Application de la méthode TAG/CF à l'exemple de l'hôpital

Pascal André et Jean-Claude Royer

22 septembre 1997

Résumé

Nous présentons dans ce document une utilisation de la méthode TAG/CF de spécification formelle et de conception à objets. Nous l'appliquons à un exemple simple de système : les entrées et les sorties d'un hôpital. Après une présentation de la spécification du comportement par un automate nous calculons les axiomes de la spécification algébrique. Une version du résultat écrite avec Asspegique permet d'illustrer la proximité syntaxique et des exemples de preuves et d'évaluations. Nous montrons également comment l'automate du comportement peut servir à guider une preuve. La conception à objets d'une telle spécification utilise un niveau intermédiaire : le modèle des classes formelles. Nous illustrons les étapes de la conception en classes formelles et décrivons deux voies possibles. Finalement nous ébauchons la production d'une prototype dans un langage à objets et donnons la conception de notre exemple en C++.

MOTS-CLÉS : spécification algébrique, conception à objets, Type Abstrait Graphique, Classe Formelle, langage à objets, C++, Asspegique.

Abstract

This paper illustrates the TAG/CF method: an object-oriented method allowing formal specification and design. We experiment the method on a simple system: the inputs and outputs of an hospital. Firstly, we describe the behavioural specification based on a guarded automaton. Secondly, we extract from this description an algebraic specification. Using an Asspegique translation of that specification we illustrate symbolic evaluations and proofs. We also show how to use the automaton to guide a proof. The object-oriented design needs an intermediate level: the formal class model. We describe several design with the formal class model. Lastly, we comment the production of an object prototype and a C++ implementation.

KEYWORDS: algebraic specification, object-oriented design, Graphic Abstract Data Type, Formal Class, object-oriented language, C++, Asspegique.

Application de la méthode TAG/CF à l'exemple de l'hôpital

Pascal André et Jean-Claude Royer[1]


Equipe Génie Logiciel, Méthodes et Spécifications formelles
Institut de Recherche en Informatique de Nantes
Université & Ecole Centrale
2, rue de Houssinière
BP 929208
44322 Nantes Cédex 03
22 septembre 1997

1. Ce travail est soutenu par le GDR Programmation du CNRS et une première version a été présenté aux journées du GDR à Lille le 22 septembre 1994.

Table des matières

1 Introduction

Les méthodes d'analyse et de conception à objets évoluent rapidement, de nombreux progrès ont été fait ces dernières années dans ce domaine. Il est depuis longtemps reconnu que différentes phases sont nécessaires à un bon développement. Aujourd'hui le modèle à objets s'est imposé dans toutes les phases du développement. On admet également qu'un problème doit être décrit sous divers aspects complémentaires. Les méthodes courantes proposent trois plans de description : le modèle dynamique, le modèle fonctionnel et le modèle structurel. Un pas important dans ce domaine est d'introduire des aspects formels dans le cycle de développement à objets. L'utilisation des spécifications formelles dans un environnement réel de développement se heurte à plusieurs problèmes. Le principal étant, pour beaucoup d'utilisateur, la difficulté d'écrire *ex nihilo* une spécification formelle. D'autre part le raffinage d'une spécification en une mise en œuvre efficace et réutilisable n'est en général pas trivial.

Nous proposons l'introduction des concepts de type abstrait graphique et de classe formelle (TAG et CF respectivement) qui sont une réponse partielle à ces problèmes. Nous nous sommes limités aux systèmes séquentiels.

Notre approche est une approche multi-langages : langage de spécification, langage de conception abstraite et langage de codage. Une des sources d'incohérences et de difficultés dans le développement logiciel est qu'il n'y a pas de passerelle facile ni rigoureuse entre les différents plans de description d'un système. Un seul langage couvrant à la fois les différentes phases et les différents plans n'est pas réaliste. Nous avons plutôt privilégié des langages avec plusieurs facettes (automate/spécification algébrique, type abstrait de données/classe). Il y a un manque de formalisme aux niveaux des spécifications et du raffinage vers la conception mais également au niveau de la comparaison de composants (indispensable pour une bonne réutilisation). Le succès d'une méthode à objets telle qu'OMT (Object Modeling Technique [RBP+91]) semble beaucoup dû à la liberté qui règne dans l'écriture des spécifications. L'introduction d'aspects formels va contraindre les différents acteurs du développement. Ces aspects doivent donc s'intégrer dans le processus de développement classique et être un plus plutôt qu'un bouleversement des habitudes. Ces contraintes doivent être introduites d'une façon naturelle et progressive et avec un support logiciel riche. On doit proposer des outils d'édition, de preuves, de simplification, d'évaluation, de gestion, ... bref un bon environnement support. Un tel environnement est à l'étude et certains outils sont implantés [AR95b]. Nous allons plutôt présenter les concepts introduits et les liens entre eux. Notre but n'est pas de définir une nouvelle méthode d'analyse et conception mais des concepts et des techniques qui peuvent s'intégrer dans une méthode à objets.

2 Présentation des concepts

Nous résumons les principaux impératifs souhaités.

- Nous désirons introduire des spécifications formelles de type abstrait. L'intérêt pour les spécifications formelles est classique [ST90, CG88]. Les types abstraits algébriques (TAA) sont choisis parce qu'il sont plus proches des classes que d'autres formalismes (Z, VDM, ...) [Hor78, EM85, GL90, Bid82, AR92b].

- Nous souhaitons faciliter l'écriture des spécifications. Dans la plupart des méthodes actuelles l'analyste élabore une description du comportement dynamique du système. Nous introduirons la notion de type abstrait graphique basé sur un automate avec gardes.

– Nous désirons faire le lien entre les différents plans de description surtout entre le modèle dynamique et le modèle fonctionnel. Le TAG est d'abord un modèle dynamique du composant mais également un modèle fonctionnel exprimé par une spécification algébrique de type de données. L'association d'aspects dynamiques à une spécification algébrique fait l'objet d'autres travaux plus théoriques notamment [Ehr94].

– Nous voulons nous dégager du niveau trop concret des langages à objets. Pour cela nous avons défini un langage de spécification pour les classes (le modèle des classes formelles [ACR94]). Ce langage est d'une implantation aisée dans un langage à objets avec classes.

– Nous souhaitons que la spécification du système et sa conception soit aussi automatisée que possible. Des techniques d'aides à l'écriture, au raffinage, à la conception et à l'optimisation doivent être étudiées.

Classiquement un composant a une vue externe et une vue interne. Nous adoptons également ce principe : la vue externe est le TAG et la vue interne est double : classes formelles et une (ou plusieurs) mise en œuvre en classes concrètes (CC). L'organisation générale des concepts est la suivante :

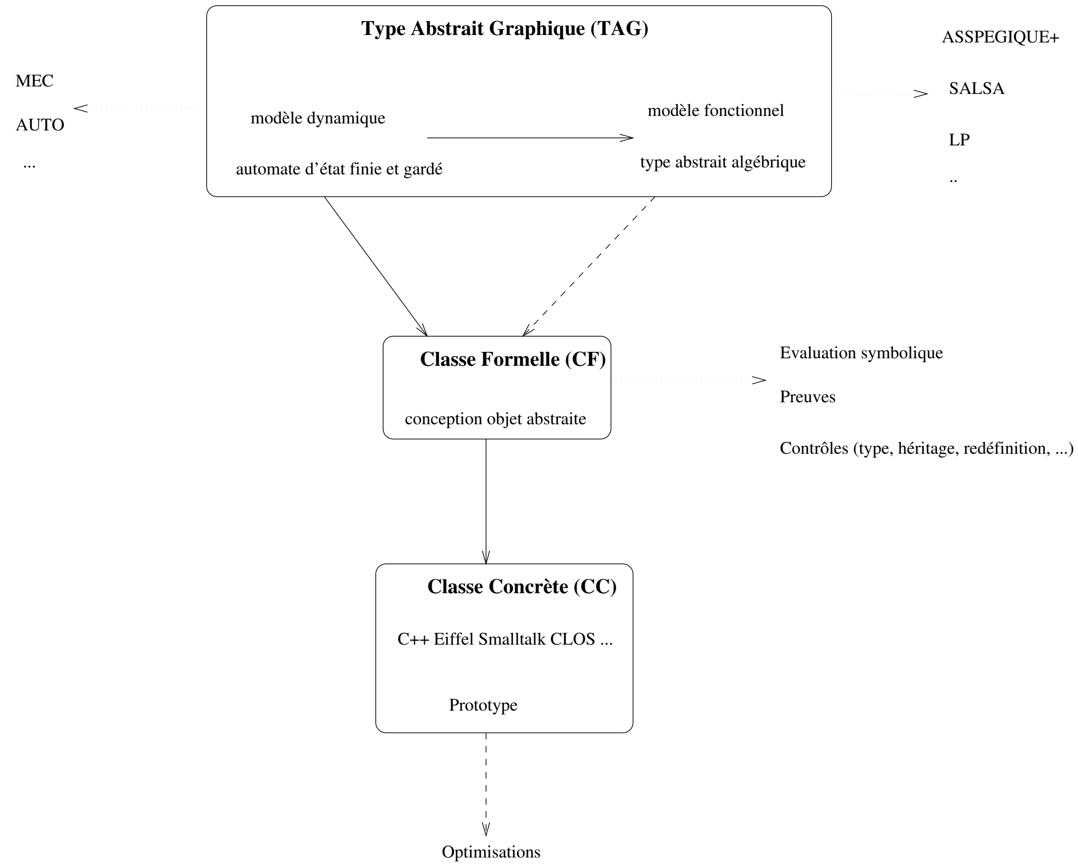

Figure 1 : Concepts et organisation générale

Les arcs pleins correspondent aux transformations étudiées en grande partie, les autres sont en cours d'études. On remarquera l'ouverture possible vers d'autres environnements (de spécifications, de preuves, de réécritures, d'automates, ...) Nous illustrerons cette possibilité par la liaison avec le système Asspegique+ [BCC+89], un autre exemple utilise Larch Prover [AR97]. Le TAG est le niveau le plus abstrait, il est le résultat d'une analyse et d'une formalisation préliminaire et incomplète. Il peut, par exemple, résulter d'une transformation des schémas OMT du système. La transformation des schémas OMT en TAG a été étudiée partiellement dans [ABR95]. Nous ne décrivons pas ici l'aspect «trouver les TAG» qui est typiquement du domaine de l'analyse. La classe formelle décrit une conception objet et abstraite du TAG. La classe concrète décrit une mise en œuvre dans un langage à classes. Nous illustrerons ces concepts par la spécification et la conception d'un système simplifié d'entrées dans un service hospitalier.

2.1 Cahier des charges

Nous désirons simuler un service hospitalier simplifié. Il y a des entrées normales (`admit`) limités à un nombre maximum (`number`) défini à l'initialisation. Il y a des entrées urgentes (`urgency`) non limitées. La notion d'urgence est liée à l'entrée à l'hopital et pas associée au patient. Les urgences sont traitées en priorité (`cure`). La prochaine personne à traiter est connue (`first`). On peut connaître le nombre d'entrées normales, le nombre courant d'urgences et le nombre total d'entrées.

3 Introduction aux TAG

Nous avons unifié la vue dynamique d'un composant (un automate gardé) et la vue fonctionnelle (spécification de TAA), pour plus de détails sur le modèle consulter [AR95a, AR95b]. De plus nous voulons faciliter la définition et l'écriture du TAG.

3.1 Modèle dynamique

Le «spécifieur» commence par donner une description du modèle dynamique sous forme d'un automate gardé. Ce style a été choisi pour sa puissance d'expression, sa proximité avec ce qui se fait en pratique (RDPO, SADT, automates, statecharts ...) et surtout sa double interprétation possible. Cette double interprétation est d'une part un système d'état fini et d'autre part un type de données. Un état correspond à une étape dans le cycle de vie et à un sous-type du type de données. Une transition correspond à une interruption par événement et à un appel d'opération. Nous distinguons les opérations qui changent d'état (constructeurs) des opérations qui observent l'état (observateurs).

Le premier travail de spécification (voir la figure 2) est de définir les états du système, les transitions entre ces états, les gardes associées et des informations complémentaires : constantes, variables de types ... Un état représente une situation de l'hôpital (vide, plein ou entre les deux). On distingue les états normaux de ceux où l'hôpital contient des urgences. Une transition indique un changement d'état dû à une action ou un événement (admission d'un patient ou d'une urgence, soin, ...).

Des hypothèses naturelles sur les TAG ont été définies : connexité, gardes exclusives, états initiaux, états non vides et finiment générés, pas de concurrence intra-objet. Dès ce stade des vérifications de l'automate sont possibles : vivacité, terminaison, interblocage. L'interfaçage avec des systèmes comme Auto, Mec, ... pour faire des vérifications plus spécifiques peut être intéressants. Nous avons étudié des moyens pour réduire l'explosion combinatoire (regroupement d'états, produit d'automates, transitions agrégées : opérateur séquentiel, collatéral et alternative simple).

3.2 Modèle fonctionnel

La seconde phase va être de produire le modèle fonctionnel sous forme d'un TAA. Au lieu d'écrire directement cette spécification le système exploite les informations de l'automate pour aider l'utilisateur

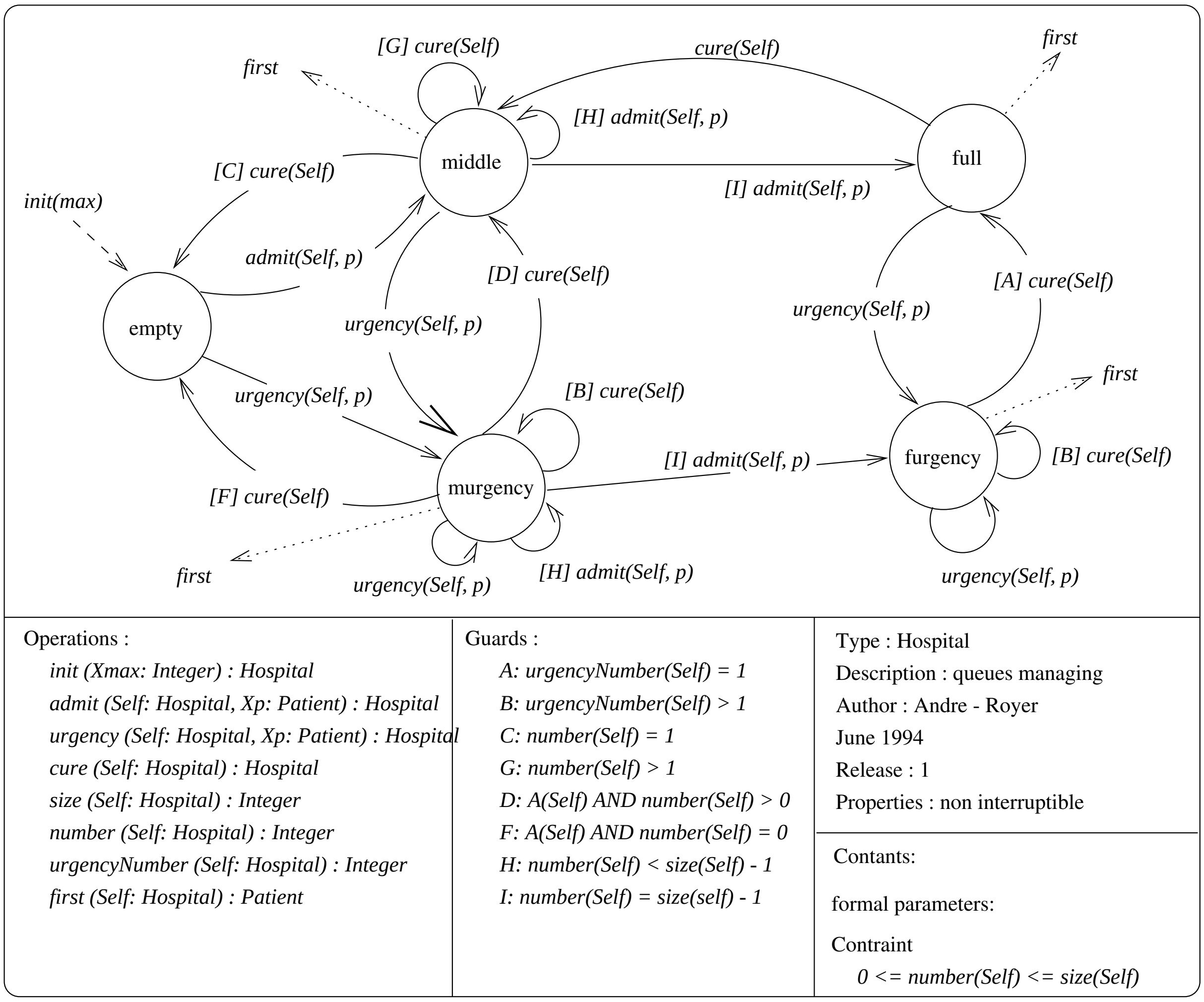

Figure 2 : L'automate du TAG de `Hospital`

dans sa tâche. L'idée est de considérer une transition comme une opération partielle qui transforme une valeur de l'état de départ en une valeur de l'état d'arrivée. Nous avons défini des algorithmes permettant d'obtenir automatiquement l'extraction de certaines informations. En particulier l'extraction de la signature est triviale.

3.2.1 Classification des opérations

La première action que l'on doit faire est de partitionner les opérations en générateurs, observateurs primitifs et opérations secondaires. Tout ceci se fait *a priori*, guidé par l'intuition ou l'expérience de l'utilisateur. Une preuve est possible une fois que les axiomes sont complètement écrit. En pratique il semble bien que cette preuve soit souvent possible et nous n'avons pas constaté de remise en cause de la partition initiale. L'extraction des générateurs est fondé sur l'idée que l'on doit pouvoir atteindre chaque état (au moins une fois) à partir de la famille génératrice [2]. Un algorithme de parcours de graphe fournit les différents familles possibles et l'utilisateur en choisit une. Des critères de taille de la famille, du nombre

2. Cette condition est nécessaire et suffisante pour que chaque état possède au moins une valeur finiment générée (si les gardes sont $\not\equiv false$).

de constructeurs de base, ... peuvent être introduits en plus. Nous obtenons ici deux familles: {**`init, admit, cure, urgency`**} et {**`init, admit, urgency`**}. De façon évidente nous choisissons la seconde ce qui nous donne l'automate réduit aux générateurs de la figure 3.

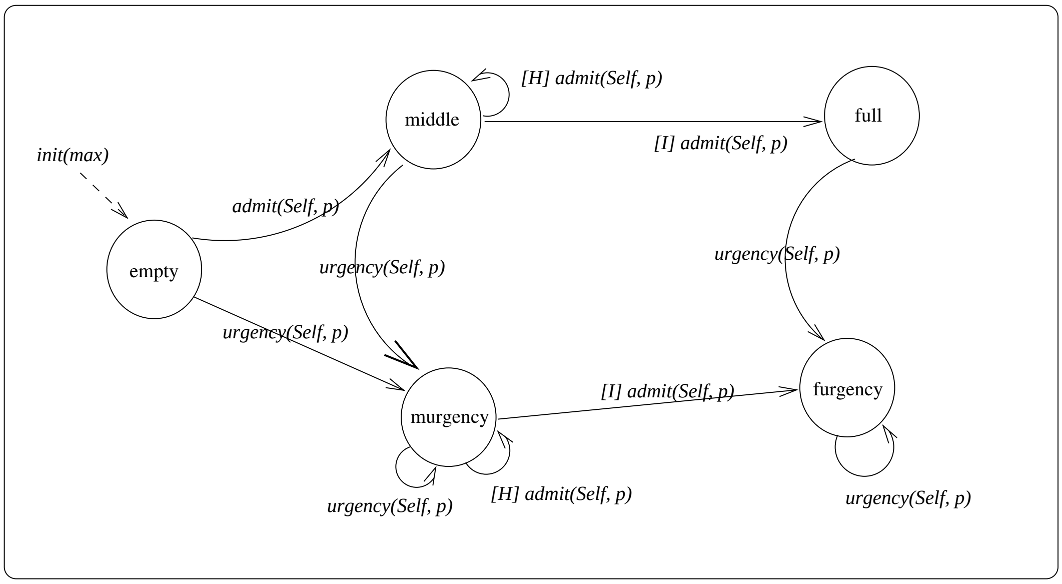

Figure 3 : L'automate réduit aux générateurs

Pour la détermination des observateurs primitifs nous ne disposons pas de principe aussi agréable mais de quelques heuristiques. En voici quelques unes : même type résultat, même domaine de définition, les deux, complémentarité du domaine de définition. Une étude précise reste à faire dans ce domaine.

Nous choisissons ici **`size, number, urgencyNumber, first`**.

3.2.2 Prédicats d'états et pré-conditions des opérations

Les informations suivantes sont extraites automatiquement du TAG. Pour les prédicats d'état, nous étudions leur valeur sur chaque générateur du type. Par exemple,

```
middle1: middle(init(Xmax)) == false
middle2: middle(admit(Self, Xp)) == empty(Self) OR [middle(Self) AND H(Self)]
middle3: middle(urgency(Self, Xp)) == false.
```

Pour les pré-conditions, nous calculons pour chaque opération une expression formée des prédicats d'état (pour les états où l'opération est définie) et des gardes correspondantes.

```
precondition(admit(Self,Xp)) = empty(Self) OR middle(Self) OR murgency(Self)
```

Notons la nécessité d'un simplifieur d'expressions ou d'un prouver, c'est utile dans de nombreux cas. Par exemple les propriétés de complémentarité et d'exclusivité des gardes permettent de simplifier les pré-conditions et les axiomes des opérations.

3.2.3 Extraction des axiomes

La difficulté principale dans un TAA est d'extraire les axiomes, pour cela il n'existe bien évidemment pas d'algorithmes. Néanmoins nous avons défini des algorithmes avec interactions de l'utilisateur pour produire un ensemble d'axiomes conduisant *a priori* à une bonne spécification. Nous pensons que ceci est un apport très intéressant en pratique car il automatise, en partie, et simplifie l'effort de l'utilisateur.

Nous nous sommes basés sur les méthodes usuelles de spécification de type de données [Hor78, Mus80, Bid82, GL90, BB93].

3.2.4 Génération des axiomes

Nous générons des axiomes conditionnels qui, *a priori*, respectent de bonnes propriétés comme la non-contradiction ou la complétude suffisante. Nous produisons toujours un ensemble d'axiomes conditionnels pour un opérateur qui décrit tous les termes constructibles sur la famille génératrice. Exemple de `first` :

```
first1: empty(Self) == true ==>  first(admit(Self, Xp)) == Xp
first2: middle(Self) == true AND H(Self) == true ==>  first(admit(Self, Xp)) == first(Self)
first3: middle(Self) == true AND I(Self) == true ==>  first(admit(Self, Xp)) == first(Self)
first4: full(Self) == true ==>  first(urgency(Self, Xp)) == Xp
first5: furgency(Self) == true ==>  first(urgency(Self, Xp)) == first(Self)
first6: murgency(Self) == true AND I(Self) == true ==>  first(admit(Self, Xp)) == first(Self)
first7: empty(Self) == true ==>  first(urgency(Self, Xp)) == Xp
first8: middle(Self) == true ==>  first(urgency(Self, Xp)) == Xp
first9: murgency(Self) == true ==>  first(urgency(Self, Xp)) == first(Self)
first10: murgency(Self) == true AND H(Self) == true ==>  first(admit(Self, Xp)) == first(Self)
```

Cette génération est exhaustive et sans paire critique car basée sur un parcours de l'automate réduit aux générateurs. En supposant que les axiomes sont orientables en règles de réécriture de gauche à droite nous obtenons un système convergent. Il est alors décidable de prouver la suffisante complétude de la spécification.

Le principe de base de la génération des axiomes est appelé une **G-dérivation** et se résume comme suit. Un terme du type d'intérêt est un chemin dans l'automate de comportement. Une G-dérivation est le «tirage» inverse d'une transition dans l'automate réduit aux générateurs (figure 3). C'est donc l'application d'un générateur à l'argument privilégié `Self`. Connaître l'effet d'une opération sur un terme `Self`, du type d'intérêt, se trouvant dans un état donné, c'est analyser toutes les transitions qui mènent dans cet état.

Il existe en réalité plusieurs applications de la G-dérivation : pour les prédicats d'états, pour les axiomes des observateurs primitifs et pour les constructeurs ou observateurs secondaires. Seuls les termes droits doivent être fournis par l'utilisateur. Il est d'ailleurs possible d'envisager une interprétation en langage naturel qui pourrait aider l'utilisateur à répondre. Notons également que des contrôles (syntaxe, type, opérations, chemin dans l'automate, ...) sont possibles sur la réponse de l'utilisateur. Nous obtenons «sans trop de peine» une spécification algébrique avec de bonnes propriétés. A partir de là, l'outillage habituel en spécification algébrique est utilisable : preuves de terminaison, de confluence, d'extensions des opérations secondaires, etc.

3.3 Axiomes du TAG `Hospital`

- Signature du type `Hospital`:

$init$	$: Integer \rightarrow Hospital$	*générateurs*
$admit$	$: Hospital\ Patient \rightarrow Hospital$	
$urgency$	$: Hospital\ Patient \rightarrow Hospital$	
$cure$	$: Hospital \rightarrow Hospital$	*constructeurs secondaires*
$size$	$: Hospital \rightarrow Integer$	*observateurs primitifs*
$number$	$: Hospital \rightarrow Integer$	
$urgencyNumber$	$: Hospital \rightarrow Integer$	
$first$	$: Hospital \rightarrow Patient$	
$totalNumber$	$: Hospital \rightarrow Integer$	*observateurs secondaires*
$empty?$	$: Hospital \rightarrow Boolean$	*observateurs d'état*
$middle?$	$: Hospital \rightarrow Boolean$	
$full?$	$: Hospital \rightarrow Boolean$	
$murgency?$	$: Hospital \rightarrow Boolean$	
$furgency?$	$: Hospital \rightarrow Boolean$	
A	$: Hospital \rightarrow Boolean$	*gardes et invariant*
...		
I	$: Hospital \rightarrow Boolean$	
NE	$: Hospital \rightarrow Boolean$	
$invariant$	$: Hospital \rightarrow Boolean$	

- Prédicats généraux :

```
gardes
        A(Self) == urgencyNumber(Self) = 1
        B(Self) == urgencyNumber(Self) >  1
        C(Self) == number(Self) = 1
        D(Self) == A(Self) AND number(Self) >  0
        F(Self) == A(Self) AND number(Self) = 0
        G(Self) == number(Self) >  1
        H(Self) == number(Self) < size(Self) - 1
        I(Self) == number(Self) = size(Self) - 1

NE(Self) == middle?(Self) OR full?(Self) OR murgency?(Self) OR furgency?(Self)

invariant(Self) == true.
```

- Preconditions des operations (axiomes simplifiés).

```
precondition(init(Xmax)) == true
precondition(size(Self)) == true
precondition(number(Self)) == true
precondition(urgencyNumber(Self)) == true
precondition(first(Self)) == NE(Self)
precondition(cure(Self)) == NE(Self)
precondition(admit(Self,Xp)) == empty?(Self) OR middle?(Self) OR murgency?(Self)
precondition(urgency(Self,Xp)) == true
```

• Observateurs d'états

```
empty1: empty?(init(Xmax)) == true
empty2: empty?(admit(Self, Xp)) == false
empty3: empty?(urgency(Self, Xp)) == false
middle1: middle?(init(Xmax)) == false
middle2: middle?(admit(Self, Xp)) == empty?(Self) OR [middle?(Self) AND H(Self)]
middle3: middle?(urgency(Self, Xp)) == false
murgency1: murgency?(init(Xmax)) == false
murgency2: murgency?(admit(Self, Xp)) == murgency?(Self) AND H(Self)
murgency3: murgency?(urgency(Self, Xp)) == empty?(Self) OR middle?(Self) OR
                       murgency?(Self)
full1: full?(init(Xmax)) == false
full2: full?(admit(Self, Xp)) == middle?(Self) and I(Self)
full3: full?(urgency(Self, Xp)) == false
furgency1: furgency?(init(Xmax)) == false
furgency2: furgency?(admit(Self, Xp)) == murgency?(Self) and K(Self)
furgency3: furgency?(urgency(Self, Xp)) == full?(Self) OR furgency?(Self)
```

• observateurs primitifs totaux

```
size1: size(init(Xmax)) == Xmax
size2: size(admit(Self, Xp)) == size(Self)
size3: size(urgency(Self, Xp)) == size(Self)
number1: number(init(Xmax)) == 0
number2: number(admit(Self, Xp)) == 1 + number(Self)
number3: number(urgency(Self, Xp)) == number(Self)
urgencyNumber1: urgencyNumber(init(Xmax)) == 0
urgencyNumber2: urgencyNumber(admit(Self, Xp)) == urgencyNumber(Self)
urgencyNumber3: urgencyNumber(urgency(Self, Xp)) == 1 + urgencyNumber(Self)
```

• observateurs primitifs partiels

```
first1: empty?(Self) == true ==>  first(admit(Self, Xp)) == Xp
first2: middle?(Self) == true AND H(Self) == true ==>
                       first(admit(Self, Xp)) == first(Self)
first3: middle?(Self) == true AND I(Self) == true ==>
                       first(admit(Self, Xp)) == first(Self)
first4: full?(Self) == true ==>  first(urgency(Self, Xp)) == Xp
first5: furgency?(Self) == true ==>  first(urgency(Self, Xp)) == first(Self)
first6: murgency?(Self) == true AND K(Self) == true ==>
                       first(admit(Self, Xp)) == first(Self)
first7: empty?(Self) == true ==>  first(urgency(Self, Xp)) == Xp
first8: middle?(Self) == true ==>  first(urgency(Self, Xp)) == Xp
first9: murgency?(Self) == true ==>  first(urgency(Self, Xp)) == first(Self)
first10: murgency?(Self) == true AND H(Self) == true ==>
                       first(admit(Self, Xp)) == first(Self)
```

• constructeurs secondaires (un seul : `cure`)

```
-- 1       G-dérivation sur middle?
-- a       G-dérivation de <middle?, empty?, cure, C>
cure1a1: C(admit(Self, Xp)) == true AND empty?(Self) == true ==>
```

```
          cure(admit(Self, Xp)) == Self
cure1a2: C(admit(Self, Xp)) == true AND middle?(Self) == true AND H(Self) == true ==>
          cure(admit(Self, Xp)) == admit(cure(Self), Xp)
-- b      G-dérivation de <middle?, middle?, cure, G>
cure1b1: G(admit(Self, Xp)) == true AND empty?(Self) == true ==>
          cure(admit(Self, Xp)) == Self
cure1b2: G(admit(Self, Xp)) == true AND middle?(Self) == true AND H(Self) == true ==>
          cure(admit(Self, Xp)) == admit(cure(Self), Xp)
-- 2      G-dérivation sur full?
-- a      G-dérivation de <full?, middle?, cure, true>
cure2a1: middle?(Self) == true AND I(Self) == true ==>
          cure(admit(Self, Xp)) == admit(cure(Self), Xp)
-- 3      G-dérivation sur furgency?
-- a      G-dérivation de <furgency?, full?, cure, A>
cure3a1: A(urgency(Self, Xp)) == true AND full?(Self) == true ==>
          cure(urgency(Self, Xp)) == Self
cure3a2: A(urgency(Self, Xp)) == true AND furgency?(Self) == true ==>
          cure(urgency(Self, Xp)) == urgency(cure(Self), Xp)
cure3a3: A(urgency(Self, Xp)) == true AND murgency?(Self) == true AND K(Self) == true ==>
          cure(admit(Self, Xp)) == admit(cure(Self), Xp)
-- b      G-dérivation de <furgency?, furgency?, cure, B>
cure3b1: B(urgency(Self, Xp)) == true AND full?(Self) == true ==>
          cure(urgency(Self, Xp)) == Self
cure3b2: B(urgency(Self, Xp)) == true AND furgency?(Self) == true ==>
          cure(urgency(Self, Xp)) == urgency(cure(Self), Xp)
cure3b3: B(admit(Self, Xp)) == true AND murgency?(Self) == true AND K(Self) == true ==>
          cure(admit(Self, Xp)) == admit(cure(Self), Xp)
-- 4      G-dérivation sur murgency?
-- a      G-dérivation de <murgency?, murgency?, cure, B>
cure4a1: B(urgency(Self, Xp)) == true AND empty?(Self) == true ==>
          cure(urgency(Self, Xp)) == Self
cure4a2: B(urgency(Self, Xp)) == true AND middle?(Self) == true ==>
          cure(urgency(Self, Xp)) == Self
cure4a3: B(urgency(Self, Xp)) == true AND murgency?(Self) == true ==>
          cure(urgency(Self, Xp)) == urgency(cure(Self), Xp)
cure4a4: B(admit(Self, Xp)) == true AND murgency?(Self) == true AND H(Self) == true ==>
          cure(admit(Self, Xp)) == admit(cure(Self), Xp)
-- b      G-dérivation de <murgency?, empty?, cure, F>
cure4b1: F(urgency(Self, Xp)) == true AND empty?(Self) == true ==>
          cure(urgency(Self, Xp)) == Self
cure4b2: F(urgency(Self, Xp)) == true AND middle?(Self) == true ==>
          cure(urgency(Self, Xp)) == Self
cure4b3: F(urgency(Self, Xp)) == true AND murgency?(Self) == true ==>
          cure(urgency(Self, Xp)) == urgency(cure(Self), Xp)
cure4b4: F(admit(Self, Xp)) == true AND murgency?(Self) == true AND H(Self) == true ==>
          cure(admit(Self, Xp)) == admit(cure(Self), Xp)
-- c      G-dérivation de <murgency?, middle?, cure, D>
cure4c1: D(urgency(Self, Xp)) == true AND empty?(Self) == true ==>
          cure(urgency(Self, Xp)) == Self
```

```
cure4c2: D(urgency(Self, Xp)) == true AND middle?(Self) == true ==>
          cure(urgency(Self, Xp)) == Self
cure4c3: D(urgency(Self, Xp)) == true AND murgency?(Self) == true ==>
          cure(urgency(Self, Xp)) == urgency(cure(Self), Xp)
cure4c4: D(admit(Self, Xp)) == true AND murgency?(Self) == true AND H(Self) == true ==>
          cure(admit(Self, Xp)) == admit(cure(Self), Xp)
```

- secondary observers

```
totalNumber: totalNumber(Self) = number(Self) + urgencyNumber(Self)
```

4 Spécification Asspegique

Il est possible d'écrire une spécification algébrique de notre système en utilisant les outils existants comme Asspegique.

```
VERSION : Hospital.spec   AUTEUR : royer   MODIFIE LE : Mon-May--6-10:33:02-1996
STATUS : compilee  REGLES : non compilees  PREUVES : non  REVE : non  SLOG : non
-------------------------------------------------------------------------------
" Hospital est une file de Patient avec urgence "
" version simplifiee de Hospital compte tenu  "
" des proprietes sur les gardes "
spec: Hospital;
    use: Patient;
    sort: Hospital;
    generated by:
        " creation de l'Hospital "
        init    _ :         Entier                          -> Hospital;
        " admission normale d'un patient "
        admit _  _ :        Hospital Patient                -> Hospital;
        " entree d'une urgence "
        urgency  _ _ :      Hospital Patient                -> Hospital;

    operations:
        " soigne le premier "
        cure _ :            Hospital                        -> Hospital;

        " nombre d'admission bornee "
        size _ :             Hospital                       -> Entier;
        " nombre d'entree normales"
        number _ :           Hospital                       -> Entier;
        " nombre d'urgences non borne"
        urgencyNumber _ :    Hospital                       -> Entier;
        " nombre total d'entrees "
        totalNumber _ :      Hospital                       -> Entier;
        " designe celui qui est sera soigne "
        first _ :            Hospital                       -> Patient;

        " un essai "
        essai :                                             -> Hospital;

    predicates:
        " hopital vide "
        empty? _ :         Hospital;
        " normal "
        middle? _ :        Hospital;
```

```
    " normal et plein "
    full? _ :        Hospital;
    " inter et urgences "
    murgency? _ :    Hospital;
    " plein et urgences "
    furgency? _ :    Hospital;
    " les gardes complementaires et exclusives "
    " A + B = 1; D + F = A; C + G = 1; I + H = 1"
    gardeA _ : Hospital ;
    gardeB _ : Hospital ;
    gardeC _ : Hospital ;
    gardeD _ : Hospital ;
    gardeF _ : Hospital ;
    gardeG _ : Hospital ;
    gardeH _ : Hospital ;
    gardeI _ : Hospital ;

preconditions:

    " size Self is-defined;"
    " number Self is-defined;"
    " urgencyNumber Self is-defined;"
    " totalNumber Self is-defined;"
    " urgency Self is-defined;"
    (max > 1) => init max is-defined;
    (empty? Self) or (middle? Self) or (murgency? Self)
                    => (admit Self p) is-defined;
    not (empty? Self) => (cure Self) is-defined;
    not (empty? Self) => (first Self) is-defined;

axioms:
    " axiomes des predicats d'etat "

E1: empty? (init max) is-true;
E2: empty? (admit Self p) is-false;
E3: empty? (urgency Self p) is-false;

M1: middle? (init max) is-false;
M2: middle? (admit Self p) =
        (empty? Self) or ((middle? Self) and (gardeH Self));
M3: middle? (urgency Self p) is-false;

full1: full? (init max) is-false;
full2: full? (admit Self p) = (middle? Self) and (gardeI Self);
full3: full? (urgency Self p) is-false;

ms1: murgency? (init max) is-false;
ms2: murgency? (admit Self p) = (murgency? Self) and (gardeH Self);
ms3: murgency? (urgency Self p) =
        (empty? Self) or (middle? Self) or (murgency? Self);

fs1: furgency? (init max) is-false;
fs2: furgency? (admit Self p) = (murgency? Self) and (gardeI Self);
fs3: furgency? (urgency Self p) = (full? Self) or (murgency? Self) ;

    " les gardes "
    " gardes de CURE "
```

```
A : gardeA Self = equal (urgencyNumber Self) 1;
B : gardeB Self = (urgencyNumber Self) > 1;
C : gardeC Self = equal (number Self) 1;
D : gardeD Self = (gardeA Self) and (number Self) > 0;
F : gardeF Self = (gardeA Self) and (equal (number Self) 0);
G : gardeG Self = (number Self) > 1;
H : gardeH Self = (number Self) < (p (size Self));
I : gardeI Self = equal (number Self) (p (size Self));

" axiomes des observateurs "
s1 : size (init max) = max;
s2 : size (admit Self p) = size Self;
s3 : size (urgency Self p) = size Self;

n1 : number (init max) = 0;
n2 : number (admit Self p) = s (number Self);
n3 : number (urgency Self p) = number Self;

u1 : urgencyNumber (init max) = 0;
u2 : urgencyNumber (admit Self p) =   urgencyNumber Self;
u3 : urgencyNumber (urgency Self p) = s (urgencyNumber Self);

t1 : totalNumber Self = (number Self) + (urgencyNumber Self);

" first"
f1 : empty? Self is-true => first (admit Self p) = p;
f2+3 : middle? Self is-true => first (admit Self p) = first Self;
f4+8 : murgency? Self is-true => first (admit Self p) = first Self;
f5 : empty? Self is-true => first (urgency Self p) = p;
f6 : middle? Self is-true => first (urgency Self p) = p;
f7 : murgency? Self is-true => first (urgency Self p) = first Self;
f9 : full? Self is-true => first (urgency Self p) = p;
f10 : furgency? Self is-true => first (urgency Self p) = first Self;

" axiomes des constructeurs - version Pascal"
c1a1+1b1 : empty? Self is-true => cure (admit Self p) = Self;
c1a2+1b2+2a1 : middle? Self is-true =>
        cure (admit Self p) = admit (cure Self) p;

c3a1+3b1 :  full? Self is-true => cure (urgency Self p) = Self;
c3a2+3b2 : furgency? Self is-true =>
        cure (urgency Self p) = urgency (cure Self) p;
c3a3+3b3 : murgency? Self is-true & gardeI Self is-true =>
     cure (urgency Self p) = urgency (cure Self) p;

c4a1+4b1+4c1 : empty? Self is-true => cure (urgency Self p) = Self;
c4a2+4b2+4c2 : middle? Self is-true => cure (urgency Self p) = Self;
c4a3+4b3+4c3 : murgency? Self is-true =>
      cure (urgency Self p) = urgency (cure Self) p;
c4a4+4b4+4c4 : murgency? Self is-true & gardeH Self is-true =>
              cure (admit Self p) = admit (cure Self) p;

" essai "
essai : essai =  admit (urgency (admit (init 3) toto) tata) tutu;

where:
        Self : Hospital;
```

```
                max : Entier;
                p, q : Patient;
end Hospital
```

Par rapport à la spécification précédente nous avons simplifié les axiomes : de 10 axiomes pour `first` et 23 pour `cure` nous obtenons respectivement 8 et 10 axiomes. Ceci correspond à ce qu'on obtient directement par les méthodes classiques, mais sans assistance.

4.1 Interface avec Asspegique+

Une ouverture vers d'autres environnements est possible et souhaitable, en particulier nous n'avons pas la prétention de fabriquer tous les outils nécessaires à un bon environnement de développement. Le cas d'utilisation d'un système de manipulation algébrique nous a semblé le plus utile dans un premier temps. Nous avons étudié l'interfaçage avec le système Asspegique+ [Cho88, BCC90, Roq92]. Bien que celui-ci n'ai pas été formellement implanté sa faisabilité est simple. Un utilisateur d'Asspegique+ peut constater que notre écriture est proche de ce qu'il utilise habituellement. L'interface consiste à générer un fichier avec quelques modifications syntaxiques mineures et à l'intégrer sous Asspegique+. Notons toutefois que les hypothèses des TAG sont sur certains aspects plus faibles que la contrainte hiérarchique des TAA. Nos hypothèses sont plus proche de celles que pose David Musser [Mus80], elles nous semblent plus adaptées au contexte objet. Nous avons privilégié Asspegique+ pour des raisons circonstancielles, tout autre environnement de ce type est a priori possible. Nous avons donc une ouverture directe et complète avec Asspegique+ mais également vers d'autres environnements par l'intermédiaire des FISCs du projet SALSA [BBC+92].

4.2 Exemple d'évaluation

Suivant les outils, les documents et les compétences dont disposent l'utilisateur divers types de preuves peuvent être faites.

Ici nous allons illustrer des exemples de preuves «à la main» sur la spécification fonctionnelle du système en utilisant une approche algébrique classique et une approche basée sur le TAG. Les outils traditionnels de réécriture permettent d'automatiser partiellement, voire parfois complètement certaines preuves. Il est par exemple assez simple de vérifier le résultat de l'évaluation de la fonction `number` sur un hopital donné :

```
expression :
number essai

 L'expression a evaluer est ...

number_
   |
 essai
    [ essai -->> admit (urgency (admit (urgency (admit (init 5)
      JeanLebon) PierreKiroule) JacquesAdit) PaulTruc) SergeManteau ] .

Cette regle nous permet d'obtenir :

   number_
      |
     ____admit_______
    /                \
     ___urgency_____    SergeManteau
```

```
      /               \
       ____admit_______    PaulTruc
      /                \
        __urgency_____    JacquesAdit
       /            \
   _admit___    PierreKiroule
  /         \
init_   JeanLebon
  |
  5
```

```
    [ 5 -->> s (s (s (s (s 0)))) ] .
```

Cette regle nous permet d'obtenir :

```
   number_
      |
       ____admit_______
      /                \
       ___urgency_____    SergeManteau
      /               \
       ____admit_______    PaulTruc
      /                \
        __urgency_____    JacquesAdit
       /            \
   _admit___    PierreKiroule
  /         \
init_   JeanLebon
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  0
```

```
    [ number (admit Self p) -->> s (number Self) ] .
```

Cette regle nous permet d'obtenir :

```
      s_
      |
   number_
      |
       ___urgency_____
      /               \
       ____admit_______    PaulTruc
      /                \
        __urgency_____    JacquesAdit
       /            \
   _admit___    PierreKiroule
  /         \
```

```
init_   JeanLebon
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  0
```

```
    [ number (urgency Self p) -->> number Self ] .
```

Cette regle nous permet d'obtenir :

```
      s_
      |
   number_
      |
       ____admit_______
      /                \
        __urgency_____    JacquesAdit
       /              \
   _admit___    PierreKiroule
  /         \
init_   JeanLebon
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  0
```

```
    [ number (admit Self p) -->> s (number Self) ] .
```

Cette regle nous permet d'obtenir :

```
      s_
      |
      s_
      |
   number_
      |
        __urgency_____
       /              \
   _admit___    PierreKiroule
  /         \
init_   JeanLebon
```

```
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  0
```

```
    [ number (urgency Self p) -->> number Self ] .
```

Cette regle nous permet d'obtenir :

```
       s_
       |
       s_
       |
    number_
       |
   _admit___
  /         \
init_   JeanLebon
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  s_
  |
  0
```

```
    [ number (admit Self p) -->> s (number Self) ] .
```

Cette regle nous permet d'obtenir :

```
   s_
   |
   s_
   |
   s_
   |
number_
   |
 init_
   |
   s_
   |
   s_
   |
```

```
  s_
  |
  s_
  |
  s_
  |
  0

   [ number (init max) -->> 0 ] .

Cette regle nous permet d'obtenir :

s_
|
s_
|
s_
|
0

s (s (s 0))
```

Ce fichier est le résultat de l'évaluation graphique de `number(essai)` avec Asspegique.

4.3 Preuve algébrique classique

En utilisant l'outillage habituel des spécifications de types abstraits algébriques il est possible de prouver des théorèmes comme par exemple le fait, relativement trivial, que `number` est toujours positif ou nul. Cette information est évidemment importante et malgré son caractère intuitif elle doit être formellement établie. En effet elle peut être utile pour des considérations de typage, pour simplifier les gardes (par exemple D+F = A) et évidemment pour éviter des situations aberrantes comme avoir -1 patient dans l'hôpital ce qui conduirait sûrement à un processus sans terminaison.

Cette technique classique se base sur une preuve par récurrence (ici le nombre total de patient dans l'hôpital), la proposition étant `Self : Hospital ==>  number(Self)` $\geq$ `0`. L'utilisation de la spécification Asspegique (et de son environnement) est immédiat dans ce cas : il y a trois générateurs **`init, admit, urgency`** les deux derniers étant récursifs. La preuve est assez directe à partir des axiomes de `number : n1, n2 et n3`. Il faut d'abord prouver le cas de base à savoir `number(init(max)))` $\geq$ `0`, ce qui est immédiat par `n1`. Dans les deux autres cas il faut poser la proposition comme hypothèse de récurrence et considérer les deux constructeurs récursifs. Les axiomes n2 et n3 donnent respectivement :
`number(admit(Self, Xp)) == 1+number(Self)`
`number(urgency(Self, Xp)) == number(Self)`
La preuve est directe à partir de là. Une telle preuve peut être faire automatiquement par certains systèmes (Asspegique, Larch Prover, ...) en utilisant soit la technique précédente soit des techniques indirectes (induction sans induction par exemple). A titre d'exemple nous avons fait une preuve semi-manuelle avec l'évaluateur symbolique d'Asspegique. L'opérateur **`preuve`** est simplement défini par **`inf 0 (number Self)`**.

```
expression :
preuve (init max:Entier)

---- premier cas
 L'expression a evaluer est ...
```

```
preuve_
   |
 init_
   |
  max
    [ preuve Self -->> inf 0 (number Self) ] .

Cette regle nous permet d'obtenir :

 inf___
/      \
0   number_
       |
     init_
       |
      max

    [ number (init max) -->> 0 ] .

Cette regle nous permet d'obtenir :

inf__
/   \
0   0

    [ inf 0 0 -->> true ] .

Cette regle nous permet d'obtenir :

true

---- deuxieme cas
expression :
preuve (admit Self:Hospital Xp:Patient)

 L'expression a evaluer est ...

 preuve_
    |
 admit__
 /     \
Self   Xp
    [ preuve Self -->> inf 0 (number Self) ] .

Cette regle nous permet d'obtenir :

 inf___
/      \
0   number_
       |
    admit__
    /     \
   Self   Xp

    [ number (admit Self p) -->> s (number Self) ] .
```

```
Cette regle nous permet d'obtenir :

inf__
/   \
0   s_
    |
 number_
    |
   Self

    [ inf 0 (s x) -->> true ] .

Cette regle nous permet d'obtenir :

true

---- troisieme cas
expression :
preuve (urgency Self:Hospital Xp:Patient)

 L'expression a evaluer est ...

 preuve_
    |
urgency__
 /     \
Self   Xp
    [ preuve Self -->> inf 0 (number Self) ] .

Cette regle nous permet d'obtenir :

 inf___
/      \
0   number_
       |
   urgency__
    /     \
   Self   Xp

    [ number (urgency Self p) -->> number Self ] .

Cette regle nous permet d'obtenir :

 inf___
/      \
0   number_
       |
      Self

inf 0 (number Self)
---- cette derniere expression etant exactement : preuve(Self)
```

On constatera que dans les deux premiers cas la réponse `true` est directe, seul le dernier cas fait explicitement apparaître la nécessité de poser une hypothèse de récurrence.

Dans le cas d'une proposition plus complexe comme celle qui suit une telle preuve n'est ni simple à faire manuellement ni automatiquement. Une raison est que l'on a simplifié la spécification originelle, ce qui demande de bonnes compétences. Par ailleurs la preuve est plus difficile à suivre par quelqu'un qui a spécifié le système initial c'est à dire le TAG. D'où l'idée d'un système de preuve plus conviviale basé sur la G-dérivation du TAG. Dans l'exemple précédent le choix des trois générateurs est à lier avec la figure 3.

4.4 Preuve TAG

La proposition informelle est : dans un état quelconque de l'hôpital, si il y a une urgence au moins, alors l'opérateur `first` trouve la plus anciennement arrivée dans l'hôpital. Elle peut se formaliser comme suit :

`Self : Hospital AND thereIsUrgency?(Self) ==>  first(Self) == oldestUrgency(Self).`

Il faut définir la fonction `oldestUrgency`, en fait nous allons la définir en même temps que nous allons faire la preuve. L'idée de base est de considérer le TAG réduit aux générateurs (voir figure 3) et de faire des G-dérivations pour prouver la proposition. La récurrence va en quelque sorte être complètement guidée par l'automate réduit aux générateurs.

Pour la condition `thereIsUrgency?(Self)` nous pouvons d'abord calculer la table de vérité à partir du TAG réduit par G-édrivation. Nous constatons facilement que le résultat est vrai si et seulement si `murgency?(Self) OR furgency?(Self)`. Pour `first(Self)` et `oldestUrgency(Self)` nous dressons, pour chacun, une table des G-dérivations compte tenu de l'hypothèse précédente.

Etat final	**Transition**	**Etat initial**	**Résultat**
`murgency`	`urgency(Self, Xp)`	`empty`	`Xp`
	`urgency(Self, Xp)`	`middle`	`Xp`
	`urgency(Self, Xp)`	`murgency`	`first(Self)`
	`admit(Self, Xp)`	`murgency`	`first(Self)`
`furgency`	`admit(Self, Xp)`	`murgency`	`first(Self)`
	`urgency(Self, Xp)`	`furgency`	`first(Self)`
	`urgency(Self, Xp)`	`full`	`Xp`

Figure 4 : G-dérivation de `first`

Nous pouvons observer que l'autre table est identique à celle-ci. Un tel mécanisme de preuve reste à affiner et à outiller mais il semble prometteur. En terme de puissance d'expression ces deux techniques sont équivalentes, l'avantage espéré est au niveau guide par un outil et «traçabilité» pour le spécifieur.

5 Modèle des classes formelles

La principale raison d'être de ce modèle est que nous souhaitons un niveau d'abstraction au-dessus des langages à objets avec classes comme C++, Smalltalk, Eiffel ou CLOS. Le modèle des CF est une sorte de langage de spécification de classes qui est intermédiaire entre les TAA et les classes concrètes. Nous avons privilégié une proximité avec les TAA car ce type de spécification est historiquement et conceptuellement proche de la PPO. Toutefois des aspects comme l'héritage mais surtout les métaclasses ne semblent pas triviales à introduire dans les TAA. Le modèle à CF doit permettre une transition plus facile des TAA vers les classes mais également de définir des règles d'écriture et des outils inspirés des spécifications algébriques. Nous n'avons pas une sémantique algébrique complète du modèle mais elle existe pour la notion de classe. Une sémantique opérationnelle abstraite existe également et elle prend en compte l'héritage. Nous avons défini un mécanisme simple et naturel de réécriture qui permet de faire de l'évaluation symbolique ou de la preuve de propriété. Des aspects divers de ce modèle sont décrits dans [Roy92, AR92b, Roy93, AR94b, ACR94, ACCR94].

Les principales caractéristiques du modèle sont les suivantes :

- Une CF est une spécification particulière de TAA.
- La conception d'une CF est inspirée de la PPO et se fait par un aspect et des méthodes secondaires.
- Une CF est un type et l'héritage implique le sous-typage, un contrôle de type simple mais sûr existe.
- Des règles diverses ont été défini concernant : la définition structurelle des classes, les relations d'héritages, l'écriture des méthodes et leur redéfinition.
- La généricité est au moins possible par héritage.

Une CF se définit par un aspect et des méthodes secondaires.

```
                              Hospital
                        inherits from OBJECT

comments: class for hospital

features: init, admit, urgency, cure, number, urgencyNumber, first
```

```
                            aspect : hospital
field selectors

size          : Hospital --> Integer
queue         : Hospital --> List[Patient]
urgencyQueue  : Hospital --> List[Patient]
constraint:
   size(Self) >  1 and number(Self) <= size(Self)
```

```
                              extensions

;; empty? : les deux listes sont vides
  empty? : Hospital  -->  Boolean
empty?(Self) == empty?(queue(Self)) and empty?(urgencyQueue(Self))

;; number : nombre de patient non urgent
  number : Hospital  -->  Integer
number(Self) == length(queue(Self))

;; full? : le nombre maximal de patient non urgent est atteint
  full? : Hospital  -->  Boolean
full?(Self) == number(Self) = size(Self)

;; urgencyNumber : nombre de patient en urgence
  urgencyNumber : Hospital  -->  Integer
urgencyNumber(Self) == length(urgencyQueue(Self))
```

```
                         Hospital -- suite --
;; thereIsUrgency? : il y a au moins un patient en urgence
  thereIsUrgency? : Hospital  --> Boolean
thereIsUrgency?(Self) == urgencyNumber(Self) > 0

;; first : prochain patient qui sera soigné
  first : Hospital  --> Patient
requires: empty?(Self) == false
thereIsUrgency?(Self) == true ==>  first(Self) == head(urgencyQueue(Self))
thereIsUrgency?(Self) == false ==>  first(Self) == head(queue(Self))

;; cure : le patient a été soigné
  cure : Hospital  --> Hospital
requires: empty?(Self) == false
thereIsUrgency?(Self) == true ==>  cure(Self) == copy(Self, urgencyQueue = tail(urgencyQueue(Self)))
thereIsUrgency?(Self) == false ==>  cure(Self) == copy(Self, queue = tail(queue(Self)))

;; admit : admit a new normal patient
  admit : Hospital Patient  --> Hospital
requires: full?(Self) == false
admit(Self, Xp) == copy(Self, queue = cons(queue(Self), Xp))

;; urgency : admit a new urgency patient
  urgency : Hospital Patient  --> Hospital
urgency(Self, Xp) == copy(Self, urgencyQueue = cons(urgencyQueue(Self), Xp))
```

```
                         class methods

;; init : open the hospital
  init : Integer  --> Hospital
init(Xmax) == newHospital(size = Xmax, queue = newEmptyList(),
                   urgencyQueue = newEmptyList())
```

Figure 5 : Exemple de classe formelle pour `Hospital`

Cette classe a été obtenu par une analyse intuitive du système. Un aspect définit un ensemble de méthodes primitives qui sont utiles à la manipulation des instances. Les principales sont : la création, la copie, l'égalité, la description et les sélecteurs de champs. Les méthodes dites secondaires sont des extensions fonctionnelles de ce noyau. La conception d'une telle classe se focalise d'abord sur la description des objets puis sur leur utilisation. Cette pratique de la conception de la classe est conforme à l'usage de la PPO et a plusieurs avantages :

- c'est une spécification algébrique particulière qui sous certaines hypothèses est suffisamment complète et non-contradictoire,
- elle vérifie la propriété naturelle en PPO que l'ajout d'une nouvelle méthode secondaire ne change pas le type des instances.

Si globalement une classe formelle représente une classe «à la Eiffel» (mais actuellement sans `like`) plusieurs originalité existent. La proximité avec les spécifications algébriques et tous les outils ou techniques qui s'en inspirent (réécriture, critères de bonnes spécifications, ...). La possibilité de pré-conditions sur les sélecteurs qui sans accroître la puissance d'expression du langage permet plus de souplesse dans la définition des classes. En particulier une spécification «hiérarchique» comme celle des liste de [AR94b] peut également être définie de façon «plate». Cet aspect est important pour la conception de composant réutilisable et pour la souplesse du modèle. Des extensions : métaclasses (MOP d'ObjVlisp), multi-sélection, type quantifié, ... sont à l'étude.

6 Conception d'un TAG en CF

Le problème du raffinage d'un TAG en CF est un problème classique de représentation d'un type de données. Notre avons étudié la transformation du TAG sans axiomes en une hiérarchie de classes formelles. Elle nous semble plus facile pour un concepteur connaissant la PPO et qui ne connaît généralement pas les spécifications algébriques. Une façon plus rigoureuse d'opérer est de transformer la spécification algébrique en classes formelles. Cette alternative a l'avantage de permettre la preuve de la représentation en même temps que sa construction. Néanmoins l'approche que nous avons faite propose des outils communs aux deux démarches et permet d'expérimenter les différents choix et de mesurer leur influence au niveau de la réutilisation et de la structuration.

Le processus de représentation n'est évidemment pas complètement automatique mais certaines parties le sont [AR94a]. Des choix sont faits : au niveau de la hiérarchie, du placement des opérations, des structures abstraites choisies, etc. Ces différents choix sont influencés par des critères de réutilisabilité, de simplicité, de taille des axiomes, etc.

Les principales phases sont :

- Simplification de l'automate par regroupement d'états connexes. Chaque état représente un sous-type.
- Calcul des profils exacts des générateurs en fonction des regroupements d'états (union de types).
- Obtention de deux hiérarchies une d'utilisation l'autre de création (regroupement sur le receveur ou sur le résultat).
- Calcul des aspects des classes avec la seconde hiérarchie. Un calcul semi-automatique est possible à partir des générateurs mais des simplifications sont nécessaires. Elles sont prouvables en utilisant un schéma inspiré de la représentation des types de données [Hoa72].
- On restructure chaque aspect et la hiérarchie d'utilisation.
- Ecriture des axiomes des méthodes secondaires de chaque classe en fonction de l'aspect précédemment défini.

Le processus général est décrit par le schéma suivant :

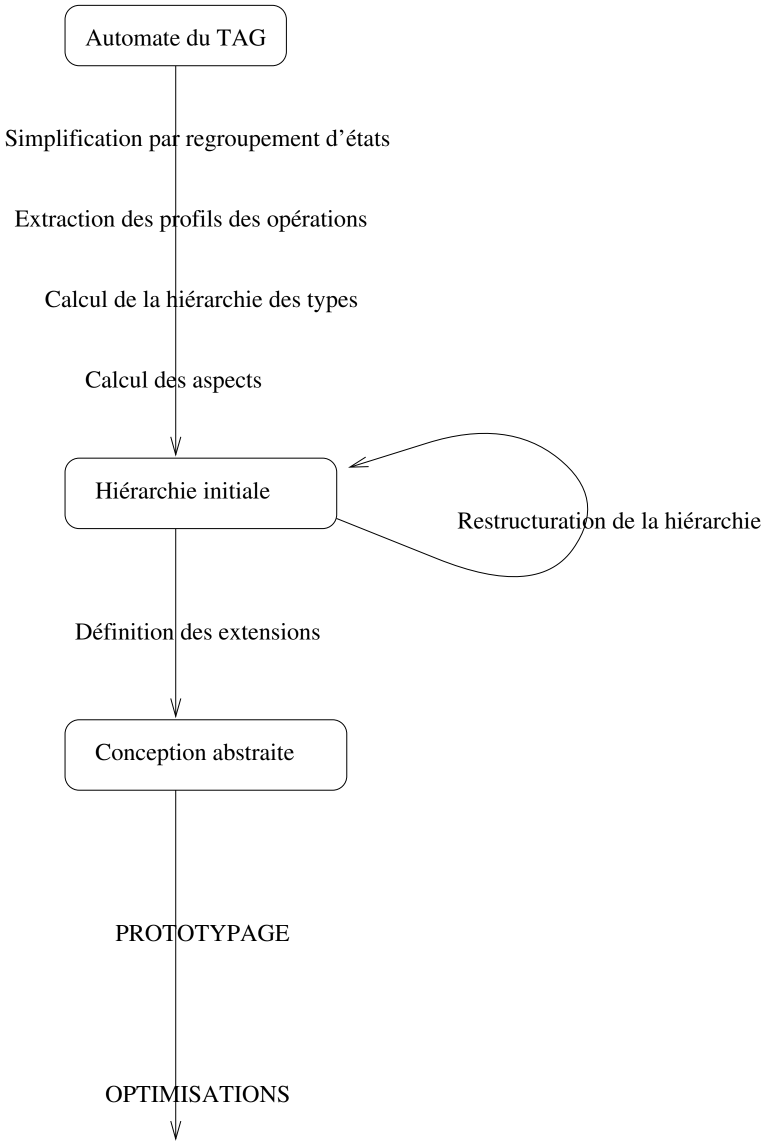

Figure 6 : Démarche générale de la représentation du TAG

Le résultat obtenu avec notre exemple est une hiérarchie à trois niveaux et cinq classes terminales. La restructuration du graphe est par exemple possible en une classe dont l'aspect est :

```
                          Hospital
                  inherits from OBJECT
-----------------------------------------------------------
                      aspect : hospital
field selectors

empty?     : Hospital ⟶ Boolean
size       : Hospital ⟶ Integer
requires: empty?(Self) == true

normal?    : Hospital ⟶ Boolean
requires: empty?(Self) == false

entry      : Hospital ⟶ Patient
requires: empty?(Self) == false

oldState   : Hospital ⟶ Hospital
requires: empty?(Self) == false

constraint:
   size(Self) >  1 and number(Self) <= size(Self) -1
```

Cette représentation est équivalente à celle définie intuitivement (figure 5) mais en moins naturelle et moins réutilisable. La version intuitive montre plus clairement que l'ordre relatif entre les urgences et les non urgences n'importe pas. Ce qui importe ce sont les deux ordres concernant les urgences et les non urgences, d'ailleurs une représentation avec une seule liste est possible en prenant soin de mettre les urgences en tête. Dans un cas d'étude plus complexe l'intuition seule ne suffirait pas à obtenir un tel résultat. Le processus que nous proposons est très méthodique mais il nécessite également de l'expérience et de l'intuition. Celle-ci ne peut être efficace que si les composants manipulés sont de complexité limitée. Cette complexité est mesurée par des critères simples qui doivent encore être améliorés.

6.1 Conception en classes formelles

La construction de classes formelles suit une démarche décrite dans [And95]. Cette démarche est brièvement illustrée dans ce qui suit.

6.1.1 Construction de la hiérarchie initiale

Il n'y a pas de simplification de l'automate car il n'y a pas d'états frères. Le calcul des opérations primitives a déjà été fait pour la spécification algébrique. Nous calculons les profils des opérations partielles:

admit	:	NONFULLHospital	Patient $\rightarrow$ NONEMPTYHospital
urgency	:	Hospital	Patient $\rightarrow$ URGENCYHospital
cure	:	NONEMPTYHospital $\rightarrow$ Hospital	
first	:	NONEMPTYHospital $\rightarrow$ Patient	

Nous obtenons les hiérarchies suivantes :

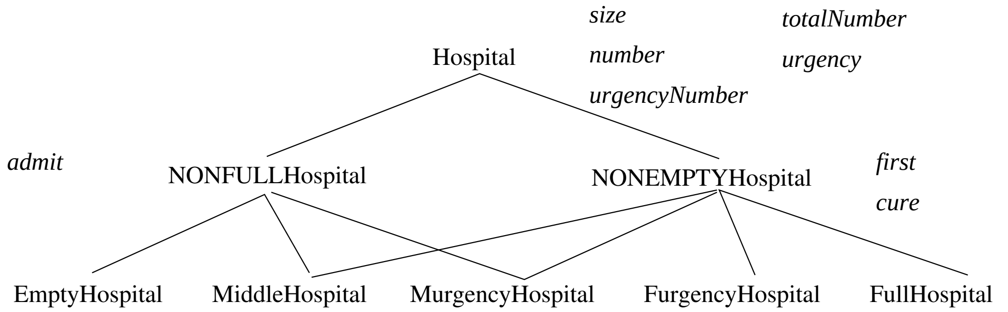

Figure 7 : Hiérarchie d'utilisation de `Hospital`

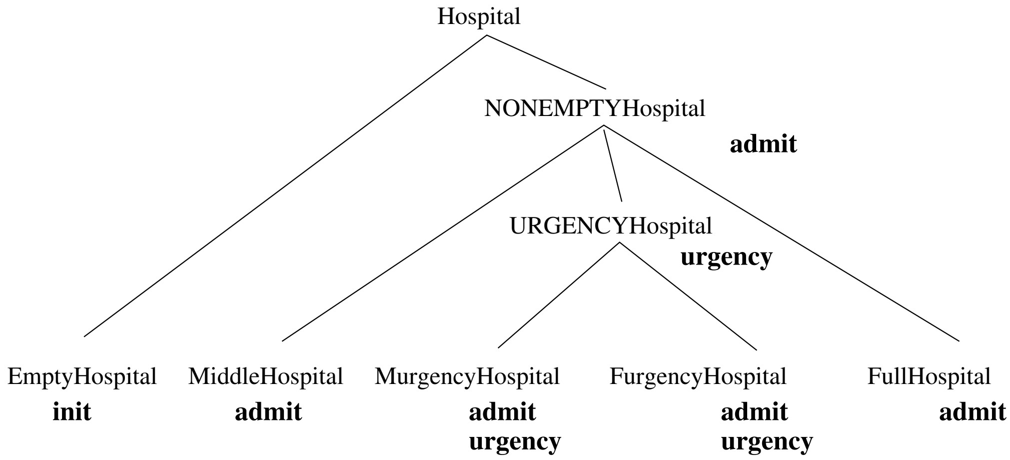

Figure 8 : Hiérarchie de création de `Hospital`

Puis nous calculons les structures par défaut pour chaque classe terminale ; les renommages des champs reste à faire.

`EmptyHospital` **inherits from** `NONFULLHospital`

aspect : hospital
field selectors
`max : EmptyHospital —→ Integer`

MiddleHospital

inherits from `NONEMPTYHospital NONFULLHospital`

aspect : hospital

field selectors

```
isFirst   : MiddleHospital ⟶ Boolean
Self1     : MiddleHospital ⟶ EmptyHospital
```
requires: `isFirst(Self) == true`

```
Xp1       : MiddleHospital ⟶ Patient
```
requires: `isFirst(Self) == true`

```
Self2     : MiddleHospital ⟶ MiddleHospital
```
requires: `isFirst(Self) == false`

```
Xp2       : MiddleHospital ⟶ Patient
```
requires: `isFirst(Self) == false`

FullHospital

inherits from `NONEMPTYHospital`

aspect : hospital

field selectors

```
Self   : FullHospital ⟶ MiddleHospital
Xp     : FullHospital ⟶ Patient
```

MurgencyHospital
inherits from NONEMPTYHospital NONFULLHospital

aspect : hospital

field selectors

```
origin  : MurgencyHospital ⟶ 1..4
Self1   : MurgencyHospital ⟶ EmptyHospital
```
requires: origin(Self) = 1 == true &

```
Xp1     : MurgencyHospital ⟶ Patient
```
requires: origin(Self) = 1 == true

```
Self2   : MurgencyHospital ⟶ MiddleHospital
```
requires: origin(Self) = 2 == true

```
Xp2     : MurgencyHospital ⟶ Patient
```
requires: origin(Self) = 2 == true

```
Self3   : MurgencyHospital ⟶ MurgencyHospital
```
requires: origin(Self) = 3 == true

```
Xp3     : MurgencyHospital ⟶ Patient
```
requires: origin(Self) = 3 == true

```
Self4   : MurgencyHospital ⟶ MurgencyHospital
```
requires: origin(Self) = 4 == true

```
Xp4     : MurgencyHospital ⟶ Patient
```
requires: origin(Self) = 4 == true

FurgencyHospital
inherits from NONEMPTYHospital

aspect : hospital

field selectors

```
origin  : FurgencyHospital ⟶ 1..3
Self1   : FurgencyHospital ⟶ FullHospital
```
requires: origin(Self) = 1 == true &

```
Xp1     : FurgencyHospital ⟶ Patient
```
requires: origin(Self) = 1 == true

```
Self2   : FurgencyHospital ⟶ MurgencyHospital
```
requires: origin(Self) = 2 == true

```
Xp2     : FurgencyHospital ⟶ Patient
```
requires: origin(Self) = 2 == true

```
Self3   : FurgencyHospital ⟶ FurgencyHospital
```
requires: origin(Self) = 3 == true

```
Xp3     : FurgencyHospital ⟶ Patient
```
requires: origin(Self) = 3 == true

6.1.2 Restructuration

La restructuration commence par un changement d'aspect, voici les nouveaux aspects[3].

```
                         MiddleHospital
        inherits from NONEMPTYHospital NONFULLHospital

                        aspect : hospital
field selectors

oldState   : MiddleHospital —→ EmptyHospital
queue      : MiddleHospital —→ List[Patient]
```

```
                        MurgencyHospital
        inherits from NONEMPTYHospital NONFULLHospital

                        aspect : hospital
field selectors

queue          : MurgencyHospital —→ List[Patient]
urgencyQueue   : MurgencyHospital —→ List[Patient]
oldState       : MurgencyHospital —→ NONFULLHospital
requires: length(urgencyQueue(Self)) = 1
```

Remarque : le type exact de `oldState` est `EmptyHospital` $\cup$ `MiddleHospital`, on le relaxe vers `NONFULLHospital` (première classe abstraite commune de la hiérarchie d'utilisation). Il y a ainsi un circuit dans $dps \cup ako$, mais cela ne pose pas de problèmes.

```
                        FurgencyHospital
              inherits from NONEMPTYHospital

                        aspect : hospital
field selectors

queue          : FurgencyHospital —→ List[Patient]
urgencyQueue   : FurgencyHospital —→ List[Patient]
oldState       : FurgencyHospital —→ NONEmptyHospital
```

Remarque : le type exact de `oldState` est `FullHospital` $\cup$ `MurgencyHospital`, on le relaxe vers `NONEMPTYHospital`.

Voyons maintenant le regroupement de classes. Plusieurs regroupements sont possibles, le plus extrême étant de tout regrouper dans la classe racine. Des niveaux intermédiaires sont bien sûr possibles. A partir de là la définition des méthodes secondaires ne pose pas trop de problème, mais il ne faut pas oublier les contraintes.

6.2 Autre structuration possible

Voici une autre restructuration plus directe et plus abstraite, avec des commentaires et sa fonction de représentation. Cette nouvelle restructuration permet d'illustrer certaines difficultés qui restent à résoudre. Elle montre que l'on peut être plus ou moins fin dans le travail des structures, ce qui conduit a des représentations plus ou moins efficaces ou réutilisables. Malheureusement ces deux aspects hautement souhaitables en génie logiciel ne vont pas manifestement de pair.

3. Nous ne donnons pas ici la fonction de transformation, qui est à démontrer.

En fait l'aspect est une sorte d'article avec variant et une généralisation des équations de types d'où la puissance d'expression mais aussi la complexité des manipulations.

1. A partir du cahier des charges et des profils établir un dictionnaire global pour les noms de sélecteurs cela évitera des ambiguïtés et permettra une meilleure qualité des composants.
2. Le principe fondamental est de minimiser la structure mais tout en pouvant distinguer les générateurs initiaux. Pour un état nous regardons les différents générateurs qui arrivent et nous en déduisons une structure. Nous pouvons associer à chaque générateur un aspect (c'est trivial). Puis à partir de plusieurs structure en faire un réunion disjointe et calculer une nouvelle structure optimale. Cette ruse est intéressante car elle permet d'uniformiser le processus de restructuration.
3. Les pré-conditions sont «élargies» et souvent transformées en contrainte si elles sont vérifiées par les autres axiomes.
4. Une réunion des générateurs de même nom et un partage des informations est utile. Mais nous pouvons pousser plus loin en faisant des tranches verticales dans les arguments et en le regroupant par héritage ou union disjointe (utilisation d'une extension permettant des sélecteurs de mêmes noms et exclusifs). Dans certains cas on doit faire des tranches verticales mais également horizontales.
5. Nous travaillons uniquement sur les structures en oubliant les extensions, par exemple tous les états connaissent `number`, `size`, etc.

Pour `EmptyHospital` nous avons un générateur avec un argument entier et une pré-condition cela donne un sélecteur et une contrainte.

EmptyHospital **inherits from** `NONFULLHospital`
aspect : hospital **field selectors** `size : EmptyHospital --> Integer` **constraint:** `size(Self) > 1`

Pour `FullHospital` nous avons un seul générateur avec 2 arguments et une pré-condition donc deux sélecteurs et une contrainte.

FullHospital **inherits from** `NONEMPTYHospital`
aspect : hospital **field selectors** `entry : FullHospital --> Patient` `oldState : FullHospital --> MiddleHospital` **constraint:** `number(Self) = size(Self)`

Noter que la contrainte est réécrite directement sur `Self` et non plus sur `oldState(Self)`.

Pour `MiddleHospital` on fait deux tranches une avec `oldState : Middle + Empty` et une avec `entry : Patient`. Nous remarquons que la garde `H` est vraie sur le cas `empty` donc cette garde passe en contrainte. Mais nous n'avons pas dans la hiérarchie une super-classe `Toto = Empty + Middle` sinon les champs `entry` et `oldState` seraient suffisants. Nous proposons la structure suivante :

```
                         MiddleHospital
          inherits from NONEMPTYHospital NONFULLHospital
---------------------------------------------------------------
                        aspect : hospital
field selectors

empty?     : MiddleHospital --> Boolean
entry      : MiddleHospital --> Patient
oldState   : MiddleHospital --> MiddleHospital
requires: empty?(oldState(Self)) == false

oldState   : MiddleHospital --> EmptyHospital
requires: empty?(oldState(Self)) == true

constraint:
   number(Self) < size(Self)
```

Pour le cas `MurgencyHospital` nous avons quatre générateurs, trois `urgency` et un `admit` (une tranche horizontale). Remarquons que la garde `H` est vraie pour tous les générateurs donc elle passe en contrainte. Il y a deux tranches : une sur `Patient` et une sur l'union des types receveurs. En utilisant la hiérarchie nous voyons une union disjointe de `MiddleHospital`, `EmptyHospital` et `MurgencyHospital` en `NonFullHospital`.

```
                         MurgencyHospital
          inherits from NONEMPTYHospital NONFULLHospital
---------------------------------------------------------------
                        aspect : hospital
field selectors

normal?    : MurgencyHospital --> Boolean
entry      : MurgencyHospital --> Patient
oldState   : MurgencyHospital --> MurgencyHospital
requires: normal?(Self) = true

oldState   : MurgencyHospital --> NonFullHospital
requires: normal?(Self) = false

constraint:
   number(Self) < size(Self)
```

Pour le cas `FurgencyHospital` il y a trois générateurs a deux arguments semblables. En utilisant la hiérarchie nous voyons une union disjointe de `MurgencyHospital`, `FurgencyHospital` et `FullHospital` en `NonEmptyHospital` pour `oldState`.

```
                    FurgencyHospital
              inherits from NONEMPTYHospital
```

aspect : hospital

field selectors

```
entry      : FurgencyHospital ⟶ Patient
oldState   : FurgencyHospital ⟶ NonEmptyHospital
```

constraint:

```
   number(Self) = size(Self)
```

La garde `I` est écrite comme contrainte en :
`full?(oldState(Self)) or furgency?(oldState(Self)) or (murgency?(oldState(Self)) and (number(oldState(Self)) = size(oldState(Self)) - 1)`.
Elle est simplifiable, en fait pour `FullHospital` et `FurgencyHospital` il y a égalité, et on peut prouver `number(Self) = size(Self)` pour le cas `admit` à partir de `MurgencyHospital`.

6.2.1 Restructuration de la hiérarchie

Voyons maintenant le regroupement de classes. Nous avons également choisi de tout regrouper dans la classe racine.

1. Il faut faire une structure commune permettant de définir toutes les sous-structures : une sorte d'union, mais attention aux conflits de noms.
2. On remarque la présence des informations suivantes : `size, entry, oldState, normal?`.
3. Nous voyons nettement qu'il faut distinguer le cas vide des autres ce qui fait une sorte de classification. Donc nous aurons un champ `empty?`.
4. L'information `size` doit être un sélecteur de champ, mais ceci est uniquement nécessaire dans le cas vide. En fait dans les autres états il suffit que `size` soit une extension.
5. Les champs `entry, oldState` sont des champs nécessaires dans le cas non vide.
6. L'information `normal?` est utile pour `MuregencyHospital` pour distinguer les deux boucles `admit` et `urgency` sur l'état correspondant. Pour les autres structures `normal?` est une extension implicite.

Tous ceci suggère donc la description suivante :

Hospital **inherits from** OBJECT
aspect : hospital **field selectors** empty? : Hospital ⟶ Boolean size : Hospital ⟶ Integer **requires:** empty(?Self) == true normal? : Hospital ⟶ Boolean **requires:** empty?(Self) == false entry : Hospital ⟶ Patient **requires:** empty?(Self) == false oldState : Hospital ⟶ Hospital **requires:** empty?(Self) == false **constraint:** size(Self) > 1 and number(Self) <= size(Self)

En terme de représentation cette définition est équivalente à celle donnée intuitivement. Nous oublions les contraintes car elles sont identiques. La fonction d'abstraction (représentation de celle-ci B par l'intuitive A) est :

```
empty?(Xln) and empty?(Xlu) ==> abstraction(newA(max, Xln, Xlu)) ==
            newB(true, max, -, -, -)
not(empty?(Xln)) AND empty?(Xlu) ==>  abstraction(newA(max, Xln, Xlu)) ==
            newB(false, -, true, head(Xln), abstraction(max, tail(Xln), Xlu))
not(empty?(Xlu)) ==>  abstraction(newA(max, Xln, Xlu)) ==
            newB(false, -, false, head(Xlu), abstraction(newA(max, Xln, tail(Xlu))))
```

La représentation inverse :

```
abstraction(newB(true, max, -, -, -)) == newA(max, empty, empty)
soit R = abstraction(Xo)
abstraction(newB(false, max, true, Xp, Xo)) == newA(max, add(Xp, queue(R)), urgencyQueue(R))
abstraction(newB(false, max, false, Xp, Xo)) == newA(max, queue(R), add(Xn, urgencyQueue(R)))
```

La technique intuitive ne peut marcher que sur des exemples plutôt petit. Nous proposons une technique plus méthodique où l'intuition est également nécessaire. Il est nécessaire de préciser et d'assister la manipulation de ces structures qui demande entre autre des outils de preuve adaptés et des critères de bonnes structuration. En particulier des critères de taille des composants et de leur agencement doit permettre au concepteur de garder la maîtrise «intellectuelle» de chaque composant. Nous avons des outils et des heuristiques qui permettent de guider ou de prouver certains pas. Il semble qu'un critère de réutilisation peut être le suivant : taille des aspects des classes utilisées / taille du composant défini. Il y a un autre critère utile : indiquant la qualité au vue d'une réutilisation future de la classe fabriquée.

6.2.2 Définition des méthodes secondaires

Pour l'écriture des méthodes secondaires «une traduction assistée» des axiomes de la spécification est possible mais elle reste à préciser. Nous avons fait ici une traduction complètement manuelle, on pourra observer que le résultat est assez proche de celle présentée précédemment. Les deux cas qui posent une difficulté sont `first` et `cure`. En effet nous ne disposons pas directement de tous les prédicats d'états nécessaires, mais il est possible de les définir. Une autre approche est de considérer une opération privée `thereIsUrgency?` qui regroupe les états `murgency` et `furgency`, elle permet d'arriver à une version simple.

```
                              Hospital
                         inherits from OBJECT

comments: class for hospital

features: init, admit, urgency, cure, number, urgencyNumber, first, empty?, full?, totalNumber

empty?     : Hospital —→ Boolean
size       : Hospital —→ Integer
requires: empty?(Self) == true

normal?    : Hospital —→ Boolean
requires: empty?(Self) == false

entry      : Hospital —→ Patient
requires: empty?(Self) == false

oldState   : Hospital —→ Hospital
requires: empty?(Self) == false

constraint:
   size(Self) > 1 and number(Self) <= size(Self)
```

```
                              extensions

;; size : le nombre maximal d'entrée non urgente
  size : Hospital  —→  Integer
not(empty?(Self)) ==> size(Self) == size(oldState(Self))

;; number : nombre de patients non urgents
  number : Hospital  —→  Integer
empty?(Self) ==> number(Self) == 0
not(empty?(Self)) AND normal?(Self) ==> number(Self) == 1+number(oldState(Self))
not(empty?(Self)) AND not(normal?(Self)) ==> number(Self) == number(oldState(Self))

;; full? : plus de place en non urgente
  full? : Hospital  —→  Boolean
full?(Self) == number(Self) = size(Self)

;; urgencyNumber : nombre des urgences
  urgencyNumber : Hospital  —→  Integer
empty?(Self) ==> urgencyNumber(Self) == 0
not(empty?(Self)) AND normal?(Self) ==> urgencyNumber(Self) == urgencyNumber(oldState(Self))
not(empty?(Self)) AND not(normal?(Self)) ==> urgencyNumber(Self) == 1+urgencyNumber(oldState(Self))

;; thereIsUrgency? : au moins un patient en urgence
  thereIsUrgency? : Hospital  —→  Boolean
thereIsUrgency?(Self) == urgencyNumber(Self) > 0
```

```
Hospital -- suite --
;; first : le patient a soigner
  first : Hospital --> Patient
requires: empty?(Self) == false
thereIsUrgency?(Self) AND normal?(Self) ==> first(Self) == first(oldState(Self))
thereIsUrgency?(Self) AND not(normal?(Self)) ==> first(Self) == entry(Self)
not(thereIsUrgency?(Self)) ==> first(Self) == entry(Self)

;; cure : le premier patient a été soigné
  cure : Hospital --> Hospital
requires: empty?(Self) == false
thereIsUrgency?(Self) AND normal?(Self) ==> cure(Self) == copy(Self, oldState=cure(oldState(Self)))
thereIsUrgency?(Self) AND not(normal?(Self)) ==> cure(Self) == oldState(Self)
not(thereIsUrgency?(Self)) ==> cure(Self) == oldState(Self)
;; admit : admission d'un patient non urgent
  admit : Hospital Patient --> Hospital
requires: full?(Self) == false
admit(Self, Xp) == copy(Self, empty?=false, normal?=true, entry=Xp, oldState=Self))

;; urgency : une admission en urgence
  urgency : Hospital Patient --> Hospital
urgency(Self, Xp) == copy(Self, empty?=false, normal?=false, entry=Xp, oldState=Self))
```

```
class methods

;; init : ouverture de l'hopital
  init : Integer --> Hospital
init(Xmax) == newHospital(size = Xmax, empty?=true)
```

Figure 9 : Classe formelle pour Hospital

Notons qu'ici `size` a un statut particulier : il est sélecteur de champs ou une extension suivant l'état de l'hôpital.

7 Classes concrètes et mise en œuvre

Les phases précédentes nous fournissent une hiérarchie des classes et des opérations, ainsi qu'une description des classes formelles (aspect et méthodes secondaires). Nous avons tout ce qu'il nous faut pour réaliser une mise en œuvre du système voulu. Le codage peut se faire facilement dans un langage à classes du genre Eiffel, C++, CLOS, Smalltalk. Suivant le langage des caractéristiques sont différentes (typage, héritage, envoi de message, ..) elles posent quelques problèmes spécifiques mais en général assez simple à résoudre. Nous avons étudié la traduction automatique des classes formelles en Smalltalk V 4.0 et en Eiffel V 2.3.

7.1 Prototypage

La traduction d'une CF en CC est assez directe ceci permet d'obtenir un prototype rapidement du système. La traduction n'est en général pas complètement automatisable, cela dépend de la forme des axiomes. Il est facile de produire la déclaration de la structure de la classe et de l'interface des méthodes. La production automatique du code des méthodes est possible si les axiomes sont orientables en règles de réécriture.

Le principe de mise en œuvre d'une CF est simple :

1. Une CF est implantée par une CC.
2. La structure de la CC est un ensemble de champs privés correspondant aux sélecteurs de champs. La notion de champ conditionnel nécessite quelques artifices supplémentaires soit des unions disjointes, soit de l'héritage, ou encore des articles avec variants.
3. Pour chaque sélecteur une méthode de lecture est définie avec une éventuelle pré-condition.
4. L'invariant est une pré-condition de la création des instances. Il est possible de le traduire directement sur `Self` l'instance qui vient d'être crée, mais il est sûrement plus efficace de le décomposer en tenant compte des valeurs des champs de la structure.
5. Il convient de prévoir le codage des méthodes primitives.
6. Une méthode secondaire est traduite par une méthode concrète.

Ceci est un schéma simple et évidemment incomplet que l'on doit adapter suivant le langage cible choisi. Le prototype obtenu est fonctionnel et en général peu efficace.

7.2 Optimisations

Pour obtenir un produit plus utilisable des optimisations sont possibles. Il est préferrable de distinguer deux types d'optimisations : celles générales et celles plus spécifiques qui dépendent du langage cible. Il est plus intéressant de faire les optimisations générales sur la conception abstraite en CF. Voici quelques exemples de telles optimisations :

- Il est possible utiliser l'extension à effet de bord du modèle CF. Nous obtenons ainsi une conception avec des effets de bords que l'on peut coder d'une façon similaire à la section précédente. Les optimisations concernent l'économie de création des objets : il faut transformer le maximum de copies en modifications sans perturber la sémantique initiale (attention aux partages de structure).
- élimination de la recursivité : on connaît les techniques de variables tampons ou de passage de continuations qui peuvent être adaptées ici.
- Changement de la structure : ajout de champ privé pour optimiser l'appel d'une méthode ou inversement utilisation d'une méthode pour un champ de valeur constante. Le codage efficace des paramètres implicites est important, nous disposons de plusieurs solutions notamment des méthodes ou des variables de classes. Le codage optimal des structures avec champs conditionnels nécessite plus de soins.
- Faire apparaître des fonctions auxiliaires privées qui permettent de partager un calcul coûteux entre plusieurs méthodes.
- Des optimisations sont possibles pour l'héritage : réduction de la hiérarchie, optimisation de la recherche, ... Pour la recherche des méthodes on peut supposer que ce problème est bien résolu au niveau du langage de mise en œuvre [AR92a].

Les optimisations spécifiques à un langage seront réalisées directement sur le prototype.

7.3 Exemple de prototype en C++

A titre de démonstration nous présentons ici le codage de la classe formelle précédemment présentée en C++. Nous essayons de montrer que le codage peut se faire directement même si on a une classe assez complexe. Il est un peu plus difficile de faire de bonnes optimisations, car cela demande de plus une bonne connaissance du langage. La difficulté ici est le codage optimal de la structure avec champ conditionnel.

```
// la classe Hospital
// JCR - HOPITAL
// Hospital.h
// 5/96

#ifndef HOSPITAL_H
#define HOSPITAL_H

#include "Patient.h"

// version simpliste
// sans redéfinition de l'affectation,
// du constructeur de recopie et du destructeur
// On suit le schema de prototypage
// quelques noms changent,
// des pointeurs sont ajoutes,
class Hospital
{

//-----------------
// les champs prives de la structure
//-----------------
// le codage n est pas optimal
// a cause du champ conditionnel size
  bool _empty;
  int _size;
  bool _normal;
  Patient *_entry;
  Hospital *_oldState;

  public :

//-----------------
// les lecteurs publiques de la structure
//-----------------
// l etat global de l hopital
    bool empty () {return _empty;};

// le nombre maximal d entree normal
    int size () {return _size;};

// l etat d urgence du dernier patient entre
    bool normal () {return _normal;};

// le dernier patient entre
Patient *entry() {return _entry;};

// l etat precedent
      Hospital * oldState() {return _oldState;};

// on fabrique deux constructeurs
// celui d un hopital vide et
// un avec une entree dans lequel on integre
// le codage de l'invariant sur oldstate

// cas de hopital vide (fonction init)
// non optimal car du size partout
// les autres champs sont indefinis
     Hospital (int s)
             {if (s>=1)
{_size=s;
 _empty=1;}};

// une entree (avec codage de l invariant)
    Hospital (bool n, Patient *p, Hospital *o)
             {if (n && o->number() <= o->size() -1
                 ||  (!n) && o->number() <= o->size()
)
{_empty=0;
 _size=o->size();
 _normal=n;
 _entry=p;
 _oldState=o;};};

//-----------------
// autres methodes
//-----------------
// le nombre d'entre normale
      int number();

// le nombre d'urgence
      int urgencyNumber();

//le nombre total
      int totalNumber();

// la taille maximale des entrees normales est atteinte
     bool full();

// il y a une urgence
     bool thereIsUrgency();

// le prochain patient  a etre examine
  Patient *first();

// le soin du prochain patien
  Hospital *cure();

//admission d un patient normal
 Hospital *admit(Patient *p);

//admission d un patient en urgence
 Hospital *urgency(Patient *p);

// voir
 void describe()
  {printf(" Un hopital :\n");
  {if (empty())
      printf(" -- Taille %d et il est vide -- \n",
                     size());
    else
     {if (normal())
         printf("       une entree normale ");
       else printf("       une entree urgente ");
```

```
      entry()->describe();
      oldState()->describe();};};};
};

#endif
// la classe Hospital
// JCR - HOPITAL
// Hospital.cpp
// 5/96

#include "Hospital.h"

//-----------------
// autres methodes
//-----------------
// le nombre d entre normale
      int Hospital::number()
          {if (empty())
              return 0;
          else
          {return (oldState()->number()
                   + ((normal()) ? 1 : 0)) ;};
  }

// nombre d'urgence
      int Hospital::urgencyNumber()
          {if (empty())
              return 0;
           else
           {return (oldState()->urgencyNumber()
                    + ((normal()) ? 0 : 1)) ;};
           }

// nombre total
      int Hospital::totalNumber()
          {return (number() + urgencyNumber());}

// il y a une urgence
     bool Hospital::thereIsUrgency()
         {return (urgencyNumber() > 0);}

// le prochain patient  a etre examine
     Patient *Hospital::first()
        {if (!empty())
             {if (thereIsUrgency())
                {if (normal())
                    return (oldState()->first());
                   else
                    return (entry());}
                else
                return (entry());};}

// le soin du prochain patient
     Hospital *Hospital::cure()
        {if (!empty())
             {if (thereIsUrgency())
                {if (normal())
                    return (new Hospital(normal(),entry(),
                                         oldState()->cure()));
                   else
                    return (oldState());}
                else
                return (oldState());};}

// entree normale
      Hospital *Hospital::admit(Patient *p)
                {return (new Hospital(true,p,this));}

// entree urgente
      Hospital *Hospital::urgency(Patient *p)
                {return (new Hospital(false,p,this));}
// essai Hospital
// JCR 9/96

#include "Patient.h"
#include "Hospital.h"

int main()
{
printf(" TEST HOSPITAL \n");

Patient *jean = new Patient("Jean","Lebon");
Patient *pierre = new Patient("Pierre", "Kiroule");
Patient *jacques = new Patient("Jacques", "Adit");
Patient *paul = new Patient("Paul", "Truc");
Patient *serge = new Patient("Serge", "Manteau");
//jean->describe();
//pierre->describe();

// creation de l'hopital vide
Hospital *CHU = new Hospital(5);
//CHU->describe();
// une arrivee normale
CHU = CHU->admit(jean);
// une arrivee urgente
CHU = CHU->urgency(pierre);
// une arrivee normale
CHU  = CHU->admit(jacques);
// une arrivee urgente
CHU = CHU->urgency(paul);
// une arrivee normale
CHU = CHU->admit(serge);

CHU->describe();

// essais
printf("\n Nombre d'urgences %d et de patients normaux
%d \n", CHU->urgencyNumber(), CHU->number());


// tester tous les first dans l'ordre
printf(" Soin du 1er \n");
CHU->first()->describe();
printf(" Soin du 2eme \n");
```

```
CHU->cure()->first()->describe();
printf(" Soin du 3eme \n");
CHU->cure()->cure()->first()->describe();
printf(" Soin du 4eme \n");
CHU->cure()->cure()->cure()->first()->describe();
printf(" Soin du 5eme \n");
CHU->cure()->cure()->cure()->cure()->first()->describe();

}
```

8 Conclusion

Nous avons explorés quelques concepts utiles pour une conception plus rigoureuse en PPO. Une idée majeure est d'utiliser des concepts avec une double interprétation (TAG = automate/TAA et CF = TAA/classe) ce qui permet de faciliter les liens entre les niveaux de description (dynamique/fonctionnel et spécification/conception). L'apport du TAG est d'abord de faire cohabiter une description dynamique et une spécification algébrique de TAA. De plus l'écriture du TAA est méthodique et très automatisée. Le résultat est plus lourd qu'avec les techniques usuelles d'écriture de spécifications mais il est plus sûr. Des simplifications ultérieures sont possibles et prouvables, par exemple en utilisant la complémentarité et l'exclusivité des gardes. Dans notre exemple ceci permet de faire passer le nombre d'axiomes de dix à huit pour `first` et de vingt-trois à dix pour `cure`. Ces résultats sont comparables à ce qu'obtiendrait un «spécifieur» expérimenté utilisant les méthodes habituelles mais avec plus ou moins de sécurité quant au résultat.

Le TAG est un outil puissant, souple et bien adapté aux systèmes avec des pré-conditions complexes. L'utilisation du TAG pour l'extraction d'une spécification algébrique nous semble donc un apport pratique intéressant qui ne se limite pas à notre cadre. Le principe des TAG peut être étendu pour intégrer des aspects concurrents dans un contexte de contrôle centralisé. Une exploitation plus fine des informations de l'automate est possible : utilisation de chemin dans l'automate pour vérifier les termes droits, simplifier la génération des axiomes en utilisant les gardes exclusives, entre autres.

La méthode de conception des TAG en CF bien que non complètement formalisable est assez stricte. Des points délicats restent encore à préciser : critères de bonne structuration, critères de réutilisabilité, de manipulation des contraintes, ... L'étude de la représentation directe du TAA en classe devrait permettre de lever ces points et de donner une démarche avec un maximum de rigueur. L'utilisation du niveau intermédiaire des classes formelles est également un plus pour la facilité de conception mais également pour la qualité des produits réalisés.

Références

[ABR95] Pascal André, Franck Barbier, and Jean-Claude Royer. Une expérimentation de développement formel à objets. *Technique et Science Informatique*, 14(8):973–1005, octobre 1995.

[ACCR94] Pascal André, Dan Chiorean, Corina Cîrstea, and Jean-Claude Royer. Object-Oriented Design With Formal Classes. In *ConTI'94: International Conference on Technical Informatics*, volume V, pages 245–254, November 1994. Timisoara, Romania.

[ACR94] Pascal André, Dan Chiorean, and Jean-Claude Royer. The Formal Class Model. In *Joint Modular Languages Conference, Modula, Oberon & friends*, pages 59–78, Ulm, Germany, 1994. GI, SIG and BCS. September, 28-30.

[And95] Pascal André. *Méthodes formelles et à objets pour le développement du logiciel : Etudes et propositions.* Thèse de doctorat, IRISA, Université de Rennes I, 7 juillet 1995. préparée à l'IRIN - Nantes.

[AR92a] Pascal André and Jean-Claude Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In Andreas Paepcke, editor, *OOPSLA'92 Proceedings*, volume 27, pages 110–126, Vancouver, October 1992. ACM, ACM Press.

[AR92b] Michel Augeraud and Jean-Claude Royer. Une interprétation du concept de classe en termes de type abstrait. *Journées GDR Programmation et outils pour l'intelligence artificielle*, pages 13–27, March 1992. Nancy 18-20 mars Rapport GDR de Programmation.

[AR94a] Pascal André and Jean-Claude Royer. Building Formal Classes from Graphic Abstract Data Types. *Submitted to Journal of Object-Oriented Programming*, August 1994.

[AR94b] Pascal André and Jean-Claude Royer. La modélisation des listes en programmation par objets. In Pierre Cointe, Christian Queinnec, and Bernard Serpette, editors, *Journées Francophones des Langages Applicatifs (JFLA'94)*, number 11 in Collection Didactique, pages 259–285, Noirmoutier, 31 janvier - 1 février 1994. INRIA.

[AR95a] Pascal André and Jean-Claude Royer. Aide à la spécification algébrique de types abstraits par l'utilisation d'automates: principes, algorithmes et exemples. Rapport de recherche 98, Institut de Recherche en Informatique de Nantes, October 1995.

[AR95b] Pascal André and Jean-Claude Royer. Formal Concepts and Tools for Object-Oriented Analysis and Design. In *5th ICC: Object-Oriented Technology: Methods and Applications*. Tata McGraw-Hill, January 1995. 9-11 January, Hyderabad, India.

[AR97] Pascal André and Jean-Claude Royer. How To Easily Extract an Abstract Data Type From a Dynamic Description. Research Report 159, Institut de Recherche en Informatique de Nantes, September 1997.

[BB93] Gilles Bernot and Michel Bidoit. *Introduction aux spécifications algébriques*. Ecole des jeunes Chercheurs du Greco de Programmation, Université de Bordeaux I, 351, Cours de la libération 33405 Talence Cedex, March 1993. 29 mars - 8 avril 1993, Saint-Malo.

[BBC+92] Dider Bert, Michel Bidoit, Christine Choppy, Rachid Echahed, Jean-Pierre Finance, Marie-Claude Gaudel, Jean-Michel Hufflen, Jean-Pierre Jacquot, Michel Lemoine, Nicole Lévy, Jean-Claude Reynaud, Clément Roques, and Frédéric Voisin. Le projet SALSA : Structure d'AccueiL pour les Spécification Algébriques. *Journées du GDR Programmation avancée et outils pour l'intelligence artificielle*, pages 157–168, mar 1992. Nancy 18-20 mars Rapport du GRECO de Programmation.

[BCC+89] Michel Bidoit, Franck Capy, Christine Choppy, Stéphane Kaplan, F. Schlienger, and Frédéric Voisin. ASSPEGIQUE: An integrated specification environment. In *Proc. of the Third International Conference on Rewriting Techniques and Applications*, page 547. Springer-Verlag Lecture Notes in Computer Science 355, 1989.

[BCC90] Michel Bidoit, Franck Capy, and Christine Choppy. The design and specification of the ASSPEGIQUE data base. In *Proc. of the Int. Symp. on Design and Implementation of Symbolic Computation Systems (DISCO)*, pages 205–214. Springer-Verlag L.N.C.S. 429, 1990.

[Bid82] Michel Bidoit. Types abstraits algébriques : spécifications structurées et présentations gracieuses. *Colloque de l'AFCET: les mathématiques de l'informatique Paris*, pages 347–357, March 1982.

[CG88] Christine Choppy and Marie-Claude Gaudel. Impact des spécifications formelles sur le développement de logiciel. *BIGRE + GLOBULE*, 58:3–11, January 1988. IRISA-AFCET.

[Cho88] Christine Choppy. *ASSPEGIQUE User's Manual.* LRI, Orsay, rapport greco de programmation edition, August 1988.

[Ehr94] Hartmut Ehrig. DYNAMIC ABSTRACT DATA TYPE: An Informal Proposal. *Bulletin of the EATCS*, (53):162–169, June 1994.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of algebraic specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monography on Theoretical Computer Science.* Springer-Verlag, 1985.

[GL90] John Guttag and Barbara H. Liskov. *La maîtrise du développement de logiciel: abstraction et spécification.* Les Editions d'Organisations, 1990.

[Hoa72] C.A.R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

[Hor78] John Guttag J.J. Horning. The algebraic Specification of Abstract Data Types. *Acta informatica*, 10:27–52, 1978.

[Mus80] David Musser. On Proving Inductive Properties of Abstract Data Types. *in Proceedings of the Seventh Annual ACM Symposium on PoPL*, pages 154–162, January 1980. Las Vegas.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederik Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

[Roq92] Clément Roques. L'environnement ASSPEGIQUE+: Le valideur. L.R.I. Research Report 727, 1992.

[Roy92] Jean-Claude Royer. A New Set Interpretation for the Inheritance Relation and its Checking. *ACM OOPS MESSENGER*, 3(3):22–40, July 1992.

[Roy93] Jean-Claude Royer. Un exercice de spécification formelle de preuve et de conception à objets. Rapport de recherche 93-39, Institut de Recherche en Informatique de Nantes, Faculté des Sciences et des Techniques, Université de Nantes, September 1993.

[ST90] Donald Sanella and Andrzej Tarlecki. Algebraic specification and formal methods for program development: waht are the real problems? *Bulletin EATCS*, 41:134–137, June 1990.