Vers un modèle formel en analyse à objets

Pascal André
IRIN - Université de Nantes & Ecole Centrale
2, rue de la Houssinière, 44072 Nantes Cedex 03
andre@irin.univ-nantes.fr

Résumé

Nous proposons dans cette communication un modèle de description formelle des objets pour la spécification abstraite des objets. Les objets sont décrits sous un aspect dynamique par un automate d'états finis gardé puis sous un aspect fonctionnel par une spécification algébrique. L'originalité de la proposition est que le modèle dynamique sert à extraire systématiquement le modèle fonctionnel, ce qui assure la cohérence entre les deux aspects. Avoir un modèle formel permet le contrôle de l'héritage et la vérification de propriétés de la spécification. Un autre intérêt de ce processus de spécification est la construction assistée de spécifications algébriques tout en ayant une description visuelle et agréable du comportement des objets.

Mots-clés : Modèle formel à objets, spécifications algébriques, automates, construction assistée.

1 Introduction

L'utilisation de l'approche à objets en analyse et conception induit une nouvelle culture du développement, plus proche des contraintes industrielles. Il s'agit d'investir sur le long terme en définissant des composants généraux réutilisables et adaptables pour rentabiliser les efforts de développement. Un composant est défini par les services qu'il offre et les contraintes qu'il impose. Les services sont les opérations du composant. Les contraintes portent sur les services attendus d'autres composants pour réaliser ses propres services. La "culture composant" nécessite des outils pour spécifier les composants, les classer dans des bibliothèques, les retrouver pour les adapter à de nouvelles applications.

Les méthodes d'analyse et conception à objets, telles qu'OMT [Rumbaugh *et al.*, 1991], Shlaer & Mellor [Shlaer et Mellor, 1992], bien que très appréciées par des développeurs, ne répondent pas vraiment à ces exigences. D'une part, la description des composants du système n'est pas modulaire mais répartie dans divers modèles. Ces modèles expriment des aspects différents du systèmes : structure du système (classes et relations), scenarii, comportement dynamique (état/événement), découpage fonctionnel du système, etc. D'autre part, les modèles sont très riches mais ne sont pas définis formellement et pas toujours exploitables. Par exemple, les métaclasses qui en OMT factorisent des propriétés communes, n'ont pas systématiquement de correspondance en programmation ou en bases de données. Enfin, la description des interfaces entre composants, souvent en langage naturel, est tardive dans le processus De ces constats, il résulte les problèmes suivants. Premièrement, il est difficile d'assurer la cohérence globale et la correction des spécifications. Par exemple, le modèle fonctionnel et le modèle dynamique d'OMT sont redondants. Deuxièmement, rien ne différencie la description des composants et leur réalisation. A terme, seule la réalisation des composants donne une sémantique formelle de ce qu'il est capable de faire. Ceci engendre une faible interopérabilité des objets. Une description trop concrète de la structure des objets du systèmes entraîne une surspécification qui est à la fois contraignante pour la conception et un handicap pour la production de composants généraux. Troisièmement, le processus de développement ne peut être rigoureux et automatisé. La généralisation des composants et leur réutilisation reste à la charge des concepteurs. Elle n'est pas possible au niveau de l'analyse ou de la conception non détaillée[1].

La culture composant implique donc un effort dans la description des composants. Pour Meyer une spécification formelle des objets est indispensable "*Object-oriented Design is the construction of software systems as structured collections of abstract data type implementations*" [Meyer, 1988]. C'est dans cet esprit que nous avons défini le modèle des Types Abstraits Graphiques (TAG en abrégé). Il s'agit d'un modèle de spécification abstraite de composants que nous avons voulu proche des méthodes d'analyse et conception et qui permet d'introduire progressivement le formalisme. A notre avis, la spécification du comportement dynamique des objets c'est-à-dire le service qu'il offre au cours du temps, prime sur les autres aspects (structurel et fonctionnel). Nous avons choisi le formalisme des automates, pour sa simplicité, sa lisibilité, sa modularité et sa proximité avec les modèles dynamiques des méthodes à objets. La structure interne des objets importe peu dans une vue abstraite des objets. La partie fonctionnelle est donnée sous forme de spécification algébrique. Nous avons choisi ce style car aucune représentation particulière de l'état des objets n'est privilégiée, contrairement aux langages de spécification par modèle abstrait tels que Z ou VDM ou leurs extensions à objets. Une autre raison est l'existence d'outils de preuve relativement sophistiqués tels Larch Prover, Asspegique [Choppy, 1988], etc.

Les motivations du modèle TAG sont : avoir un modèle algébrique pour les objets, assurer des propriétés de cohérence et complétude des spécifications, faciliter l'écriture et la compréhension de spécifications formelles, décrire des aspects dynamiques et fonctionnels cohérents, permettre la preuve de spécification. Le modèle des TAG s'inscrit dans un processus de développement plus complet décrit dans [André *et al.*, 1995]. Ce processus comprend (1) une description rapide et approximative des objets en utilisant une méthode d'analyse à objets, (2) une formalisation des objets au moyen

1. Notons tout de même que les méthodes se focalisant uniquement sur la conception sont plus proches de l'approche composant.

des TAG, (3) une conception abstraite en termes de classes formelles [André *et al.*, 1994] -modèle à objets algébrique commun au langages de programmation- et enfin (4) une traduction en partie automatisée dans les langages de programmation à classes.

Dans la section 2, nous présentons rapidement le modèle, et plus particulièrement l'aspect dynamique, sous forme d'un automate. Nous illustrons notre propos par un exemple simple de construction : la classe `Cabane`. Nous montrons dans la section suivante comment construire une description fonctionnelle complète et cohérente du modèle grâce aux spécifications algébriques. Nous en discutons les propriétés dans la section 4. La section 5 est consacrée à une discussion sur les extensions en termes de structuration et d'héritage du modèle TAG. Le modèle proposé est adapté aux systèmes séquentiels, comme d'autres propositions que nous présentons dans la section 6.

2 Spécification du comportement dynamique des objets

Un **T**ype **A**bstrait **G**raphique est défini par un ensemble de méthodes et un ensemble d'axiomes exprimant la sémantique et les contraintes d'utilisation de ces méthodes. L'écriture des axiomes passe par la description d'un automate de comportement dynamique. Cet automate exprime les séquences autorisées d'envoi de message sur le receveur noté `self`. Le type en cours de définition est appelé **type d'intérêt**. Ce sont des types et non des classes car aucune structure en termes d'attributs n'est donnée. C'est pourquoi dans la suite nous utiliserons le terme d'opération plutôt que celui de méthode. Il n'y a pas non plus d'identité d'objet comme dans les langages de programmation à objets : les instances du TAG sont des valeurs.

Prenons l'exemple de la construction d'une cabane, dans un matériau tel que le bois ou le béton et selon un plan précisé. Le plan décrit des murs et un toit, constitué de façades. La construction de la cabane est progressive : les murs sont montés puis les façades sont couvertes. La figure 1 montre le comportement dynamique du TAG générique `Cabane`, paramétré par un type `Matériau`. L'exemple est un peu simpliste mais il permet d'illustrer notre propos sans égarer le lecteur dans des considérations telles que l'interaction entre deux composants. Nous supposons l'existence d'un type générique `Set[X]` pour désigner des ensembles d'objets.

2.1 Opérations des objets

Les opérations sont décrites par un profil, lors de la construction de l'automate. Une opération définie sur tous les états et qui ne change pas d'état, est dite **neutre** et n'apparait pas sur l'automate. Toute opération contient au moins une fois le type d'intérêt dans son profil, sinon cette opération relève forcément d'une autre abstraction de données. Dans l'exemple de la figure 1, les profils sont :

```
plan (façades : Set[Façade], murs : Set[Mur]) : Cabane
monter (self : Cabane, m : Mur) : Cabane
démonter (self : Cabane, m : Mur) : Cabane
```

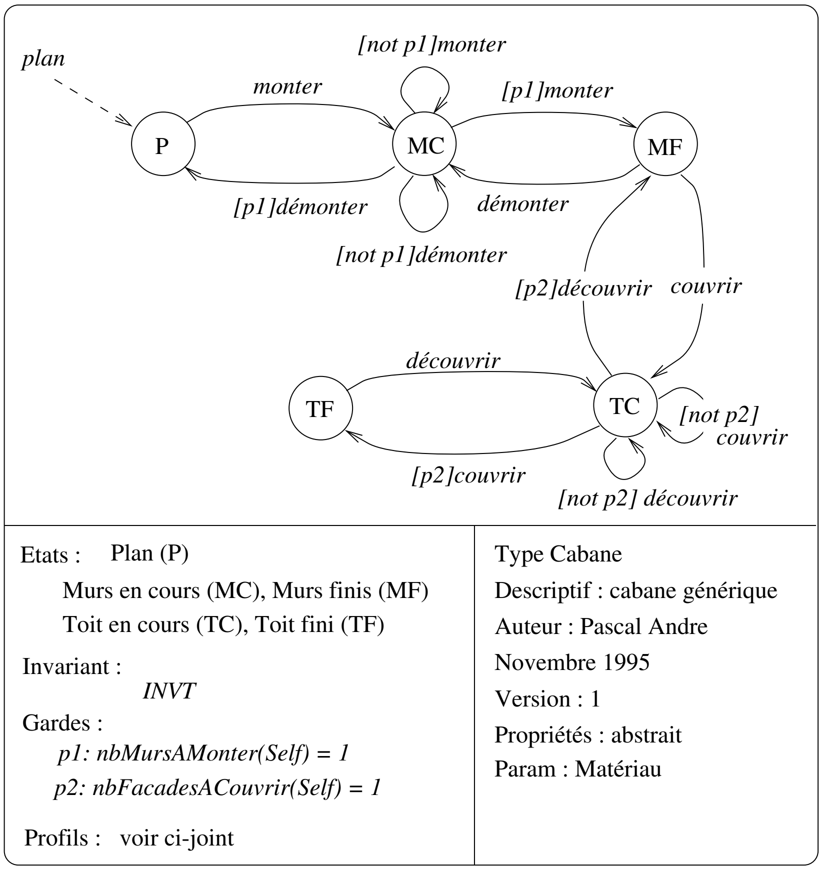

FIG. 1 - *La spécification graphique de la cabane.*

```
murs (self : Cabane) : Set[Mur]
nbMurs (self : Cabane) : Entier
mursAMonter (self : Cabane) : Set[Mur]
nbMursAMonter (self : Cabane) : Entier
mursMontés (self : Cabane) : Set[Mur]
nbMursMontés (self : Cabane) : Entier
couvrir (self : Cabane, f : Façade) : Cabane
découvrir (self : Cabane, f : Façade) : Cabane
façades (self : Cabane) : Set[Façade]
nbFaçades (self : Cabane) : Entier
façadesACouvrir (self : Cabane) : Set[Façade]
nbFaçadesACouvrir (self : Cabane) : Entier
façadesCouvertes (self : Cabane) : Set[Façade]
nbFaçadesCouvertes (self : Cabane) : Entier
matériau (self : Cabane) : Matériau
soufler (self : Cabane) : Cabane
brûler (self : Cabane) : Cabane
```

Comme d'usage en spécifications algébriques, un **constructeur** est une opération dont le résultat est le type d'intérêt. Il est dit **constructeur de base** s'il ne contient pas le type d'intérêt en paramètre. Un **observateur** est une opération dont le résultat est différent du type d'intérêt. Les constructeurs de base correspondent aux méthodes d'**instanciation** dans le modèle à classes. Les autres opérations correspondent aux méthodes d'instances et elles ont, par convention syntaxique, comme premier argument la variable receveur `self : Cabane`. L'ensemble des constructeurs Γ et l'ensemble des observateurs φ forment une partition des opérations. L'ensemble des constructeurs de base est noté Γ_b.

$\Gamma_b = \{$`plan`$\}, \Gamma = \{$`plan, monter, démonter, couvrir , découvrir, soufler, brûler`$\}$
$\varphi = \{$`murs, nbMurs, mursAMonter, nbMursAMonter, mursMontés, nbMursMontés, façades, nbFaçades, façadesACouvrir, nbFaçadesACouvrir, façadesCouvertes, nbFaçadesCouvertes, matériau`$\}$

2.2 Gardes

Afin de manipuler des automates finis nous utilisons des gardes. Une **garde** est une expression booléenne associée à une transition et notée [*exprbool*]. Elle est définie à partir des paramètres de l'opération correspondante et des observateurs du type d'intérêt accessibles dans l'état initial de la transition. Une transition est dite **passante** si la garde associée est instanciée à vrai.

2.3 Invariant

L'invariant est un prédicat que tous les objets du TAG doivent vérifier. Par exemple, le nombre de murs montés ou à monter (resp. de façades couvertes ou à couvrir) est toujours égal au nombre de murs (resp. de façades) du plan : `INVT(self) == equal(add(nbMursAMonter(self), nbMursMontés(self)), nbMurs(self)) AND equal(add(nbFaçadesACouvrir(self), nbFaçadesCouvertes(self)), nbFaçades(self))`.

2.4 Contrôle de la spécification

A ce niveau de la description seules des vérifications syntaxiques peuvent être établies, contrôlant notamment la cohérence entre les profils et l'automate, et sur l'automate lui-même (au moins un constructeur de base, pas de gardes fausses, états accessibles, profils des opérations correspondant à leur utilisation...). Ces contrôles sont détaillés dans [André et Royer, 1995].

3 Spécification algébrique du comportement des objets

Jusqu'à maintenant, nous avons donné les règles d'utilisation des opérations (ordre, paramètres, conditions dans les gardes). Il reste à donner une sémantique à ces opérations. Nous le faisons en construisant une spécification algébrique.

Nous supposons le lecteur familier avec les spécifications algébriques de types abstraits de données. Consulter par exemple [Gaudel, 1990; Ehrig et Mahr, 1985]. Une spécification algébrique comprend une signature (noms des types utilisées, profils des opérations), les préconditions des opérations et des axiomes. La signature et les préconditions définissent les règles de manipulation du type abstrait de données. Les axiomes sont des formules logiques définies sur les opérations, qui leur donnent une sémantique. Nous nous restreignons aux algèbres autorisant les axiomes conditionnels positifs (clauses de Horn) pour lesquels la cohérence et la complétude peuvent être démontrées. Ces axiomes sont de la forme $\mathtt{e}_1$ `AND` $\mathtt{e}_2$ `AND ... AND` $\mathtt{e}_n$ `==> e` où $\mathtt{e}_1$, $\mathtt{e}_2$, ... $\mathtt{e}_n$ et `e` sont des équations `terme1 == terme2`.

Dans la section 3.1, nous calculons la signature et les préconditions directement à partir de la spécification graphique du TAG. Le calcul des axiomes est décrit dans la section 3.2. Les vérifications de propriétés de la spécification (section 4) et les preuves de théorèmes sont effectuées dans un environnement de spécification algébrique.

3.1 Calcul de la signature et des préconditions

L'automate est représenté par des observateurs booléens. A chaque état est associé un **observateur d'état** de même nom que l'état, qui est vrai si l'objet est dans cet état ou encore si la valeur de l'objet est du sous-type correspondant à cet état. Les observateurs d'état sont ajoutés aux profils, de même que les gardes et l'invariant. La signature est la suivante (`S` est l'ensemble des types utilisés.) :

```
S = {Cabane, Booléen, Entier, Matériau, Façade, Mur, Set[Façade], Set[Mur]}
plan             : Set[Façade] Set[Mur] → Cabane
monter           : Cabane Mur → Cabane
...
brûler           : Cabane → Cabane
P                : Cabane → Boolean
...
TF               : Cabane → Boolean
p1               : Cabane → Boolean
p2               : Cabane → Boolean
INVT             : Cabane → Boolean
```

La génération des préconditions ci-après est automatique. Les opérations neutres, les observateurs d'état, les gardes et l'invariant sont des opérations totales : elles ont pour précondition vrai. Les opérations suffixées par (‡) correspondent aux méthodes abstraites et ne sont pas applicables : elles ont pour précondition faux. Pour les autres opérations, il suffit d'examiner toutes les transitions et de construire incrémentalement les préconditions par union des gardes de cette opération, compte-tenu des états de départ. Les termes suffixés par (†) ont été simplifiés par une procédure de décision interactive (un outil de simplification est utile ici). Par exemple, la précondition générée pour `monter` est `R(self) OR ( MC (self) AND p1(self) ) OR ( MC (self) AND NOT p1(self) )`. Elle s'écrit plus simplement `R(self) OR MC (self)`; ce qui signifie qu'un mur peut être monté si les murs sont en plan ou en cours de construction. Voici les préconditions des opérations de la cabane :

```
∀ self : Cabane, façades : Set[Façade], murs : Set[Mur], f : Façade, m : Mur
precondition(plan(façades, murs)) = true
precondition(monter(self, m)) = P(self) OR MC(self)
precondition(démonter(self,m)) = MF(self) OR MC(self)†
precondition(murs(self)) = true
// idem pour nbMurs, mursAMonter, nbMursAMonter, mursMontés, nbMursMontés //
precondition(couvrir(self,f)) = MF(self) OR TC(self)†
precondition(découvrir(self,f)) = TF(self) OR CC(self)†
precondition(façades(self) = true
// idem pour nbFaçades, façadesACouvrir, nbFaçadesACouvrir //
// idem pour façadesCouvertes, nbFaçadesCouvertes //
precondition(matériau(self) = false‡
precondition(soufler(self)) = false‡
precondition(brûler(self)) = false‡
precondition(MC(self)) = true // idem pour P, MF, TC, TF p1 p2 INVT//
```

3.2 Calcul de l'axiomatique

L'axiomatique est un ensemble d'axiomes que doit vérifier la spécification. Deux propriétés sont à assurer : (1) complétude : il ne manque pas d'axiomes et (2) cohérence : vrai=faux ne peut être démontré. Pour cela nous utilisons une méthode de construction systématique des axiomes inspirée de Guttag [Guttag et Horning, 1978] et Bidoit [Bidoit, 1982]. Elle consiste à définir la sémantique d'une opération en donnant sous forme d'axiomes son résultat sur un terme quelconque du type d'intérêt. Ces termes sont de la forme $\mathtt{c}_n\mathtt{(c}_{n-1}\mathtt{(...(c}_0\mathtt{(args}_0\mathtt{),...),args}_{n-1}\mathtt{),\ args}_n\mathtt{)}$, où les c_i sont des constructeurs et $\mathtt{c}_0$ un constructeur de base. Ce sont des chemins dans l'automate de comportement dynamique. Nous utilisons une induction sur les constructeurs pour diminuer le nombre d'axiomes.

Définition 3.1 (Générateur, famille génératrice)
Un sous-ensemble Ω de Γ_T, l'ensemble des constructeurs d'un type T, est une famille génératrice de T ssi tout terme clos de type T peut être prouvé égal à un terme dont le symbole de tête est un générateur (i.e. élément de la famille).

Séparer les observateurs en observateurs primaires et secondaires est aussi intéressant. En particulier, l'héritage des observateurs secondaires ne pose aucun problème. Nous ne connaissons pas de technique systématique pour partitionner les observateurs tandis que l'automate sert à la détermination de la famille génératrice.

3.2.1 Détermination des générateurs

L'ensemble des générateurs doit au moins permettre d'atteindre chaque état de l'automate. C'est une condition nécessaire mais non suffisante, car les gardes ont une sémantique dynamique. Prendre les gardes en compte revient à associer un ensemble de conditions logiques à chaque chemin, ensemble difficile à simplifier statiquement. L'idée intuitive de l'algorithme de calcul de la famille génératrice est de parcourir le graphe, en partant d'états initiaux (un ou plusieurs) et de calculer les chemins minimaux pour atteindre chaque état. Nous proposons un algorithme glouton de détermination de cet ensemble, en partant de Γ_b pour atteindre tous les états en utilisant le moins possible de nouveaux générateurs. Cet algorithme donne de bons résultats dans les cas pratiques étudiés. Dans notre exemple une famille génératrice est {`plan`, `monter`, `couvrir`}.

3.2.2 Génération des termes : le principe de la G-dérivation

Comme nous l'avons vu précédemment, le langage engendré par l'automate réduit aux générateurs donne tous les termes du type d'intérêt. A l'inverse, une G-dérivation d'un terme contenant une variable `self` du type d'intérêt et se trouvant dans un état `e(self)` est une substitution de `self` par un générateur menant à l'état `e(self)`. Cette définition correspond à celle de Bidoit [Bidoit, 1982] appliquée à un unique argument : le receveur. Les algorithmes de G-dérivation correspondent ainsi à des parcours inverses de l'automate de comportement réduit aux transitions des générateurs. La G-dérivation va servir à automatiser la construction des axiomes. Pour calculer le résultat d'une opération sur un terme quelconque, dont l'état est connu, nous effectuons des G-dérivations du terme jusqu'à pouvoir fournir interactivement la partie droite de l'axiome.

3.2.3 Détermination des axiomes

Nous distinguons deux cas : a) les opérations issues de l'automate et b) les autres opérations. Les axiomes des opérations issues de l'automate (observateurs d'états, gardes et invariant) sont calculés directement. Les axiomes des autres opérations (hormis les générateurs) sont obtenus par G-dérivation.

Opérations issues de l'automate L'axiome d'une garde ou de l'invariant est sa définition. Les axiomes des observateurs d'état expriment le résultat de chaque observateur sur chaque générateur. Si le générateur ne mène pas dans l'état, le résultat est faux. Sinon, le résultat est la disjonction des prédicats associés aux transitions supportant ce générateur. Le prédicat associé à une transition est la conjonction de la garde et de l'observateur d'état de départ de la transition. Prenons l'observateur d'état `MC`.

```
∀ self : Cabane, façades : Set[Façade], murs : Set[Mur], f : Façade, m : Mur
MC1: MC(plan(façades, murs)) == false
MC2: MC(monter(self,m)) == P(self) OR ( MC(self) AND NOT p1(self) )
MC3: MC(couvrir(self, f)) == false
p1(self) == nbMursAMonter(self) = 1
p2(self) == nbFaçadesACouvrir(self) = 1
INVT(self) == add(nbMursAMonter(self), nbMursMontés(self)) = nbMurs(self) AND
add(nbFaçadesACouvrir(self), nbFaçadesCouvertes(self)) = nbFaçades(self).
```

Autres opérations Construire l'axiomatique c'est donner le résultat de chaque opération sur une variable receveur dans chaque état où elle est définie. L'algorithme construit ainsi systématiquement les parties gauches des axiomes. Si le concepteur ne peut pas fournir la partie droite de l'axiome, une G-dérivation est appliquée pour toutes les transitions menant à cet état. Par exemple, examinons le cas du constructeur `démonter`. Il est défini sur les états `MF` et `MC`.

1) Dans l'état `MF`: dém1: `MF(self) == true ==> démonter(self,m) ==??`
Le résultat de l'opération dans cet état ne pouvant être connu facilement une G-dérivation est appliquée, qui remonte à l'état `MC`.
dém1: `MC(self) AND p1(self) == true ==> démonter(monter(self, m1),m) ==`
`IF equal(m,m1) THEN self ELSE monter(démonter(self, m),m1)`

2) Dans l'état `MC`, le constructeur `démonter` est supporté par deux transitions :
a- dém2: `MC(self) AND p1(self) == true ==> démonter(self,m) ==??`
dém2a: `P(self) AND p1(monter(self, m1)) == true ==>`
`démonter(monter(self, m1),m) == self`
dém2b: cas `NOT equal(m,m1)` impossible : pas d'axiome
dém2c: cas `MC(self) AND NOT p1(self)` impossible : pas d'axiome

b- dém3: `MC(self) AND NOT p1(self) == true ==> démonter(self,m) ==??`
dém3a: cas `P(self)` impossible : pas d'axiome
dém3b: `MC(self) AND NOT p1(self) AND NOT p1(monter(self, m1)) == true ==>`
`démonter(monter(self, m1),m) == IF equal(m,m1) THEN self`
`ELSE monter(démonter(self, m),m1)`

Cet exemple montre la nécessité d'un prouveur pour simplifier les expressions et démontrer les situations (telles que 2b, 2c, 3a) générées par l'automate mais impossible à cause des gardes. Nous procédons de même pour les autres constructeurs et les observateurs. Les méthodes abstraites (`matériau, souffler, brûler`) ne donnent pas lieu à des calculs d'axiomes. Nous les ignorons dans le raisonnement pour l'instant.

4 Propriétés générales d'une spécification

L'écriture de spécifications modulaires se fait en décrivant le type d'intérêt à partir d'autres types existant. Il faut prouver que cet enrichissement n'introduit pas d'erreurs dans les autres types [Gaudel, 1990]. Les contraintes hiérarchiques expriment le fait que l'utilisation d'une spécification ne modifie pas la classe de ses modèles, c'est-à-dire que ajouter de nouvelles opérations (les observateurs de la sorte utilisatrice sont constructeurs de la sorte utilisée) n'ajoute pas de nouvelles valeurs (suffisante complétude de Guttag [Guttag et Horning, 1978]). La contrainte hiérarchique est la propriété d'absence de circuit dans le graphe de d'utilisation.

Dans un contexte objet, cette contrainte hiérarchique est trop forte, la collaboration d'objets implique souvent d'exprimer des types mutuellement dépendants. Par exemple, supposons qu'un rectangle soit construit à partir de deux points. Il est naturel en objet, d'avoir une méthode `extrémité : Point` $\times$ `Point` $\rightarrow$ `Rectangle` dans la spécification du type `Point`, qui à partir de deux points crée un rectangle. Cet observateur sera traduit par un appel à un générateur de base du type `Rectangle`, de profil `new : Point` $\times$ `Point` $\rightarrow$ `Rectangle`. C'est donc un observateur secondaire. La relation d'utilisation inclue le type `Point` sur `Rectangle` et le type `Rectangle` sur `Point`. Il existe d'autres cas typiques où la contrainte hiérarchique est trop forte : ce sont les définitions récursives de types de données, conflits entre un type et un sous-type (exemple : une liste non-vide est définie à partir d'une liste et d'un élément, c'est en même temps une liste), ou les conflits entre un paramètre et un type paramétrique (exemple : une liste de liste). Nous affaiblissons la contrainte en la posant sur les types de générateurs uniquement.

Une signature est **raisonnable** si tous les types de la signature sont atteints. Pour cela il suffit qu'il existe un élément dans chaque type. Le type est alors dit **habité**.

Définition 4.1 (contrainte hiérarchique TAG)
Une spécification est dite bien structurée s'il existe un constructeur de base, muni d'une garde non trivialement fausse et dont les types paramètres sont habités et ne dépendent pas du type d'intérêt.

Par construction, la signature produite par un TAG `TI` vérifiant la contrainte hiérarchique TAG est raisonnable. En effet, la spécification possède au moins un constructeur de base $\texttt{cb : T}_1 \texttt{ ... T}_n \rightarrow \texttt{TI}$. Chaque type $\texttt{T}_i$ du constructeur de base étant habité, il possède au moins une valeur $\texttt{v}_i$. Donc le terme $\texttt{cb(v}_1\texttt{, ..., v}_n\texttt{)}$ existe et le type `TI` est habité. Ces hypothèses sont équivalentes à celles de Musser [Musser, 1980]. Ce dernier autorise des dépendances mutuelles entre spécifications car la définition inductive est basée sur la structure de l'expression plutôt que sur une hiérarchie dans l'organisation des spécifications. Dans l'hypothèse d'une signature raisonnable, la déduction équationnelle usuelle est bien fondée.

Intéressons-nous maintenant aux propriétés de complétude et de cohérence. En général, deux alternatives sont utilisées pour montrer ces propriétés : le raisonnement équationnel ou les systèmes de réécriture [Jouannaud et Lescanne, 1986]. Nous utiliserons la seconde alternative pour deux raisons : premièrement la forme des axiomes permet assez naturellement de remplacer les axiomes par des règles de réécriture conditionnelle, deuxièmement les systèmes de réécriture possèdent des algorithmes puissants (Knuth-Bendix) pour compléter un système non cohérent. Un autre intérêt des systèmes de réécriture est de pouvoir exécuter les spécifications et valider la spécification.

Supposons les équations orientées de gauche à droite. Chaque axiome $\texttt{t}_1 \texttt{ == t}_1\texttt{'}$ $\texttt{AND .. AND t}_n \texttt{ == t}_n\texttt{' ==> l == r}$ devient une règle de réécriture. $\texttt{t}_1 \texttt{ == t}_1\texttt{' \& ..}$ $\texttt{\& t}_n \texttt{ == t}_n\texttt{' ==> l} \rightarrow \texttt{r}$. Une réécriture est un remplacement d'un sous-terme d'un terme par un autre terme selon une règle du système de réécriture. Un terme est irréductible si aucune réécriture ne peut lui être appliquée. Les formes irréductibles ou normales sont calculées à partir des termes par réécriture successive. La réécriture permet de démontrer un théorème équationnel $t = t'$ par un calcul "t et t' ont la même forme normale" (théorème de Church-Rosser) ou encore de prouver des théorèmes inductifs par différentes techniques.

Cette affirmation est valide si le système de réécriture est **convergent**. Cette propriété est la conjonction de deux propriétés : la **terminaison** et la **confluence**. La terminaison exprime le fait que les calculs sont finis. Cette propriété est indécidable mais elle peut être démontrée en exhibant un ordre de réduction sur les termes. S'il existe un unique état initial, alors un parcours de l'automate en profondeur d'abord donne un ordre sur les générateurs, qui est parfois suffisant. La confluence exprime le fait que tout terme se réécrit en un seul terme irréductible, quelque soit l'ordre d'application des règles de réécriture. Si le système termine, il suffit alors de montrer la confluence locale en étudiant uniquement les règles de réécriture. Une paire critique est un couple de règles tel qu'un terme quelconque puisse se réécrire indifféremment par l'une ou l'autre des règles. Le système est localement confluent si les paires critiques sont confluentes.

Théorème 4.2 (confluence locale)
Le système de réécriture associé à une spécification algébrique TAG est localement confluent si l'automate est déterministe.

Cette propriété découle de l'algorithme de construction automatique des parties gauches des axiomes. Il y aura une paire critique si pour un état et une opération donnés, deux transitions sont possibles. Comme l'automate est déterministe, les gardes de ces transitions sont mutuellement disjointes. Donc un seul des axiomes est applicable.

5 Extension du modèle

5.1 Structuration d'automates

Un des reproches souvent fait aux automates est de devenir ingérables lorsque le nombre d'états est trop important. En réponse à ce problème, nous proposons des notations pour structurer les automates. Ces notations ne sont pas ad hoc comme dans beaucoup de modèles mais basées sur des relations calculables : le produit d'automates, l'agrégation ou la généralisation d'états ou de transitions.

L'agrégation d'états est proche de la notion de composante connexe dans les graphes. En pratique, le graphe est souvent fortement connexe (tous les états sont accessibles de n'importe quel état) car les automates décrivent des cycles de vie en général réinitialisables. Nous utilisons alors la relation suivante de connexité plus faible mais plus efficace en pratique.

Définition 5.1 (états frères)
Deux états `e` et `e'` sont frères si `e'` est l'unique successeur de `e` et `e` est un successeur de `e'`.

L'algorithme de restructuration est alors le suivant. L'application de la relation produit des groupes d'états. Ce ne sont pas des classes d'équivalence car ils ont des intersections non-vides. Pour chaque état d'une intersection non vide, il est demandé à l'utilisateur de choisir le groupe d'affectation. Cet algorithme est ré-itéré jusqu'à ce que le graphe soit suffisamment simple. Il est souhaitable de définir une notion d'interface d'états pour pouvoir encapsuler des automates dans des états comme dans les statecharts de Harel. La difficulté est de pouvoir calculer une telle relation entre états.

Pour les transitions, trois relations permettent de factoriser l'automate. Soient `t=<op, g, orig, dest>` et `t'=<op', g', orig', dest'>` deux transitions, et `succ` la fonction successeur.

Définition 5.2
t et t' sont ***similaires*** *ssi*
$op = op' \wedge g = g' \wedge (dest = dest' \vee (orig = dest \wedge orig' = dest'))$
t et t' sont ***séquentielles*** *ssi*
$orig' = dest \wedge succ(orig) = \{dest\} \wedge succ(orig') = \{dest'\}$
t et t' sont ***parallèles*** *ssi* $orig' = orig \wedge dest = dest' \wedge t \neq t'$

5.2 Héritage

L'héritage de TAG sert à construire incrémentalement des spécifications, à partager les axiomes et à classer les spécifications en hiérarchies de spécialisation. Ce concept n'est défini que dans un contexte opérationnel, dans lequel un système de réécriture est associé à chaque spécification. L'évaluation des spécifications se fait par réécriture. Les règles sont classées par opération et les axiomes relatifs à une opération sont mis dans une seule spécification. Il y a d'abord évaluation du receveur pour chercher les règles à appliquer. S'il n'y a pas d'axiomes pour l'opération considérée du type du receveur, les règles sont cherchées dans la super-spécification et ainsi de suite. Cette solution nécessite une évaluation par valeur i.e. la forme normale des arguments est calculée avant d'appliquer l'opération.

L'héritage est une notion complexe, qui couvre à la fois le sous-typage, le raffinage, l'extension ou la redéfinition. Nous avons dénombré quatre cas d'héritage pour les langages à objets basés sur les spécifications algébriques :

1. extension : spécialisation d'une spécification complètement définie (e.g. `Pixel inherit Point`),

2. concrétisation : spécialisation d'une spécification incomplète (e.g. `Entier inherit Comparable`),

3. raffinage : changement de représentation (e.g. `EnsembleListe inherit Ensemble`),

4. duplication : renommage des sortes et des opérations (e.g. `FileCinema inherit FileFIFO`).

Dans le modèle des TAG, nous avons commencé par définir la relation d'héritage sur la partie algébrique. L'héritage est contrôlé : ce n'est pas une relation posée a priori. Nous souhaitons que l'héritage entraîne le sous-typage en respectant le principe de substitution : *"toute opération applicable au super-type est applicable au sous-type"*. Les quatre cas ci-dessus se résument à deux situations : si la super-spécification est finiment engendrée alors c'est une **extension** sinon c'est une **concrétisation**. Les spécifications non finiment engendrées correspondent aux classes abstraites des modèles à objets i.e. sans constructeurs de base.

Définition 5.3 (héritage d'extension)
Une spécification $spec_1$ hérite par extension de $spec_2$ ssi il existe une fonction surjective $f_{abst} : termes_{spec_1} \rightarrow termes_{spec_2}$ telle que

$$\forall t_1 \in termes_{spec_1},\ \exists t_2 \in termes_{spec_2},\ f_{abst}(t_1) = t_2.$$

L'héritage d'extension est proche de la coercition de type et de la notion de représentation dans les spécifications algébriques. Nous utilisons une fonction d'abstraction de la sous-spécification vers la super-spécification. La preuve de l'héritage d'extension se fait en exprimant chaque générateur de la sous-spécification comme une extension des générateurs de la super-spécification. Si le modèle à objets comporte des attributs (variables d'instances) alors la relation ci-dessus se traduit par une comparaison de ces attributs; ce qui est évidemment plus simple à démontrer. C'est ainsi qu'est défini l'héritage dans le modèle des classes formelles [André *et al.*, 1994].

Définition 5.4 (héritage de concrétisation)
Une spécification $spec_1$ concrétise $spec_2$ ssi $\forall t_1 \in termes_{spec_1},\ t_1 \in termes_{spec_2}$.

Lorsqu'une spécification concrétise une spécification non finiment engendrée, la fonction d'abstraction est l'identité. Il s'agit alors de montrer que toutes les opérations de la super-spécification s'appliquent aux termes de la sous-spécification. Premièrement, chaque générateur de la super-spécification est exprimé en fonction des générateurs de la sous-spécification. Ces générateurs correspondent en quelque sorte à des méthodes abstraites qu'il faut définir. Deuxièmement, toutes les opérations qui ont un générateur de la super-spécification en partie gauche sont redéfinies avec les nouveaux générateurs pour que la spécification soit complète. Ici aussi, les vérifications sont plus simples avec des attributs.

6 Travaux connexes

Certains langages de spécification algébrique et à objets ne comportent que des extensions structurelles comme l'héritage ou l'encapsulation. D'autres intègrent parfois des mécanismes comme l'envoi de message à des niveaux de description plus opérationnels. Les définitions de l'héritage varient d'un auteur à l'autre, mais en général le sous-typage est séparé de l'héritage d'implantation, parfois appelé raffinage. OBJ2

[Futatsugi *et al.*, 1985] définit uniquement le sous-typage strict, par inclusion d'ensembles supports. La notion de sous-classe en GSBL [Clerici et Orejas, 1988] correspond à l'héritage bien que les auteurs distinguent définitions complètes et incomplètes de sortes ou même d'opérations. Le sous-typage d'OS/OP [Breu, 1991] est une notion plus forte que la concrétisation des TAG, car elle est définie sur des spécifications de classes complètes, tandis que l'héritage d'OS/OP est équivalent à l'extension. Dans le langage NDL [Parisi et Pierantonio, 1991], les auteurs séparent sous-typage, héritage de spécialisation et héritage d'implantation. La différence entre les deux premiers porte uniquement sur une partie interface de classe, correspondant à des opérations redéfinissables dans les sous-classes.

Les langages ci-dessus ne proposent pas de notation graphique pour décrire les types de données. La prise en compte d'aspects dynamiques n'est pas abordée dans ces langages. Le passage de la spécification se fait par raffinage, il n'y a pas de véritable étape de conception, telle que nous la proposons dans le passage des spécifications TAG aux classes formelles dans [André *et al.*, 1995]. De même, la distinction entre niveaux d'abstraction est totalement floue, du fait du raffinage. Par contre, des travaux théoriques plus solides valident la sémantique de ces langages et notamment celle de l'héritage [Breu, 1991; Parisi et Pierantonio, 1991].

Les travaux les plus proches sont ceux de [Paech et Rumpe, 1994]. Les auteurs définissent un type de données par des attributs, des opérations et un automate. Les états de l'automate constituent une partition des valeurs des attributs, et non des étapes dans un cycle de vie. Cette définition implique un choix immédiat dans la représentation des objets alors que les états des TAG sont purement abstraits. Des rôles établissent des vues partielles de l'automate mais les automates ne sont pas structurés par des opérateurs comme dans les TAG. L'héritage est défini par une relation de spécialisation de type qui sert aussi au raffinement. La relation de spécialisation de type implique une inclusion des opérations et attributs du super-type dans le sous-type et une relation de spécialisation d'automates. La relation de spécialisation d'automates indique que le nombre d'états n'est pas étendu et que les transitions sont préservées. Un état non initial peut disparaître si un invariant est renforcé. Un état du super-automate peut être partitionné en plusieurs états du sous-automate. Les liens entre les deux automates sont difficiles à assurer car les auteurs autorisent à la fois l'extension et la restriction. De plus les définitions sont basées sur des langages d'automates alors que le contrôle n'est possible qu'à partir de règles précises et concrètes. Le comportement fonctionnel n'est pas abordé par les auteurs tandis qu'il est construit progressivement dans les TAG.

D'autres travaux visent à apporter de la rigueur dans les modèles et le processus d'analyse et conception à objets. La méthode Fusion [Derek Coleman *et al.*, 1994] propose un processus précis. L'analyse décrit les objets par un modèle entité/association enrichi et l'interface du système par des expressions régulières d'opérations. Ces opérations sont décrites au plus bas niveau par des assertions, malheureusement en langage naturel. La conception consiste ensuite à répartir les opérations dans les objets en utilisant des graphes d'interaction. L'idée est simple, mais s'applique uniquement aux systèmes séquentiels. Les notations sont définies informellement et aucune vérification automatique n'est possible sur la sémantique des modèles. La méthode Syntropy [Cook et Daniels, 1994] est une autre approche en ce sens. Le processus consiste à construire trois modèles : le modèle essentiel décrivant le domaine de l'application, le modèle de spécification décrivant le logiciel et le modèle d'implantation. Chaque modèle est décrit par des vues et notamment une vue structurelle (*type views*) et une vue dynamique (*Statecharts*) mieux définies qu'en OMT. Les vues sont enrichis au fil du développement. Par exemple, les événements du modèle essentiel s'enrichissent d'une description interne (effets en pré/post conditions et génération d'événements) puis décrits par des envois de messages séquencés dans le modèle d'implantation. L'apport de Syntropy est le typage systématique des concepts via la théorie des ensembles : attributs, paramètres et résultats d'opération, objets. Des règles de conformité de type sont énoncées. Les modèles et le processus sont plus riches que celui des TAG. La cohérence semble pouvoir être assurée mais le formalisme sert plus à la précision qu'à la vérification. Il se pose le problème des représentations particulière pour les objets (les attributs devient des observateurs). De plus le processus nécessite beaucoup d'expérience et d'intervention pour passer au modèle d'implantation.

7 Conclusion

Nous proposons dans ce papier un modèle pour la spécification abstraite d'objets. Ce modèle est inspiré à la fois des modèles en analyse à objets et des types abstraits de données. Le modèle est basé sur la notion d'état abstrait, ce qui donne une représentation à la fois plus opérationnelle que celle des types abstraits algébriques purs mais plus naturelle et plus facile à construire, et à la fois plus abstraite que celle des spécifications par modèle abstrait ou même à objets. Nous avons donné une définition de l'héritage qui peut servir de base au raffinement des spécifications : un objet peut avoir une nouvelle implantation tant qu'elle respecte sa définition abstraite. L'évaluation se fait par réécriture conditionnelle avec un mécanisme de recherche de la méthode à appliquer à partir de l'objet receveur (évaluation par valeur).

La méthode peut sembler un peu lourde sur de petits exemples. Mais le processus de spécification est naturel, méthodique et largement automatisable. Ce qui le rend praticable par des non-spécialistes des spécifications formelles. La spécification obtenue possède de bonnes propriétés (complétude suffisante, cohérence). Le noyau présenté ici doit être étendu afin de faciliter la structuration des spécifications et la concurrence. Le traitement de la concurrence dans les TAG est simple et se base sur les produits synchronisés d'automates. Ces produits permettent de synchroniser les opérations se déroulant en même temps sur les automates de deux objets distincts. Une synchronisation se traduit par une opération générale du système incluant ces deux objets et faisant appel à l'opération de chacun des objets. Cette façon de faire est lourde mais habituelle dans les spécifications formelles. Nous étudions aussi l'héritage du comportement dynamique. Actuellement nombre d'auteurs utilisent une notion d'équivalence observationnelle. Nous souhaitons une définition structurelle de l'héritage des automates, qui bien qu'un peu rigide, doit essayer de coller à la définition algébrique.

La spécification algébrique que nous calculons est plate, c'est-à-dire constituée d'un seul type. Notre style de spécification permet aussi de construire des spécifications algébriques ordonnées. Une spécification ordonnée est une hiérarchie de types liés par une relation d'héritage. Cette hiérarchie est obtenue en considérant chaque état comme un sous-type du type d'intérêt et en groupant des états. Dans notre exemple, la méthode de construction donnerait une hiérarchie en un TAG abstrait (Cabane) et trois TAG concrets (ConstructionDesMurs, ConstructionDuToit, CabaneFinie). Nous obtenons ainsi une description plus riche permettant de détecter plus d'erreurs dans le contrôle de type et une réutilisation plus fine des objets. Ce travail est similaire à la conception des TAG en classes formelles, qui consiste à déterminer une structure aux objets par raffinements successifs. L'ensemble du processus est présentée dans [André *et al.*, 1995].

Références

[André *et al.*, 1994] Pascal André, Dan Chiorean, et Jean-Claude Royer. The Formal Class Model. In *Joint Modular Languages Conference*, pages 59–78, Ulm, Germany, Septembre 1994. GI, SIG and BCS.

[André *et al.*, 1995] Pascal André, Frank Barbier, et Jean-Claude Royer. Une expérimentation de développement formel à objets. *Techniques et Sciences Informatique*, 14(8):973–1005, Octobre 1995.

[André et Royer, 1995] Pascal André et Jean-Claude Royer. Aide à la spécification algébrique de types abstraits par l'utilisation d'automates : principes, algorithmes et exemples. Rapport de Recherche RR-98, IRIN, Octobre 1995. (53 pages).

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

[Breu, 1991] Ruth Breu. *Algebraic Specification Techniques in Object-Oriented Programming Environments*, volume 562 de *Lecture Notes in Computer Science*. Springer Verlag, Juin 1991. Thesis, Universität Passau, Germany.

[Choppy, 1988] Christine Choppy. *ASSPEGIQUE User's Manual*. LRI, Orsay, Août 1988. Rapport Greco de Programmation.

[Clerici et Orejas, 1988] S. Clerici et F. Orejas. GSBL: An Algebraic Specification Language Based on Inheritance. In *Proceedings of ECOOP '88*, volume 322 de *LNCS*, pages 78–92, Oslo, Août 1988. Springer Verlag.

[Cook et Daniels, 1994] Steve Cook et John Daniels. *Designing Object Systems* OBJECT-ORIENTED MODELLING WITH SYNTROPY. Object-Oriented Series. Prentice Hall, 1994.

[Derek Coleman *et al.*, 1994] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes et Paul Jeremaes. *Object-Oriented Development* THE FUSION METHOD. Object-Oriented Series. Prentice Hall, 1994.

[Ehrig et Mahr, 1985] Harmut Ehrig et Bernd Mahr. *Fundamentals of Algebraic Specifications 1 : Equations and Initial Semantics*, volume 6 of *EATCS Monograph on Theorical Computer Science*. Springer-verlag, 1985.

[Futatsugi *et al.*, 1985] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, et José Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Janvier 1985. SIGACT, SIGPLAN.

[Gaudel, 1990] Marie-Claude Gaudel. *Algebraic Specifications*, chapitre 22. Software Engineers Reference Book. Butterworths, Mc Dermid, J. (Ed.), Mars 1990.

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

[Jouannaud et Lescanne, 1986] Jean-Pierre Jouannaud et Pierre Lescanne. La réécriture. *Technique et Science Informatique*, 5(6):433–452, 1986.

[Meyer, 1988] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall International, 1988.

[Musser, 1980] David R. Musser. On proving Inductive Properties of Abstract Data Types. In *Proceedings of the Seventh Annual ACM Symposium on PoPL*, pages 154–162, Las Vegas, Janvier 1980.

[Paech et Rumpe, 1994] Barbara Paech et Bernhard Rumpe. A new Concept of Refinement used for Behavior Modelling with Automata. In *FME'94 Symposium*, éditeurs Naftalin. Maurice, Tim Denvir, et Miquel Bertran, volume 973 de *LNCS*, pages 306–325, Barcelona, Spain, Octobre 1994. Springer Verlag.

[Parisi et Pierantonio, 1991] F. Parisi Presicce et A. Pierantonio. An Algebraic View of Inheritance and Sybtyping in Object Oriented Programming. In *3rd European Software Engineering Conference, ESEC'91*, volume 550 de *LNCS*, pages 364–379, Milan, Italy, 1991. A. van Lamsweerde and A. Gugetta (Eds.), Springer Verlag.

[Rumbaugh *et al.*, 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, et William Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall International, 1991.

[Shlaer et Mellor, 1992] Sally Shlaer et Stephen J. Mellor. *Object Lifecycles*. Yourdon Press Computing. Prentice Hall, 1992. Modeling the World in States.