeur)
                    >> Lc(i).Lac(Lcoeur(j)).suiv.lit.fixeLitOccupéPar
                          (Lc(i).Lac(Lcoeur(j)).suiv.dde.pat)
              End-For
              Lj := Lj + i
        ELSE
              IF Lc(i).Lac(1).marque = "Impossible"
              THEN
                    For j := 1 To length (Lc(i).Lac)
                          IF Lc(i).Lac(j).suiv.dde <> NULL
                          THEN >> Lc(i).Lac(j).suiv.dde.refusée
                    End-for
                    Lj := Lj + i
  End-For
  For j := 1 To Length (Lj)
        Lc := Lc - Lc(Lj(j))
  End-for
```

11. {Exécution des composantes linéraires}

```
  Lj := EMPTY
  For i := 1 to length (Lc)
        Lineaire := TRUE
        For j := 1 to length (Lc(i).Lac)
              IF CARD (Lc(i).Lac(j).prec) > 1
              THEN Lineaire := FALSE
        End-for
        If Lineaire = TRUE
        THEN  j := 1
              While not (Lc(i).Lac(j).lit.litOccupé=FALSE)
                    j := j+1
              End-While
              J := POS (Lc(i).Lac(j).prec(1), Lc(i).Lac)
              While (Lc(i).Lac(j).prec <> EMPTY)
                    >> Lc(i).Lac(j).lit.fixeLitLibre
                    >> Lc(i).Lac(j).suiv.lit.fixeLitOccupéPar
                          (Lc(i).Lac(j).suiv.dde.pat)
                    >> Lc(i).Lac(j).suiv.dde.accordée
                    j := POS (Lc(i).Lac(j).prec(1), Lc(i).lac)
              End-while
        Lj := Lj + i
  End-For
  For j := 1 to length (Lj)
        Lc := Lc-Lc(Lj(j))
  End-for
```

12. /

13. a) Pattern CalculerPoids

{N(k) est l'indice de l'élément de la liste Lc(i).Lac à partir duquel il faut calculer le poids}

```
  l := N(k); poidsT := 0, chT.chemin := N(k)
  While not (Lc(i).Lac(l).suiv.dde=NULL)
```

```
            poidsT := PoidsT + Lc(i).Lac(l).suiv.dde.poids
            l := POS (Lc(i).Lac(l).suiv, Lc(i).Lac)
            chT.chemin := chT.chemin + l
    End-while
b) Optimisation
Variables :       chT, ch : CP [chemin : LIST of NAT, ddeE : NAT],
                  N : LIST of NAT
                  En : LIST of DDE_MVT
For i := 1 to length (Lc)
            ch := EMPTY; Max := 0
            For j := 1 to length (Lc(i).Lac)
                  IF Lc(i).Lac(j).prec=EMPTY
                  THEN   N := N + j
            End-For
            k := 1
    While not (k>length (N))
                  En := ALL[d:DDE_MVT|d.ser=Lc(i).Lac(j).lit.serDelit
                              AND d in Lde]
                  Pattern calculerPoids
                  IF poidsT > Max
                  THEN   ch.chemin := chT.chemin
                         Max := poidsT
                  m := 1
                  While not (m>length(En))
                         IF Lc(i).Lac(N(k)).lit.serDeLit=En(m).ser
                         THEN pdsE := En(m).poids+2
                         ELSE pdsE := En(m).poids
                         IF poidsT + pdsE > max
                         THEN   ch.ddE := En(m)
                                max := poidsT + pdsE
                         m := m + 1
                  End-while
                  k := k + 1
            End-while
            {Dans ch, j'ai le chemin optimal. Il suffit donc de l'exécuter}
            j := 1
            while not (j>length(ch.chemin))
                  >> Lc(i).Lac(ch.chemin(j)).dde.accordée
                  j := j + 1
            end-while
            IF ch.ddeE <> NULL
            THEN   >> ch.ddeE.accordée
      End-for
```

18.2. ALLOCATION-PRESCRIPTION-SOIN

18.2.1. La classe d'objets DDE-PS

La classe d'objets DDE-PS contient les demandes des différentes fonctionnalités[2] qui souhaitent allouer une prescription de soins. Une instance de la classe sera créée pour chaque demande émise et sera détruite lorsqu'elle aura été satisfaite. Rappelons, en effet, que la fonctionnalité émet une demande pour une prescription à telle heure, tel jour et que c'est en fonction du planning et de la priorité de cette demande que cet prescription est alloué comme souhaité ou plus tard s'il n'y a plus de place dans le planning. Cette décision est, comme auparavant, prise par la fonction allocationPrescr de l'objet Allocation. Cette fonction ne subit pas de modifications.

La spécification de la classe d'objet est la suivante :

OBJET DDE-PS

INTERFACE

- **Observations**
  - pat : PAT
  - med : MED
  - prest : PREST
  - datePr : DATE
  - heurePr : TIME
  - priorité : NAT

- **Keys**
- **Events**
  - * birth (pat : PAT, med : MED, prest : PREST, datePr : DATE, heurePr : TIME, priorité : NAT, acceptEv : EVENT)

[2] En réalité, elle contient les demandes des différentes instances de l'objet-fonctionnalité PRESCR_SOIN.

BODY

- **Attributes**
  - pat : PAT
  - med : MED
  - prest : PREST
  - datePr : DATE
  - heurePr : TIME
  - priorité : NAT
  - acceptEv : EVENT

- **Derivations**

- **Actions**
  - * birth (pat : PAT, med : MED, prest : PREST, datePr : DATE, heurePr : TIME, priorité : NAT, acceptEv : EVENT)
    ⇒ pat := birth.pat
    ⇒ ...
  - accordée
    >> acceptEv
  - !+ suicide

BEHAVIOUR

birth

sit1

accordée

sit2

suicide

18.2.2. Implémentation

La solution au problème de l'optimisation de l'allocation des prescriptions de soins est beaucoup plus simple que la précédente, notamment grâce au fait que nous avons déjà développé la fonction qui place une prescription de soins dans le planning en fonction de sa priorité. Il nous reste à gérer le fait que plusieurs demandes sont émises en même temps et il faut donc les soumettre à la fonction allocPrescr une par une et dans l'ordre des priorités.

Comme pour la spécification précédente, nous avons opté pour une solution décrite par un pseudo-algorithme proche de OBLOGlight.

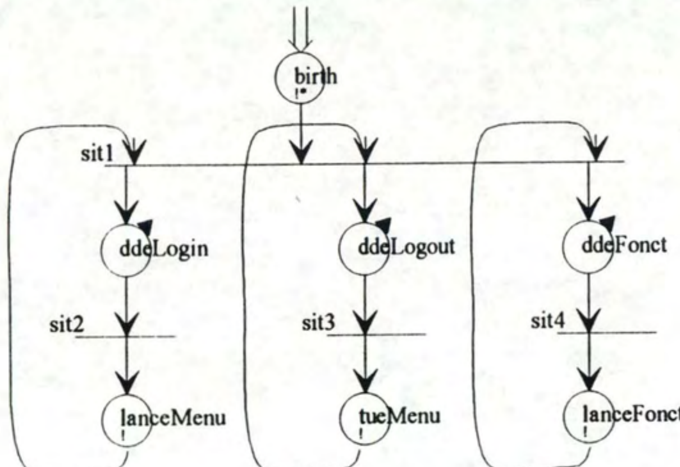
1-2. Ldps := ALL[dde:DDE-PS|TRUE] SORTED ON priorité
{Liste des demandes de prescriptions de soins triée par priorité}
3. For i := 1 TO length(Ldps)
>> allocation.allocPrescr (Ldps(i).pat, Ldps(i).med, Ldps(i).prest, Ldps(i).datePr,
Ldps(i).heurePr,
Ldps(i).priorité)
{Appel à la fonction allocPrescr de l'objet Allocation}
>> Ldps(i).accordée
End-for

19. OBJET SESSION

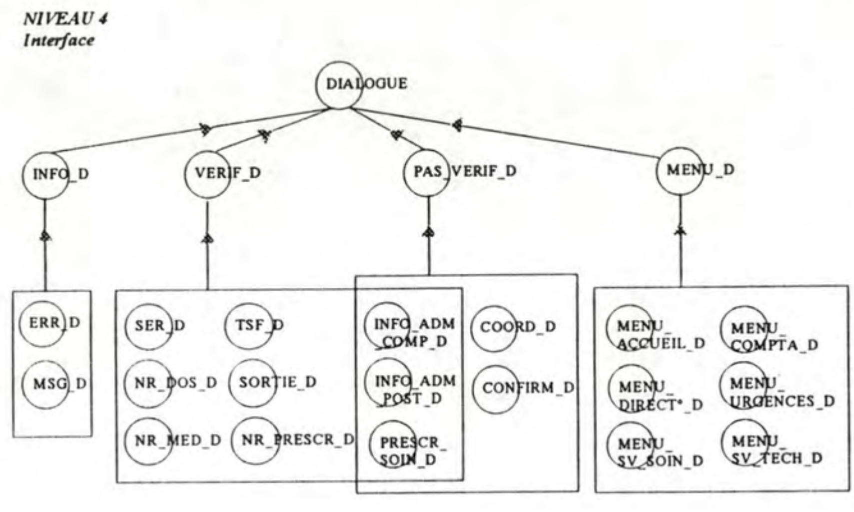
Pour cet objet, nous nous contenterons d'une description sommaire.

Cet objet dispose de trois fonctions qui sont `ddeLogin`, `ddeLogout` et `ddeFonct`. La fonction `ddeLogin` est activée lorsque, depuis son terminal, une personne demande à entrer dans le système. En fonction du service concerné, le menu adéquat est alors affiché (action `lanceMenu`). La fonction `ddeLogout` est activée lorsque, depuis son terminal, une personne demande à sortir du système. Le menu est alors effacé de son terminal (action `tueMenu`). La fonction `ddeFonct` est activée lorsque, à partir de son menu, une personne sélectionne une fonctionnalité. La fonctionnalité concernée est alors déclenchée (action `lanceFonct`).

BEHAVIOUR



1. ARCHITECTURE GLOBALE



2. OBJET DIALOGUE

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * birth
  - \+ death

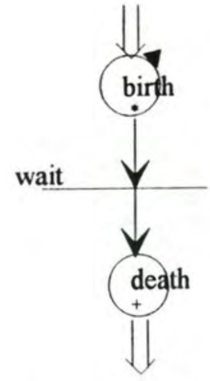
BODY

- **Attributes**

- **Derivations**
  - titre : STRING := "Dialogue"

- **Actions**
  - * birth
  - \+ death

BEHAVIOUR



3. OBJET INFO_D

Rem : IS-A DIALOGUE

INTERFACE

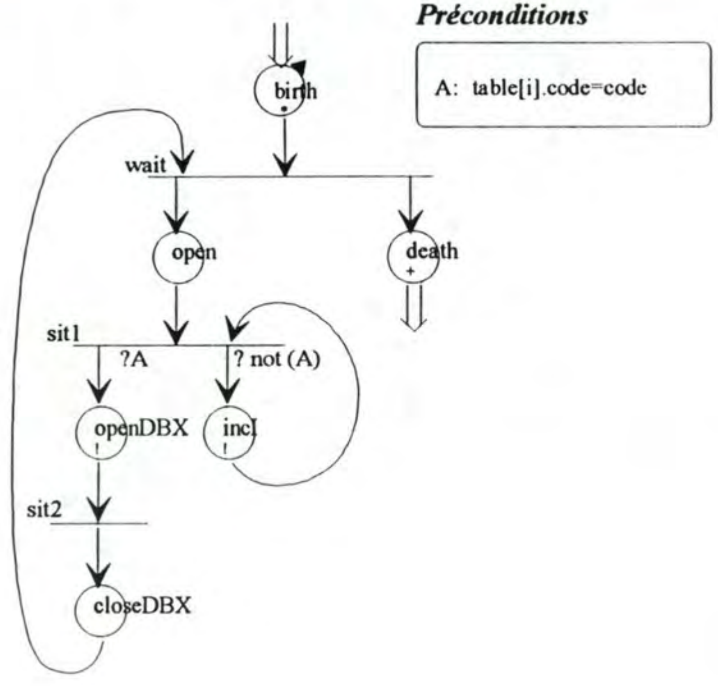
- **Observations**
- **Keys**
- **Events**
  - * birth
  - open (code : INFO_CODE_DT)
    {cette spécif est incorrecte car il n'existe pas de notion de sur-type d'un type de donnée. L'idéal serait de pouvoir déclarer INFO_CODE_DT comme le sur-type de ERR_CODE_DT et de MSG_CODE_DT. }
  - + death

BODY

- **Attributes**
  - table : ARRAY of CP[code : INFO_CODE_DT, chaîne : STRING]
  - code : INFO_CODE_DT
  - i : NAT
- **Derivations**
  - titre : STRING
    := "Information"
- **Actions**
  - * birth
    ⇒ table[1].code := ...
    ⇒ table[1].chaîne := "..."
    ⇒ etc.
  - open (code)
    ⇒ code := open.code
    ⇒ i := 1
  - ! incI
    ⇒ i := i + 1
  - ! openDBX
    CALLS to EXT with (table[i].chaîne, titre)
    {ouvre la boîte}
  - closeDBX
    CALLED from EXT
    {l'utilisateur a cliqué sur le seul bouton OK}
    CALLS to EXT
    {ferme la boîte}
  - + death

Je vais, pour l'instant, me limiter à spécifier le sur-type de chaque classe sans entrer dans les détails des différents objets spécialisés. Seules une image de la boîte et, si nécessaire, une liste des primitives spécifiques à un objet spécialisé seront exposées. Je ne crois pas que j'irais plus loin dans la suite.

BEHAVIOUR



4. OBJET ERR_D

Rem : IS-A INFO_D

APERÇU DE LA BOÎTE

Erreur

< Message d'erreur >

OK

5. OBJET MSG_D

Rem : IS-A INFO_D

APERÇU DE LA BOÎTE

6. OBJET VERIF_D

Rem : IS-A DIALOGUE

But : saisir un (plusieurs) identifiant(s) relatif(s) à un (plusieurs) objet(s). Vérifier leur existence et en obtenir la référence interne. Dans les sous-types de cet objet (ainsi que des autres), il sera fait mention des observations utilisées.

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * birth
  - open *(okEv, cancelEv : EVENT)*
    {cette spécif est doublement incorrecte car (1) il n'existe pas de notion de sur-type d'un type événement. L'idéal serait de pouvoir déclarer un type EVENT représentant un événement quelconque : cela allégerait le mécanisme lourd du stored action. (2) Par ailleurs, ces paramètres ne devraient pas être spécifiés à cette phase : à discuter (cf. problème soulevé au début du document}
  - + death

BODY

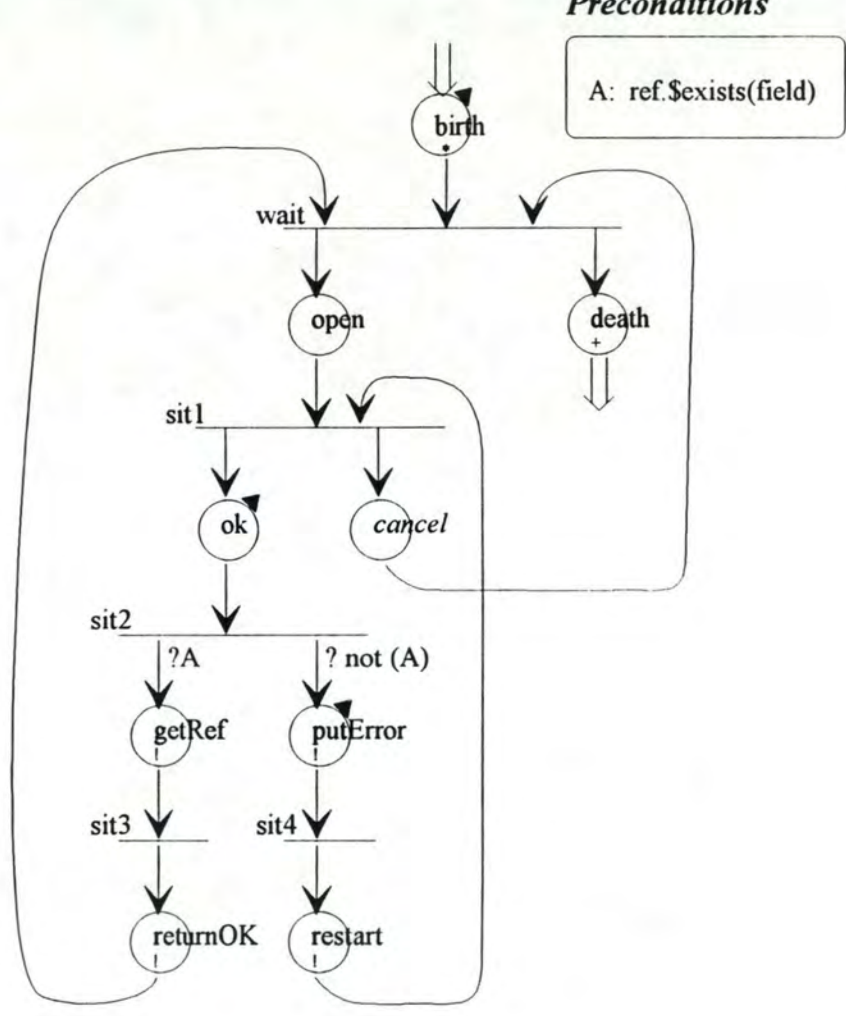
- **Attributes**
  - field : STRING
  - ref : OBJECT *{n'importe quel objet : à redéfinir dans la spécialisation}*
  - *okEv, cancelEv : EVENT*

- **Derivations**
  - titre : STRING
    := "Vérification"

- **Actions**
  - * birth
  - open *(okEv, cancelEv)*
    ⇒ *okEv, cancelEv := open.okEv, open.cancelEv*
    CALLS to EXT with (titre) *{ouvre la boîte}*

- ok (field : STRING)
  CALLED from EXT
  ⇒ field := ok.field
- *cancel*
  CALLED from EXT
  >> cancelEv
  CALLS to EXT {ferme la boîte}
- ! getRef
  ⇒ ref := ref.$key (field) *{obtient la référence de l'objet sur base de son identifiant : à redéfinir dans la spécialisation}*
- ! putError (code : ERR_CODE_DT)
  := code = ...
  >> ERR_D.open (code)
- ! restart
  CALLS to EXT *{redémarrer la boîte : la réactiver}*
- *! returnOk*
  CALLS to EXT {ferme la boîte}
  >> okEv
  := okEv.ref := ref {rappelle où on l'a appelé et passe ref dans le paramètre de l'événement okEv}
- + death

BEHAVIOUR



7. OBJET SER_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE

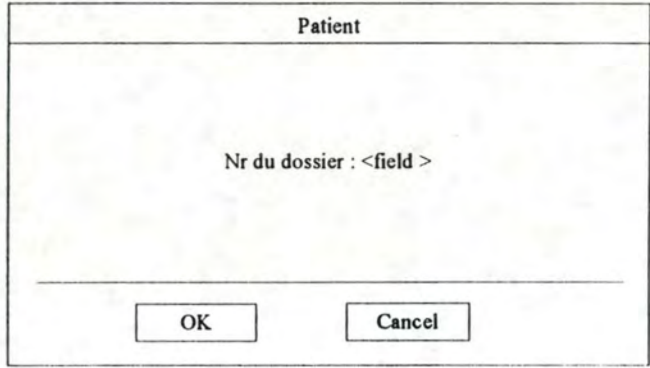
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : SER_MED
- **Primitives**
  - $serMedExistant(nom)
  - $serviceMédical(nom)
- **Code d'erreur** : SER_MED_NOT_EXISTS

8. OBJET NR_DOS_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE



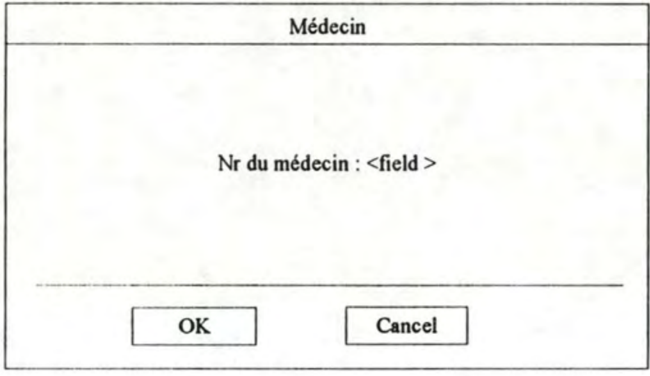
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : PAT
- **Primitives**
  - $patExistantParNrDos(nrDos)
  - $patParNrDos(nrDos)
- **Code d'erreur** : PAT_NOT_EXISTS

9. OBJET NR_MED_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE



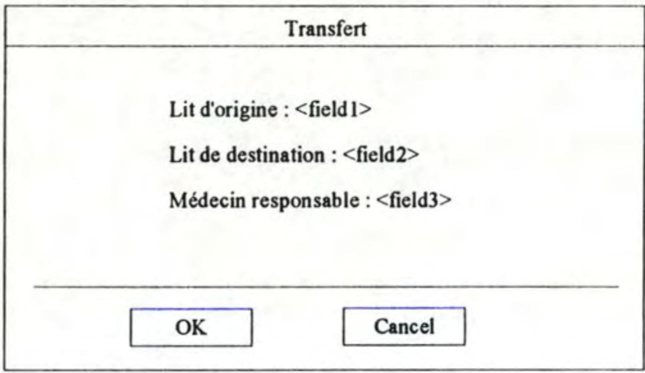
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : MED
- **Primitives**
  - $medExistant(nrMed)
  - $médecin(nrMed)
- **Code d'erreur** : MED_NOT_EXISTS

10. OBJET TSF_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE



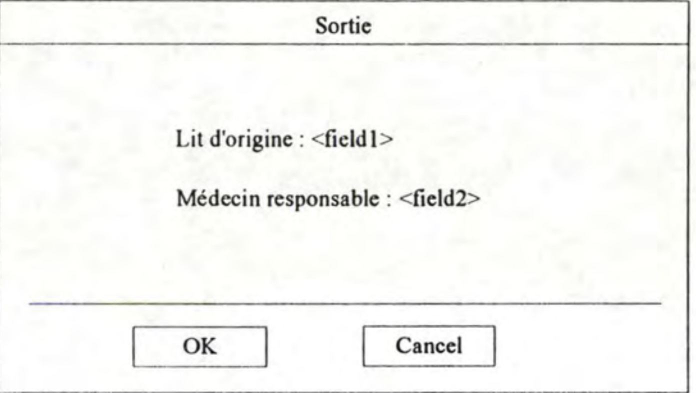
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : MED, LIT
- **Primitives**
  - $litExistant(nrLit)
  - $lit(nrLit)
  - $medExistant(nrMed)
  - $médecin(nrMed)
- **Code d'erreur** : MED_NOT_EXISTS, LIT_NOT_EXISTS

11. OBJET SORTIE_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE



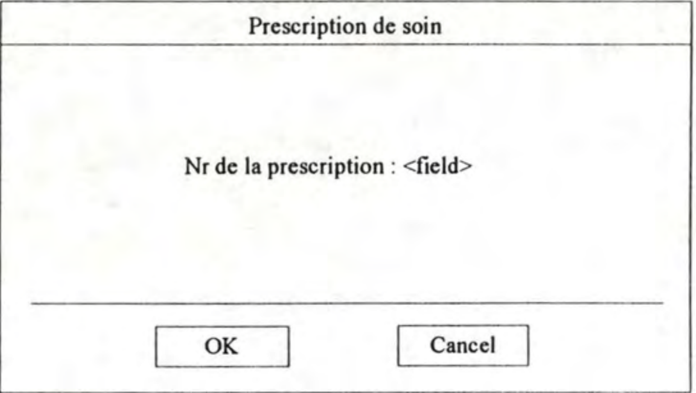
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : MED, LIT
- **Primitives**
  - $litExistant(nrLit)
  - $lit(nrLit)
  - $medExistant(nrMed)
  - $médecin(nrMed)
- **Code d'erreur** : MED_NOT_EXISTS, LIT_NOT_EXISTS

12. OBJET NR_PRESCR_D

Rem : IS-A VERIF_D

APERÇU DE LA BOÎTE



OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : PRESCR_SOIN
- **Primitives**
  - $prescrSoinExistante(nrPrescr)
  - $prescriptionSoin(nrPrescr)
- **Code d'erreur** : PRESCR_SOIN_NOT_EXISTS

13. OBJET PAS_VERIF_D

Rem : IS-A DIALOGUE

But : saisir un ou plusieurs renseignement(s) ne nécessitant pas de vérification d'existence (adresse, téléphone, etc.).

INTERFACE

- **Observations**
- **Keys**
- **Events**
  - * birth
  - open *(okEv, cancelEv : EVENT)*
    {cette spécif est doublement incorrecte : cf. VERIF_D}
  - + death

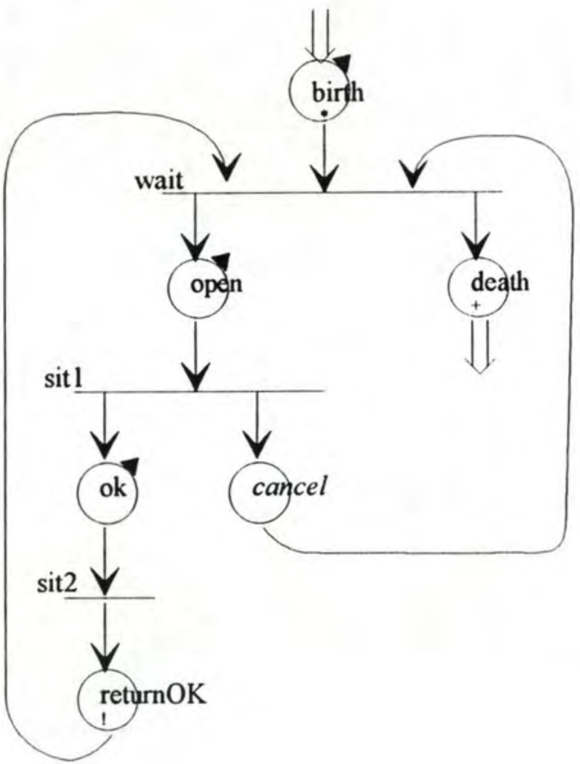
BODY

- **Attributes**
  - field : STRING
  - *okEv, cancelEv : EVENT*

- **Derivations**
  - titre : STRING
    := "Sans Vérification"

- **Actions**
  - * birth
  - open *(okEv, cancelEv)*
    ⇒ *okEv, cancelEv := open.okEv, open.cancelEv*
    CALLS to EXT with (titre) *{ouvre la boîte}*
  - ok (field : STRING)
    CALLED from EXT
    ⇒ field := ok.field
  - *cancel*
    CALLED from EXT
    >> cancelEv
    CALLS to EXT {ferme la boîte}
  - *! returnOk*
    CALLS to EXT {ferme la boîte}
    >> okEv
    := okEv.field := field {rappelle où on l'a appelé et passe field dans le paramètre de l'événement okEv}
  - + death

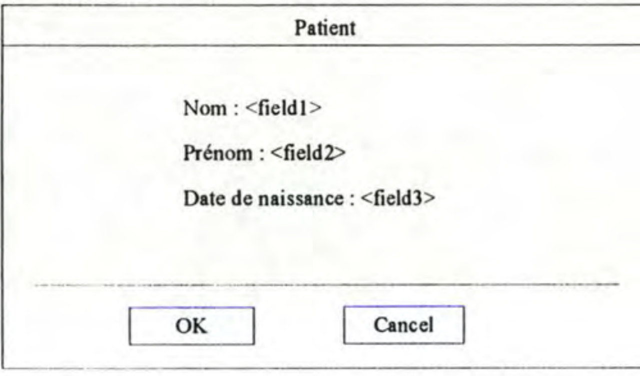
BEHAVIOUR



14. OBJET COORD_D

Rem : IS-A PAS_VERIF_D

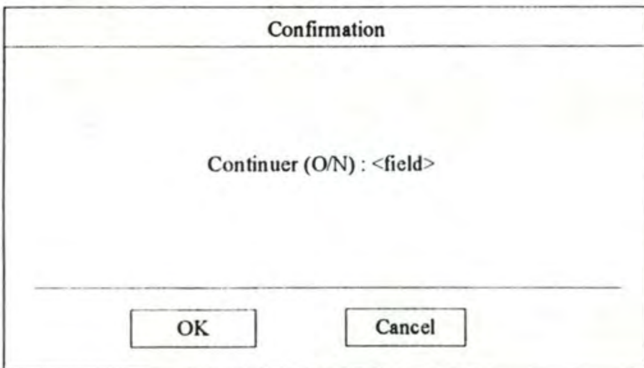
APERÇU DE LA BOÎTE



15. OBJET CONFIRM_D

Rem : IS-A PAS_VERIF_D

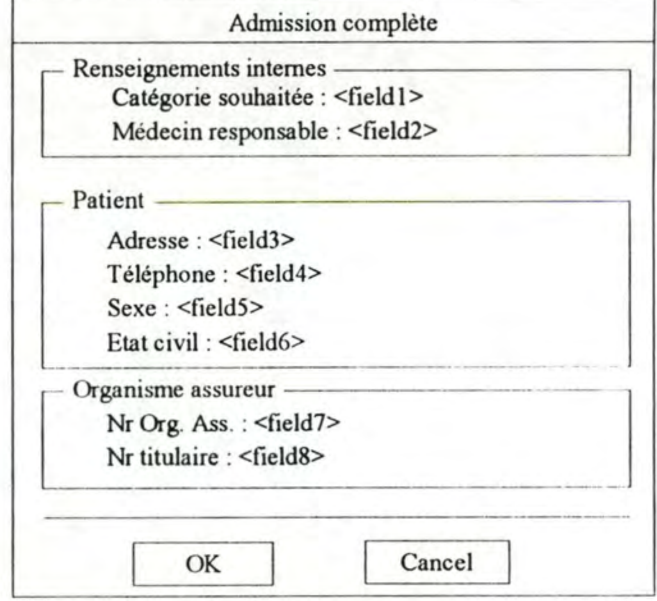
APERÇU DE LA BOÎTE



16. OBJET INFO_ADM_FULL_D

Rem : IS-A PAS_VERIF_D AND VERIF_D
{les boîtes qui suivent mélangent les services des deux types proposés jusqu'ici : certaines informations sont vérifiées et d'autres pas. Le problème est que OBLOGlight ne permet pas d'héritage multiple !}

APERÇU DE LA BOÎTE



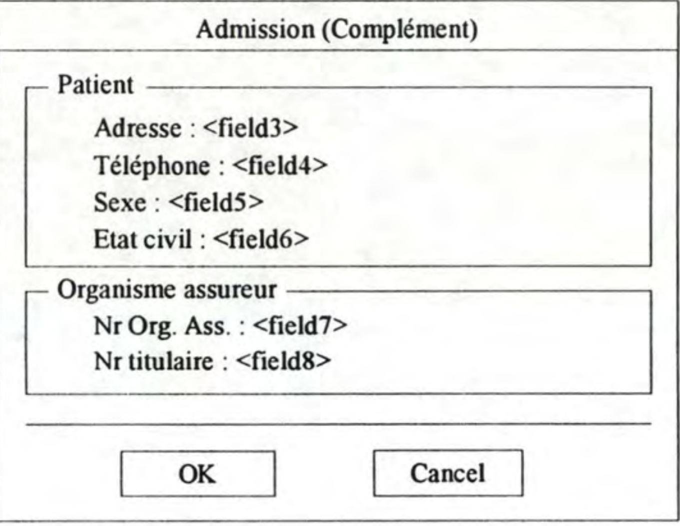
OBJET ET PRIMITIVES DE L'OBJET

- **Objet** : CAT, MED (cf. supra), ORG_ASS
- **Primitives**
  - $catExistante
  - $catégorie
  - $oAExistant
  - $organismeAssureur
- **Code d'erreur** : CAT_NOT_EXISTS, OA_NOT_EXISTS

17. OBJET INFO_ADM_POST_D

Rem : IS-A PAS_VERIF_D AND VERIF_D

APERÇU DE LA BOÎTE



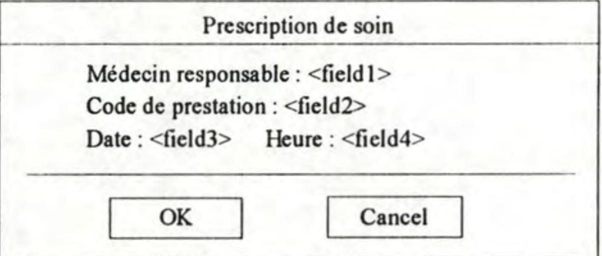
OBJET ET PRIMITIVES DE L'OBJET

déjà défini.

18. OBJET PRESCR_SOIN_D

Rem : IS-A PAS_VERIF_D AND VERIF_D

APERÇU DE LA BOÎTE



OBJET ET PRIMITIVES DE L'OBJET

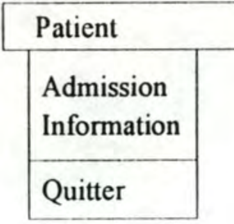
- **Objet** : PREST
- **Primitives**
  - **$prestExistante**
  - **$prestation**
- **Code d'erreur** : PREST_NOT_EXISTS

19. OBJET MENU_ACCUEIL_D

Rem : IS-A MENU_D

APERÇU DU MENU

SERVICE ACCUEIL

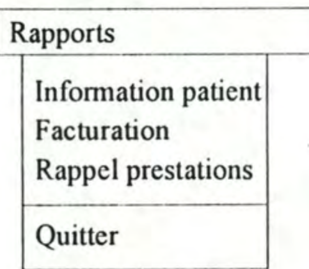


20. OBJET MENU_DIRECTION_D

Rem : IS-A MENU_D

APERÇU DU MENU

SERVICE DIRECTION

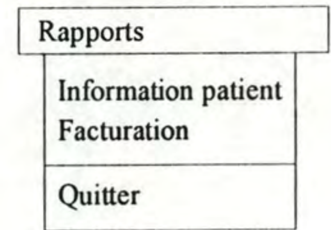


21. OBJET MENU_COMPTA_D

Rem : IS-A MENU_D

APERÇU DU MENU

SERVICE COMPTABILITE

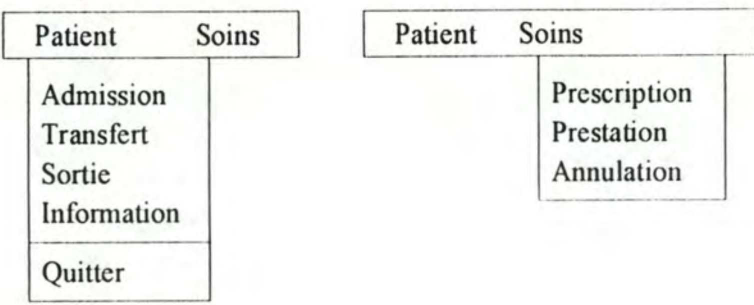


22. OBJET MENU_URGENCES_D

Rem : IS-A MENU_D

APERÇU DU MENU

SERVICE URGENCES

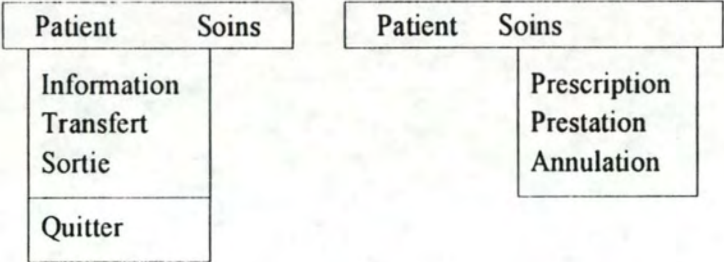


23. OBJET MENU_SV_SOIN_D

Rem : IS-A MENU_D

APERÇU DU MENU

SERVICE <service de soin>



24. OBJET MENU_SV_TECH_D

Rem : IS-A MENU_D

APERÇU DU MENU

SERVICE <service technique>

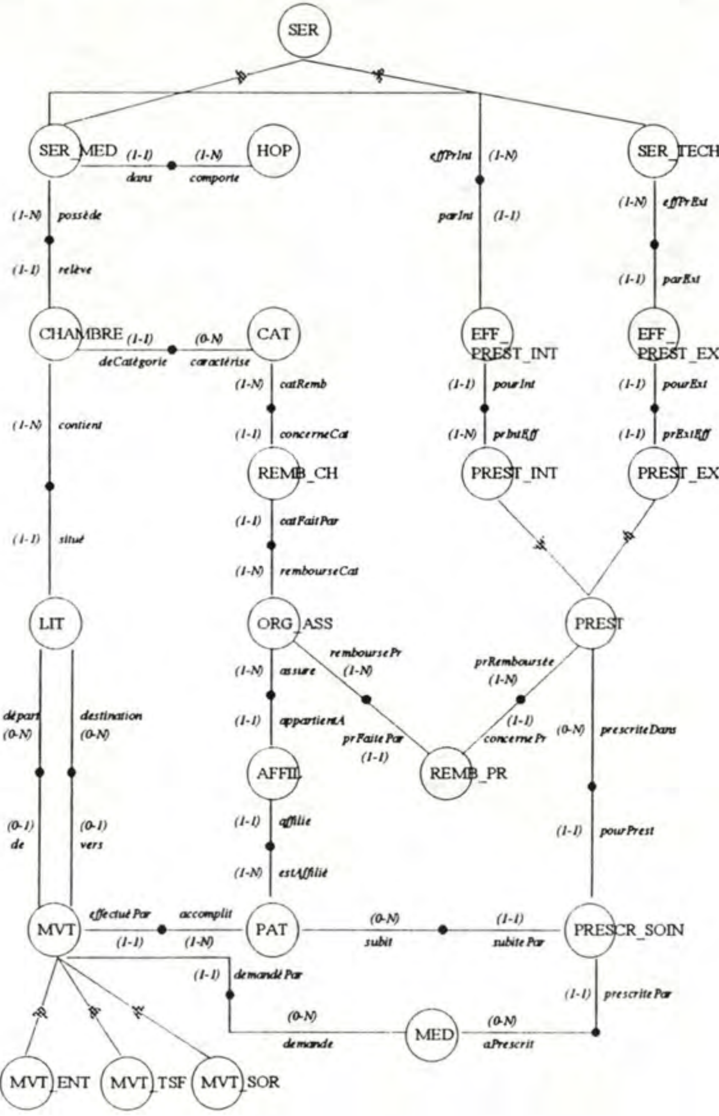
Opérations

Prestation prescription
Information patient

Quitter

III. LE NIVEAU 3

1. ARCHITECTURE GLOBALE



2. OBJET HOP

<u>Rem</u> : tous les objets sont de types tables (TBL) et la base de données est supposée initialisée.

INTERFACE

- **Observations**
  - dateDerFact : DATE
    {? état=libre}
  - dateDerStat : DATE
    {? état=libre}
  - comporte : LIST of SER_MED
    {? état=libre}
  - nom : STRING
    {? état=libre}

- **Keys**
  - $hôpital (nom : STRING) : HOP
    {? état=libre}
    {? $hopExistant (nom)}

- **Events**
  - *** birth (nom : STRING, dateDerFact, dateDerStat : DATE)**
  - fixeDateDerFact (date : DATE)

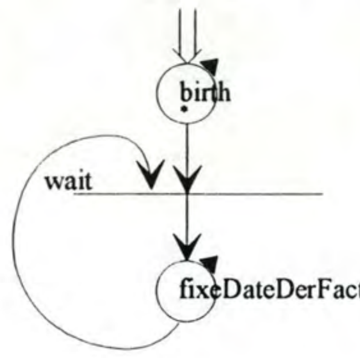
BODY

- **Attributes**
  - dateDerFact : DATE
  - dateDerStat : DATE
  - nom : STRING
  - état : STATE_DT

- **Derivations**
  - comporte
    := ALL[sM : SER_MED | sM.dans = SELF]
  - $hôpital (nom)
    := ONE[h:HOP | h.nom = nom]

- **Actions**
  - * birth (nom, dateDerFact, dateDerStat)
    ⇒ nom := birth.nom
    ⇒ dateDerFact := birth.dateDerFact
    ⇒ dateDerStat := birth.dateDerSat
    ⇒ état := libre
  - fixeDateDerFact (date)
    ⇒ dateDerFact := fixeDateDerFact.date

BEHAVIOUR



3. OBJET SER

INTERFACE

- **Observations**
  - nom : STRING
    {? état=libre}
  - $serExistant (nom : STRING) : BOOL
    {? état=libre}

- **Keys**
  - $service (nom : STRING) : SER_MED
    {? état=libre}
    {? $serExistant (nom)}

- **Events**
  - * birth (nom : STRING)

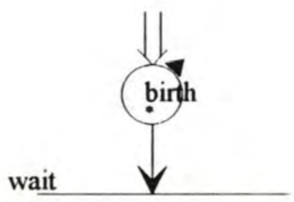
BODY

- **Attributes**
  - nom : STRING
  - état : STATE_DT

- **Derivations**
  - $service (nom)
    := ONE[s : SER | s.nom = nom]
  - $serExistant (nom)
    := EXISTS[s : SER | s.nom = nom]

- **Actions**
  - * birth (nom)
    ⇒ nom := birth.nom
    ⇒ état := libre

BEHAVIOUR



4. OBJET SER_MED

Rem : IS-A SER

INTERFACE

- **Observations**
  - dans : HOP
    {? état=libre}
  - possède : LIST of CHAMBRE

{? état=libre}
• serPlein : BOOL
{? état=libre}
• nom : STRING
{? état=libre}
• effPrInt : LIST of EFF_PREST_INT
{? état=libre}
• $serMedExistant (nom : STRING) : BOOL
{? état=libre}

- **Keys**
  • $serviceMédical (nom : STRING) : SER_MED
  {? état=libre}
  {? $serMedExistant (nom)}

- **Events**
  • * birth (nom : STRING, hop : HOP)

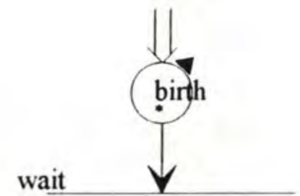
BODY

- **Attributes**
  • dans : HOP
  • nom : STRING
  • état : STATE_DT

- **Derivations**
  • possède
  := ALL[ch : CHAMBRE | ch.relève = SELF]
  • $serviceMédical (nom)
  := ONE[sM : SER_MED | sM.nom = nom]
  • serPlein
  := NOT EXISTS[l : LIT | l.serDeLit = SELF AND l.litOccupé=FALSE]
  • $serMedExistant (nom)
  := EXISTS[sM : SER_MED | sM.nom = nom]
  • effPrInt
  := ALL[eff : EFF_PREST_INT | eff.parInt = SELF]

- **Actions**
  • * birth (nom, hop)
  ⇒ nom := birth.nom
  ⇒ dans := birth.hop
  ⇒ état := libre

BEHAVIOUR



5. OBJET CHAMBRE

INTERFACE

- **Observations**
  • deCatégorie : CAT
  {? état=libre}
  • contient : LIST of LIT
  {? état=libre}
  • relève : SER_MED
  {? état=libre}
  • nrCh : NAT
  {? état=libre}
  • $chambreExistante (nrCh : NAT) : BOOL
  {? état=libre}

- **Keys**
  • $chambre (nrCh : NAT) : CHAMBRE
  {? état=libre}
  {? $chambreExistante (nrCh)}

- **Events**
  • * birth (serMed : SER_MED, cat : CAT)

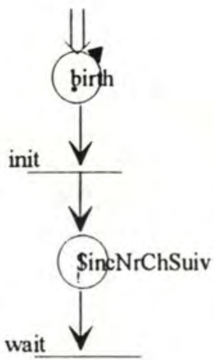
Body

- **Attributes**
  - nrCh, deCatégorie, relève
  - $nrChSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - contient
    := ALL[l : LIT | l.situé = SELF]
  - $chambre (nrCh)
    := ONE[ch : CHAMBRE | ch.nrCh = nrCh]
  - $chambreExistante (nrCh)
    := EXISTS[ch : CHAMBRE | ch.nrCh = nrCh]

- **Actions**
  - * birth (serMed, cat)
    ⇒ nrCh := $nrChSuiv
    ⇒ deCatégorie := birth.cat
    ⇒ relève := birth.serMed
    ⇒ état := occupé
  - ! $incNrChSuiv
    ⇒ $NrChSuiv := $nrChSuiv+1
    ⇒ état := libre

Behaviour



6. OBJET CAT

Interface

- **Observations**
  - cat : STRING
    {? état=libre}
  - caractérise : LIST of CHAMBRE
    {? état=libre}
  - prixPlein : NAT
    {? état=libre}
  - catRemb : LIST of REMB_CH
    {? état=libre}
  - $catExistante (cat : STRING) : BOOL
    {? état=libre}

- **Keys**
  - $catégorie (cat : STRING) : CAT
    {? état=libre}
    {? $catExistante (cat)}

- **Events**
  - * birth (cat : STRING, prixPlein : NAT)

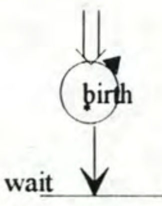
Body

- **Attributes**
  - cat, prixPlein
  - état : STATE_DT

- **Derivations**
  - caractérise
    := ALL[ch : CHAMBRE | ch.deCatégorie = SELF]
  - $catégorie (cat)
    := ONE[c : CAT | c.cat = cat]
  - catRemb
    := ALL[rCh : REMB_CH | rCh.concerneCat = SELF]
  - $catExistante (cat)
    := EXISTS[c : CAT | c.cat = cat]

- **Actions**
  - * birth (cat, prixPlein)
    ⇒ cat := birth.cat
    ⇒ prixPlein := birth.prixPlein
    ⇒ état := libre

BEHAVIOUR



7. OBJET LIT

INTERFACE

- **Observations**
  - catégorieLit : CAT
    {? état=libre}
  - situé : CHAMBRE
    {? état=libre}
  - relève : SER_MED
    {? état=libre}
  - nrLit : NAT
    {? état=libre}
  - départ, destination : LIST of MVT
    {? état=libre}
  - litOccupé : BOOL
    {? état=libre}
  - serDeLit : SER_MED
    {? état=libre}
  - litOccupéPar : PAT
    {? état=libre}
  - $litExistant (nrLit : NAT) : BOOL
    {? état=libre}

- **Keys**
  - $lit (nrLit : NAT) : LIT
    {? état=libre}
    {? $litExistant (nrLit)}

- **Events**
  - * birth (ch : CHAMBRE)
  - fixeLitOccupéPar (pat :PAT)
  - fixeLitLibre

BODY

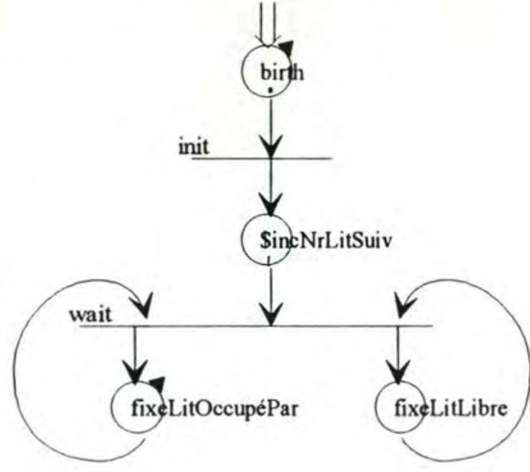
- **Attributes**
  - nrLit, situé, litOccupéPar
  - $nrLitSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - départ
    := ALL[m : MVT_TSF/SOR | m.de = SELF]
  - destination
    := ALL[m : MVT_TSF/ENT | m.vers = SELF]
  - litOccupé
    := litOccupéPar <> NULL
  - catégorieLit
    := SELF.situé.deCatégorie
  - serDeLit
    := SELF.situé.relève
  - $lit (nrLit)
    := ONE[l : LIT | l.nrLit = nrLit]
  - $litExistant (nrLit)
    := EXISTS[l : LIT | l.nrLit = nrLit]

- **Actions**
  - * birth (ch)
    ⇒ nrLit := $nrLitSuiv
    ⇒ situé := birth.ch
    ⇒ litOccupéPar := NULL
    ⇒ état := occupé

- ! $incNrLitSuiv
  ⇒ $NrLitSuiv := $nrLitSuiv+1
  ⇒ état := libre
- fixeLitOccupéPar(p)
  ⇒ litOccupéPar := fixeLitOccupéPar.p
- fixeLitLibre
  ⇒ litOccupéPar := NULL

BEHAVIOUR



8. OBJET MVT

INTERFACE

- **Observations**
  - nrMvt : NAT
    {? état=libre}
  - dateMvt : DATE
    {? état=libre}
  - heureMvt : TIME
    {? état=libre}
  - typeMvt : TYPE_MVT_DT
    {? état=libre}
  - *de, vers : LIT*
    {? état=libre}
  - demandéPar : MED
    {? état=libre}
  - effectuéPar : PAT
    {? état=libre}
  - $mvtExistant (nrMvt : NAT) : BOOL
    {? état=libre}

- **Keys**
  - $mouvement (nrMvt : NAT) : MVT
    {? état=libre}
    {? $mvtExistant (nrMvt)}
- **Events**
  - * birth (p : PAT, med : MED, date : DATE, heure : TIME)

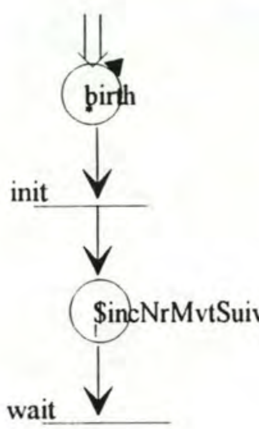
BODY

- **Attributes**
  - nrMvt, dateMvt, heureMvt, typeMvt, demandéPar, effectuéPar, *de, vers*
  - $nrMvtSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - $mouvement (nrMvt)
    := ONE[m:MVT | m.nrMvt = nrMvt]
  - $litExistant (nrMvt)
    := EXISTS[m:MVT | m.nrMvt = nrMvt]

- **Actions**
  - * birth (p, med, date, heure)
    ⇒ nrMvt := $nrMvtSuiv
    ⇒ dateMvt := birth.date
    ⇒ heureMvt := birth.heure
    ⇒ effectuéPar := birth.pat
    ⇒ demandéPar := birth.med
    ⇒ état := occupé
  - ! $incNrMvtSuiv
    ⇒ $NrMvtSuiv := $nrMvtSuiv+1
    ⇒ état := libre

BEHAVIOUR



9. OBJET MVT_ENT

Rem : IS-A MVT

INTERFACE

- **Observations**
  - nrMvt : NAT
    {? état=libre}
  - dateMvt : DATE
    {? état=libre}
  - heureMvt : TIME
    {? état=libre}
  - typeMvt : TYPE_MVT_DT
    {? état=libre}
  - vers : LIT
    {? état=libre}
  - demandéPar : MED
    {? état=libre}
  - effectuéPar : PAT
    {? état=libre}
  - $mvtExistant (nrMvt : NAT) : BOOL
    {? état=libre}
- **Keys**
  - $mouvement (nrMvt : NAT) : MVT
    {? état=libre}
    {? $mvtExistant (nrMvt)}
- **Events**
  - * birth (litDest : LIT, p : PAT, med : MED, date : DATE, heure : TIME)

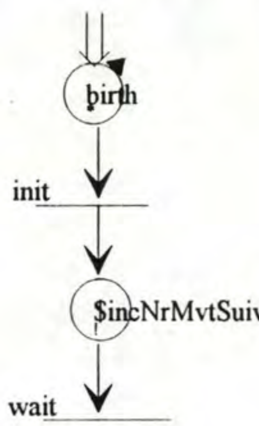
BODY

- **Attributes**
  - nrMvt, dateMvt, heureMvt, typeMvt, demandéPar, effectuéPar, vers
  - $nrMvtSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - $mouvement (nrMvt)
    := ONE[m:MVT | m.nrMvt = nrMvt]
  - $litExistant (nrMvt)
    := EXISTS[m:MVT | m.nrMvt = nrMvt]

- **Actions**
  - * birth (p, med, date, heure)
    ⇒ nrMvt := $nrMvtSuiv
    ⇒ dateMvt := birth.date
    ⇒ heureMvt := birth.heure
    ⇒ effectuéPar := birth.pat
    ⇒ demandéPar := birth.med
    ⇒ vers := birth.litDest
    ⇒ typeMvt = ENTREE
    ⇒ état := occupé
  - ! $incNrMvtSuiv
    ⇒ $NrMvtSuiv := $nrMvtSuiv+1
    ⇒ état := libre

BEHAVIOUR



10. OBJET MVT_TSF

Rem : IS-A MVT

INTERFACE

- **Observations**
  - nrMvt : NAT
    {? état=libre}
  - dateMvt : DATE
    {? état=libre}
  - heureMvt : TIME
    {? état=libre}
  - typeMvt : TYPE_MVT_DT
    {? état=libre}
  - de, vers : LIT
    {? état=libre}
  - demandéPar : MED
    {? état=libre}
  - effectuéPar : PAT
    {? état=libre}
  - $mvtExistant (nrMvt : NAT) : BOOL
    {? état=libre}
- **Keys**
  - $mouvement (nrMvt : NAT) : MVT
    {? état=libre}
    {? $mvtExistant (nrMvt)}
- **Events**
  - * birth (litOr, litDest : LIT, p : PAT, med : MED, date : DATE, heure : TIME)

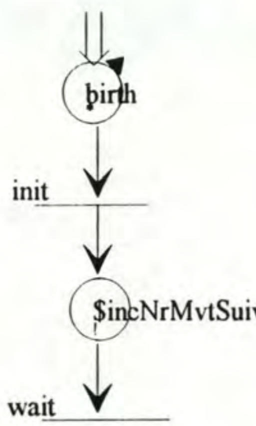
BODY

- **Attributes**
  - nrMvt, dateMvt, heureMvt, typeMvt, demandéPar, effectuéPar, de, vers
  - $nrMvtSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - $mouvement (nrMvt)
    := ONE[m:MVT | m.nrMvt = nrMvt]
  - $litExistant (nrMvt)
    := EXISTS[m:MVT | m.nrMvt = nrMvt]

- **Actions**
  - * birth (p, med, date, heure)
    ⇒ nrMvt := $nrMvtSuiv
    ⇒ dateMvt := birth.date
    ⇒ heureMvt := birth.heure
    ⇒ effectuéPar := birth.pat
    ⇒ demandéPar := birth.med
    ⇒ vers := birth.litDest
    ⇒ de := birth.litOr
    ⇒ typeMvt = TRANSFERT
    ⇒ état := occupé
  - ! $incNrMvtSuiv
    ⇒ $NrMvtSuiv := $nrMvtSuiv+1
    ⇒ état := libre

BEHAVIOUR



11. OBJET MVT_SOR

Rem : IS-A MVT

INTERFACE

- **Observations**
  - nrMvt : NAT
    {? état=libre}
  - dateMvt : DATE
    {? état=libre}
  - heureMvt : TIME
    {? état=libre}
  - typeMvt : TYPE_MVT_DT
    {? état=libre}
  - de : LIT
    {? état=libre}
  - demandéPar : MED
    {? état=libre}
  - effectuéPar : PAT
    {? état=libre}
  - $mvtExistant (nrMvt : NAT) : BOOL
    {? état=libre}
- **Keys**
  - $mouvement (nrMvt : NAT) : MVT
    {? état=libre}
    {? $mvtExistant (nrMvt)}
- **Events**
  - * birth (litOr : LIT, p : PAT, med : MED, date : DATE, heure : TIME)

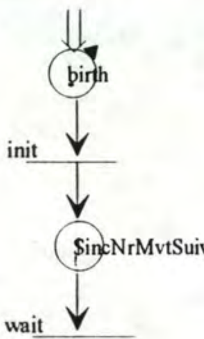
BODY

- **Attributes**
  - nrMvt, dateMvt, heureMvt, typeMvt, demandéPar, effectuéPar, de
  - $nrMvtSuiv : NAT
  - état : STATE_DT

- **Derivations**
  - $mouvement (nrMvt)
    := ONE[m:MVT | m.nrMvt = nrMvt]
  - $litExistant (nrMvt)
    := EXISTS[m:MVT | m.nrMvt = nrMvt]

- **Actions**
  - * birth (litOr, p, med, date, heure)
    ⇒ nrMvt := $nrMvtSuiv
    ⇒ dateMvt := birth.date
    ⇒ heureMvt := birth.heure
    ⇒ effectuéPar := birth.pat
    ⇒ demandéPar := birth.med
    ⇒ de := birth.litOr
    ⇒ typeMvt = SORTIE
    ⇒ état := occupé
  - ! $incNrMvtSuiv
    ⇒ $NrMvtSuiv := $nrMvtSuiv+1
    ⇒ état := libre

BEHAVIOUR



12. OBJET REMB_CH

INTERFACE

- **Observations**
  - concerneCat : CAT
    {? état=libre}
  - catFaitPar : ORG_ASS
    {? état=libre}
  - prixRemb : NAT
    {? état=libre}
  - $rembChExistant (cat : CAT, oA : ORG_ASS) : BOOL
    {? état=libre}

- **Keys**
  - $remboursementChambre (cat : CAT, oA : ORG_ASS) : REMB_CH
    {? état=libre}
    {? $rembChExistant (cat, oA)}
- **Events**
  - * birth (cat : CAT, oA : ORG_ASS, prix : NAT)

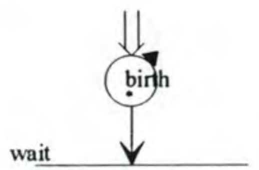
BODY

- **Attributes**
  - concerneCat, catFaitPar, prixRemb
  - état : STATE_DT

- **Derivations**
  - $remboursementChambre (cat, oA)
    := ONE[rCh : REMB_CH | rCh.concerneCat = cat AND rCh.catFaitPar = oA]
  - $rembChExistant (cat, oA)
    := EXISTS[rCh : REMB_CH | rCh.concerneCat = cat AND rCh.catFaitPar = oA]

- **Actions**
  - * birth (cat, oA, prix)
    ⇒ concerneCat := birth.cat
    ⇒ catFaitpar := birth.oA
    ⇒ prixRemb := birth.prix
    ⇒ état := libre

BEHAVIOUR



13. OBJET REMB_PR

INTERFACE

- **Observations**
  - concernePr : PREST
    {? état=libre}
  - prFaitPar : ORG_ASS
    {? état=libre}
  - prixRemb : NAT
    {? état=libre}
  - $rembPrExistant (prest : PREST, oA : ORG_ASS) : BOOL

{? état=libre}

- **Keys**
  - $remboursementPrestation (prest : PREST, oA : ORG_ASS) : REMB_CH
    {? état=libre}
    {? $rembPrExistant (prest, oA)}
- **Events**
  - * birth (prest : PREST, oA : ORG_ASS, prix : NAT)

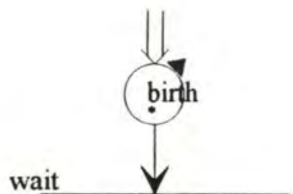
BODY

- **Attributes**
  - concernePr, prFaitPar, prixRemb
  - état : STATE_DT

- **Derivations**
  - $remboursementPrestation (prest, oA)
    := ONE[rPr : REMB_PR | rPr.concernePr = prest AND rPr.prFaitPar = oA]
  - $rembPrExistant (prest, oA)
    := EXISTS[rPr : REMB_PR | rPr.concernePr = prest AND rPr.prFaitPar = oA]

- **Actions**
  - * birth (prest, oA, prix)
    ⇒ concernePr := birth.prest
    ⇒ prFaitpar := birth.oA
    ⇒ prixRemb := birth.prix
    ⇒ état := libre

BEHAVIOUR



14. OBJET ORG_ASS

INTERFACE

- **Observations**
  - nrOA : NAT
    {? état=libre}
  - adrOA : STRING
    {? état=libre}
  - nomOA : STRING
    {? état=libre}
  - rembourseCat : LIST of REMB_CH
    {? état=libre}
  - assure : LIST of AFFIL
    {? état=libre}
  - remboursePr : LIST of PREST
    {? état=libre}
  - $oAExistant (nrOA : NAT) : BOOL
    {? état=libre}

- **Keys**
  - $organismeAssureur (nrOA : NAT) : ORG_ASS
    {? état=libre}
    {? $oAExistant (nrOA)}
- **Events**
  - * birth (nrOA : NAT, adrOA : STRING, nomOA : STRING)

BODY

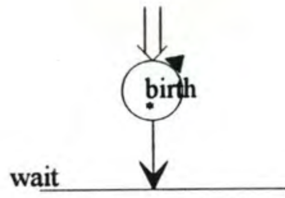
- **Attributes**
  - nrOA, adrOA, nomOA
  - état : STATE_DT

- **Derivations**
  - $organismeAssureur(nrOA)
    := ONE[oA : ORG_ASS | oA.nrOA = nrOA]
  - $oAExistant (nrOA)
    := EXISTS[oA : ORG_ASS | oA.nrOA = nrOA]
  - rembourseCat
    := ALL[rCh : REMB_CH | rCh.catFaitPar=SELF]

- assure
  := ALL[aff : AFFIL | aff.appartientA=SELF]
- remboursePr
  := ALL[rPr : REMB_PR | rPr.prFaitPar=SELF]

- **Actions**
  - * birth (nrOA, adrOA, nomOA)
    ⇒ nrOA := birth.nrOA
    ⇒ adrOA := birth.adrOA
    ⇒ nomOA := birth.nomOA
    ⇒ état := libre

BEHAVIOUR



15. OBJET SER_TECH

Rem : IS-A SER

INTERFACE

- **Observations**
  - nom : STRING
    {? état=libre}
  - $serTechExistant (nom : STRING) : BOOL
    {? état=libre}
  - effPrExt : LIST of EFF_PREST_EXT
    {? état=libre}

- **Keys**
  - $serviceTechnique (nom : STRING) : SER_TECH
    {? état=libre}
    {? $serTechExistant (nom)}

- **Events**
  - * birth (nom : STRING)

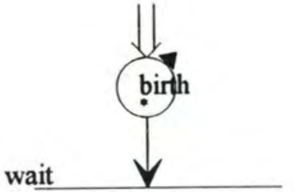
BODY

- **Attributes**
  - nom : STRING
  - état : STATE_DT

- **Derivations**
  - $serviceTechnique (nom)
    := ONE[sT : SER_TECH | sT.nom = nom]
  - $serTechExistant (nom)
    := EXISTS[sT : SER_TECH | sT.nom = nom]
  - effPrExt
    := ALL[eff : EFF_PREST_EXT | eff.parExt = SELF]

- **Actions**
  - * birth (nom)
    ⇒ nom := birth.nom
    ⇒ état := libre

BEHAVIOUR



16. OBJET AFFIL

INTERFACE

- **Observations**
  - appartientA : ORG_ASS
  - affilie : PAT
  - nrTit : NAT
  - typeAffil : TYPE_AFFIL_DT
  - $affilExistante (oA, pat) : BOOL

- **Keys**
  - $affiliation (oA, pat) : AFFIL
    ? $affilExistante (oA, pat)

- **Events**
  - * birth (pat, oA, nrTit, typeAffil)

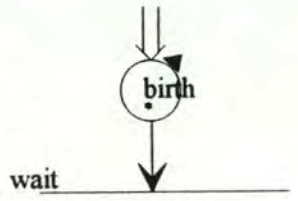
BODY

- **Attributes**
  - appartientA, affilie, nrTit, typeAffil

- **Derivations**
  - $affiliation (oA, pat)
    := ONE[aff : AFFIL | aff.appartientA=oA AND aff.affilie=pat]
  - $affilExistante (oA, pat)
    := EXISTS[aff : AFFIL | aff.appartientA=oA AND aff.affilie=pat]

- **Actions**
  - * birth (pat, oA, nrTit, typeAffil)
    ⇒ affilie := birth.pat
    ⇒ appartientA := birth.oA
    ⇒ nrTit := birth.nrTit
    ⇒ typeAffil := birth.typeAffil

BEHAVIOUR



17. OBJET PAT

INTERFACE

- **Observations**
  - nrDos : NAT
  - coord : COORD_DT
  - nom, prenom : STRING
  - dateNaiss : DATE
  - adrPat, tél : STRING
  - etatCivil : ETAT_CIVIL_DT
  - sexe : SEX_DT
  - estAffilié : AFFIL
  - subit : LIST of PRESCR_SOIN
  - accomplit : LIST of MVT
  - $patExistantParCoord (coord : COORD_DT) : BOOL
  - $patExistantParNrDos (nrDos : NAT) : BOOL
  - patDansSer(serMed : SER_MED) : BOOL
  - oADePat : ORG_ASS
  - listeMvtDates (dateD, heureD, DateF, heureF) :
    CP[listesMvtDates : LIST of MVT, nbHosp : NAT]
  - listePrescrDates (dateD, heureD, DateF, heureF) : LIST of PRESCR_SOIN
  - patMalade : BOOL
  - litDePat : LIT
  - patDansLit (lit : LIT) : BOOL

- **Keys**
  - $patParCoord (coord : COORD_DT) : PAT
    ? $patExistantParCoord (coord)
  - $patParNrDos (nrDos : NAT) : PAT

? $patExistantParNrDos (nrDos)

- **Events**
  - * birth (coord : COORD_DT)
  - fixeAdr (adr : STRING)
  - fixeTel (tel : STRING)
  - fixeEtatCivil (etatCivil : ETAT_CIVIL_DT)
  - fixeSexe (sexe : SEX_DT)

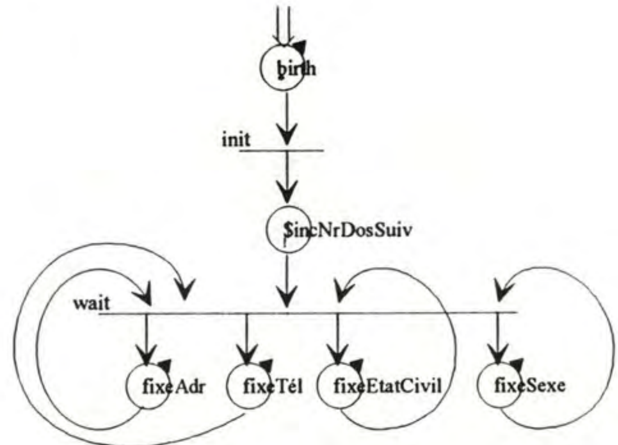
BODY

- **Attributes**
  - nrDos, coord, adr, etatCivil, sexe, tel, $nrDosSuiv : NAT

- **Derivations**
  - estAffilié
    := ONE[aff : AFFIL | aff.affilie=SELF]
  - subit
    := ALL[pS : PRESCR_SOIN | pS.subiePar = SELF]
  - accomplit
    := ALL[m : MVT | m.effectuéPar = SELF]
  - nom, prénom, dateNaiss
    := coord.nom, .prénom, .dateNaiss
  - $patExistantParCoord/NrDos (coord/nrDos)
    := EXISTS[p : PAT | p.coord/nrDos = coord/nrDos]
  - patDansSer (serMed)
    := SELF.litDePat.serDeLit = serMed
  - oADePat
    := SELF.estAffilié.appartientA
  - listeMvtDates (...)
    := listeMvtDates := ALL[m : MVT | m in SELF.accomplit AND dateF >= m.dateMvt > dateD AND heureF >= m.heureMvt > heureD] SORTED ON (dateMvt, heureMvt)
    := nbHosp := #(ALL[m : MVT | m in listeMvtDates AND (CLASS_OF (m) = "MVT_ENT" OR CLASS_OF (m) = "MVT_SOR")]) DIV 2
  - listePrescrDates (...)
    := listePrescrDates := ALL[pS : PRESCR_SOIN | pS in SELF.subit AND dateF >= pS.datePr > dateD AND heureF >= pS.heurePr > heureD AND pS.enAttente= FALSE] SORTED ON (datePr, heurePr)
  - patMalade
    := EXISTS[l : LIT | l.litOccupéPar = SELF]
  - litDePat
    := ONE[l : LIT | l.litOccupéPar = SELF]
  - patDansLit (lit)
    := lit.litOccupéPar = SELF
  - $patParCoord/NrDos (coord/nrDos)
    := ONE[p : PAT | p.coord/nrDos = coord/nrDos]

- **Actions**
  - * birth (coord)
    ⇒ coord := birth.coord
    ⇒ nrDos := $nrDosSuiv
  - $incNrDosSuiv
    ⇒ $nrDosSuiv := $nrDosSuiv + 1
  - fixeAdr (adr)
    ⇒ adr := fixeAdr.adr
  - fixeTel (tel)
    ⇒ tel := fixeTel.tel
  - fixeEtatCivil (EtatCivil)
    ⇒ EtatCivil := fixeEtatCivil.EtatCivil
  - fixeSexe (sexe)
    ⇒ Sexe := fixeSexe.Sexe

BEHAVIOUR



18. OBJET MED

INTERFACE

- **Observations**
  - nrMed : NAT
  - nom, prenom : STRING
  - adr : STRING
  - aPrescrit : LIST of PRESCR_SOIN
  - demande : LIST of MVT
  - $medExistant (nrMed : NAT) : BOOL

- **Keys**
  - $médecin (nrMed : NAT) : MED
    ? $medExistant (nrMed)

- **Events**
  - * birth (nom, prenom adr : STRING)

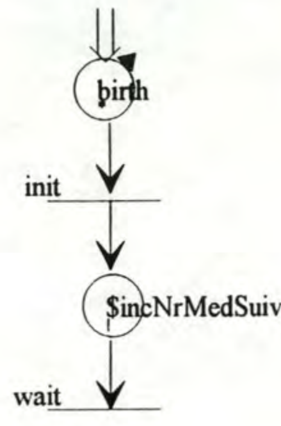
BODY

- **Attributes**
  - nrMed, nom, prenom, adr,$nrMedSuiv : NAT

- **Derivations**
  - aPrescrit
    := ALL[pS : PRESCR_SOIN | pS.prescritePar = SELF]
  - demande
    := ALL[m : MVT | m.demandéPar = SELF]
  - $medExistant(nrMed)
    := EXISTS [med : MED | med.nrMed = nrMed]
  - $médecin (nrMed)
    := ONE[med : MED | med.nrMed = nrMed]

- **Actions**
  - * birth (...)
    ⇒ nom; prenom, adr := birth.nom, prenom, adr
    ⇒ nrMed := $nrMedSuiv
  - $incNrMedSuiv
    ⇒ $nrMedSuiv := $nrMedSuiv + 1

BEHAVIOUR



19. OBJET PRESCR_SOIN

INTERFACE

- **Observations**
  - nrPr : NAT
  - datePr/heurePr : DATE/TIME
  - enAttente : BOOL
  - subitePar : PAT
  - prescritePar : MED
  - pourPrest : PREST
  - $prescrExistante (nrPr : NAT) : BOOL

- **Keys**
  - $prescriptionSoin (nrPr : NAT) : PRESCR_SOIN
    ? $prescrExistante (nrPr)

- **Events**
  - * birth (pat, med, prest, date, heure)
  - fixeEtat (etat)
  - + death

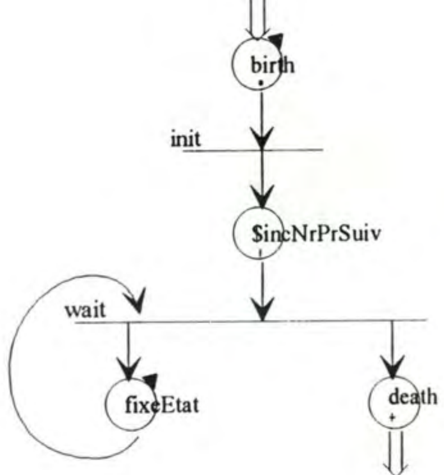
Body

- **Attributes**
  - nrPr, datePr, heurePr, etatPr : ETAT_PRESCR_DT, subitePar, prescritePar, pourPrest, $nrPrSuiv : NAT

- **Derivations**
  - $prescrExistante(nrPr)
    := EXISTS [pS : PRESCR_SOIN | pS.nrPrescr = nrPr]
  - $prescriptionSoin (nrPr)
    := ONE[pS : PRESCR_SOIN | pS.nrPrescr = nrPr]

- **Actions**
  - * birth (...)
    ⇒ etc. := birth.etc.
    ⇒ nrPr := $nrPrSuiv
  - $incNrPrSuiv
    ⇒ $nrPrSuiv := $nrPrSuiv + 1
  - fixeEtat (etat)
    ⇒ etatPr := fixeEtat.etat
  - death

Behaviour



20. OBJET PREST

Interface

- **Observations**
  - codePrest : STRING
  - nomPr : STRING
  - typePrest : TYPE_PREST_DT
  - capMaxi, prixPleinPr : NAT
  - prRemboursée : LIST of REMB_PR
  - prescriteDans : LIST of PRESCR_SOIN
  - $prestExistante (code) : BOOL

- **Keys**
  - $prestation (code) : PREST
    ? $prestExistante (code)

- **Events**
  - * birth (code, nom, type, capMaxi, prix)

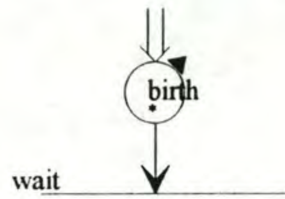
Body

- **Attributes**
  - codePrest, nomPrest, typeprest, capMaxi, prixPlein

- **Derivations**
  - $prestExistante(code)
    := EXISTS [prest : PREST | prest.codePrest = code]
  - $prestation (code)
    := ONE[prest : PREST | prest.codePrest = code]
  - prRemboursée
    := ALL[rPr : REMB_PR | rPr.concernePr=SELF]
  - prescriteDans
    := ALL[pS : PRESCR_SOIN | pS.pourPrest=SELF]

- **Actions**
  - * birth (...)
    ⇒ etc. := birth.etc.

BEHAVIOUR



21. OBJET PREST_EXT

INTERFACE

- **Observations**
  - codePrest : STRING
  - nomPr : STRING
  - typePrest : TYPE_PREST_DT
  - capMaxi, prixPleinPr : NAT
  - prRemboursée : LIST of REMB_PR
  - prescriteDans : LIST of PRESCR_SOIN
  - $prestExistante (code) : BOOL
  - prExtEff : EFF_PREST_EXT
  - serDePrest : SER_MED

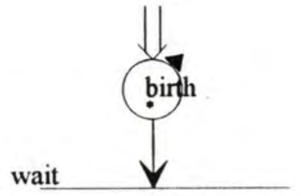
- **Keys**
  - $prestation (code) : PREST
    ? $prestExistante (code)
- **Events**
  - * birth (code, nom, type, capMaxi, prix)

BODY
- **Attributes**
  - codePrest, nomPrest, typeprest, capMaxi, prixPlein

- **Derivations**
  - $prestExistante(code)
    := EXISTS [prest : PREST | prest.codePrest = code]
  - $prestation (code)
    := ONE[prest : PREST | prest.codePrest = code]
  - prRemboursée
    := ALL[rPr : REMB_PR | rPr.concernePr=SELF]
  - prescriteDans
    := ALL[pS : PRESCR_SOIN | pS.pourPrest=SELF]
  - prExtEff
    := ONE [eff : EFF_PREST_EXT | pourExt=SELF]
  - serDePrest
    := SELF.prExtEff.parExt

- **Actions**
  - * birth (...)
    ⇒ etc. := birth.etc.

BEHAVIOUR



22. OBJET PREST_INT

INTERFACE

- **Observations**
  - codePrest : STRING
  - nomPr : STRING
  - typePrest : TYPE_PREST_DT
  - capMaxi, prixPleinPr : NAT
  - prRemboursée : LIST of REMB_PR
  - prescriteDans : LIST of PRESCR_SOIN
  - $prestExistante (code) : BOOL
  - prIntEff : LIST of EFF_PREST_INT

- **Keys**
  - $prestation (code) : PREST
    ? $prestExistante (code)

- **Events**
  - * birth (code, nom, type, capMaxi, prix)

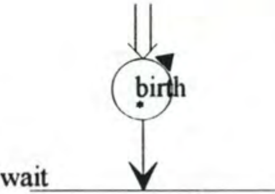
BODY

- **Attributes**
  - codePrest, nomPrest, typeprest, capMaxi, prixPlein

- **Derivations**
  - $prestExistante(code)
    := EXISTS [prest : PREST | prest.codePrest = code]
  - $prestation (code)
    := ONE[prest : PREST | prest.codePrest = code]
  - prRemboursée
    := ALL[rPr : REMB_PR | rPr.concernePr=SELF]
  - prescriteDans
    := ALL[pS : PRESCR_SOIN | pS.pourPrest=SELF]
  - prExtEff
    := ALL [eff : EFF_PREST_INT | pourInt=SELF]

- **Actions**
  - * birth (...)
    ⇒ etc. := birth.etc.

BEHAVIOUR



23. OBJET EFF_PREST_INT

INTERFACE

- **Observations**
  - pourInt : PREST_INT
  - parInt : SER_MED
  - nbUnitesExec : NAT

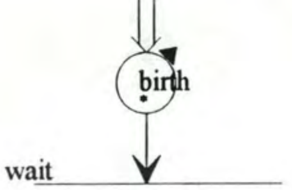
- **Keys**
- **Events**
  - * birth (serMed : SER_MED, prestInt : PREST_INT, nbExec : NAT)

BODY

- **Attributes**
  - pourInt, parInt, nbUnitesExec

- **Actions**
  - * birth (serMed, prestInt, nbExec)
    ⇒ parInt := birth.serMed
    ⇒ pourInt := birth.prestInt
    ⇒ nbUnitesExec := birth.nbExec

BEHAVIOUR



24. OBJET EFF_PREST_EXT

INTERFACE

- **Observations**
  - pourExt : PREST_EXT
  - parExt : SER_TECH
  - nbUnitesExec : NAT

- **Keys**
- **Events**
  - * birth (serTech : SER_TECH, prestExt : PREST_EXT, nbExec : NAT)

BODY

- **Attributes**
  - pourExt, parExt, nbUnitesExec

- **Actions**
  - * birth (serTech, prestExt, nbExec)
    ⇒ parExt := birth.serTech
    ⇒ pourExt := birth.prestExt
    ⇒ nbUnitesExec := birth.nbExec

BEHAVIOUR