Spécification multi-formalismes : l'exemple de la facturation

Pascal André
Equipe Génie Logiciel, Méthodes et Spécifications Formelles
IRIN - Université de Nantes
2, rue de la Houssinière
BP 92208, 44072 Nantes Cedex 03
andre@irin.univ-nantes.fr

25 septembre 1996

RÉSUMÉ

Nous traitons dans ce rapport la spécification d'un problème sous différents formalismes. Le cas étudié est la facturation de commandes. Un énoncé très concis est proposé. Quatre formalismes sont utilisés pour modéliser ce cas d'étude : Merise, OMT, Z et TAG/CF. Pour chaque modélisation, nous présentons la technique de spécification, nous l'appliquons à l'exemple et nous notons quelques remarques. Enfin nous comparons les modélisations et concluons sur le problème du multi-formalisme.

MOTS-CLÉS

Multi-formalismes, méthodes de développement, spécifications formelles, modèles à objets, automates, Merise, OMT, Z, TAG/CF.

ABSTRACT

In this report, we deal with the specification of a problem with different formalisms. The case study is invoicing orders. A short statement of the problem is provided. Different models of invoicing are given using four formalisms: Merise, OMT, Z and TAG/CF. For each modelling, we present the specification technique, we apply it to the example and sum up some remarks. Last we compare the models and we conclude on multi-formalism.

KEY WORDS

Multi-formalisms, development methods, formal methods, object orientation, finite state machines, Merise, OMT, Z, TAG/CF.

SPÉCIFICATION MULTI-FORMALISMES : l'exemple de la facturation

Pascal André
Equipe Génie Logiciel, Méthodes et Spécifications Formelles
IRIN - Université de Nantes
2, rue de la Houssinière
BP 92208, 44072 Nantes Cedex 03
andre@irin.univ-nantes.fr

RÉSUMÉ. *Nous traitons dans ce rapport la spécification d'un problème sous différents formalismes. Le cas étudié est la facturation de commandes. Un énoncé très concis est proposé. Quatre formalismes sont utilisés pour modéliser ce cas d'étude : Merise, OMT, Z et TAG/CF. Pour chaque modélisation, nous présentons la technique de spécification, nous l'appliquons à l'exemple et nous notons quelques remarques. Enfin nous comparons les modélisations et concluons sur le problème du multi-formalisme.*
MOTS-CLÉS : *Multi-formalismes, méthodes de développement, spécifications formelles, modèles à objets, automates, Merise, OMT, Z, TAG/CF.*
ABSTRACT. *In this report, we deal with the specification of a problem with different formalisms. The case study is invoicing orders. A short statement of the problem is provided. Different models of invoicing are given using four formalisms: Merise, OMT, Z and TAG/CF. For each modelling, we present the specification technique, we apply it to the example and sum up some remarks. Last we compare the models and we conclude on multi-formalism.*
KEY WORDS : *Multi-formalisms, development methods, formal methods, object orientation, finite state machines, Merise, OMT, Z, TAG/CF.*

---

1 Introduction

La modélisation consiste à donner une représentation d'un problème afin de le rendre plus compréhensible et de permettre de raisonner sur ce problème. Au cours du développement de systèmes d'information, plusieurs modèles sont élaborés en analyse, conception et réalisation. Ces modèles sont complémentaires. Soit ils expriment les mêmes choses mais à des niveaux d'abstraction différents (i.e. plus ou moins proche de la machine), soit ils expriment des aspects différents du système à modéliser (dans les systèmes d'information il est d'usage de prendre en compte des aspects données, traitements, temps...).

Nous nous situons dans ce document à un niveau très abstrait dans lequel le problème est posé dans un énoncé informel en langage naturel. Appelons analyste la (les) personne(s) qui modélise(nt). La modélisation est donc la représention par l'analyste d'un énoncé dans un formalisme donné. Par définition, toute modélisation est subjective et dépend de deux facteurs : le formalisme utilisé et la connaissance de l'analyste. Plus précisément l'influence du formalisme vient des paradigmes (concepts) sous-jacents qui orientent la modélisation en la dirigeant sur certains concepts et éventuellement une démarche de modélisation. La connaissance de l'analyste est un ensemble assez flou de choses, comprenant son expérience sur ce type d'énoncé, sa familiarité avec le formalisme utilisé, etc. Cette connaissance va influer sur la compréhension du sujet par l'analyste. Enfin, si l'analyste effectue différentes modélisations dans des formalismes différents, l'ordre dans lequel il les fait oriente sa compréhension.

Nous étudions dans ce rapport l'influence du formalisme sur la modélisation. Nous illustrons ce problème en modélisant un exemple selon différents formalismes. L'exemple traité est celui de la facturation d'une commande. Nous utilisons différents formalismes issus de différentes méthodes. Pour simplifier nous traitons le cas par méthode et non par formalisme car une méthode inclut différents formalismes complémentaires (dont la cohérence reste parfois floue). Les méthodes abordées ici sont Merise [TRC83], OMT [RBP+91], Z [Spi89] et TAG/CF [ABR95]. Les deux premières sont dites semi-formelles tandis que les dernières sont formelles.

Notre réflexion est organisée comme suit. Nous rappelons l'énoncé dans la section 2. Pour chaque méthode traitée, nous faisons un bref rappel de la méthode puis nous présentons l'utilisation de la méthode pour l'étude de cas, notamment en décrivant succinctement les formalismes utilisés et enfin nous construisons différents modèles pour la spécification du cas. La section 3 est dédiée à Merise. L'angle des données est décrit par un modèle entité/association tandis que les traitements sont modélisés par un réseau de Petri. Dans la section suivante, consacrée à OMT, nous donnons le modèle des objets et le modèle dynamique de ces objets. Puis nous détaillons la spécification Z en termes de schéma d'état et schémas d'opérations. Enfin, nous terminons par une spécification TAG/CF avec une description dynamique par automates et une description fonctionnelle par spécification algébrique. Il s'agit ici aussi d'un modèle à objets. En guise de conclusion, nous faisons une synthèse sur la contribution de chaque méthode à la résolution du problème de la facturation.

2 Présentation rapide du sujet

Le cas d'étude concerne la facturation de commandes. L'énoncé, posé par Henri Habrias, est le suivant : "*... qui consiste à facturer des commandes, i.e. à faire passer des commandes de l'état 'en attente' à l'état 'facturée' ...*". La concision de l'énoncé implique évidemment des choix de modélisation et des questions différentes. Voici une liste non exhaustive de questions légitimes sur cet énoncé. Qu'est-ce qu'une commande ? Qu'est-ce que la facturation de commande ? Des frais sont-ils pris en compte pour une commande ? Qui commande et qui paie ? La remise faite au client apparaît-elle dans la commande ?

Nous n'y répondons pas d'emblée, le sujet est affiné progressivement au cours de la modélisation. En fonction des questions suggérées par le formalisme, des hypothèses sont ajoutées, qui enrichissent la modélisation et donc la compréhension du sujet. Nous essayons de faire apparaître ce cheminement dans nos spécifications. L'ordre de réalisation des spécifications est le suivant : Merise, Z, OMT, TAG/CF. Les questions que nous nous sommes posés sont implicitement liées à cet ordre, et les choix de conception aussi. Nous conseillons donc au lecteur de lire les modélisations dans cet ordre. Il apparaîtra alors la distinction entre problèmes issus des règles de gestion (domaine du problème) et ceux du au formalisme de modélisation.

3 Merise

3.1 Présentation de Merise

3.1.1 La méthode

Merise [TRC83] est une méthode de conception et de développement de systèmes d'information. Merise est une méthode complète, comprenant par exemple des éléments de la gestion de projets. Elle est applicable à des domaines variés : systèmes d'information, systèmes temps réels, etc. L'intérêt dans notre contexte est que ces formalismes sont connus de beaucoup de personnes. Les particularités de la méthode sont : une séparation des données et des traitements avec des vérifications croisées, l'étude du système existant et du système futur, l'adaptation de la méthode au projet étudié et enfin l'évolution permanente de la méthode en fonction des technologies nouvelles (des bases de données relationnelles aux objets). Merise s'articule autour de deux démarches : l'une par étape et l'autre par niveau d'abstraction. La démarche par étape comprend entre autres l'étude préalable, l'étude détaillée et la réalisation. Les modèles produits peuvent appartenir à différents niveaux d'abstraction. La démarche par niveau d'abstractions distingue trois niveaux :

- Le niveau conceptuel décrit l'application indépendamment de l'organisation.
- Le niveau organisationnel décrit l'application dans l'organisation (aspects temporels, humains).
- Le niveau opérationnel décrit l'implantation de l'application.

A chaque niveau correspond un modèle pour les données, un modèle pour les traitements et éventuellement d'autres modèles (diagrammes de flux, diagramme de circulation des informations, diagrammes d'interactions, etc.).

3.1.2 Restriction au cas d'étude

Dans le cadre de l'énoncé de la facturation, il s'agit de représenter le problème uniquement au niveau conceptuel. A ce niveau, les diagrammes de flux exprimant les liens entre acteurs et les diagrammes de circulation des informations sont inutiles car hors du contexte. Nous nous focalisons sur les deux modèles principaux : le modèle conceptuel des données et le modèle conceptuel des traitements. Le modèle des données décrit les informations du système par un schéma Entité-Association. Le modèle des traitements décrit les opérations du système et leurs enchaînements par un réseau de Petri enrichi. En général, l'analyste commence d'abord par le modèle conceptuel des données qu'il affine ensuite par le modèle conceptuel des traitements. C'est la démarche que nous adoptons ici.

3.1.3 Formalismes utilisés

Modèle conceptuel des données Le formalisme Entité/association structure les informations en attributs, entités et associations. Une entité est un objet abstrait ou concret du réel perçu considéré comme un tout et ayant une existence autonome. Une association est une liaison entre plusieurs entités. Les attributs (ou propriétés) sont des qualités d'une entité ou une association. Des types correspondent à chacun de ces concepts, mais par abus de langage le terme type sera omis dans notre discours. La clé d'une entité est un ensemble d'attributs suffisant pour différencier deux entités du même type d'entité. Des contraintes d'intégrité traduisent des règles du réel perçu. Certaines sont notées sur le schéma, c'est le cas des cardinalités ou de l'appartenance à un type, d'autres sont exprimées en langue naturelle. Les cardinalités expriment le nombre de fois qu'une entité apparaît dans une association. Elles ont le sens suivant :

- 0 : aucune association ne fait référence à l'entité,
- 1 : une association fait référence à l'entité,
- n : plusieurs associations font référence à l'entité.

Pour chaque couple (entité,association) une cardinalité minimale et une cardinalité maximale sont fournies. Les cardinalités minimales sont habituellement 0 ou 1 et les cardinalités maximales 1 ou n. La notation graphique associée est schématisée dans la figure 1.

Nous avons deux entités : commande et facture. Ce sont bien des choses autonomes et cohérentes. Le fait que la facture soit passée par un client est un lien entre les deux entités car cette information ne dépend pas que de la commande ou de la facture. Les attributs soulignés sont la clé.

La cardinalité se lit dans le sens de l'entité vers l'association. Par exemple, la cardinalité (1,1) de `Commande` vers `passée_par` signifie qu'une commande est passée par un et un seul client, ou dit autrement "à toute commande correspond un client et un seul". Les propriétés attendues sont la non-redondance des informations, l'accessibilité...

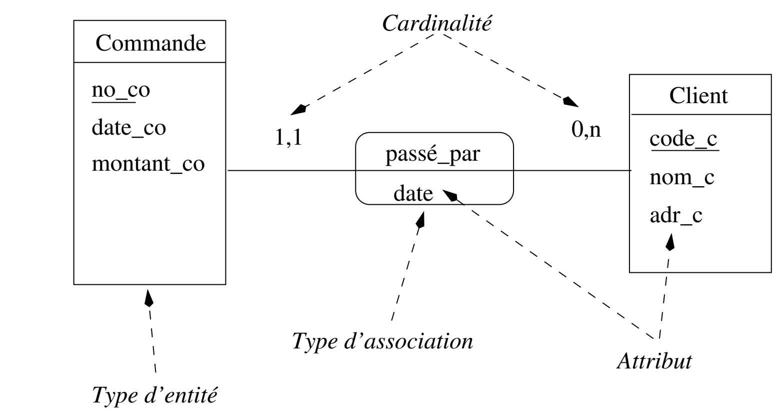


Figure 1 : Formalisme Entité/Association (Merise)

Modèle conceptuel des traitements Les traitements sont modélisés par un réseau de Petri (RdP en abrégé) enrichi, qui modélise l'évolution dynamique du système, l'enchaînement des opérations et les fonctions du système. Un événement est le fait que quelque chose se passe dans le système ou dans son environnement. Le type d'événement est représenté par une place du RdP dont les jetons représentent les occurrences de ce type d'événement. Une opération est un ensemble d'actions effectuées en réponse à un ou plusieurs événements. Les opérations sont les transitions du RdP, elles ont des places d'entrée (événements requis) et des places de sortie (événements produits). Une synchronisation est une condition à réaliser pour qu'une opération démarre. Elle sert à la mise à feu des transitions du RdP. Les transitions consomment des jetons des places d'entrée et produisent des jetons dans les places de sortie. Il est possible de conditionner l'émission d'événements et de quantifier la consommation (participation) et la production d'événements (cardinalités). La notation est résumée dans le schéma de la figure 2.

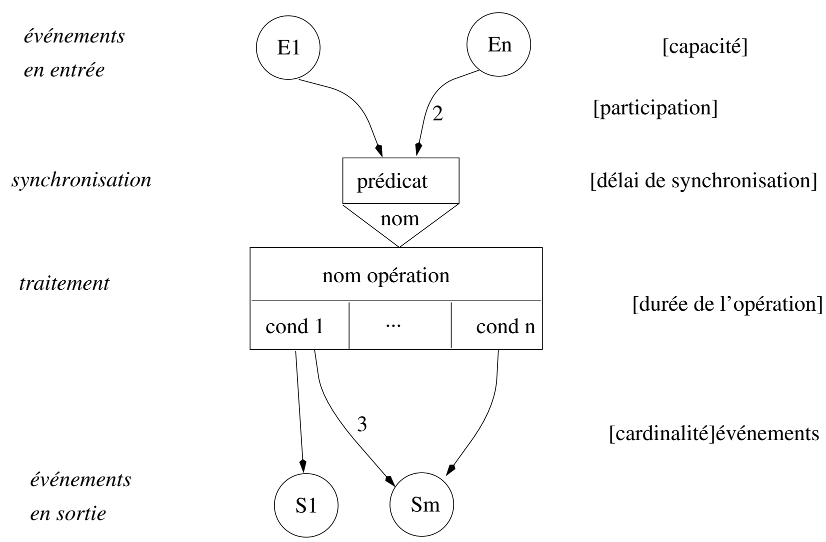


Figure 2 : Formalisme Réseau de Petri enrichi (Merise)

Le fonctionnement est le suivant : s'il y a suffisamment d'occurrence d'événements (de jetons) en entrée et que la condition de synchronisation est vérifiée alors la transition est tirable. Parmi les transitions tirables, une transition est mise à feu (le choix est non-déterministe). Il peut y avoir plusieurs niveaux de structuration si le réseau est complexe : les opérations sont décomposables en sous-diagrammes. Sont alors produits par exemple un diagramme d'enchainement des processus, modèle conceptuel du processus. Les propriétés attendues sont celles des réseaux de Petri (vivacité, terminaison, absence de famine).

Le reste de la notation et le processus de modélisation sont décrits dans [TRC83]. Nous allons maintenant construire le modèle des données dans la section 3.2 puis celui des traitements dans la section 3.3. Plusieurs versions sont déclinées en fonction des hypothèses prises pour répondre aux questions posées lors de la modélisation.

3.2 Modèle Conceptuel des Données

La modélisation débute par le recensement des entités candidates issues de la lecture du texte : commande, produit, facture. Ensuite, nous raisonnons sur ces entités : quels sont leurs attributs (une commande a des lignes indiquant des produits, des articles et des quantités), quels sont les liens entre ces entités (une ligne de commande est un lien entre la commande et un produit), y-t-il d'autres entités associées (une commande est passée par un client). Apparaissent alors des notions de clients, lignes de commande et de facture, des dates diverses, des adresses de client, de livraison de facturation. Ces informations ne sont pas extraites de l'énoncé mais de notre expérience et de l'idée que nous nous faisons de la facturation. Ce qui est souvent le cas en pratique, car les cahiers des charges sont généralement incomplets.

Nous commençons alors une construction initiale et intuitive du modèle Entité/Association. A ce stade, plusieurs questions plus ou moins proches du sujet nous sont apparues. Nous les classons par entités concernées.

1. Gestion des commandes :
   - Les notions comme la date de commande, les frais, les remises, la TVA, le montant total sont-elles d'intérêt ?
   - Une ligne de commande concerne-t-elle un seul produit ?
   - Le prix d'un produit à la commande est-il celui du produit en stock ?
   - Peut-on commander une quantité de produit supérieure à celle du stock ?
2. Gestion des clients :
   - Tient-on compte des clients en général ? des seuls clients qui commandent ?
   - Le client qui commande est-il celui qui paie ?
   - Une commande (resp. une facture) peut-elle être ventilée sur plusieurs clients ?
   - Comment sont gérées les remises, avoirs, etc...
3. Gestion des factures :
   - Fait-on de la multi-facturation (plusieurs factures pour une commande) ?
   - Fait-on de la facturation multi-commandes (plusieurs commandes pour une facture) ?
   - Une commande (resp. une facture) peut-elle être ventilée sur plusieurs clients ?
   - Y-a-t-il des factures d'avoir, si oui sont elles-liées à des factures et/ou des commandes (achat/remboursement) ?
4. Gestion des commandes/factures/produits :
   - La commande fait-elle apparaître le prix ? la TVA ? la remise ? la ristourne ? l'avoir ?

- Le prix à la facturation est-il celui à la commande ?
- La TVA est-elle globale ou par produit ?
- Toute commande est-elle (sera-t-elle) facturée ?
- Une commande (resp. une facture) peut-elle être ventilée sur plusieurs clients ?
- Les commandes entières sont-elles les seules facturées ?
- Une ligne de commande peut-elle être facturée en partie ?
- Une ligne de facture fait-elle référence à plusieurs commandes ? Par exemple, un même produit commandé plusieurs fois sera facturé pour une quantité égale à la somme des quantités commandées pour ce produit.
- A-t-on dans le temps des commandes ou factures sans contenu ?

Les réponses à ces questions donnent des versions différentes du modèle. La gestion de stock n'est pas considérée ici. Plus le modèle est large, plus les traitements doivent être rigoureux et vice-versa. Les types sont simples et sont donc soit omis soit non détaillés pour ne pas surcharger les schémas. Les identifiants sont soulignés.

3.2.1 Version stricte

Parmi les hypothèses ci-dessus, nous prenons les plus simples d'un point de vue formalisme Entité/Associations, c'est-à-dire celles qui engendrent le moins d'entités, d'associations, d'attributs et de contraintes fonctionnelles. L'énoncé le plus restrictif est le suivant. Chaque commande donne lieu à une seule facture. Une commande est la donnée d'un ensemble de lignes de commande. Chaque ligne de commande correspond à un produit commandé pour une quantité donnée et à un prix donné. Aucune commande vide n'est acceptée. Des produits non commandés sont gérés car présents au catalogue (ce catalogue n'est pas modélisé). Une commande est effectuée par un seul client. Elle est facturée ensuite à ce même client. Le fait qu'elle soit facturée est indiqué par un attribut booléen `est_facturé`. Cet énoncé se modélise simplement avec trois entités et deux associations.

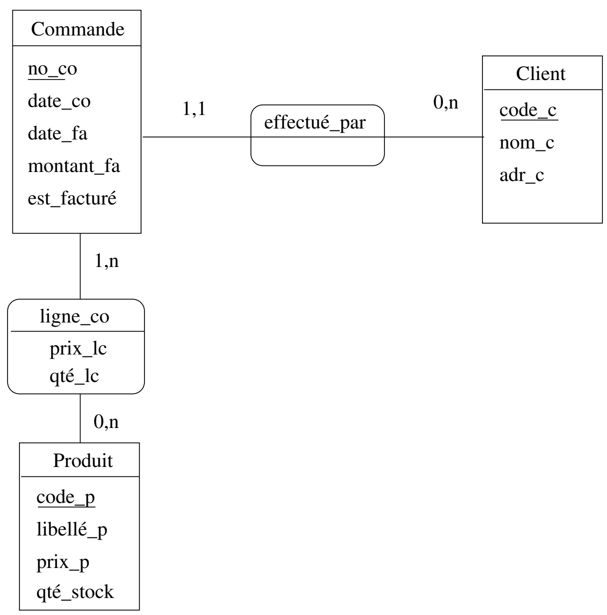


Figure 3 : Modèle Conceptuel des Données de la facture - cas 1 (Merise)

Les cardinalités minimales de un expriment que la commande est forcément effectuée par un client et contient au moins une ligne de commande. La facturation de commandes consiste donc à faire passer le booléen `est_facturé` de faux à vrai.

3.2.2 Version souple

Nous donnons maintenant une version plus générale dans la figure 4 en prenant des hypothèses moins strictes. Une commande est effectuée par un seul client mais peut être imputée à plusieurs clients pour deux raisons : il y a multi-facturation d'une commande et plusieurs payeurs pour une facture. Une facture peut aussi regrouper plusieurs commandes pour un paiement en fin de période (mois, trimestre) par exemple. Ces choix de gestion entraînent une séparation entre la commande et ses lignes. Chaque ligne de commande concerne un produit et correspondra à une ligne de facture. Une ligne de facture concerne un seul produit mais éventuellement sur plusieurs commandes. Il y a donc un assouplissement très important des contraintes. Cet assouplissement entraîne de réelles difficultés dans le contrôle de la cohérence. La seule règle globale à vérifier est qu'un produit commandé devra être facturé. Le laps de temps entre la commande et la facturation est un paramètre du système. Cet énoncé se modélise en six entités et huit associations.

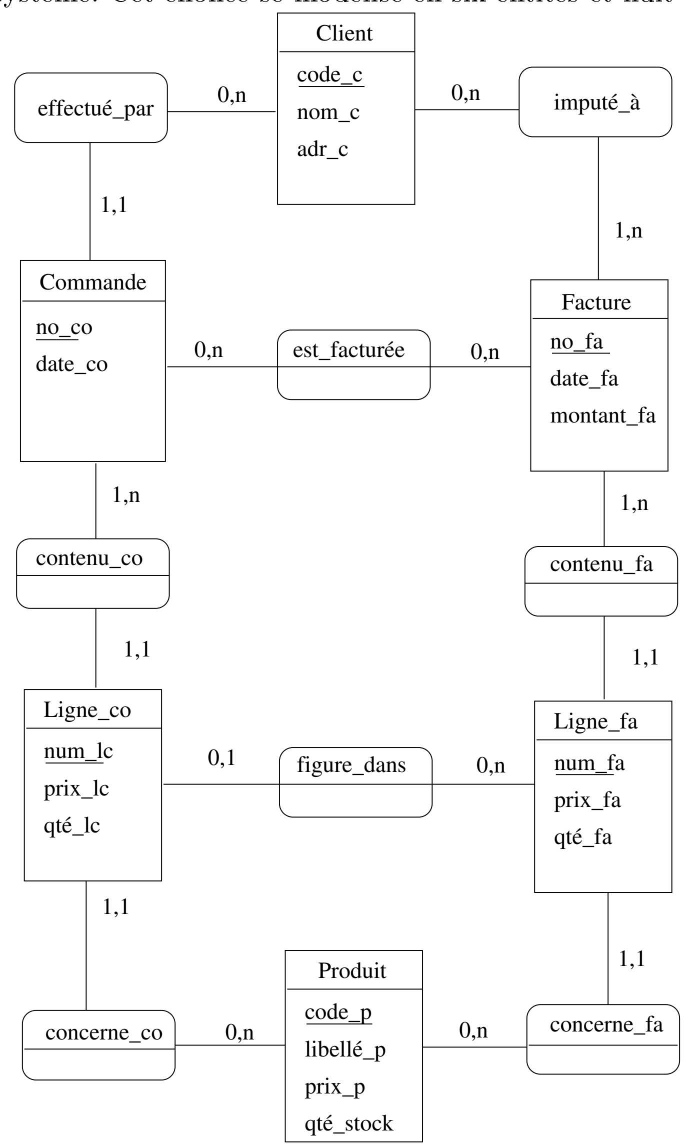


Figure 4 : Modèle Conceptuel des Données de la facture - cas 2 (Merise)

Nous n'avons pas ajouté de concepts par rapport au schéma de la figure 3 mais distingué commandes et factures, qui restent d'ailleurs reliées. Ce qui donne de l'importance à la ligne de commande qui passe du statut d'association dans la figure 3 au statut d'entité dans la figure 4. L'assouplissement des règles de gestion entraîne aussi l'ajout d'attributs et donc des problèmes de gestion des données. Remarquons enfin, que les cardinalités sont peu significatives dans beaucoup de cas `0, n`. Ceci réduit l'espace de contrainte sur le modèle des données mais ajoute des invariants non modélisables. Par exemple, quand peut-on considérer qu'une commande est entièrement facturée ? De nouvelles questions apparaissent alors : Les prix à la facturation sont-ils ceux à la commande ? Ce sont bien là des questions relatives au formalisme utilisé.

3.3 Modèle Conceptuel des Traitements

Au niveau conceptuel il n'est pas tenu compte de l'organisation (qui fait les opérations, où, à quel moment précis). Par exemple, savoir si c'est une personne, un employé ou un ordinateur qui passe les commandes n'est pas notre préoccupation. Nous étudions uniquement ce qui est fait et la causalité des opérations et événements. Par exemple, il existe un événement `nouvelle_commande` qui déclenche la prise en compte d'une commande. Nous notons les événements émis et reçus par le système mais les documents restent implicites. La modélisation débute donc par le recensement des opérations du système et leurs conditions d'activation. Nous notons immédiatement deux processus distincts (deux lots d'opérations) : l'un pour les commandes et l'autre pour les factures. Les commandes sont saisies, validées et imprimées. Les factures sont établies à partir des commandes et imprimées. Voici une ébauche de modèle conceptuel des traitements.

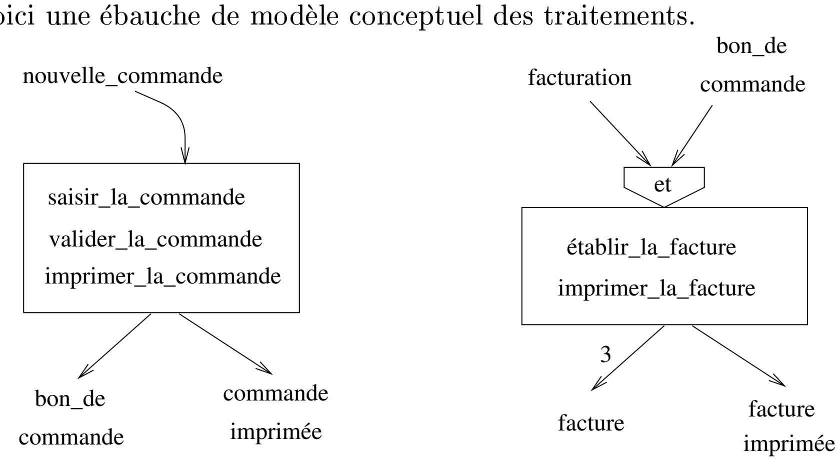


Figure 5 : Modèle Conceptuel des Traitements de la facture - cas 0 (Merise)

Cette modélisation est insuffisante pour répondre aux questions de l'analyste. En effet des choix apparaissent ici en supplément aux questions générées par la construction du modèle conceptuel des données. Ils sont issus de la modélisation en réseaux de Petri.

1. Gestion des commandes.
   - La commande est-elle saisie et vérifiée ligne par ligne ou globalement ?
   - Comment sont traitées les lignes de commandes ou les commandes erronées ?
   - Que signifie valider une commande ?
   - La commande est-elle réalisée en une seule fois ou à différents moments ?
   - Peut-on modifier une commande ?
2. Gestion des clients.

- Doit-on saisir des informations concernant le client (à la commande) ?
- Doit-on vérifier l'existence des clients pour une commande ou une facture ?
- Y-a-t'il des motifs de rejets d'une commande (client débiteurs, ...) ?

3. Gestion des factures :
   - La facturation est-elle périodique ou à la demande ?
   - Le paiement effectif est-il pris en compte (bon de caisse, récépissés divers) ?
   - La facturation se fait à partir de plusieurs commandes ?
4. Gestion des commandes/factures/produits :
   - La facturation est-elle consécutive à la commande ?
   - Comment la cohérence est-elle gérée dynamiquement ?
   - En combien d'exemplaires sont produits les commandes et factures ?

Nous supposons que le paiement et la gestion des clients font l'objet d'autres processus du système.

3.3.1 Version stricte

Nous reprenons les hypothèses du MCD de la figure 6. Nous supposons que la facturation et la commande sont liées : la facturation sera déclenchée sur l'arrivée d'un bon de commande (événement bon de commande). Cette liaison n'implique pas une continuité directe dans le temps : elle se fait lors d'une demande de facturation par une événement `facturation`. La validation entraîne l'omission des articles erronés. Une commande ne peut être modifée une fois terminée. La facturation est à la demande et produit une facture en trois exemplaires.

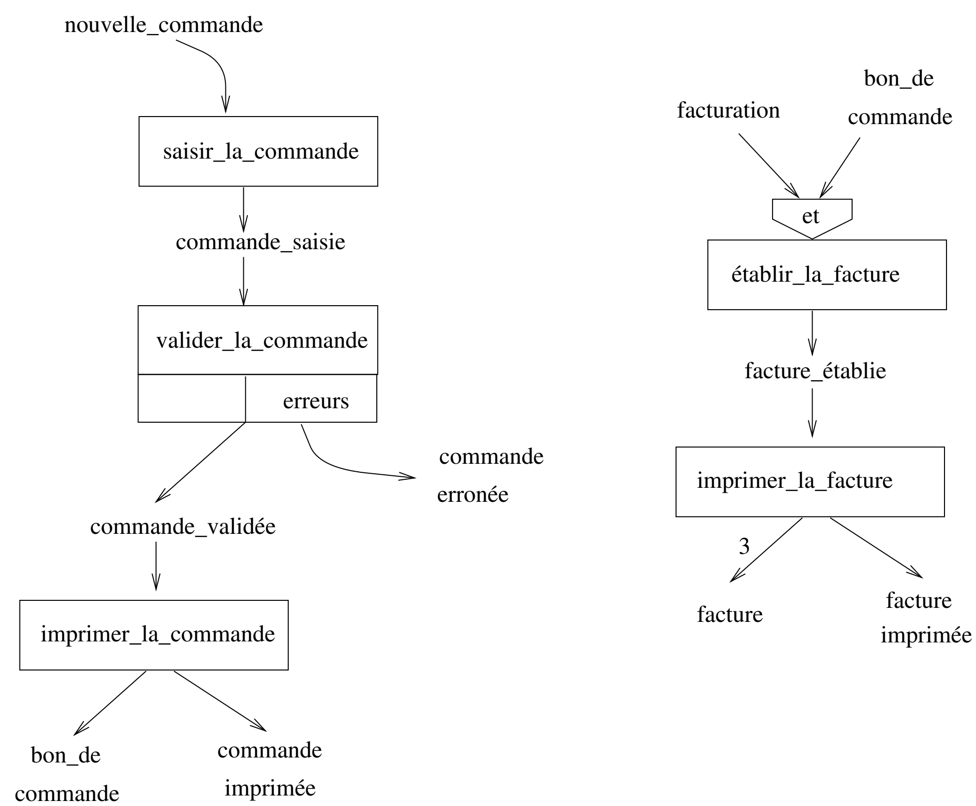


Figure 6 : Modèle Conceptuel des Traitements de la facture - cas 1 (Merise)

Le fait qu'il existe une seule facture par commande permet de lier ces deux processus par l'événement `bon_de_commande`. Notons que l'événement `commande_validée` est toujours émis, même s'il y a des erreurs. L'opération d'impression produit deux événements `bon_de_commande` et `commande_imprimée`.

3.3.2 Version souple

Nous reprenons les hypothèses du MCD de la figure 6. Nous supposons que la facturation et la commande sont disjointes. Le traitement se fait maintenant au niveau des lignes. Pour la commande, les lignes sont entrées au fur et à mesure et non pas en une fois comme dans le cas 1. Pour la facturation, plusieurs bons de commande sont examinés pour la facture. Rien n'indique que ces bons soient mis sur une seule facture. Ici aussi, la facturation peut durer dans le temps, le processus n'est pas linéaire.

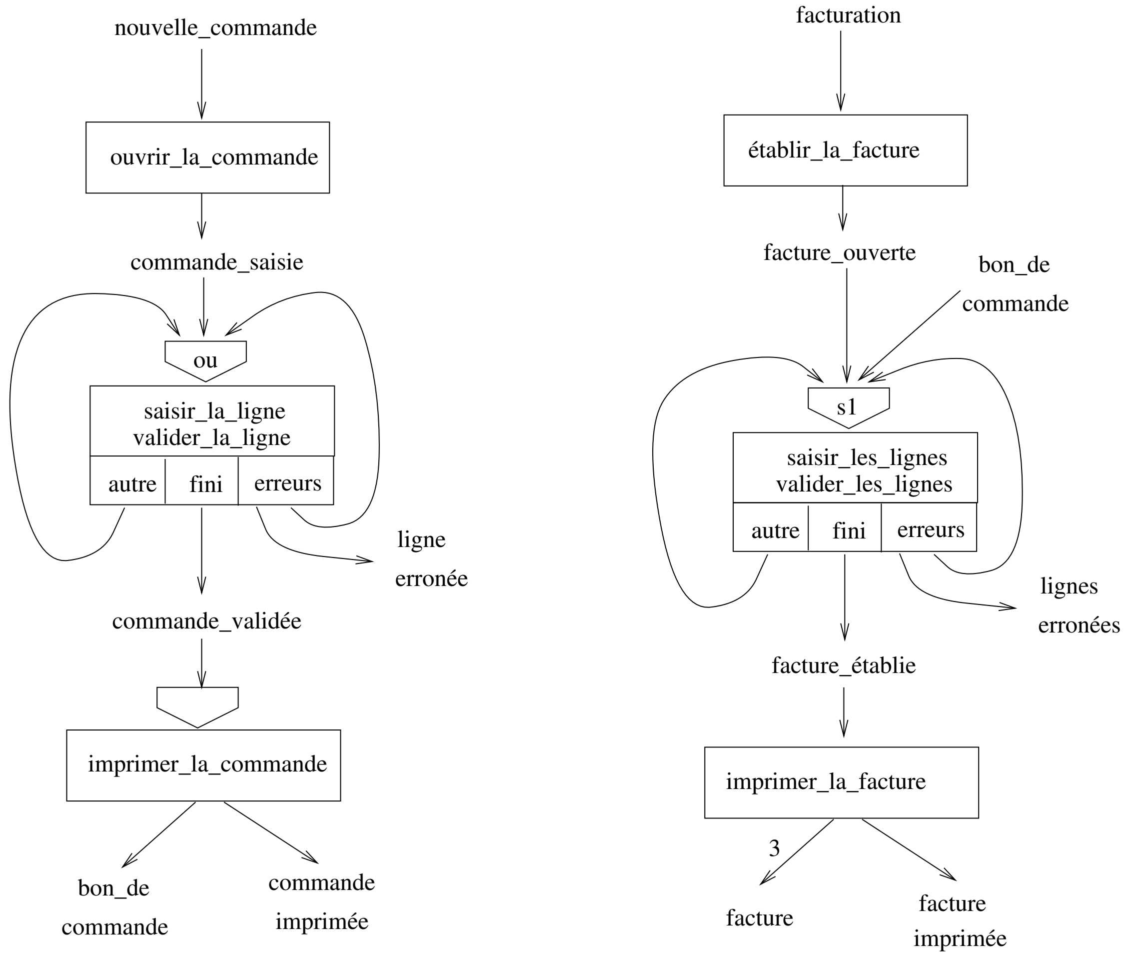


Figure 7 : Modèle Concetuel des Traitements de la facture - cas 2 (Merise)

Nous avons évité les contrôles de cohérence entre les lignes de facture et les lignes de commande. La description précise des opérations sort du contexte de travail, nous en avons donné les grandes lignes pour une compréhension globale.

3.4 Remarques

Le modèle conceptuel des données implique une réflexion sur les informations conservées et leur structuration. Il suggère un certain nombre de questions que voici. Telle information est-elle d'intérêt ou pas. Si oui, comment la représente-on ? Comment s'intègre-t-elle avec les autres informations ? Séparer ou regrouper des entités permet de clarifier le vocabulaire de l'énoncé : traiter les homonymes, les synonymes, les polysèmes.

Le modèle conceptuel des traitements induit une réflexion sur l'ordre et le contenu des opérations. Ce traitement est-il à réaliser ou non ? Si oui, quand et sous quelle forme l'est-il ? Toutefois la description doit être rigoureuse et précise pour être exploitable. Notre modélisation est trop floue et ambiguë pour comprendre les traitements effectués. C'est souvent le cas dans les modèles produits par les analystes.

Les hypothèses prises pour un modèle ont généralement de l'influence sur l'autre modèle. Lier deux entités implique un traitement commun et inversement séparer deux entités sous-entend des traitements différents (e.g. multi-facturation). La réflexion est globale au système et non modulaire.

4 OMT

4.1 Présentation de la méthode Object Modeling Technique

4.1.1 La méthode

Object Modeling Technique [RBP+91] (OMT en abrégé) est une méthode d'analyse et conception à objets. Elle couvre le cycle de l'analyse à la conception détaillée des objets. En développement à objets sous OMT se déroule en quatre phases : analyse, conception système, conception des objets et implantation. L'analyse consiste à donner un modèle de l'application et du domaine dans lequel elle opère, appelé domaine du problème. Elle met en évidence les propriétés importantes du système indépendamment des objets informatiques. La conception du système décline l'analyse dans une architecture d'ensemble du système prenant en compte les aspects bases de données, concurrence, interfaces homme/machine, distribution du contrôle, etc. Elle introduit des objets de l'informatique. La conception des objets affine la description des objets et des classes de l'application et met en oeuvre des décisions de conception. L'implantation concerne la programmation de l'application dans un environnement cible, qu'il soit objet ou non.

OMT s'applique théoriquement à toutes les applications informatiques. La méthode présente l'intérêt d'être largement utilisée dans l'industrie. Elle est représentative des méthodes à objets actuelles mais n'est pas exempte de toutes critiques. Si les modèles sont riches sémantiquement, leur interaction, c'est-à-dire la cohérence entre les plans de description du système est approximative et à la charge de l'analyste. Le processus est aussi une grande faiblesse de la méthode. L'analyste ne sait pas comment passer des modèles d'analyse à ceux de conception.

Que ce soit en analyse ou en conception, OMT emploie trois modèles différents pour décrire un système : le modèle des objets, le modèle dynamique et le modèle fonctionnel. Il s'agit là des modèles principaux, d'autres modèles annexes complètent les descriptions : diagrammes d'interaction, diagrammes d'architecture, etc. Le modèle des objets donne la structure du système en termes d'objets et de liens entre objets. Le modèle dynamique décrit l'évolution du système dans le temps et les interactions entre les objets. Le modèle fonctionnel établit les calculs dans le système et les transformations des données. Nous nous intéressons dans la suite du discours à ces modèles, communément appelés *plans de description*.

Les formalismes utilisés sont issus de techniques éprouvées par des méthodes plus anciennes. L'"astuce" a consisté à mettre dans un même concept les différents points de vue d'un système, avec quelques enrichissements. Ainsi le modèle statique est une révision des formalismes de modélisation des données (Entité-Association), incluant l'héritage, la relation d'agrégation, les opérations et parfois les métaclasses. La notion de relation en OMT est plus riche que celle d'association dans le modèle entité/association. Le modèle dynamique, qui définit le cycle de vie des objets, reprend les modèles des méthodes temps-réel (SA/RT, RdP, Statecharts). Le modèle fonctionnel correspond à l'approche des flots de données des méthodes SADT ou SA/RT.

4.1.2 Restriction au cas d'étude

Après une description des trois formalismes dans la section suivante, nous produisons deux modèles : le modèle des objets et le modèle dynamique. Comme beaucoup d'auteurs, nous considérons que le modèle fonctionnel est redondant avec les autres modèles. Ces redondances entraînent des problèmes de cohérences pour la modélisation. Nous ne décrivons donc pas de modèle fonctionnel. La méthode préconise de commencer par le modèle des objets et de décrire ensuite le comportement dynamique de ces objets. Nous suivons cette démarche. Le modèle des objets, ou plus précisément ses différentes déclinaisons, est présenté dans la section 4.2. Dans l'intérêt de l'exposé, seuls les modèles dynamiques les plus représentatifs sont décrits dans la section 4.3. Ils suffisent à extraire les questions induites par le formalisme.

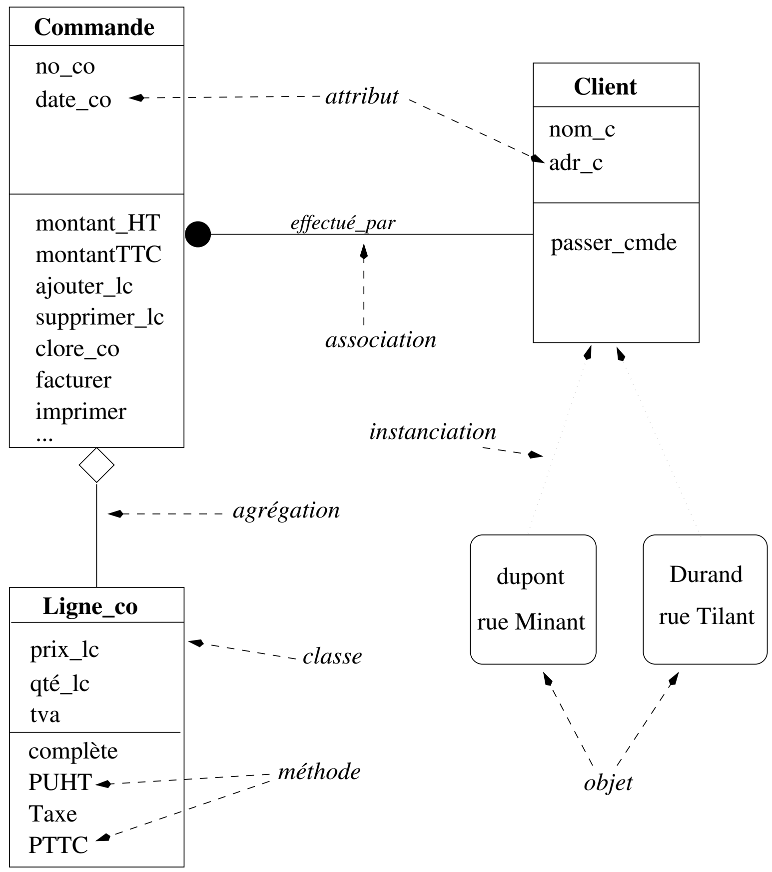
4.1.3 Formalismes utilisés

Modèle des objets Le modèle des objets décrit les objets du système et les relations statiques entre ces objets. Plus exactement, ce sont les modèles de ces objets, les classes, qui apparaissent dans les diagrammes. Il s'inspire des concepts et notations issues des modèles de données de type Entité/Association et de la programmation à objets. Les objets correspondent alors aux entités et les classes correspondent aux types d'entité.

Un *objet* est un concept, une abstraction ou une chose ayant des limites très claires et un sens précis dans le contexte du problème étudié [RBP+91]. Cette définition est large et plusieurs représentations d'un même problème existent. Chaque objet a une identité et peut être distingué des autres. En OMT, l'identité est implicite. Dans d'autres modèles, comme celui des TAG de la section 6, l'identité est explicite. Un objet est caractérisé par des propriétés ou attributs appelés aussi variables d'instance et par un comportement. Le comportement est l'ensemble des traitements applicables à l'objet. Ces traitement sont appelés *opérations* ou *méthodes*. La notion de clé ou identifiant d'entité est parfois redondante avec l'identité implicite des objets : le numéro de la commande est-il le numéro d'objet associé dans le système ou un numéro propre à l'application. Si l'analyste souhaite une codification alors il rajoute un attribut *no_commande*.

Une *classe* décrit les objets ayant même structure (les types des attributs) et même comportement (les méthodes). Par exemple, toutes les commandes sont similaires, à ceci près que le contenu varie. Les méthodes d'un objet sont regroupées dans sa classe. Les attributs sont décrits dans la classe et valorisés dans les objets. La différence entre types d'entité et classes est donc que ces dernières expriment non seulement une notion de donnée mais aussi de traitement (méthodes associées). Les objets de la classe sont appelés instances et la relation d'*instanciation* relie un objet à sa classe. Chaque objet est instance d'une seule classe.

Les attributs ne sont pas des objets. Si un objet est associé à d'autres objets alors ce lien figure dans une relation d'*association*. Par exemple, une commande est passée par un client. Si un objet est composé d'autres objets alors ce lien figure dans une relation d'*agrégation*. Par exemple, une commande est composée de lignes de commandes. Ces deux relations sont une abstraction entre deux classes de liens entre deux objets instances de ces classes. Les différences entre ces deux relations sont les suivantes. L'agrégation est une relation orientée tandis que l'association ne l'est pas. Dans l'association, les deux objets sont indépendants tandis que dans l'agrégation l'objet agrégé (ou composant) n'existe qu'au travers de l'objet agrégat. Cette distinction n'existe pas en programmation à objets. Voici la notation utilisée pour le modèle des objets.



———	association entre deux classes
$\triangle$	relation de spécialisation/généralisation
$\diamond$	relation d'agrégation (attachée à une classe)
$\Diamond$	relation ternaire (entre trois classes)
$\bullet$	cardinalité 0-n
$\circ$	cardinalité 0-1
1+	cardinalité 1-n
.... >	relation d'instanciation

Figure 8 : La notation du modèle à objets (OMT)

Une troisième relation existe entre classes : la relation de *généralisation/spécialisation*. C'est une relation orientée entre une classe, appelée super-classe de la relation, et une ou plusieurs versions affinées de celle-ci, appelées sous-classes. Il s'agit bien d'une relation entre classes uniquement et non d'une abstraction de liens entre instances de la classe. Cette relation va de pair avec la notion d'*héritage*. L'héritage est un mécanisme qui permet de définir une classe à partir de la description d'une ou plusieurs classes. La sous-classe hérite de la description de la super-classe (héritage simple) ou des super-classes (héritage multiple). La relation sert à la classification des objets. L'héritage est un mécanisme de réutilisation de code. Prenons un exemple. Considérons deux sortes de commandes : les commandes normales et les commandes facturées. Elles n'ont pas le même comportement : les commandes normales sont modifiables (ajouter des lignes, ...) et les commandes facturées sont closes (montantHT, frais, TVA, ...). La classe Facture factorise le comportement commun, et chaque classe spécialise et enrichit ce comportement. Le schéma de figure 9 modélise cet exemple. La notation $\triangle$ signifie que l'héritage est disjoint : une commande est exclusivement en cours ou facturée. L'héritage non disjoint est noté soit par deux relations d'héritage soit par un triangle plein. La relation de spécialisation est orthogonale aux relations d'agrégation et d'association. En effet, les commandes facturées sont passées par des clients et contiennent des lignes de commandes. Parfois les sous-classes correspondent à des étapes dans le cycle de vie des objets. Ceci fait référence au modèle dynamique que nous examinons dans le paragraphe suivant. Une notation particulière correspond à cette notion d'état comme classe.

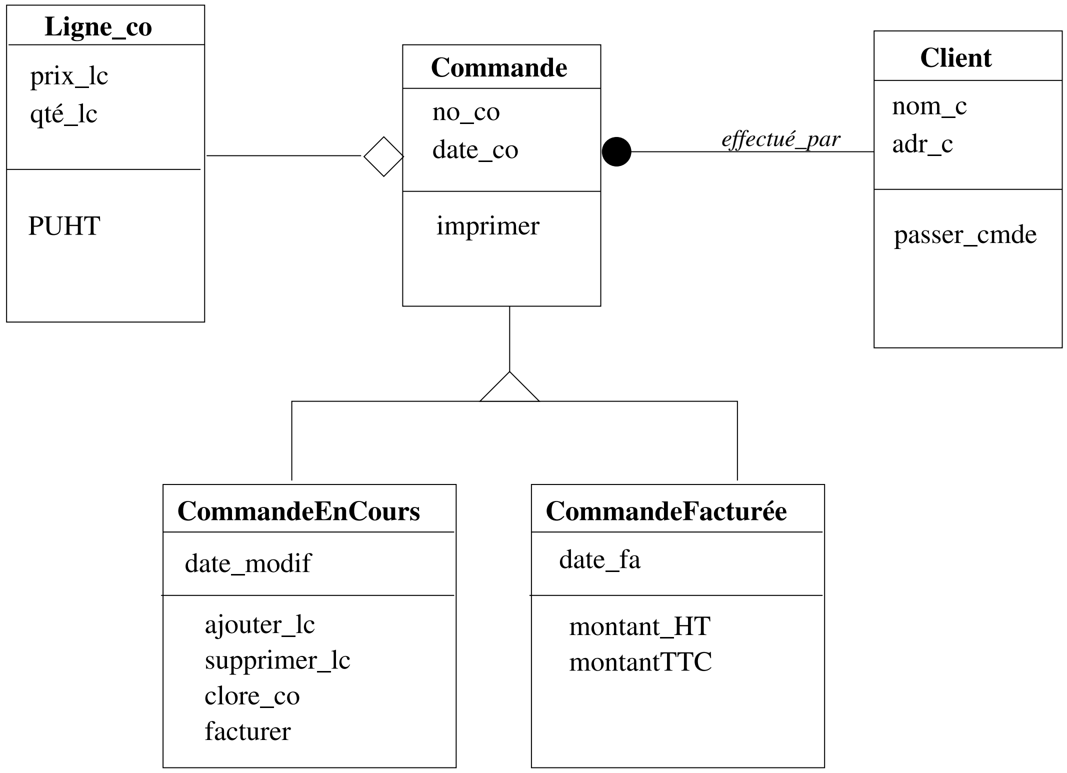


Figure 9 : Modèle des objets de la facture - cas 1 (OMT)

Le formalisme comprend beaucoup d'autres concepts que nous ne détaillons pas ici. Citons entre autres les associations n-aires, les attributs de liens, les associations ordonnées, les associations comme classe, les rôles dans les associations, la qualification des associations, l'agrégation récursive, la propagation d'opérations, la redéfinition de méthodes, les classes abstraites, les méta-objets, etc. Nous ne définissons pas ces concepts. Nous n'étudions pas non plus les dépendances entre concepts.

Modèle dynamique Ce modèle, appelé aussi diagramme d'états, représente sous forme d'un automate structuré de type StateCharts [Har87] le comportement des objets, c'est-à-dire les événements reçus par l'objet et les traitements qu'il réalise pour répondre à ces événements. Deux sortes de traitements sont distingués en OMT : les actions et les activités. Les actions sont des traitements atomiques ininterruptibles. Les activités sont des traitements interruptibles par réception d'événements. Les deux sont décrits par des méthodes ou plus précisément des noms de méthodes car en général, le corps des méthodes n'est pas spécifié. Par contre les messages émis par ces méthodes vers d'autres objets sont signalés.

Un état correspond à un intervalle de temps entre deux réceptions d'événements. A un état peut être associée une activité et des actions en entrée ou sortie de l'état. Une fin d'activité est un événement sans nom. Une transition est un changement d'état déclenché par une réception d'événement. A une transition peut être associée une action (avec éventuellement des émissions d'événements vers d'autres objets) ou une garde. La garde est un prédicat qui conditionne le déclenchement d'une transition. Le nom de l'événement et celui de l'action associée sont souvent distingués.

Un diagramme d'état peut être fini ou infini. Un état initial (•) exprime la création d'un objet et un état final (⊙) sa destruction. Dans un diagramme infini, l'état initial est facultatif et il n'y a pas d'état final.

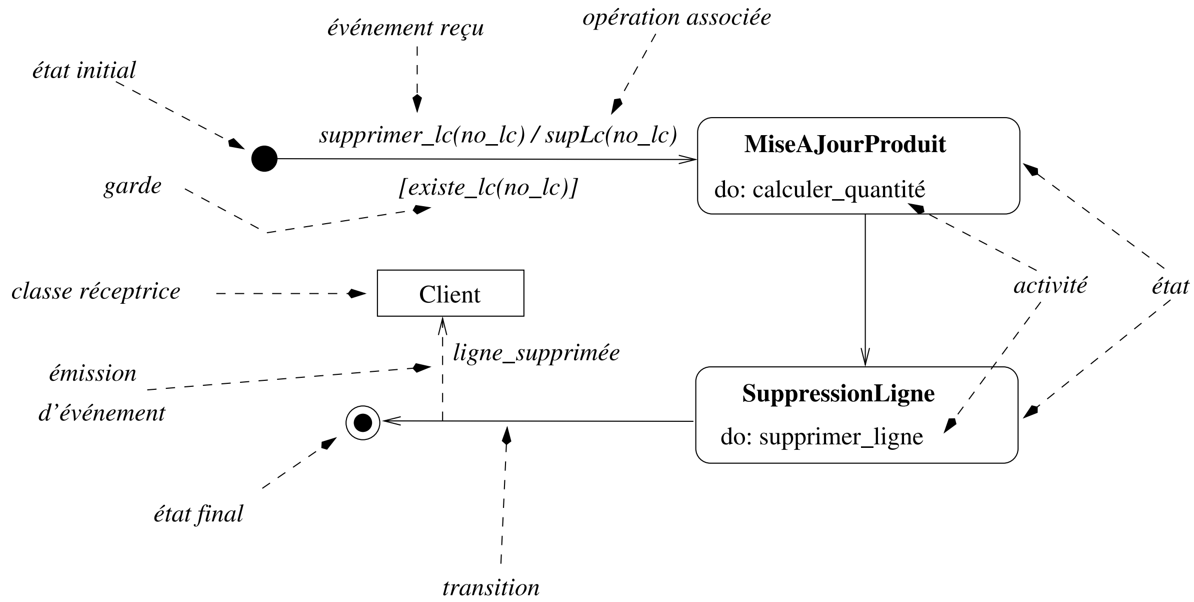


Figure 10 : La notation du modèle dynamique (OMT)

Le formalisme autorise la généralisation et l'agrégation d'états. La généralisation est l'expansion d'activités emboitées, elle permet de définir des hiérarchies d'états (inclusion de super-états) et d'événements (hiérarchies d'abstraction). L'agrégation est la concurrence entre états. Ceci permet une représentation abstraite et descendante mais aussi une structuration ascendante en remontant les opérations et les événements déclenchés aux niveaux inférieurs.
Attention : Différentes notations existent : l'ancienne, la nouvelle et les exotiques. Nous essayons autant que possible de respecter la nouvelle, mais des petites variantes sont possibles. Nous nous en excusons. Ceci étant, la sémantique de ces notations n'apparaît clairement que sur les exemples.

Modèle fonctionnel Le modèle fonctionnel exprime les transformations de données dans les traitements par un diagramme de flots de données [Vai95]. Le traitement transforme des valeurs des données. Les traitements de plus bas niveau sont des fonctions pures sans effet de bord. Les effets de bord sont modélisés au travers des réservoirs de données (stockage). Les traitements sont implantés comme des méthodes ou des fragments de méthodes d'opérations sur les classes. Il se pose à ce sujet un réel problème d'identification de l'objet receveur de la méthode (i.e. l'objet dont la classe détient la méthode) : est-ce un réservoir, un objet actif ou autre ? Les acteurs sont des objets actifs qui dirigent les flots de données en produisant ou consommant des valeurs. Ils délimitent le flot de données comme sources et puits du flot. Un flot de données relie la sortie d'un objet ou d'un traitement à l'entrée d'un autre objet ou d'un autre traitement. Il représente les valeurs de données intermédiaires dans le calcul. La figure 11 présente la notation générale du modèle. Il est possible d'imbriquer les diagrammes de flots de données, d'ajouter des flots de contrôle et de spécifier les opérations.

Le modèle fonctionnel décrit les fonctions globales du système sans spécifier qui les réalise ni à quel moment elles le sont. Il n'est pas très développé dans la méthode, mais correspond pourtant à un besoin majeur d'abstraction du comportement du système. En général, il est redondant avec le modèle dynamique, c'est pourquoi il n'apparaît guère dans les dossiers de spécification. Dans notre exemple, le modèle fonctionnel serait sans doute plus adapté que le modèle dynamique, mais nous préférons utiliser le modèle dynamique pour son originalité.

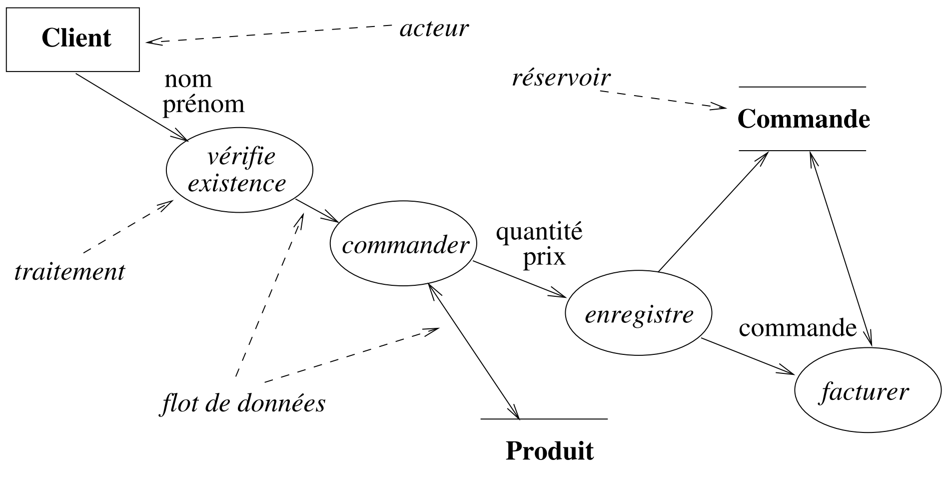


Figure 11 : La notation du modèle fonctionnel (OMT)

4.2 Modèle des objets

La modélisation débute par le recensement des objets. Cette phase est similaire à celle de la recherche des entités en Merise. L'analyste examine ensuite ce qui a trait à chaque objet, indépendamment du problème en lui même, on parle de modélisation des objets du domaine du problème et non du problème. Ainsi, nous nous interrogeons sur les clients, les commandes et les produits. Les questions induites sont relatives à la structure et au comportement des objets.

La description de la structure est proche de ce que nous avons vu en Merise à ceci près qu'une information d'un objet peut être un attribut, une méthode, un objet ou un lien (association ou une agrégation). Un attribut est une valeur simple (pas un objet) et non redondante (pas une méthode). Une méthode lit ou modifie les attributs. Un objet doit avoir un comportement et pas seulement une valeur. Un lien est un objet connu de l'objet en cours de description. Si cet objet connu fait partie intégrante de l'objet en cours de description alors la relation entre leur classe respective est une agrégation sinon c'est une association. Par exemple, une commande est composée de lignes de commandes : c'est une agrégation. Une ligne de commande fait référence à un produit mais ne contient pas ce produit : c'est une association. La différence est parfois mal perçue par les analystes. Elle a une influence sur le comportement des objets : la suppression de l'objet agrégat entraîne la suppression des objets agrégés. Par exemple, les lignes de commandes n'existent que vis-à-vis de leur commande.

Contrairement au modèle Entité/Association, la notion de méthode d'un objet nous oblige à réfléchir précisément sur le comportement des objets. Cet aspect est crucial et malheureusement souvant délaissé par les analystes. Il faut donc se poser la question suivante : qu'est-ce qu'une facture (resp. un produit, un client, une commande) est capable de faire ? Certains attributs sont calculables : ils apparaissent plus logiquement en tant que méthode et qu'en tant qu'attribut (exemple du montant de la facture). Nous évitons ici les méthodes triviales (accesseurs) qui consistent à lire ou modifier les valeurs des attributs.

La description de la structure met en évidence les relations entre classes. De nombreuses variantes existent en OMT pour représenter les associations. Donc une fois mises en évidences, les associations doivent être spécialisées et enrichies (rôles, cardinalités, attributs de liens, ordre, contraintes, etc.). L'héritage apparaît plus tard dans la modélisation pour mettre en valeur les comportements communs. Les données à un méta-niveau sont représentées par des méta-classes et un lien d'instanciation ou des attributs ou méthodes de classe (préfixées par \$). Par exemple, le nombre de commandes (ou le numéro de la dernière commande) est un attribut de classe. Supposons plusieurs exemplaires d'une même commande destinés à des usages différents, la description des commandes est représentée par une méta-classe `MetaCommande`, chaque commande est représentée par une classe `Commande`, et chaque exemplaire par un objet.

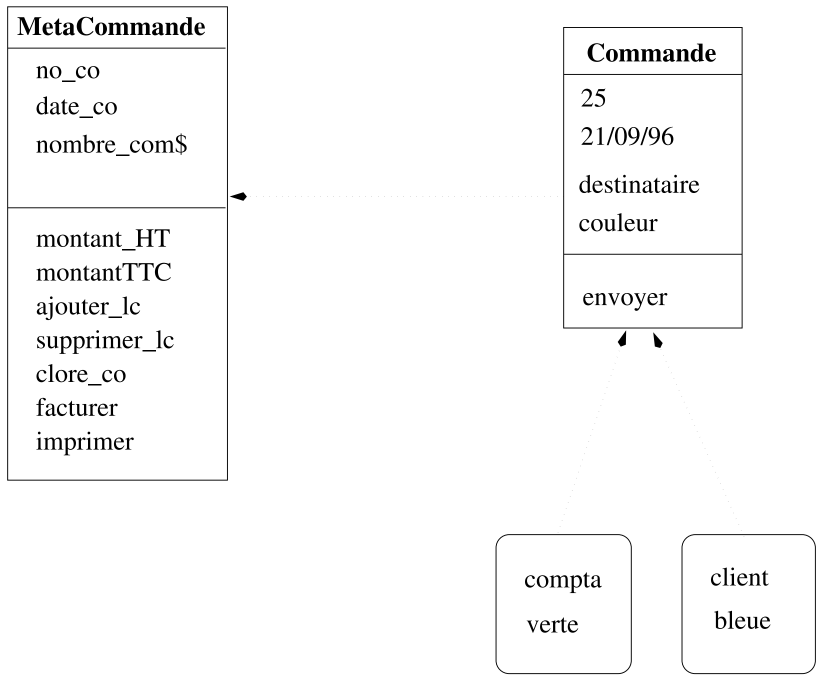


Figure 12 : Un exemple de méta-donnée (OMT)

Il y a parfois confusion entre agrégation, héritage, instanciation et association. Une telle discussion sort du cadre de ce travail, mais l'analyste doit veiller à la bonne utilisation qu'il fait des relations. La solution la plus simple et la moins erronée consiste à n'utiliser que des associations.

Nous déclinons maintenant deux versions de la modélisation du cas de la facturation. Dans la première, la commande et la facture sont un même objet qui évolue au cours du temps. Dans la seconde la commande et la facture sont clairement distinctes.

4.2.1 Version 1 : confusion de la commande et de la facture

Implicitement confondre commande et facture signifie qu'à une commande correspond une seule facture. Par contre, il reste la possibilité de facturer entièrement ou partiellement les lignes de commandes. Nous supposons une facturation entière. Elle conduit au schéma simple de la figure 13.

Notre réflexion sur les objets nous amène à sortir du contexte de la facturation. Par exemple, pour un client nous pensons à son chiffre annuel, ou les produits les plus commandés. Ce sont de nouvelles hypothèses. De même nous avons ajouté des frais à la commande, une T.V.A. par ligne de commande et non globale à la facture. La description est plus riche que celle du modèle entité/association du fait que des traitements locaux sont ajoutés. Par exemple, le chiffre d'affaire, le produit le plus acheté, les commandes impayées sont des opérations du client. Le stock est géré sommairement pour le produit par une quantité en stock. L'ajout de lignes de commande est une opération de la commande.

Les cardinalités du modèle des objets de figure 13 se lisent comme suit : "un client effectue un nombre quelconque de commandes", "une commande est effectuée par un seul client", "une commande contient plusieurs lignes de commandes", "une ligne de commande est contenue dans une seule commande", "une ligne de commande concerne un seul produit", "un produit est référencé dans un nombre quelconque de lignes de commandes".

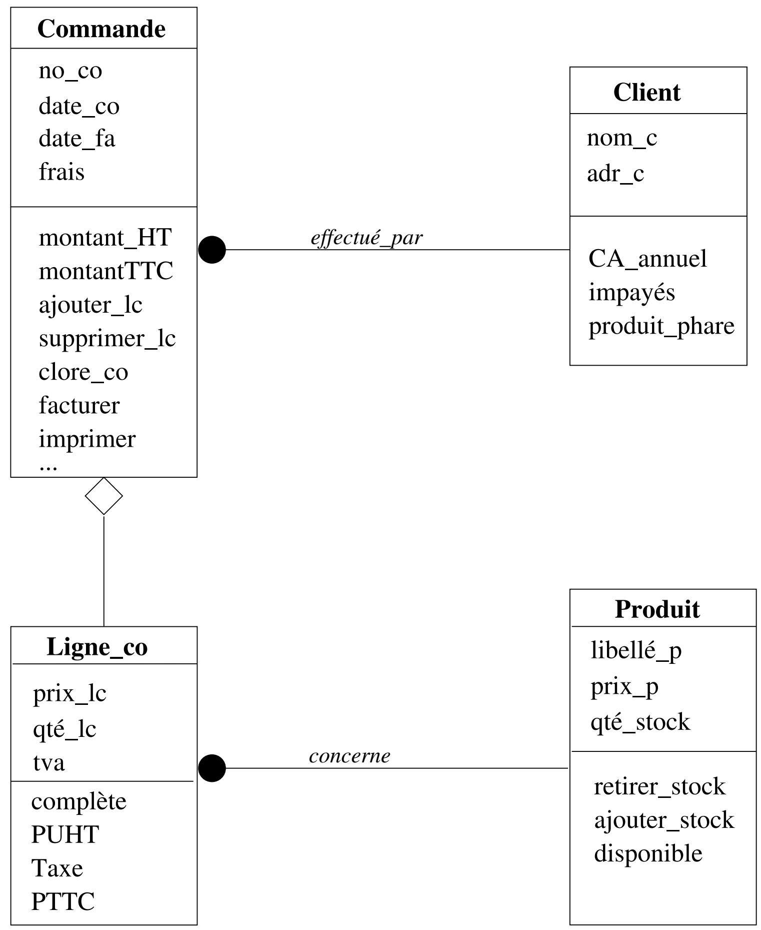


Figure 13 : Modèle des objets de la facture - version 1 (OMT)

4.2.2 Version 2 : séparation de la commande et de la facture

Les commandes et les factures sont séparées et leurs lignes aussi. Une distinction plus nette peut alors être faite entre les attributs et méthodes relatifs à la commande et ceux de la facture : par exemple les frais sont liés à la facture. La méthode `commandes` fournit les commandes d'une facture. La méthode `factures` fournit les factures d'une commande.

Notez que par rapport aux modèles de données tels que les schémas Entité/Association, la notation est plus chargée à cause des méthodes. Dans la pratique, beaucoup d'auteurs donnent des descriptions largement incomplètes, le discours fournissant des informations précieuses mais informelles. A notre avis, il vaut mieux séparer que sous-entendre. Nous avons donc décidé de séparer la description des classes de celle de leurs liens. La figure 14 montre les classes. Les figure 15 et figure 16 sont deux variantes de modélisation des relations suivant les similarités souhaitées entre commande et facture.

Nous avons ajouté de nouvelles informations en scindant la commande en deux : ouverture et clôture de la commande et de la facture, donner les factures d'une commande et vice-versa. Notez que des redondances apparaissent : avoir les factures d'une commande s'obtient par la méthode `factures` mais aussi par l'association `est_facturée`. Les auteurs de la méthode OMT sont silencieux au sujet de tels problèmes. Notez aussi que nous retrouvons les mêmes problèmes que dans Merise : une commande est-elle facturée une ou plusieurs fois. Un formalisme similaire apporte des questions similaires.

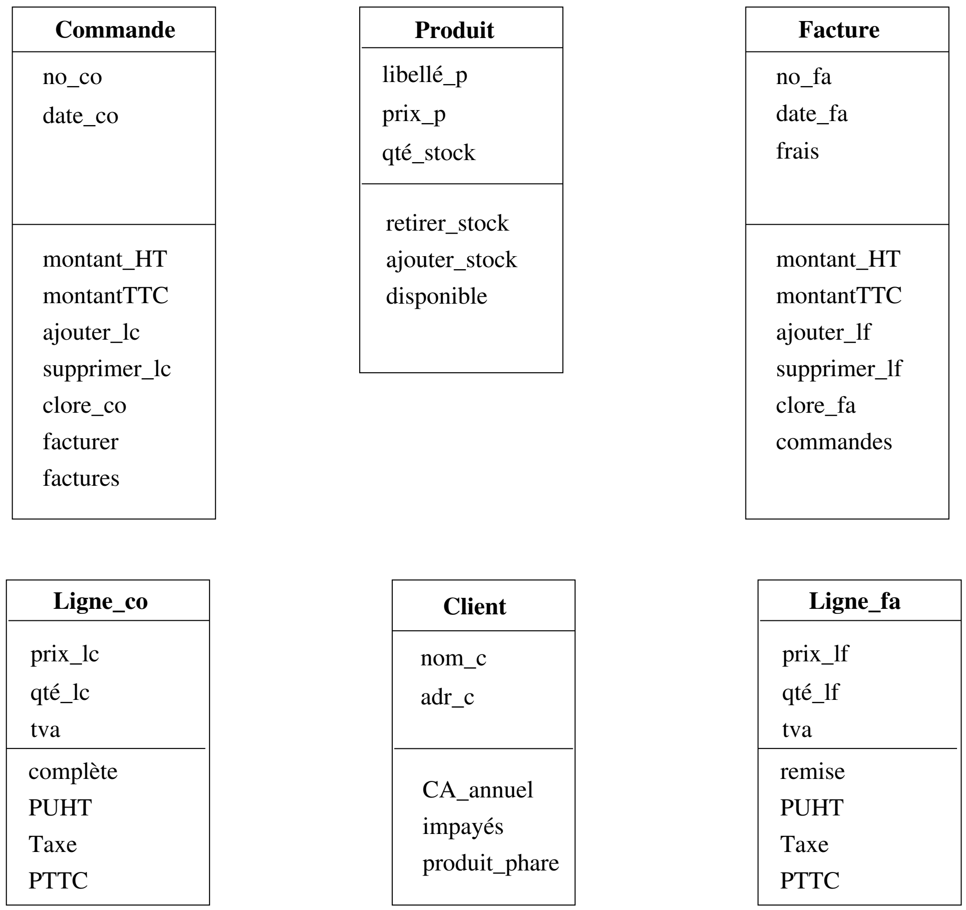


Figure 14 : Modèle des objets de la facture - version 2 : les classes (OMT)

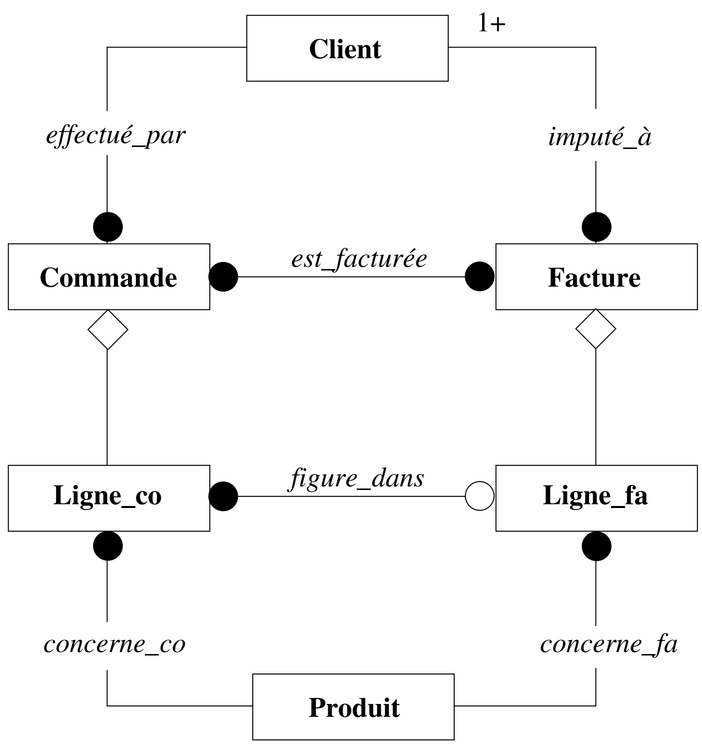


Figure 15 : Modèle des objets de la facture - version 2.1 : les relations (OMT)

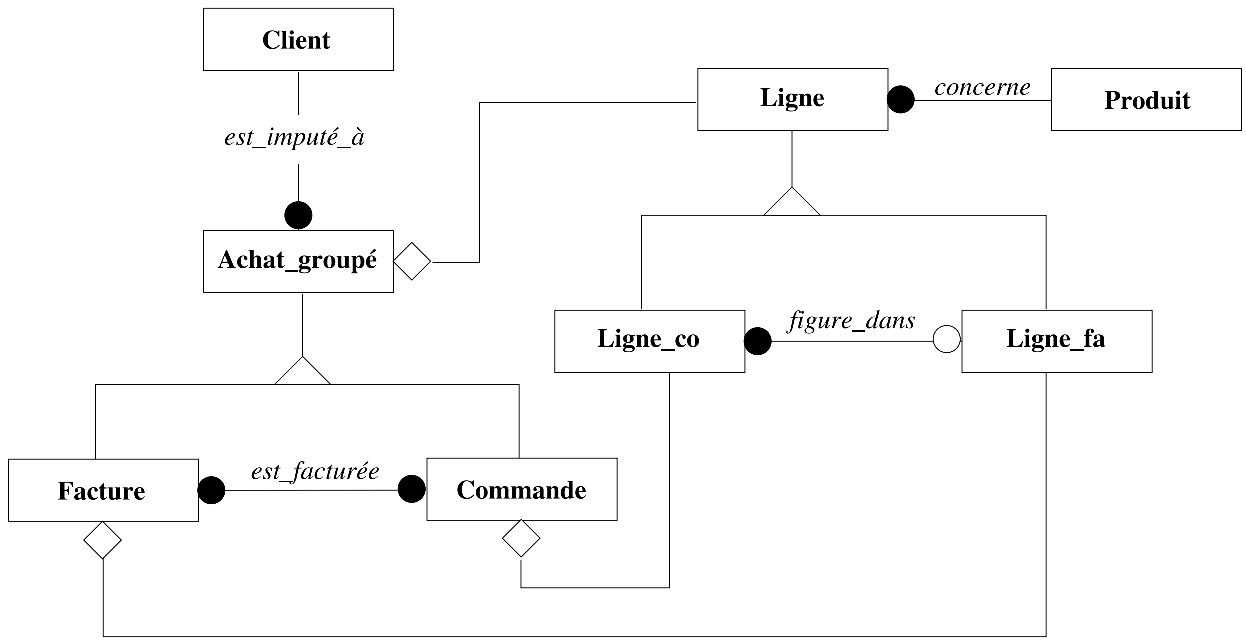


Figure 16 : Modèle des objets de la facture - version 2.2 : les relations (OMT)

Le second schéma paraît plus compliqué, mais il est aussi plus réutilisable car les comportements communs ont été généralisés. Les classes `Commande` et `Facture` ont une description très restreinte. Un schéma plus simple fait diparaître le lien entre les lignes de `Commande` et celles de `Facture`.

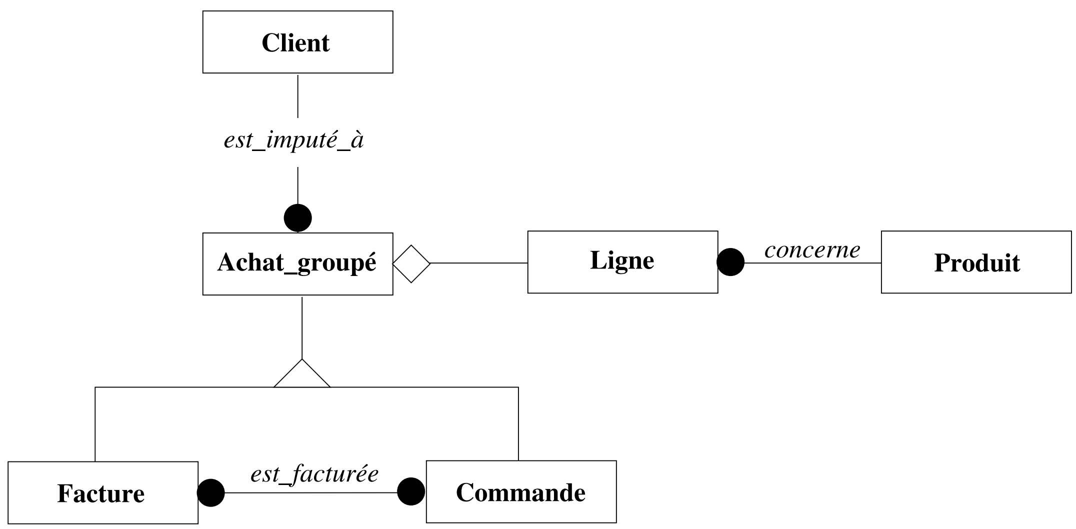


Figure 17 : Modèle des objets de la facture - version 2.3 : les relations (OMT)

Note sur la méthode : Les descriptions sont insuffisantes, surtout en termes de méthodes. Mais c'est normal : le processus est itératif : on fait un peu de structurel, un peu de dynamique et ainsi de suite jusqu'à avoir explorer entièrement le domaine de l'application.

4.3 Modèle dynamique

Le modèle dynamique exprime l'évolution du système dans le temps au travers de différents formalismes dans OMT : diagramme d'états, scénarii, etc. Cette évolution est perçue en termes d'événements. Les événements représentent des stimuli circulant dans le système. La découverte de ces événements se fait au travers de scenarii, qui indiquent les échanges entre objets. Par exemple, modélisons un exemple simple de commande-facturation. La commande est ouverte, remplie de lignes de commande, close et facturée. La figure 18 représente le diagramme de suivi d'événements de ce scenario.

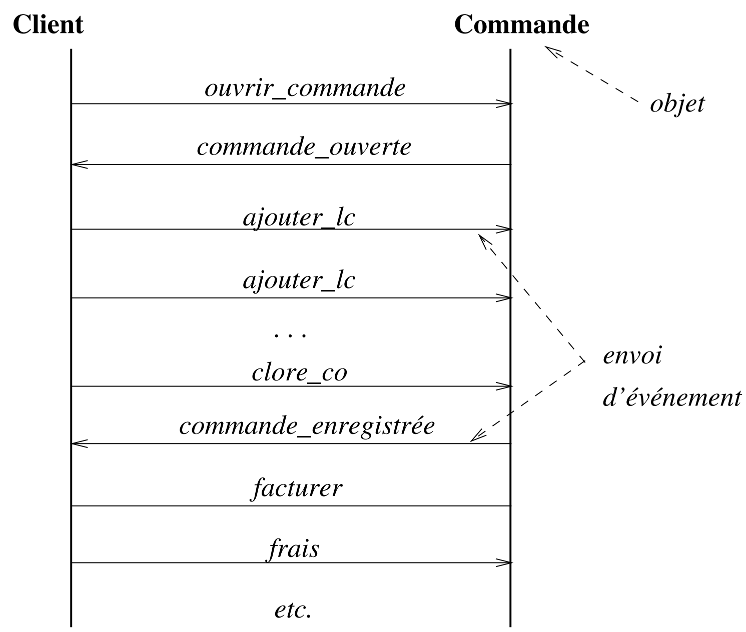


Figure 18 : Scenario simple de commande (OMT)

Le temps s'incrémente du haut vers le bas de la figure. Les flèches sont orientées de l'objet émetteur vers l'objet récepteur. L'émission des événements est uniquement asynchrone. Un appel synchrone se traduira alors par deux événements (asynchrones). Par exemple un appel d'une méthode m_A d'un objet O_A par un objet O_B avec retour d'un résultat se traduit par : l'appel de m_A de O_B vers O_A et le retour du résultat de O_A vers O_B. Un scenario représente une utilisation particulière du système. Il y a autant de scenarii que de *cas d'utilisation* du système[1]. Notez que les scenarii permettent de découvrir d'autres méthodes que celles perçues dans la construction du modèle des objets.

L'objectif est ensuite de construire ensuite un modèle dynamique pour chaque objet actif (i.e. objet ayant un comportement propre). Le comportement dynamique est donc un ensemble d'automates. Il y a plusieurs manières de construire le modèle dynamique : soit les objets sont étudiés individuellement, soit on part d'un contrôleur global et on examine les réactions des objets aux ordres du contrôleur. D'une manière générale, l'analyste commence par établir une liste (incomplète) d'événements qu'il va structurer par la suite. Ce type de modélisation n'est guère adapté dans notre cas car il n'y a pas vraiment d'objets actifs i.e. des objets dont le contrôle est propre. Toutefois, nous pouvons modéliser différentes étapes dans la vie de la commande par exemple. Nous nous plaçons dans le cadre de la version 1, car dans la version 2 les objets sont atomiques et donc très réduits d'un point de vue dynamique. Nous simplifions la notation des transitions en unifiant les noms d'événements et les noms des opérations associées : une transition labellée `evt(args)` est équivalente à une transition labellée `evt(args) / op(args)`. Nous traitons uniquement le cas de la commande et du client. Traiter le produit revient à faire de la gestion de stock, ce qui sort de notre cadre de travail.

[1] Consulter [JCJO91] pour une étude approfondie sur les cas d'utilisation.

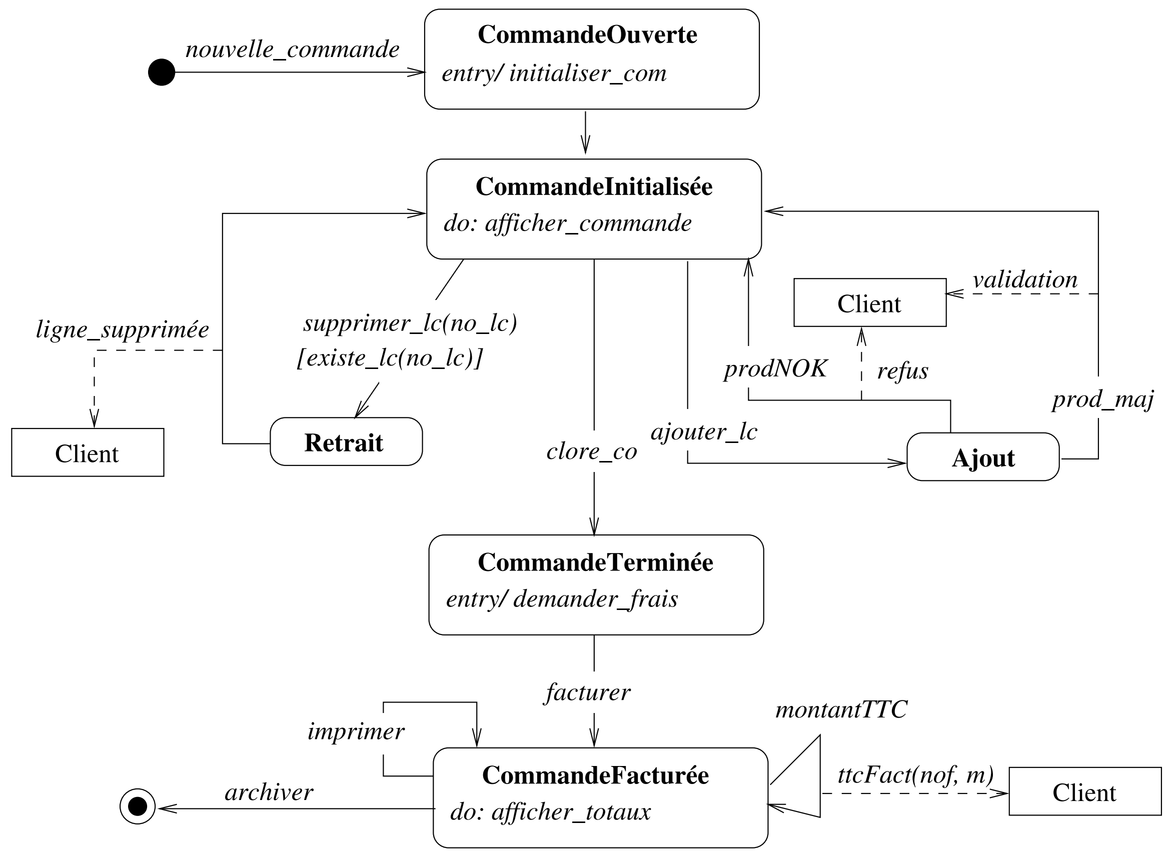


Figure 19 : Le modèle dynamique de l'objet `Commande` - version 1 (OMT)

Le préfixe `entry` dans un état signifie qu'une action précisée est exécutée à l'entré dans l'état. Deux états comportent un niveau de description supplémentaire : l'état `Ajout` et l'état `Retrait`. Ils consistent en des vérifications et mises-à-jour de produits. Voici leur description.

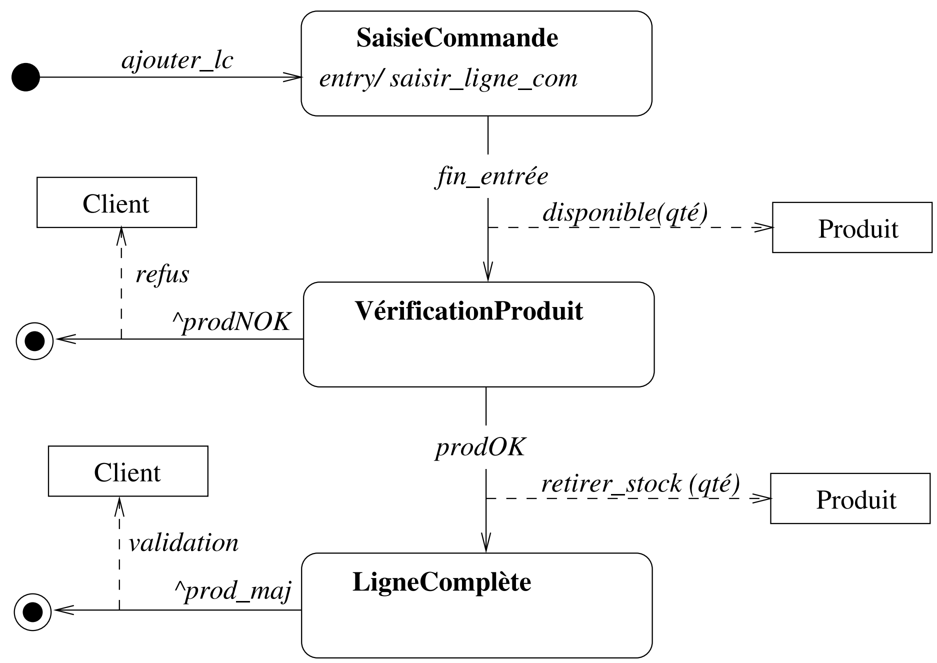


Figure 20 : Le modèle dynamique de l'objet `Commande` : état `Ajout` - version 1 (OMT)

Une transition dans un état final correspond à une réception d'événement dans l'état de niveau supérieur ou à une fin d'activité. Le préfixe '^' pour un événement signifie qu'il est visible au niveau supérieur de l'emboitement. Par exemple, l'événement `prod_maj` dans l'état `LigneComplète` de la figure 20 est l'événement `prod_maj` dans l'état`Ajout` de la figure 19. Lorsqu'une activité se termine, elle émet un événement "fin d'activité". Ainsi, une transition non labellée par un événement est une transition consécutive à une fin d'activité. C'est le cas entre l'état `MiseAJourProduit` et l'état `SuppressionLigne`.

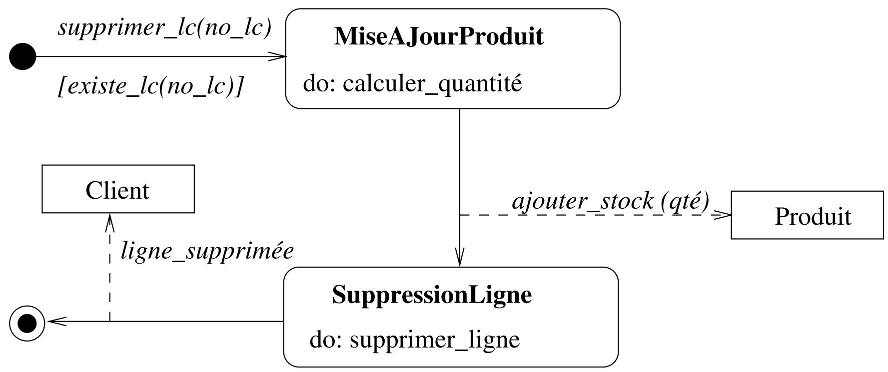


Figure 21 : Le modèle dynamique de l'objet `Commande` : état `Retrait` - version 1 (OMT)

Les échanges ont lieu avec des clients et des produits. Nous l'indiquons en précisant le type d'objet vers lequel est émis un événement. Par exemple, dans la figure 21, l'événement `ajouter_stock` est envoyé à un objet de type `Produit`. Il est évident qu'au niveau de la conception, une interface doit préciser ces échanges.

4.4 Remarques

Un intérêt du modèle objet est la traçabilité entre les objets d'analyse, de conception et d'implantation. En maintenance, il est théoriquement possible de retrouver les liens et modifier les dossiers de spécifications. En OMT, le même modèle (notation) est utilisé ce qui implique un effort d'abstraction de la part de l'analyste pour ne pas surspécifier.

La modélisation des relations entre objets pose un problème : la notion de service implicite aux objets est mal cernée par les relations qu'elles soient simples, d'agrégation ou de spécialisation. Qui est le client, qui est le serveur ? Une des difficultés de la conception par objets vient de là.

Par rapport à Merise, l'analyste a une vision très partielle et orientée des traitements (une vision plus globale est donnée par les flots de données). L'étude précise des objets amène de nouveaux comportements ignorés jusque-là : archivage des commandes, impression, lecture de valeurs, vérifications produit-commande dans une interaction d'objets et non une recherche d'information dans la base. La création dynamique d'objets est un aspect "oublié" dans OMT. Pourtant nous souhaitons créer des commandes, des factures, des lignes de commandes. Sans doute est-ce parce qu'en général les modélisations OMT dans la littérature portent sur un nombre fixe d'objets actifs.

OMT se veut fédératrice en ce sens que n'importe quelle application peut être modélisée. De ce fait les trois formalismes représentent (individuellement) tous les aspects à modéliser. Cependant le lien entre ces aspects n'est traité que superficiellement par les auteurs de la méthode. Ainsi il est souvent aisé de décrire indépendamment trois modèles en pensant à des choses différentes mais l'objet ne fédère pas trivialement les trois réflexions. La modélisation par flots de données semble plus adaptée car il y a plus de calculs et de mises-à-jour de valeurs que d'interactions par événements.

A notre avis, une méthode à objet telle que Fusion [Der94] est plus adaptée à notre cas d'étude. Le principe est le suivant : extraction des objets et des opérations du système puis répartition des opérations dans les objets. La méthode OOM [RB93], version orientée objet de Merise, abonde aussi en ce sens : écriture de modèles conceptuels classiques puis répartition des traitements dans les objets.

Notons pour terminer que, comme dans les autres méthodes, il faut savoir délimiter le cadre de l'étude et ne pas décrire pour les objets retenus des aspects qui ne concernent pas directement l'application (ex: gestion de stock, gestion des paiements...).

5 Z

Z symbolise, avec VDM [Jon93], l'école de la spécification par modèle abstrait, qui repose sur la définition d'un modèle du système avec des entités mathématiques bien comprises telles que les ensembles ou les fonctions [Som92]. Les deux techniques reposent sur les même principes (spécification mathématique, preuve de propriétés abstraites, raffinement ou réification). Nous présentons plutôt Z, car les "schémas", notion propre à Z, sont plus lisibles que les spécifications textuelles de VDM.

5.1 Présentation du langage Z

5.1.1 La méthode

Z [Spi89, WL88] est une notation pour la spécification formelle. Ce langage a été développé à partir des travaux de Jean-Raymond Abrial au PRG de l'université d'Oxford. La sémantique de Z est basée sur la théorie des ensembles et la logique des prédicats du premier ordre. Dans l'approche de spécification par modèle abstrait, le système est décrit par un état et des opérations sur cet état. En Z, l'état est décrit en termes d'ensembles, de relations, de fonctions et de prédicats. Les opérations sont définies axiomatiquement par pré- et post-conditions sur l'état. Ces axiomes sont des prédicats.

La démarche induite par Z est celle du cycle linéaire dit contractuel. Une spécification formelle est écrite et validée (preuve de propriétés). Puis l'état et les opérations sont transformés (raffinés) progressivement jusqu'à contenir des structures de données et de contrôle du langage de programmation cible. Ces transformations sont prouvées méthodiquement.

Z n'est pas une méthode car elle ne possède pas de démarche et d'environnement de développement. La démarche de Z est trop succincte et trop fastidieuse pour être appliquée en pratique. Il manque un environnement de développement avec des outils de spécification, de validation, d'assistance à la preuve et de raffinement. En effet, la saisie des spécifications se fait en utilisant le traitement de texte scientifique LaTeX. La validation n'existe que pour le typage : l'outil FUZZ détecte les incohérences de types d'une spécification écrite en LaTeX. Les preuves sont faites "à la main". Le raffinement est à la charge du concepteur. L'outil B [Abr96] répond à ces besoins. Une spécification B est structurée en termes de machines, approche plus modulaire que Z.

5.1.2 Restriction au cas d'étude

Nous utilisons le langage Z uniquement pour la spécification du cas "facturation". Nous partons du texte initial et non d'une version issue d'une méthode semi-formelle telle que Merise. Un schéma de traduction du formalisme Entité/Association en Z est décrit dans [And96]. Nous nous focalisons sur la description de l'état du système et des opérations. Nous classons ces dernières en opération de modification et opérations de consultation. Ces dernières ne modifient pas l'état du système. Le raffinement et la preuve sont anecdotiques dans ce document.

5.1.3 Formalismes utilisés

Une spécification Z se présente comme une compilation de déclarations globales (types, variables, fonctions,...) et de schémas d'états et d'opérations [Som92]. La syntaxe utilisée ici est une version personnalisée du format LaTeX de Mike Spivey [Spi90].

Les types prédéfinis en Z sont les ensembles, les relations, les fonctions (partielles, totales, injectives, ...), les séquences, les multi-ensembles, les séquences injectives. Décrire ces types et leurs opérations associées ici est aussi inutile que fastidieux, nous renvoyons donc le lecteur à [Spi90] ou [WL88]. Ils découlent tous de la notion d'ensemble. Une relation $X \leftrightarrow Y$ est un ensemble de couples de la forme (x, y) où $x \in X$ est l'antécédent de la relation et $y \in Y$ est l'image de la relation. Une fonction est une relation telle que tout élément de X a au plus une image dans Y.

Les ensembles sont définis en extension ou en compréhension. En Z, un type est un ensemble d'éléments, à la restriction près que deux types ont une intersection vide (i.e. un élément possède un seul type). Dans une spécification Z apparaissent trois sortes de types :

- Les types de base (*given set*), qui sont des ensembles supposés exister et dont les éléments ne sont pas connus. Par exemple, un ensemble de commandes et un ensemble de dates:

  $$[COMMANDE, DATE]$$

  Les entiers ($\mathbb{Z}$ et $\mathbb{N}$), les booléens ($\mathbb{B}$) et les réels ($\mathbb{R}$) sont aussi de tels ensembles.

- Les types libres sont des ensembles finis énumérés. Par exemple, une commande est en cours, close ou facturée.

  $$\textit{état\_commande} ::= \textit{en\_cours} \mid \textit{close} \mid \textit{facturée}$$

- Les types construits sont définis à partir d'opérateurs sur les types prédéfinis. Par exemple, nous considérons un sous-ensemble des commandes possibles, les commandes de notre société :

  $$\textit{les\_commandes} \subseteq COMMANDE$$

Les variables utilisées en Z sont typées. Une déclaration simple de variable est la suivante :

$$\textit{une\_commande} : COMMANDE$$

Les définitions axiomatiques sont des déclarations de variables (simples ou fonctions) munies d'un prédicat ou contrainte.

$$\begin{array}{|l}
\textit{une_commande} : COMMANDE \\
\hline
\textit{une_commande} \in \textit{les_commandes}
\end{array}$$

$$\begin{array}{|l}
\textit{état_de_la_commande} : COMMANDE \nrightarrow \textit{état_commande} \\
\hline
\operatorname{dom} \textit{état_de_la_commande} \subseteq \textit{les_commandes}
\end{array}$$

La fonction dom représente le domaine d'une relation (l'ensemble des antécédents). La notation $\nrightarrow$ désigne une fonction partielle : tous les antécédents n'ont pas d'image. Cette information est précisée ensuite : "n'ont une image que les antécédents membres de l'ensemble *les_commandes*". Une définition générique est une définition paramétrée par des types. Par exemple, la fonction dom utilisée ci-dessus est définie comme suit :

$$[X, Y]$$
$$\mathrm{dom} : (X \leftrightarrow Y) \rightarrow \mathbb{P}\, X$$
$$\forall R : X \leftrightarrow Y \bullet \mathrm{dom}\, R = \{x : X, y : Y \mid x \mapsto y \in R \bullet x\}$$

Cette définition est utilisable quels que soient les ensembles X et Y. La fonction $\mathbb{P}\ X$ désigne la puissance d'un ensemble X, c'est-à-dire tous les ensembles constructibles à partir de X. Par exemple, $\mathbb{P}\,\{1, 2\} = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

Les prédicats sont définis en utilisant la logique des prédicats du premier ordre et ses opérateurs habituels ($\neg$, $\vee, \wedge, \Rightarrow, \forall$ ...). Ils sont de la forme suivante :

$$(quantifieurs \mid restriction \bullet \textit{prédicat})$$
$$(\forall c : COMMANDE \mid c \in lesCommandes \bullet a\_une\_facture\ c)$$

Ce prédicat signifie que tout commande référencée dans le système a une facture.

Un schéma est utilisé pour donner un nom à un groupe de déclarations et de prédicats. Il est présenté dans une boîte.

$$\textit{nom\_du\_schéma}[\textit{types paramètres}]$$
$$variable_1 : TYPE_1$$
$$\vdots$$
$$variable_n : TYPE_n$$
$$\textit{prédicat}$$

Un schéma est décrit textuellement par une λ-expression ou par la notation linéaire suivante :

$$\textit{nom\_du\_schéma}\ [\textit{types paramètres}] \mathrel{\widehat=} [variable_1 : TYPE_1; \ldots; variable_n : TYPE_n \mid \textit{prédicat}]$$

Des opérateurs permettent de combiner les schémas. Le calcul des schémas [Spi89, Hay92] est une algèbre des schémas avec des opérateurs d'ajout de prédicat $S \mid P$, fusion de déclarations $S1; D$, renommage $S[new/old]$, négation $\neq S$, produit $S \vee T$, union $S \wedge T$, implication $S \Rightarrow T$, équivalence $S \Leftrightarrow T$, masquage $S \setminus (v_1,\ v_2, ..., v_n)$, projection $S \upharpoonright (v_1,\ v_2, ..., v_n)$, pré-condition pre , postcondition *post*, composition $S; T$, surcharge $S \oplus T$, ...

Ce mécanisme complexe permet la réutilisation, la preuve et le raffinement de spécifications. Cette manière de procéder, moins courante que le schéma $nouveau := fonction(ancien)$, montre bien que Z est une approche déclarative et non pas impérative. Nous présentons trois opérateurs qui sont utilisés par la suite.

- $\Delta SCHEMA$ est un raccourcis d'écriture pour définir un nouveau schéma contenant les déclarations du schéma $SCHEMA$ et une duplication de ces déclarations, suffixées par une apostrophe (**'**) exprimant la nouvelle valeur des variables.
- $\Xi SCHEMA$ est le schéma invariant : les nouvelles valeurs des variables sont égales aux anciennes valeurs.
- $SCHEMA_2 \ \wedge\ SCHEMA_2$ indique une fusion des déclarations et une conjonction des prédicats.

Présentation d'une spécification La spécification Z est organisée de la façon suivante :

1. Les déclarations globales débutent la spécification. Elles contiennent :
   - des déclarations de types de base et de types libres,
   - des déclarations de constantes, variables et fonctions globales.
2. Les types structurés sont définis par l'utilisateur de la manière suivante :
   - Un schéma d'état regroupe des variables et un prédicat sur ces variables.
   - Un schéma initial est donné pour chaque schéma d'état.
   - Des schémas d'opérations décrivent chaque opération du type structuré. Ils se d'écomposent en :
     - Un schéma du nom de l'opération décrit les variables d'état, les paramètres et un prédicat. Le prédicat établit une relation entre les variables d'état et paramètres. Les paramètres en entrée (resp. en sortie) sont suffixés par un "? " (resp. par un "! "). La définition d'une opération complexe, contenant par exemple des erreurs, se fait par fusion de différents schémas de pré-conditions complémentaires.
     - Un schéma de précondition comprenant uniquement des paramètres en entrée est calculé ensuite.
3. L'état du système est défini par un schéma d'état comprenant :
   - un schéma donnant la structure du système (variables et invariant),
   - un schéma initial décrivant l'état initial (ou l'ensemble des états initiaux) du système.
4. Des schémas d'opérations décrivent chaque opération du système. Ils comprenent :
   - un schéma décrivant les variables d"état, les paramètres et un prédicat,
   - un schéma de précondition.
5. Des théorèmes et des preuves enrichissent la spécification.

La construction est itérative sur ce modèle. Pour simplifier nous allons d'abord définir l'état du système dans la section 5.2, son état initial dans la section 5.3 puis nous modélisons les opérations dans la section 5.4.

5.2 Spécification de l'état

5.2.1 Préambule : définition de types propres

Le modèle doit faire apparaître des clients, des produits, des commandes et des factures. Nous supposons que commandes et factures sont séparées (cas 2 de la section 3.2.2). Avant de nous intéresser à la facturation de la commande globalement, nous commençons par définir les clients, les produits, les commandes et les factures. Chacun de ces objets est défini par un type.

Pour chaque type, nous nous posons la question suivante : y-at-il un seul attribut ou plusieurs attributs? S'il n'y a qu'un attribut, nous définissons le type par un type libre; c'est le cas pour le type client. Pour les autres types, S'il y a plusieurs informations à représenter, nous utilisons des types construits; c'est le cas pour les types commande et produit. Deux constructions sont alors employées le produit cartésien lorsque les attributs ne sont pas nommés et le schéma dans le cas contraire. Dans le cas des commandes et des factures, nous utilisons le produit cartésien. Pour les lignes et les produits, nous utilisons le schéma parce qu'il est plus informatif et parce que les attributs peuvent changer de valeur. Nous ne définissons pas d'opérations pour ces types. Dans les deux cas se pose le problème de l'identifiant. Nous l'étudions dans le cadre du produit.

Clients Les clients sont représentés par un type prédéfini $CLIENT$. Aucune autre information n'est conservée. Ce type pourra être précisé par la suite, lors du raffinement de la spécification vers le code.

$$[CLIENT]$$

Produits Un produit est défini par un code, un libellé, un prix hors-taxe et une quantité. Prix et quantités sont représentés par des entiers naturels. Pour ne pas faire d'hypothèses particulières, nous supposons l'existence du type prédéfini pour les libellés et le code.

$$[LIBELLE, PRODUIT\_ID]$$

L'identifiant est une propriété qui distingue deux produits. Pour nous, l'identifiant est simplement le code du produit. Il peut faire partie intégrante du produit ou en être extrait. Examinons les deux cas.

1. Premier cas : l'identifiant fait partie intégrante du produit

   $$\begin{array}{|l}
   _\ PRODUIT1\ ____________________ \\
   code : PRODUIT_ID \\
   lib : LIBELLE \\
   qté : \mathbb{N} \\
   prix : \mathbb{N} \\
   \hline
   \end{array}$$

   $$\begin{array}{|l}
   _\ Produit1\ ____________________ \\
   produits : \mathbb{P}\ PRODUIT1 \\
   \hline
   \forall p_1, p_2 : produits \mid p_1 \neq p_2 \bullet p_1.code \neq p2.code \\
   \hline
   \end{array}$$

   Le problème de cette solution est l'écriture des contraintes de l'ensemble.

2. Second cas : l'identifiant est indépendant Nous allons donc modéliser séparément le type de valeurs et l'identifiant. L'avantage certain est que l'identifiant est ensuite utilisé seul et représente l'ensemble des valeurs (concision de la représentation).

   $$\begin{array}{|l}
   _\ PRODUIT\ ____________________ \\
   lib : LIBELLE \\
   qté : \mathbb{N} \\
   prix : \mathbb{N} \\
   \hline
   \end{array}$$

   L'ensemble suivant est mis dans un schéma ou inclus dans le schéma général de l'état du système.

   $$produits : PRODUIT\_ID \twoheadrightarrow PRODUIT$$

   Nous utilisons une surjection partielle car tous les produits du système ont un identifiant. L'identifiant sert à distinguer les homonymes : deux identifiants peuvent avoir la même valeur d'attribut (i.e. le même libellé, prix, quantité) et provenir de fournisseurs différents par exemple.

Le modèle ci-dessus s'applique dans de nombreux cas de figures, notamment chaque fois qu'un identifiant est utilisé. Nous l'utilisons systématiquement pour les commande et les factures.

Lignes Le schéma de spécification est celui des produits. La liaison est implantée de deux manières : soit par inclusion d'un attributs (exemple : produit), soit par relation entre identifiants (vers commande ou facture). Il s'agit de surjections car les commandes et les factures ont au moins une ligne. Le double identifiant des lignes de commandes et de factures est traduit par un produit cartésien pour le domaine. Nous évitons ainsi une nouvelle fonction faisant le lien entre la ligne et son constituant.

$LIGNE$

$prod : PRODUIT\_ID$
$prix : \mathbb{N}$
$qté : \mathbb{N}$

$[LIGNEC\_ID, LIGNEF\_ID]$
$lignes\_co : (COMMANDE\_ID \times LIGNEC\_ID) \twoheadrightarrow LIGNE$
$lignes\_fa : (FACTURE\_ID \times LIGNEF\_ID) \twoheadrightarrow LIGNE$

Commandes, factures Le schéma de spécification est le même que celui des produits. Il est implicitement supposé qu'à une facture correspond un seul client et une seule date. Ceci se traduit par l'usage de fonctions.

$REFERENCE$

$cli : CLIENT$
$date : DATE$

$[COMMANDE\_ID, FACTURE\_ID, DATE]$
$commandes : COMMANDE\_ID \rightarrow REFERENCE$
$factures : FACTURE\_ID \rightarrow REFERENCE$

5.2.2 Description de l'état du système

Récapitulons les descriptions de types du préambule.

$[CLIENT, LIBELLE, PRODUIT\_ID, COMMANDE\_ID, FACTURE\_ID, DATE,$
$LIGNEC\_ID, LIGNEF\_ID]$

$PRODUIT$

$lib : LIBELLE$
$qté : \mathbb{N}$
$prix : \mathbb{N}$

$LIGNE$

$prod : PRODUIT\_ID$
$prix : \mathbb{N}$
$qté : \mathbb{N}$

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{REFERENCE} \\
cli : CLIENT \\
date : DATE \\
\hline
\end{array}$$

Les déclarations de variables et de fonctions du préambule sont regroupées dans le schéma `Facturation` de la page 30. Les clients considérés sont un sous-ensemble des clients possibles. Les liens entre commande et facture, ligne de commande et ligne de facture sont ajoutés. L'identifiant d'une ligne de facture est un couple $(fact\_ID, lig\_ID)$. Le fait qu'une facture comporte plusieurs lignes est traduit par une fonction partielle des lignes dans les factures. Cette subtilité simplifie les prédicats par rapport au produit cartésien. Nous faisons les hypothèses suivantes : une commande est facturée un nombre quelconque de fois, une facture regroupe plusieurs commandes. Ceci implique que le lien entre commande et facture est de type relation et non fonction $(commande\_fact : COMMANDE\_ID \leftrightarrow FACTURE\_ID)$. Une ligne est facturée une seule fois, lig_fact est une fonction partielle.

L'invariant est un ensemble de contraintes sur l'état. Il existe deux façons de l'écrire : soit en partant des contraintes du texte informel, soit syntaxiquement en examinant tous les types apparaissant plusieurs fois. C'est ainsi que nous procédons méthodiquement pour les contraintes de domaines (1-8). Par contre, les contraintes portant sur les valeurs (9-12) sont issues de l'analyse de départ.

1. Les clients figurant sur les commandes ou les factures sont des clients du système modélisé.
2. Les produits commandés et/ou facturés sont des produits catalogués.
3. Les lignes de commandes font référence à des commandes existantes.
4. Les lignes de factures font référence à des factures existantes.
5. Les commandes facturées sont à la fois des commandes et des factures.
6. Une commande est facturée par une facture si et seulement si au moins une de ses lignes de commandes l'est.
7. Les lignes de commandes facturées ont fait l'objet d'une commande.
8. Les lignes de commandes facturées font référence à une facture.
9. La date des factures d'une commande est ultérieure à la date de la commande. Mais rien n'oblige à ce que ce soit le même client qui paie.
10. Pour une ligne facturée, la quantité commandée est inférieure à la quantité de la ligne de facture correspondante et le produit est le même. Elle n'est pas égale car une ligne de facture peut regrouper plusieurs lignes de commande.
11. Un produit donné n'apparaît qu'une fois par commande.
12. Un produit donné n'apparaît qu'une fois par facture.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{Facturation} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
clients : \mathbb{P}\, CLIENT \\
produits : PRODUIT\_ID \twoheadrightarrow\!\!\!\!\!\!\!+\;\, PRODUIT \\
commandes : COMMANDE\_ID \nrightarrow REFERENCE \\
factures : FACTURE\_ID \nrightarrow REFERENCE \\
lignes\_co : (LIGNEC\_ID \nrightarrow COMMANDE\_ID) \twoheadrightarrow\!\!\!\!\!\!\!+\;\, LIGNE \\
lignes\_fa : (LIGNEF\_ID \nrightarrow FACTURE\_ID) \twoheadrightarrow\!\!\!\!\!\!\!+\;\, LIGNE \\
commande\_fact : COMMANDE\_ID \leftrightarrow FACTURE\_ID \\
lig\_fact : (LIGNEC\_ID \nrightarrow COMMANDE\_ID) \nrightarrow (LIGNEF\_ID \nrightarrow FACTURE\_ID) \\
\hline
\forall r : REFERENCE \mid r \in \operatorname{ran} commandes \lor r \in \operatorname{ran} factures \bullet r.cli \in clients \\
\forall l : LIGNE \mid l \in \operatorname{ran} lignes\_co \lor l \in \operatorname{ran} lignes\_fa \bullet l.prod \in \operatorname{dom} produits \\
\operatorname{ran} \operatorname{dom} lignes\_co \subseteq \operatorname{dom} commandes \\
\operatorname{ran} \operatorname{dom} lignes\_fa \subseteq \operatorname{dom} factures \\
\operatorname{dom} commande\_fact \subseteq \operatorname{dom} commandes \land \operatorname{ran} commande\_fact \subseteq \operatorname{dom} factures \\
\operatorname{dom} commande\_fact = \operatorname{ran} \operatorname{dom} lig\_fact \land \operatorname{ran} commande\_fact = \operatorname{ran} \operatorname{ran} lig\_fact \\
\operatorname{dom} lig\_fact \subseteq \operatorname{dom} lignes\_co \\
\operatorname{ran} lig\_fact \subseteq \operatorname{dom} lignes\_fa \\
\forall fa : FACTURE\_ID, co : COMMANDE\_ID \mid (co, fa) \in commande\_fact \bullet \\
\quad commandes(co).date \leq factures(fa).date \\
\forall co : COMMANDE\_ID, lc : LIGNEC\_ID \mid (lc, co) \in \operatorname{dom} lig\_fact \bullet \\
\quad lignes\_co(lc, co).qt\acute{e} \leq lignes\_fa(lig\_fact(lc, co)).qt\acute{e} \land \\
\quad lignes\_co(lc, co).prod = lignes\_fa(lig\_fact(lc, co)).prod \\
\forall co : COMMANDE\_ID, lc_1 : LIGNEC\_ID, lc_2 : LIGNEC\_ID \mid \\
\quad (lc_1, co) \in \operatorname{dom} lignes\_co \land (lc_2, co) \in \operatorname{dom} lignes\_co \bullet \\
\quad lignes\_co(lc_1, co).prod \neq lignes\_co(lc_2, co).prod \\
\forall fa : FACTURE\_ID, lf_1 : LIGNEF\_ID, lf_2 : LIGNEF\_ID \mid \\
\quad (lf_1, fa) \in \operatorname{dom} lignes\_fa \land (lf_2, fa) \in \operatorname{dom} lignes\_fa \bullet \\
\quad lignes\_fa(lf_1, fa).prod \neq lignes\_fa(lf_2, fa).prod \\
\hline
\end{array}
$$

5.3 Spécification de l'état initial

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{InitFacturation} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
Facturation' \\
\hline
clients' = \{\} \\
produits' = \{\} \\
commandes' = \{\} \\
factures' = \{\} \\
\hline
\end{array}
$$

Le lecteur pourra montrer facilement qu'un tel état initial existe. L'invariant implique que les autres variables d'état sont vides.

5.4 Spécification des opérations

Chaque opération du système donne lieu à un schéma de la forme suivante :

$$
\begin{array}{|l}
\hline
\textit{ClientsDeLaCommande} \\
inclusion de l'état \\
arguments en entrée \\
arguments en sortie \\
\hline
prédicats sur l'état et les arguments \\
\hline
\end{array}
$$

L'inclusion se fait habituellement avec l'opérateur Δ ou l'opérateur Ξ. Nous distinguons les opérations selon l'opérateur utilisé en deux catégories : les opérations d'interrogation qui ne changent pas l'état du système (Ξ) et les opérations qui modifient l'état (Δ). Avec ce type de modélisation, les opérations sont proches des requêtes de consultation et de mise-à-jour dans les bases de données.

5.4.1 Opérations d'interrogation

Les requêtes à décrire sont :

- Quels est le client d'une commande de numéro donné ? `ClientsDeLaCommande`
- Quels sont les factures d'une commande ? `FacturesDeLaCommande`
- Quels sont les produits d'une commande ? `ProduitsDeLaCommande`
- Quels sont les clients d'une commande de numéro donné ?
- Quels sont les produits en stocks ?
- Quel est le montant HT d'une commande ?
- Quelles sont les commandes ou factures vides ?
- Quels sont les clients qui ont commandé un produit donné ?
- Pour quelles commandes le payeur n'est pas le client qui a commandé ?
- Quelles sont les commandes entièrement facturées ? Les commandes non facturées ? `CommandesFacturées`
- Qui paie les factures de qui ?
- Quels sont les clients qui ont commandé plus d'une quantité donnée d'un produit donné ?
- Quel est le prix d'un produit dont on connait le code ?
- Quelle est la quantité restante d'un produit dont on connait le code ?
- Quelles sont les factures complètes qui ne concernent qu'une seule commande ?
- etc.

$$
\begin{array}{|l}
\hline
\textit{ClientsDeLaCommande} \\
\Xi Facturation \\
co? : COMMANDE\_ID \\
cli! : CLIENT \\
\hline
co? \in \operatorname{dom} commandes \\
cli! = commandes(co?).cli \\
\hline
\end{array}
$$

La précondition est $co? \in \mathrm{dom}\, commandes$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;FacturesDeLaCommande\;}}\\
\Xi Facturation\\
co? : COMMANDE_ID\\
fact! : \mathbb{P}\ FACTURE_I D\\
\hline
fact! = commande_fact(\!| \{co?\} |\!)\\
\hline
\end{array}$$

$R(\!|S|\!)$ est l'image de l'ensemble S par la relation R. La précondition est vraie et le résultat peut être vide pour deux raisons : soit la commande n'existe pas, soit elle n'a pas encore de facture.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;ProduitsDeLaCommande\;}}\\
\Xi Facturation\\
co? : COMMANDE_ID\\
prods! : \mathbb{P}\ LIBELLE\\
\hline
prod! = \{l : LIGNE \mid l \in lignes_co(\!|(\mathrm{dom}\, lignes_co) \rhd \{co?\}|\!) \bullet produits(l.prod).lib\}\\
\hline
\end{array}$$

Il s'agit d'une définition en compréhension de l'ensemble avec une condition et un motif $\{decl \mid cond \bullet motif\}$. $R \rhd S$ est la restriction d'image d'une relation R à un ensemble S. La précondition est vraie et le résultat peut être vide pour deux raisons : soit la commande n'existe pas, soit elle n'a pas encore de lignes de commandes.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;CommandesFactur\acute{e}es\;}}\\
\Xi Facturation\\
co_ef! : \mathbb{P}\ COMMANDE_ID\\
co_nf! : \mathbb{P}\ COMMANDE_ID\\
\hline
co_ef! = \{co : COMMANDE_ID \mid (\mathrm{dom}\, lignes_co) \rhd \{co\}) = (\mathrm{dom}\, lig_fact) \rhd \{co\})\}\\
co_nf! = \{co : COMMANDE_ID \mid (\mathrm{dom}\, lig_fact) \rhd \{co\}) = \}\\
\hline
\end{array}$$

Une commande est entièrement facturée si chaque ligne de la commande est facturée. Une commande est non facturée si aucune de ses lignes de la commande est facturée.

5.4.2 Opérations de modification

Les opérations à décrire sont entre autres :

- Ajouter un client, un produit.
- Ouvrir une commande.
- Ajouter des lignes dans une commande (non facturée).
- Annuler une ligne de commande (non facturée).
- Facturer des lignes de commandes.
- Supprimer des clients, des produits, des commandes, des factures.

Il faut prendre soin de vérifier les préconditions qui assurent l'invariant. Par exemple, l'ajout d'une ligne de commande implique la création de la commande si elle n'existe pas. Une bonne méthode de modélisation consiste à séparer les cas dans des schémas différents puis à faire l'union des schémas. Prenons l'exemple de l'ajout d'une commande. Plusieurs cas de figure sont envisageables :

1. la commande existe, elle n'est pas enregistrée
2. la commande n'existe pas
   (a) le client existe, la commande est enregistrée
   (b) le client n'existe pas, l'enregistrement est refusé (pour ne pas boucler, nous ajoutons l'opération enregistrer auteur et ouvrage dans la spécification).

$$REPONSE ::= CommandeExistante \mid ClientInexistant \mid CommandeEnregistrée$$

$$\begin{array}{|l}
\multicolumn{1}{l}{AjoutCommandeExistant} \\
\Delta Facturation \\
re? : REFERENCE \\
co? : COMMANDE_ID \\
c? : CLIENT \\
rep! : REPONSE \\
\hline
co? \in \operatorname{dom} commandes \\
rep! = CommandeExistante \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{AjoutCommandeClientInexistant} \\
\Delta Facturation \\
re? : REFERENCE \\
co? : COMMANDE_ID \\
c? : CLIENT \\
rep! : REPONSE \\
\hline
co? \notin \operatorname{dom} commandes \\
re?.cli \notin \operatorname{dom} clients \\
rep! = ClientInexistant \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{AjoutCommandeClientExistant} \\
\Delta Facturation \\
re? : REFERENCE \\
co? : COMMANDE_ID \\
c? : CLIENT \\
rep! : REPONSE \\
\hline
co? \notin \operatorname{dom} commandes \\
re?.cli \in \operatorname{dom} clients \\
commandes' = commandes \bigcup \{co? \mapsto re?\} \\
rep! = CommandeEnregistrée \\
\hline
\end{array}$$

$$\begin{aligned}AjoutCommande == {} & AjoutCommandeExistante \lor AjoutCommandeClientInexistant \\ & \lor AjoutCommandeClientExistant\end{aligned}$$

Nous laissons au lecteur le soin de calculer et vérifier la précondition. Le principe est similaire pour les factures et les lignes. Pour les lignes l'ajout se fait aussi si la commande n'est pas facturée.

5.5 Remarques

Nous avons donné des éléments de la spécification. Une spécification complète contient des théorèmes et des preuves. Différents types de théorèmes sont à démontrer : l'existence d'un état initial, la validité de la précondition et les propriétés générales du système. La validation de la spécification se fait au travers des preuves de propriétés générales mais aussi du paraphrasage de la spécification en langage naturel. Ce travail très important sort du cadre que nous nous sommes fixés ici.

La spécification, une fois établie et validée, est ensuite raffinée pour introduire des structures de données proches de celles des langages de programmation. Chaque pas de raffinement doit être prouvé à la main, avec les risques d'erreurs liés à l'absence d'une vérification automatisée. Une autre approche du raffinement est de transcrire les spécifications en machines B et d'utiliser l'outillage de l'outil `B`[Abr96, Lan96].

Contrairement à Merise ou OMT, Z oblige à réfléchir très précisément sur les types des attributs et les opérations. La notation Z nécessite un apprentissage important à cause de la syntaxe complexe, mais les concepts sont moins nombreux qu'en OMT. L'utilisateur doit être familiarisé avec la logique des prédicats et la théorie des ensembles.

Z ne prend pas en compte les aspects dynamiques du système. Des extensions de Z aux processus concurrents ou à la logique temporelle pallient en partie à ce problème [GS96, Lan94, Smi93]. Mais les extensions ne sont pas toujours cohérentes avec Z.

Z souffre du manque de constructions modulaires pour spécifier des systèmes complexes. La spécification est à un seul niveau d'abstraction et de représentation. En effet les schémas et leur calcul sont des raccourcis syntaxiques mais pas des structures modulaires. Les machines B et les extensions à objets de Z [SBC92, DD90, AG91] répondent à ce besoin.

6 TAG/CF

6.1 Présentation des TAG/CF

6.1.1 La méthode

La méthode Types Abstraits Graphiques et Classes Formelles est synthétisée dans [And95, ABR95]. Dans cette approche, les composants logiciels sont exprimés à trois niveaux d'abstraction : types abstraits graphiques (TAG en abrégé), classes formelles (CF) et langages de programmation. Les niveaux d'abstraction n'expriment pas une dichotomie entre analyse, conception et réalisation mais s'interprètent plutôt comme une présentation distincte de raffinement en trois étapes. Les principes présentés ici servent à la fois à la spécification/prototypage des besoins et à la description/implantation du logiciel.

Comme dans OS/OP [Bre91], il y a en fait deux modèles à objets. Dans ces modèles, une classe définit un type et il n'y a pas d'identité d'objet. L'envoi de message correspond à un appel fonctionnel. L'héritage entraîne le sous-typage. Les classes abstraites définissent des classes non instanciables, décrivant en général des comportements partiels. La relation de clientèle des modèles à objets est représentée par la relation d'importation des modules.

Le modèle TAG peut être utilisé pour décrire sommairement les objets de l'analyse ou bien en complément d'autres formalismes. Il permet une description abstraite des objets sans se soucier de leur structure (agrégation, attributs) mais en tenant compte des contraintes d'enchaînement des opérations. Cette description peut ensuite être affinée dans une spécification algébrique pour mettre en valeur les propriétés fonctionnelles ou simulée par exécution concurrente d'automates communicants pour dégager les propriétés dynamiques.

Le modèle des classes formelles est utilisé pour la conception abstraite des classes. La conception par classes formelles comprend plusieurs tâches : extraire des classes à partir des TAG (structure et comportement), restructurer les classes obtenues pour les intégrer aux classes existantes (raffinement pour réutiliser des classes existantes, généraliser pour prendre en compte la hiérarchie d'héritage, optimisations pour rendre plus efficace). Il sert enfin de support pour la traduction dans différents environnements de programmation.

Conception des TAGs en CF La modélisation en termes de classes formelles peut se faire intuitivement à partir de la description informelle ou par conception des types abstraits graphiques. Cette conception se fait à partir de l'automate et optionnellement de la spécification algébrique associée si elle existe. Les classes formelles raffinent les TAG en donnant une représentation particulière mais abstraite. Les opérations deviennent des méthodes, et les axiomes, s'ils existent, sont groupés dans les méthodes. Deux approches extrêmes sont possibles pour passer d'un type (abstrait) de données à une classe (formelle) : la conception plate et la conception ordonnée.

Conception plate En conception plate, une seule classe décrit le TAG. Si l'automate est représenté par des prédicats d'état ou une fonction de transition, alors les préconditions des méthodes sont calculables directement. Dans le cas contraire, le concepteur les exprime en fonction d'une structure abstraite qu'il aura dégagé. Supposons maintenant une structure abstraite quelconque. Puisque les observateurs peuvent être partiels, certains sélecteurs de champ peuvent ne pas avoir de valeur à un instant donné. Nous utilisons de préférence un champ conditionnel à la valeur indéfinie (`nil` ou $\perp$) usuelle en programmation. La conception d'une telle classe n'est pas triviale à partir du TAG. Les inconvénients de cette approche sont : descriptions souvent opérationnelles, préconditions complexes, classes volumineuses pas faciles à lire, à comprendre et à spécialiser.

Conception ordonnée En conception ordonnée, un TAG est défini par un schéma. Rappelons qu'un schéma est un sous-ensemble du graphe d'héritage de racine unique une classe abstraite représentant le type de données. Par défaut, le schéma est initial, c'est-à-dire qu'il comporte deux niveaux : une racine et des feuilles appelées classes terminales. La racine est une classe abstraite correspondant au TAG. Les classes terminales représentent les états (le nom de la classe est la concaténation du nom du type et du nom de l'état).

6.1.2 Restriction au cas d'étude

Comme pour les autres méthodes utilisées dans notre étude, nous nous contentons d'écrire des spécifications. Nous allons donc modéliser le cas de la facturation avec les TAGs dans la section 6.2 en détaillant la partie fonctionnelle dans la section 6.3. Puis nous donnons anecdotiquement une modélisation avec les classes formelles dans la section 6.4.

6.1.3 Formalismes utilisés

Types abstraits graphiques Le premier modèle, dit des TAG, est la spécification la plus abstraite des composants logiciels, appelés types abstraits graphiques. Seuls les aspects dynamiques sont décrits sous forme d'un automate gardé. La représentation graphique est plus naturelle et plus intuitive que la spécification formelle. Elle est composée de quatre encarts : (1) le nom du type en cours de définition, le **type d'intérêt** avec les déclarations de constantes et de types paramètres de la spécification, (2) les profils des opérations (3) les noms d'états, les gardes et l'invariant, (4) l'automate du comportement dynamique. Par convention, l'encart (2) sera écrit dans le texte et non sur le schéma pour alléger l'écriture.

eth$_1$) dtri$_1$(sparam nom_m

...

eth$_n$) dtr$_n$(sparam nom_m

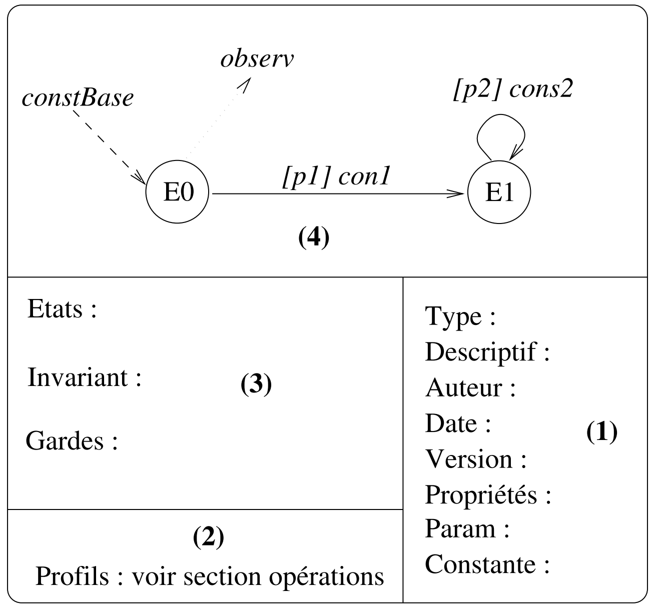


Figure 22 : Structure d'un TAG

Toute opération contient au moins une fois le type d'intérêt dans son profil, sinon cette opération relève forcément d'une autre abstraction de donnée. Nous utilisons la convention syntaxique suivante : *si le type d'intérêt est en paramètre d'une opération, alors il se trouve en première position et le nom de la variable associée est* `self` (connotation d'objet receveur). Comme d'usage, un **constructeur** est une opération dont le résultat est le type d'intérêt. Il est dit **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 état correspondent à un ensemble de valeurs et les transitions à une invocation d'opération. Trois types d'arcs existent :

1. les arcs entre deux états : ils correspondent à des constructeurs.
2. les arcs sans état d'arrivée : il correspondent à des observateurs. Ils sont représentés par des vecteurs en pointillé.
3. les arcs qui n'ont pas d'état de départ. Ils sont représentés par un vecteur en ligne discontinue, labellé par le nom d'un constructeur de base.

Les transitions sont parfois enrichies de gardes, prédicat portant sur les observateurs autorisés dans l'état et les paramètres de l'opération associée à la transition.

Nous utilisons diverses notations pour réduire la taille de l'automate (inconvénient majeur de ce formalisme), et basées sur l'agrégation et la généralisation d'états ou de transitions (voir [And95]). Nous en utilisons certaines ici.

- Profils, noms d'état et de gardes sont décrits dans des encarts.
- Une transition peut supporter plusieurs opérations : ceci correspond en réalité à plusieurs transitions.
- Les opérations invocables dans tous les états et qui ne changent pas d'état sont qualifiées de neutres et ne sont pas notées sur l'automate (exemple : l'opération `client` dans la figure 24).
- Les transitions similaires sont généralisées dans des super-états, notation inspirée de [Har87](exemple : état `EC` de la figure 24).

- Une transition est décomposable en plusieurs opérations.
- Un état est décomposable en un automate interne.

Aucune structure d'objet n'est exhibée contrairement aux objets d'OMT. Tout le modèle repose sur le comportement (les méthodes), ce qui favorise une certaine abstraction de la description. Le formalisme des automates a été choisi parce qu'il est visuel, modulaire, bien défini, qu'il existe des systèmes de preuve et surtout pour sa double interprétation possible. Cette double interprétation est d'une part un système d'états 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. La notion d'état est une abstraction de la structure des objets. Une transition correspond à un événement et à un appel d'opération.

Des hypothèses naturelles sur les TAG sont définies : connexité, gardes exclusives, états initiaux, états non vides et finiment engendrés, pas de concurrence intra-objet. Le problème de l'explosion combinatoire du nombre d'états et de la structuration des automates complexes est étudié dans [And95].

La sémantique formelle des TAG est donnée en construisant un type abstrait algébrique à partir de l'automate. L'originalité du modèle TAG est que le système exploite les informations de l'automate pour aider l'utilisateur à construire une spécification fonctionnelle de type abstrait algébrique. 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. Des algorithmes permettent d'obtenir automatiquement l'extraction de certaines informations. En particulier, l'extraction de la signature est triviale. La construction de l'axiomatique se fait ensuite en trois étapes comme nous le verrons dans la section 6.3

Dans les TAG, une distinction est faite entre héritage d'extension (proche de la coercition) pour les classes complètement définies et héritage de concrétisation pour des classes définissant des comportements partiels. Cette définition ne tient pas compte de l'héritage d'automates, pour lequel seules des heuristiques existent actuellement.

La concurrence n'est actuellement possible qu'au travers d'objets modélisant la communication. Une extension aux automates communicants est à l'étude.

Classes Formelles Le second modèle dit des Classes Formelles est un modèle commun au langages de programmation à objets. Il a pour objet la conception abstraite, la preuve de propriétés de la conception et l'automatisation du codage vers les langages de programmation à objets tels que Smalltalk, Eiffel, C++ ou CLOS. Un certain nombre de contrôles sont mis en œuvre concernant le typage, l'héritage et l'évaluation des spécifications. L'accent est mis sur la rigueur d'un formalisme de description des objets par des classes formelles et la conception à objets d'un logiciel.

Une CF est définie par un aspect,des extensions et des méthodes de classe. La figure 23 illustre cette décomposition. L'aspect est l'ensemble suffisant des méthodes (primitives) qui caractérisent un objet : création, copie, égalité, accesseurs. Il est décrit syntaxiquement en un ensemble de sélecteurs de champs (nom de méthode, type du résultat) et une contrainte portant sur ces sélecteurs de champs. Cette description syntaxique engendre automatiquement les méthodes primitives. Les sélecteurs de champs peuvent être conditionnées par un axiome. Dans le modèle CF, les axiomes sont des équations conditionnelles positives de la forme

$$\mathtt{t}_1 \;\mathtt{==}\; \mathtt{v}_1 \;\mathtt{AND}\; \ldots \;\mathtt{AND}\; \mathtt{t}_n \;\mathtt{==}\; \mathtt{v}_n \;\mathtt{==>}\; \mathtt{t}_{n+1} \;\mathtt{==}\; \mathtt{v}_{n+1}$$

où les $\mathtt{t}_i$ `==` $\mathtt{v}_i$ sont des équations de termes. Les extensions appelées aussi méthodes secondaires sont définies à partir de l'aspect. Elles sont définies par un profil (nom, paramètres typés, type du résultat), un commentaire et des axiomes. Les méthodes de classe sont définies comme des extensions à ceci près qu'elles ne contiennent pas le nom de la classe en cours de définition dans les paramètres. Elles servent notamment à l'instanciation des objets.

L'écriture d'une CF est inspirée de la programmation à objets. L'aspect décrit le noyau nécessaire et suffisant à la description complète des objets de la classe. Les extensions enrichissent le noyau sans le modifier. Cette pratique est conforme à l'usage de la programmation à objets. Elle a plusieurs avantages : d'une part, c'est une spécification algébrique particulière qui sous certaines hypothèses est suffisamment complète et non-contradictoire et d'autre part, elle vérifie la propriété naturelle en programmation à objets que l'ajout d'une nouvelle méthode secondaire ne change pas le type des instances.

```
                         Nom_Classe
              inherits from super-classes

comments: commentaire

features: caractéristiques importées

from: type import: opérations
---------------------------------------------------------------
                 aspect : nom de l'aspect
field selectors
 sélecteur_de_champ                : Nom_Classe ⟶ Type
 sélecteur_de_champ_conditionnel   : Nom_Classe ⟶ Type
   requires: condition
---------------------------------------------------------------
                       extensions
nom_méthode_instance : Nom_Classe Types_des_Paramètres ⟶ Type_du_Résultat
;; nom_méthode_instance : commentaire
var: variable : Type
 équations ou équations conditionnelles
---------------------------------------------------------------
                      class methods
nom_méthode_de_classe : Types_des_Paramètres ⟶ Type_du_Résultat
;; nom_méthode_de_classe : commentaire
var: variable : Type
 équations ou équations conditionnelles
```

Figure 23 : Structure d'une classe formelle

Si globalement une classe formelle est proche d'une classe Eiffel plusieurs originalités existent : la proximité avec les spécifications algébriques et tous les outils ou techniques qui s'en inspirent, les pré-conditions sur les sélecteurs de champs, permettent plus de souplesse dans la définition des classes. Le modèle possède un système statique de contrôle de types sûr [CCR95]. Le passage aux langages de programmation à objets est simple et automatisable, mais une phase d'optimisation est nécessaire pour obtenir un prototype efficace.

6.2 Modélisation avec les Types Abstraits Graphiques

La modélisation TAG débute par le recensement des objets. Ces objets sont inventoriés intuitivement ou proviennent d'une analyse avec les méthodes à objets telles qu'OMT (cf section 4.1). La lecture de l'énoncé fait apparaître les objets suivants : commande, facture, produit, client. Le modèle TAG amène naturellement à s'intéresser aux objets dont le comportement évolue au fil du temps. Pour chacun des objets recensés, on se pose les questions suivantes. Quel est le comportement propre (méthodes) de l'objet ? Quels sont les états du cycle de vie de l'objet ? Quelles sont les méthodes disponibles dans ces états ? Quels sont les profils des méthodes ? Lorsqu'on s'intéresse à un ensemble d'objets, l'analyste se pose les questions suivantes. Quelles sont les transitions qu'il faut synchroniser ? Quels sont les communications, les paramètres, les objets connus ou référencés dans d'autres objets ? Comment évoluent les relations entre objets du point de vue de la durée (permanente, temporaire) de la visibilité (partagé, exclusif) ?

Problème spécifique de la représentation des instances La spécification commence par la description des types d'objets puis celle des instances. L'étude d'un objet fait apparaître des liens vers des objets déjà recensés ou de nouveaux objets. Il faut comprendre ces objets pour bien modéliser l'objet en cours de description : la modélisation TAG se fait donc en parallèle pour chaque objet. Le lien est alors perçu de deux manières :

- soit l'objet est exclusif : dans ce cas une valeur du type suffit pour en représenter l'instance,
- soit l'objet est partagé : dans ce cas, la valeur est insuffisante pour représenter l'instance car si l'instance évolue dans le temps il faudrait mettre à jour dans tous les objets où elle apparaît. C'est le cas entre la commande et la facture qui se référencent mutuellement ou du produit dans la ligne de commande dont la quantité en stock évolue. Il faut alors représenter les instances du système par un identifiant et y accéder via une indirection. Cette façon de faire est habituelle en Z où les identifiants sont liés à leur information par une fonction.

Dans les spécifications algébriques la construction de la spécification est ascendante (ou hiérarchique). Des types de base sont d'abord définis ; les autres types sont définis par inclusion (enrichissement) de ces types. Ici ce n'est pas le cas du fait des dépendances mutuelles entre objets.

Nous commençons par donner un aperçu du TAG commande dans le §6.2.1. Nous l'approfondissons ensuite par la description des lignes de commande dans le §6.2.2 et la facturation dans le §6.2.3 avant d'en donner une version finale dans le §6.2.4.

6.2.1 Aperçu de la commande

Nous commençons par décrire la commande, un objet essentiel de l'application, dans la figure 24. Une commande est effectée par un client, puis remplie, consultable et facturée. Le client est décrit uniquement par un nom pour simplifier. Le remplissage d'une commande est la donnée de lignes de commande chacune comprenant un produit, une quantité, un prix unitaire hors-taxe et une T.V.A. Les lignes sont numérotées à l'intérieur de la commande par ordre de remplissage. Nous supposons que la T.V.A. est donnée par ligne de commande et non globalement. Les profils des opérations sont les suivants :

```
ouvrir(cli : Car, d : Date ) : Commande
client(self : Commande) : Car
dateCommande(self : Commande) : Date
ajouter_ligne(self : Commande, prod : Produit, qt
                         prix : R
nbLignes(self : Commande) : Entier
produitLigne(self : Commande, n : Entier) : Produit
prixLigne(self : Commande, n : Entier) : R
qt
tvaLigne(self : Commande, n : Entier) : TVA
supprimer_ligne(self : Commande, n : Entier) : Commande
modifier_ligne(self : Commande, n : Entier, , prod : Produit, prix : R
                         qt
ajouter_frais(self : Commande, frais : R
modifier_frais(self : Commande, frais : R
frais(self : Commande) : R
fixer_remise(self : Commande, taux : R
modifier_remise(self : Commande, taux : R
remise(self : Commande) : R
totalHT(self : Commande) : R
totalTVA(self : Commande) : R
totalTTC(self : Commande) : R
imprimer(self : Commande) : Commande
facturer(self : Commande) : Commande
```

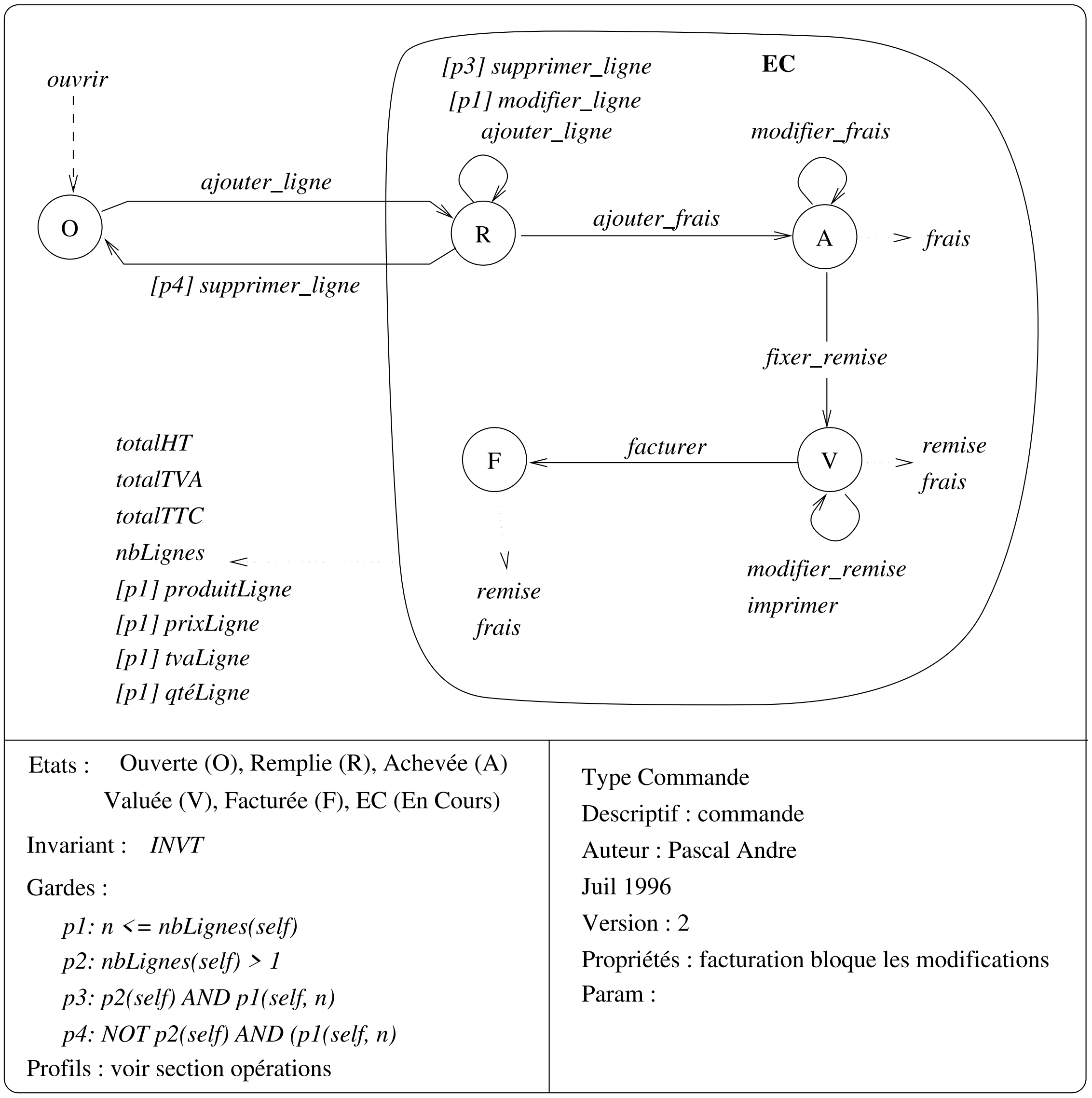


Figure 24 : L'automate du TAG `Commande` : version préliminaire

6.2.2 La commande et ses composants

Décrivons maintenant les éléments de la commande, en insistant notamment sur les lignes de commandes et les produits.

Description des prix, quantité et T.V.A. La quantité et le prix sont définis par des types de base (`Entier` et `Réel`). La T.V.A. est définie par un type `TVA` comprenant un taux et un libellé.

1- `newTVA(taux : R`
`taux(self : TVA) : R`
`intitul`
`set_taux(self : TVA, taux : R`
`set_intitul`

2- `newTVABase : TVA`
`newTVANorm : TVA`
`newTVALuxe : TVA`
`taux(self : TVA) : R`
`intitul`

Dans la seconde définition, trois taux sont distingués. Les axiomes doivent donner la valeur du taux et son intitulé. Cette définition est plus statique que la précédente et donc moins évolutive.

Description des lignes de commande Dans le modèle du §6.2.1, l'ajout d'une ligne de commande se fait par l'opération `ajouter_ligne(self : Commande, prod : Produit, qté : Entier, prix : Réel, tva : TVA) : Commande`. Pour éviter la manipulation d'un grand nombre d'arguments et rendre plus concise la spécification, nous introduisons un TAG simple `LigCom`. Ce TAG a un comportement dynamique trivial avec un seul état. Il comporte uniquement des accesseurs (opérations de lecture et modification). Pour éclaircir la description en précisant que le prix est unitaire et hors-taxe, nous appelons ce prix `puht`. Les profils sont les suivants :

```
e : Entier, puht : R newLC(prod : Produit, qt
prod_lig(self : LigCom) : Produit
puht_lig(self : LigCom) : R
qt
tva_lig(self : LigCom) : TVA
set_prod_lig(self : LigCom, prod : Produit) : LigCom
set_puht_lig(self : LigCom, puht : R
e_lig(self : LigCom, qt set_qt
set_tva(self : LigCom, tva : TVA) : LigCom
```

Description des produits Un produit est défini par un type `Produit` comprenant un libellé, un prix unitaire hors-taxe et une quantité en stock. Séparons la notion de produit au catalogue (avec sa description) de produit en stock car ce qui apparaît dans la commande c'est le produit au catalogue.

```
newProduitAuCatalogue(prix : R
puht(self : ProduitAuCatalogue) : R
libell
set_puht(self : ProduitAuCatalogue, taux : R
set_libell
```

Le produit en stock est défini par héritage de la façon suivante :

```
ProduitEnStock inherits from ProduitAuCatalogue
eel, lib : Car, qt newProduitEnStock(prix : R
quantit
ajouter(self : Produit, qt
retirer(self : Produit, qt
```

Un produit en stock est donc par défaut un produit au catalogue. Maintenant, considérons le produit du point de vue de la commande. Nous avons une référence à un produit au catalogue. Pour commander nous devons nous assurer de la quantité disponible. Si le produit au catalogue est dynamiquement un produit en stock (polymorphisme d'héritage) alors nous pouvons consulter la quantité[2]. Sinon il faut chercher le produit en stock correspondant. Il y a alors une distinction nette entre produit au catalogue et produit en stock, et il faut ajouter une bijection entre les deux. Nous avons mis en valeur une confusion entre héritage et structuration. En réponse à ce problème, nous faisons abstraction du stock pour ne conserver qu'un seul TAG : `Produit`.

Il se pose aussi le problème de la liaison du produit à la commande. Ce problème de liaison apparaît plusieurs fois dans la modélisation. Il est habituel dans les modèles à effets de bord. Si un objet est représenté par une valeur et référencé dans d'autres objets. Modifier la valeur de l'objet équivaut à modifier toutes ses références. La solution classique consiste à introduire une indirection pour séparer la référence à l'objet de l'objet lui-même. Nous l'examinons maintenant.

[2] Cette consultation sera rejetée par un contrôleur de type statique, puisque la méthode `quantité_stock` ne fait pas partie du comportement de la classe `ProduitAuCatalogue`.

Une réponse au problème spécifique de la représentation des instances Lorsqu'un produit apparaît dans plusieurs commandes, la spécification précédente du produit implique que le libellé et le prix (actuel) est dupliqué autant de fois qu'il y a de commandes le concernant. C'est un problème classique de redondance dans les bases de données. Il entraîne des problèmes de cohérence de mise-à-jour. Par exemple, si le libellé ou le prix change, il faut répercuter le changement dans toutes les commandes. Une réponse simple est de donner des noms distincts (identifiants) aux produits et référencer les produits uniquement par ces noms. Dit autrement, il faut modéliser de manière unique les instances de la classe `Produit`. Nous utilisons pour cela un type générique `Dictionnaire[Clé, Valeur]` supposé exister. La spécification des instances de la classe `Produit` est une actualisation du type générique `Produits-Inst : Dictionnaire[ProduitID, Produit]` Il en est de même pour la T.V.A. : `TVA-Inst : Dictionnaire[TVAID, TVA]`.

La spécification des produits est modifiée. Ce type est complexe. Il nécessite un TAG à deux états `EnStock` et `Rupture`, distingués par la quantité en stock. La constante `qtéMin` indique la valeur minimale du stock, elle se traduit par un paramètre supplémentaire dans le profil de `newProduit` et un observateur (`qtéMin`).

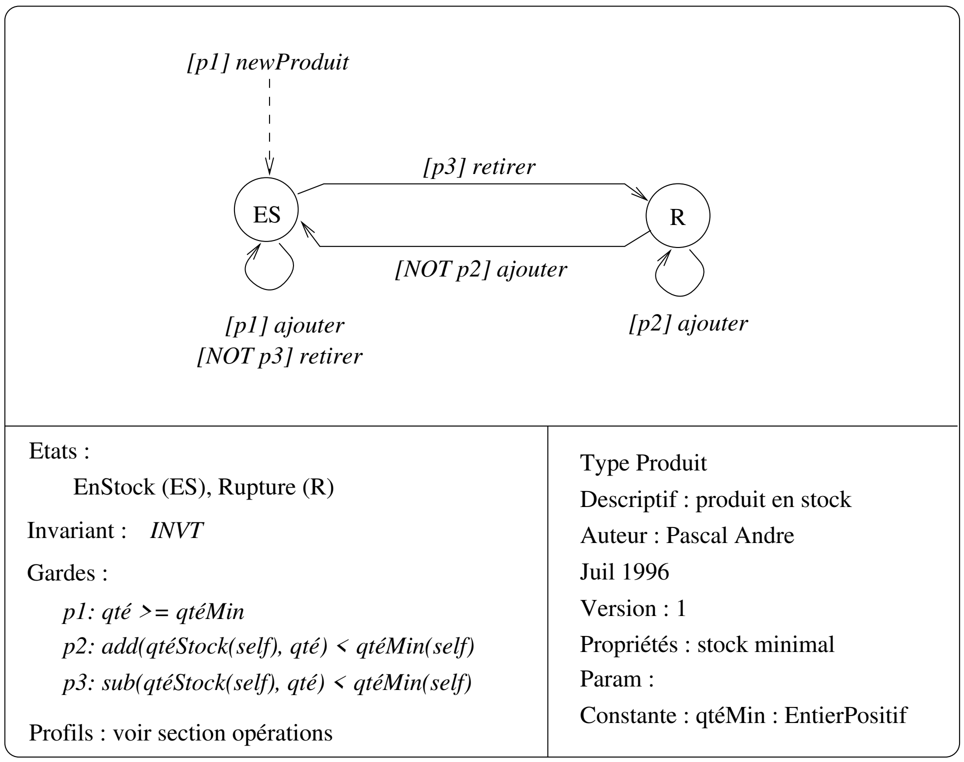


Figure 25 : L'automate du TAG `Produit`

eel,elibEnCla, qt

```
newProduit(prix : R
puht(self : Produit) : R
libell
qt
qt
set_puht(self : Produit, taux : R
set_libell
ajouter(self : Produit, qt
retirer(self : Produit, qt
```

Description des lignes de commande (suite et fin) La nouvelle spécification de la ligne de commande prend en compte les identifiants de produits et de T.V.A..

e : Entier, puht : R	newLC(prod : ProduitID, qt
	prod_lig(self : LigCom) : Produit
	puht_lig(self : LigCom) : R
	qt
	tva_lig(self : LigCom) : TVA
	set_prod_lig(self : LigCom, prod : ProduitID) : LigCom
	set_puht_lig(self : LigCom, puht : R
e_lig(self : LigCom, qt	set_qt
	set_tva(self : LigCom, tva : TVAID) : LigCom

6.2.3 Prise en compte de la facturation

La facturation engendre la création d'une facture pour le même client. Pour comprendre l'objet facture, il faut comprendre sa liaison avec la commande. Plusieurs cas sont possibles :

1. une commande donne lieu à une seule facture
   (a) une facture correspond à une seule commande : la facturation de la commande consiste à la passer à un nouvel état `facturé`. Facture et commande sont un même objet à des étapes de vie différentes..
   (b) une facture correspond à plusieurs commandes : la facture est un objet autonome dans lequel il y a plusieurs opérations de facturation, une par commande. Voici différentes présentations pour les lignes de la facture :

   Facture

n^o_com$_1$, n^o_lig_com$_1$1	n^o_com$_1$	n^o_com$_1$, ...n^o_com$_n$
	n^o_lig_com$_1$1	n^o_lig_fact$_1$
	⋮	
⋮	n^o_lig_com$_1 m$	⋮
	n^o_com$_2$	
	⋮	
n^o_com$_n$, n^o_lig_com$_n m$	n^o_lig_com$_n m$	n^o_lig_fact$_m$

   dans tous les cas, il faut vérifier que la somme des lignes d'une commande correspond à la somme de la commande.

2. une commande donne lieu à plusieurs factures
   (a) une facture correspond à une seule commande : c'est le dual du cas 1.b.
   (b) une facture correspond à plusieurs commandes : la facture est un objet autonome dans lequel il y a plusieurs opérations de facturation, une par commande. La commande elle-même comporte plusieurs opérations de facturation. Il existe des similarités avec le cas 1.b.

Lorsque facture et commande sont une même chose, alors la commande est modélisée comme dans la figure 24. La facture est l'évolution finale de la commande. On peut lui ajouter de nouveaux états : payé, en conflit... Lorsque la commande et la facture sont séparées, la prise en compte des frais et de la remise se fait dans la facture. Nous supposons, pour simplifier, qu'une commande est facturée en plusieurs fois mais qu'une facture correspond à une seule commande cas(2.a). Il n'a pas besoin alors de distinguer ligne de commande et ligne de facture[3].

[3] Un héritage permettrait alors de factoriser le comportement commun.

```
newfacture(com: CommandeID, d: Date, d: Date): Facture
client(self: Facture): Car
dateFacture(self: Facture): Date
dateCommande(self: Facture): Date
nbLignes(self: Facture): Entier
lignes(self: Facture): Liste[Ligcom]
ligne(self: Facture, n: Entier): Ligcom
ajouter_frais(self: Facture, frais: R
modifier_frais(self: Facture, frais: R
frais(self: Facture): R
fixer_remise(self: Facture, taux: R
modifier_remise(self: Facture, taux: R
remise(self: Facture): R
clôturer(self: Facture, tva: TVA): Facture
totalHT(self: Facture): R
totalTVA(self: Facture): R
totalTTC(self: Facture): R
imprimer(self: Facture): Facture
```

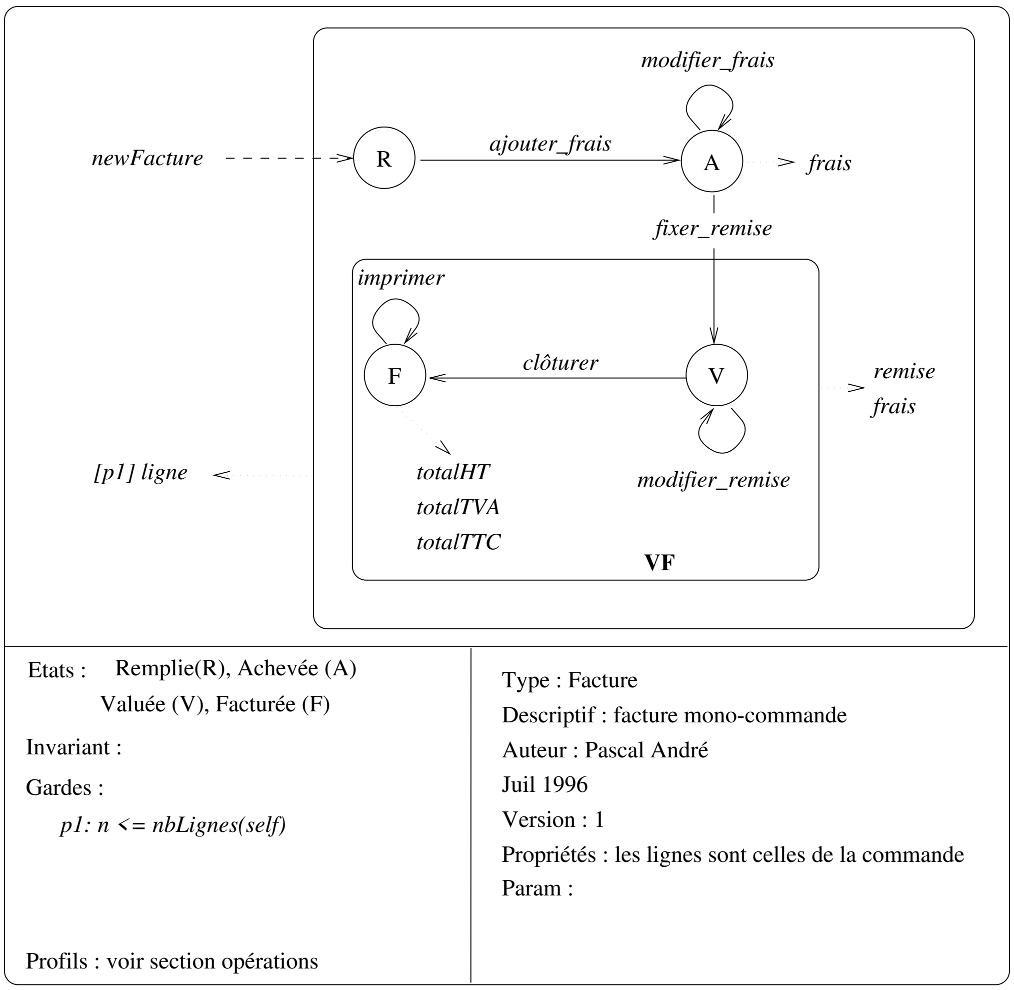


Figure 26 : L'automate du TAG `Facture`

6.2.4 Retour sur la commande

Cet automate, comme le précédent, est inspiré de celui de la commande dans la figure 24.

```
ouvrir(cli: Car, d: Date ): Commande
client(self: Commande): Car
dateCommande(self: Commande): Date
ajouter_ligne(self: Commande, lig: LigCom): Commande
nbLignes(self: Commande): Entier
lignes(self: Commande): Liste[LigCom]
ligne(self: Commande, n: Entier): LigCom
supprimer_ligne(self: Commande, n: Entier): Commande
modifier_ligne(self: Commande, n: Entier, lig: LigCom): Commande
totalHT(self: Commande): R
totalTVA(self: Commande): R
totalTTC(self: Commande): R
imprimer(self: Commande): Commande
facturer(self: Commande, f: FactureID): Commande
factures(self: Commande): Liste[Facture]
```

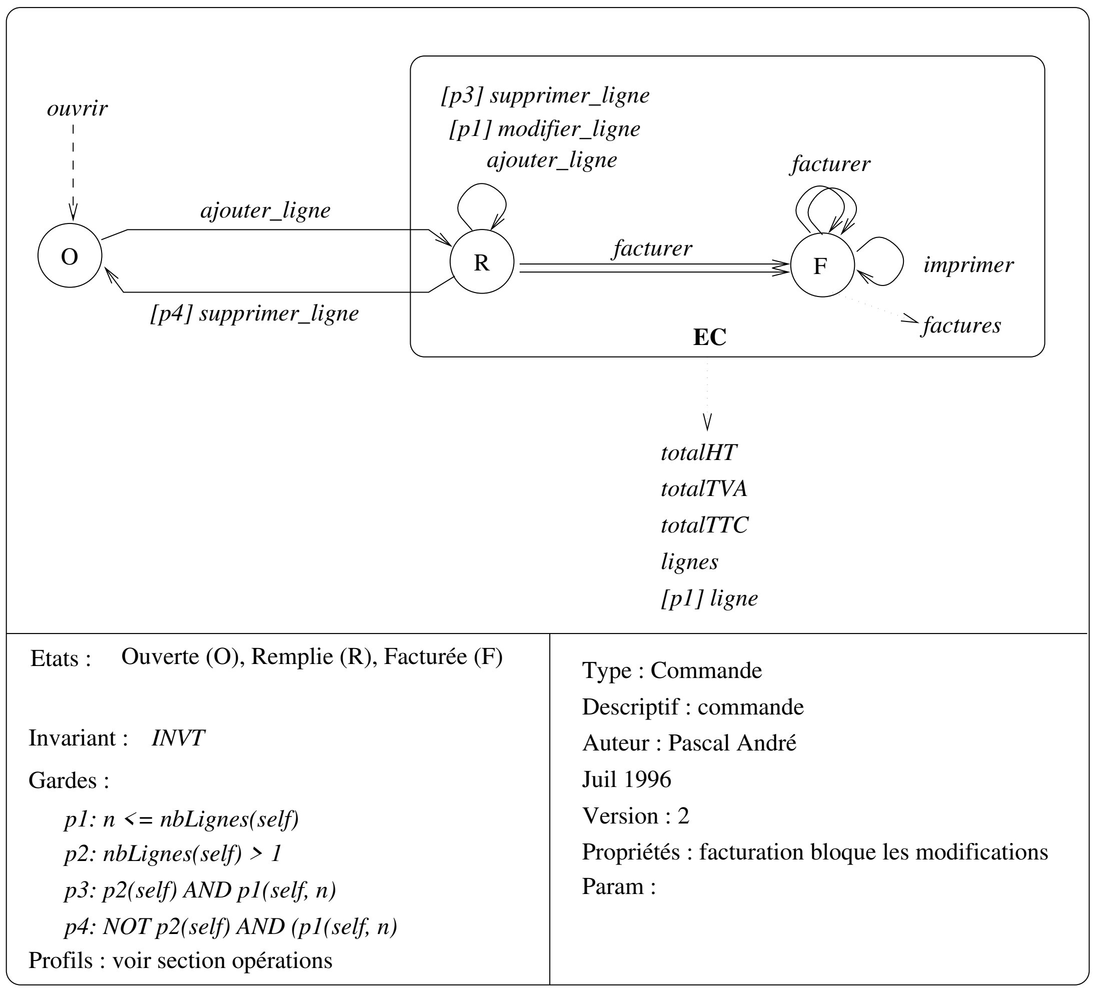


Figure 27 : L'automate du TAG `Commande` : version améliorée

Cette modélisation fait apparaître un lien entre deux objets : la facture et la commande. Ces deux objets pouvant évoluer indépendamment, pour connaître la valeur actuelle d'un objet, sa référence doit être un identifiant et non une valeur (i.e. un numéro de commande et de facture). Cette démarche est équivalente à celle qui modélise une association entre les deux objets. "Facturer une commande" se traduit alors par deux opérations synchrones : `facturer` (dans la commande) et `newFacture` (dans la facture). Une opération synchrone se traduit par deux envois de message (un pour la requête l'autre pour la réponse) et se note par une transition complexe (deux flèches).

6.3 Construction de la spécification algébrique du TAG

Seule cette phase fait de la spécification TAG une spécification formelle. Nous énonçons son principe et l'appliquons sur l'exemple de la figure 26. L'algorithme de construction de la spécification algébrique est détaillé dans[AR95]. Il est le suivant.

Etape 1 : construction de la signature Aux opérations du TAG sont ajoutés un observateur pour l'invariant, un observateur par état et un observateur par garde. Ces trois types d'opération ont pour résultat le type `Booléen`. Par exemple,`R(self : Facture)` est vrai si `self` se trouve dans l'état `R`. La signature d'un type abstrait algébrique comprend un ensemble `S` de sortes (de types manipulés) et un ensemble Σ de profils d'opérations. Dans notre exemple, la signature est la suivante :

een, Entier, Car, CommandeID, Date, Ligcom, Liste, R $S = \{$ `Facture, Bool`
$\Sigma = \{$ `newfacture : CommandeID Date` $\rightarrow$ `Facture`
`client : Facture` $\rightarrow$ `Car`
`dateFacture : Facture` $\rightarrow$ `Date`
`dateCommande : Facture` $\rightarrow$ `Date`
`nbLignes : Facture` $\rightarrow$ `Entier`
`lignes : Facture` $\rightarrow$ `Liste[Ligcom]`
`ligne : Facture Entier` $\rightarrow$ `Ligcom`
`ajouter_frais : Facture R`
`modifier_frais : Facture R`
`frais : Facture` $\rightarrow$ `R`
`fixer_remise : Facture R`
`modifier_remise : Facture R`
`remise : Facture` $\rightarrow$ `R`
`clôturer : Facture TVA` $\rightarrow$ `Facture`
`totalHT : Facture` $\rightarrow$ `R`
`totalTVA : Facture` $\rightarrow$ `R`
`totalTTC : Facture` $\rightarrow$ `R`
`imprimer : Facture` $\rightarrow$ `Facture`
`R : Facture` $\rightarrow$ `Bool` // etats //
`A : Facture` $\rightarrow$ `Bool`
`V : Facture` $\rightarrow$ `Bool`
`F : Facture` $\rightarrow$ `Bool`
`VF: Facture` $\rightarrow$ `Bool`
`p1 : Facture` $\rightarrow$ `Bool` // garde // $\}$

Etape 2 : extraction des générateurs Les opérations sont classées en opérations primitives et opérations secondaires. Cette partition permet de réduire le nombre d'axiomes de la spécification. Un parcours d'automate permet de calculer une famille de constructeurs primitifs appelés générateurs. L'extraction des générateurs est fondée sur l'idée que l'on doit pouvoir atteindre chaque état. Nous obtenons ici la famille : {`newFacture, ajouter_frais, fixer_remise, clôturer`}.

Etape 3 : calcul des préconditions Les pré-conditions des opérations sont exprimés en fonction des générateurs. La pré-condition d'une opération est une union des prédicats d'état et des gardes correspondantes. Les expressions sont alors simplifiées. La précondition des observateurs d'état est toujours vraie. Les variables utilisées dont déclarées pour l'ensemble des préconditions et des axiomes. Par exemple :

```
eel; taux, taux1 : R ∀ self : Facture; com : CommandeID; d : Date; n : Entier; frais, frais1 : R
      precondition(newFacture(com, d)) == true
      precondition(client(self)) == true
      precondition(dateFacture(self)) == true
      precondition(dateCommande(self)) == true
      precondition(lignes(self)) == true
      precondition(ligne(self, n)) == p1(self)
      precondition(ajouter_frais(self, frais)) == R(self)
      precondition(modifier_frais(self, frais)) == A(self)
      precondition(frais(self)) == A(self) OR VF(self)
      precondition(fixer_remise(self, taux)) == A(self)
      precondition(modifier_remise(self, taux)) == V(self)
      precondition(remise(self)) == VF(self)
      precondition(clôturer(self, tva)) == V(self)
      precondition(totalHT(self)) == F(self)
      precondition(totalTVA(self)) == F(self)
      precondition(totalTTC(self)) == F(self)
      precondition(imprimer(self)) == F(self)
```

Etape 4 : construction de l'axiomatique Nous avons défini des algorithmes pour construire un ensemble d'axiomes suffisamment complets. Cette construction est automatique pour les observateurs d'état, l'invariant et les gardes. Elle est assistée, en interaction avec l'utilisateur pour les autres opérations.

Commençons par la génération des axiomes des observateurs d'état et de gardes. Les axiomes de l'invariant et des gardes sont leur définition dans le TAG de la figure 26. Les axiomes d'un observateur d'état booléen sont définis sur chaque générateurs par disjonction des transtions entrantes supportées par le générateur en question. Dans notre exemple, nous avons les axiomes suivants :

```
p1(self, n) == leq(n, nbLignes(self))
R(newFacture(com, d)) == true
R(ajouter_frais(self, frais)) == false
R(fixer_remise(self, taux)) == false
R(clôturer(self, tva)) == false
A(newFacture(com, d)) == false
A(ajouter_frais(self, frais)) == true
A(fixer_remise(self, taux)) == false
A(clôturer(self, tva)) == false
V(newFacture(com, d)) == false
V(ajouter_frais(self, frais)) == false
V(fixer_remise(self, taux)) == A(self)
V(clôturer(self, tva)) == false
F(newFacture(com, d)) == false
F(ajouter_frais(self, frais)) == false
F(fixer_remise(self, taux)) == false
F(clôturer(self, tva)) == V(self)
VF(self) == V(self) OR F(self)
```

Examinons maintenant la construction assistée des axiomes pour les autres opérations. Nous nous sommes basés sur les méthodes usuelles de spécification de types abstraits de données [GH78, Mus80, LG90, Bid82]. Le principe de base de la génération des axiomes est appelé G-dérivation. Une G-dérivation d'un terme `self` dans un état `E` est sa substitution par un générateur arrivant dans l'état `E`. Connaître l'effet d'une opération sur ce terme `self` dans un état, c'est donc analyser toutes les transitions qui mènent à cet état. Les axiomes sont nommés.

Prenons l'exemple de l'observateur `remise`. Tel quel, on ne sait pas s'il s'agit de la remise effective (pourcentage du montant HT) ou du taux de remise. Puisque le montant HT est une opération interdite dans l'état `V` nous en déduisons qu'il s'agit du taux.

```
∀ self : Facture, taux : R
  VF(self) == true ==> remise(self) ==  ? ?
```

chaque cas est étudi e.

1. V(self)
   `V(self) == true ==> remise(self) ==  ? ?`
   les tra[illegible] erateurs
   on substitue `fixer_remise(self, taux)` a `self`
   la partie droite est alors trivialement egale a `taux`
   `remise1:  A(self) == true ==> remise(fixer_remise(self, taux)) == taux`
2. F(self)
   `F(self) == true ==>  remise(self) ==  ? ?`
   les tra[illegible] erateurs
   `remise2:  V(self) == true ==> remise(clôturer(self, tva)) == remise(self)`

Nous procédons ainsi pour toutes les opérations non générateur que nous définissons à partir des générateurs. où `longueur` est une opération du type `Liste`. Nous donnons maintenant les axiomes pour quelques exemples représentatifs : les observateurs `lignes` et `nbLignes` définis dans tous les états, l'observateur `modifier_remise` défini dans l'état `V`, les observateurs `totalHT`, `totalTVA`, `totalTTC` et le constructeur `imprimer`, définis dans l'état `F`.

Nous utilisons les opérations suivantes supposées définies dans le type `Liste[LigCom]`. L'opération `longueur(l: Liste[LigCom]) : Entier` fournit la longueur de la liste `l` donnée en paramètre. L'opération `somme_tva(l: Liste[LigCom]) : Réel` (resp. `somme_mht(l: Liste[LigCom]) : Entier`) fournit la somme des TVA (resp. des montants hors-taxe) d'une liste de lignes de commandes. La T.V.A. de la facture connue dans `clôturer` s'applique aux frais. La remise s'applique après le calcul de la T.V.A.. Ce mode de calcul n'est pas celui appliqué dans bon nombre de sociétés commerciales...

```
lignes1:  lignes(newFacture(com, d)) == lignes(com)
lignes1:  lignes(ajouter_frais(self, frais)) == lignes(self)
lignes1:  lignes(fixer_remise(self, taux)) == lignes(self)
lignes1:  lignes(clôturer(self, tva)) == lignes(self)
nbLignes1:  nbLignes(self) == longueur(lignes(self))
modifier_remise1:  A(self) == true ==>
    modifier_remise(fixer_remise(self,taux1), taux) == fixer_remise(self,taux)
totalHT1:  F(self) == true ==> totalHT(self) ==
    mult(add(somme_mht(lignes(self)),frais(self)),sub(1,remise(self)))
```

  // $((\Sigma_{lc \in lignes(self)} puht\_lig(lc) * qt$ //

```
totalTVA1:  V(self) == true ==> totalTVA(clôturer(self, tva)) ==
    mult(add(somme_tva(lignes(self)), mult(frais(self),taux(tva))),sub(1,remise(self)))
```

  // $((\Sigma_{lc \in lignes(self)} taux(tva\_lig(lc)) * puht\_lig(lc) * qt$
    $*(1 - remise(self))$ //

```
totalTTC1:  F(self) == true ==> totalTTC(self) ==
    add(totalHT(self)), totalTVA(self))
imprimer1:  F(self) == true ==>  imprimer(self) == self
    // les effets de bord ne sont pas exprim
```

De bonne propriétés de cohérence et complétude de la spécification sont apportées par cette méthode (voir [And95, AR95] pour une démonstration).

6.4 Modélisation avec les Classes Formelles

Puisqu'il s'agit d'un exercice de modélisation et non de conception, nous donnons une version intuitive des classes formelles et non une conception des TAG en CF. Ceci permet de mettre en valeur les questions propres au modèle des classes formelles. La classe modélisée est celle de la facture (figure 26). L'aspect reprend l'ensemble des paramètres des générateurs. L'état est représenté par des sélecteurs de champs booléens qui conditionnent les sélecteurs de champs `frais`, `remise` et `tva`. Une autre représentation par valeurs nulle (`nil` en C) est moins rigoureuse.

```
                              Facture
                         inherits from OBJECT

 comments: class for Facture

 features: newfacture, client, dateFacture, dateCommande, lignes ...

 from: Real  import: +, 0, 1, =, >, <=

 from: Boolean  import:
---------------------------------------------------------------------
                          aspect : Facture
field selectors
 commande      : Facture ⟶ CommandeID
 dateFacture   : Facture ⟶ Date
 aFrais        : Facture ⟶ Boolean
 frais         : Facture ⟶ Real
   requires: aFrais(self)
 aRemise       : Facture ⟶ Boolean
 remise        : Facture ⟶ Real
   requires: aRemise(self)
 clos          : Facture ⟶ Booléan
 tva           : Facture ⟶ TVA
   requires: clos(self)
---------------------------------------------------------------------
                            extensions
ajouter_frais : Facture Real ⟶ Facture
;; ajouter_frais : ajoute des frais sur la facture
var: frais : Real
 aFrais(self) == false ==> ajouter_frais(self,frais) == copy(self, aFrais = true, frais = frais)

fixer_remise : Facture Real ⟶ Facture
;; fixer_remise : fixe la remise client sur la facture
var: taux : Real
 aRemise(self) == false ==> fixer_remise(self,taux) == copy(self, frais = taux, aRemise = true)

clôturer : Facture TVA ⟶ Facture
;; clôturer : clos la facture
var: tva : TVA
 aRemise(self) == true ==> clôturer(self, tva) == copy(self, clos = true, tva = tva)

 ... autres extensions ...
---------------------------------------------------------------------
                           class methods
newFacture : CommandeID Date ⟶ Facture
;; newFacture : initialise la facture
var: com : CommandeID, d : Date
 newFacture(com, d) == new(Facture, commande = com, dateFacture = d,
 aFrais = false, aRemise = false, clos = false)
```

Figure 28 : La classe formelle `Facture`

6.5 Remarques

Un des atouts de ces deux modèles est leur sémantique algébrique commune. Ceci se traduit par l'écriture d'outils communs et une vérification de correction du raffinement des TAG en classes formelles, lorsqu'une description fonctionnelle a été calculée. L'évaluation symbolique et la preuve de propriétés se font par un mécanisme simple et naturel de réécriture. La démonstration de ces propriétés et la validation de la spécification sont faites via des interfaces avec d'autres environnements de spécification. Les preuves sur les automates ne sont que partiellement abordées dans [And95]. Un contrôle de type fort a été défini pour les classes formelles [CCR95] et ses principes sont applicables également aux TAG.

Un des objectifs de ces modèles est de définir des modèles utilisables en pratique. Les efforts portent tant sur les concepts que leur intégration dans un cycle de développement plus général. Une expérimentation d'utilisation d'OMT pour la partie analyse et de la méthode TAG/CF pour la partie spécification formelle et conception abstraite est relatée dans [ABR95]. Les lacunes actuelles sont une trop grande orientation composant (définition formelle, raffinement, réutilisation) au détriment de la conception de systèmes. Les modèles restent assez pauvres sémantiquement, des extensions sont prévues à la concurrence. Les aspects typage et preuve sont toujours à l'étude.

Malgré tout, la spécification TAG met bien en valeur les problèmes de modularité dans les spécifications, de communications entre modules et d'opération partiellement définies. Elle met bien en valeur à quel moment telle ou telle opération est applicable. Contrairement à OMT, elle renforce la sémantique donnée aux opérations et les liens entre opérations, Par exemple, la modélisation TAG nous a obligé a préciser l'ordre d'application de la remise, des frais et de la T.V.A., ordre qui est important pour le calcul du montant total toutes taxes comprises.

L'application "facturation" est de type mémorisation de données et contrôle global centralisé. Ce style de spécification ne correspond pas vraiment à la modélisation en TAG, plus adapté à une modélisation en termes d'objets coopérants et de contrôle réparti. C'est pourquoi les choix de modélisation et le résultat ne sont pas très probants. Notons tout de même une meilleure description des objets que celle d'OMT.

Le modèle est construit à partir d'états abstraits des objets : une idée du cycle de vie des objets. La description précise des observateurs du TAG indépendants des états permet de définir les états comme une fonction de ces observateurs et ainsi de simplifier la description algébrique. Par exemple, la distinction entre l'état `ES` et `R` du TAG `Produit` de la figure 25 s'écrit simplement :

```
eStock(self) ¿= qt ES(self) == qt
  eStock(self) ¡ qt R(self) == qt
```

7 Conclusions

Ce document a tenté de montrer brièvement l'esprit qui anime l'analyste. Le travail est limité à un cadre d'étude. Nous avons donné une modélisation en insistant sur les étapes nécessitant des choix de modélisation, et les conséquences de ces choix. Le ton est donc pédagogique et non pas d'un document de référence produit par l'analyste. Beaucoup de concepts sont supposés connus et/ou assimilés. Pour les concepts objets, consulter [MNC$^+$90, Mey88]. Il n'est pas fait mention de la conception du programme, qui constitue une étape ultérieure à l'analyse.

Nous avons donné quatre modélisations du cas de la facture. Chacune apporte sa contribution à l'éclaircissement de l'énoncé en soulevant des problèmes spécifiques. Un premier enseignement concerne le processus de modélisation. Une bonne façon de procéder est donc de réécrire l'énoncé (le cahier des charges) à la lumière des premières modélisation.

Les modèles varient en fonction du formalisme choisi (plus ou moins formel, paradigmes associés). Nous retrouvons certains problèmes quel que soit le formalisme :

- Les choix de gestion : ce sont des difficultés propres au problème (multi-facturation, ...).
- Le choix et la représentation de la séparation entre valeurs et identifiant lorsqu'il existe des relations entre "objets" indépendants.
- La réalisation des traitements. Le mode de calcul est unique et sa réalisation est plus ou moins modulaire selon la méthode choisie.

Un avantage de la modélisation à objets est que la modification d'un objet entraîne peu de modification sur les autres objets qui l'utilisent. Cela est vrai pour l'héritage et lorsque les interfaces entre objets sont bien définies. Nous nous en sommes plus rendu compte dans la modélisation TAG que dans celle d'OMT parce que les objets sont décrits plus finement. Les choix de modélisation ont moins de répercussion dans un système à objet que dans une modélisation globale à la Merise ou à la Z.

D'autres points de la modélisation sont mieux mis en valeur dans certaines méthodes que dans d'autres.

- Une séparation des données et des traitements est faite en Merise, OMT et Z.
- Un état global est modélisé en Merise, OMT et Z.
- Les traitement sont perçus globalement en Merise et en Z tandis que l'approche est modulaire en OMT et TAG.
- La description est précise en Z et TAG, informelle en Merise et OMT. La preuve de propriétés de la spécification dépend de l'existence d'une sémantique formelle.
- Les enchaînements d'opérations sont étudiés en Merise, OMT et TAG mais sous des angles différents (causalité globale en Merise), flot de contrôle local en OMT et TAG.
- La communication est abordée en OMT et TAG mais pas dans Merise et Z.
- La spécification est graphique en Merise, OMT et TAG, ce qui en facilite la lecture et la communication aux autres acteurs du développement.

La variété des formalismes fait naître des questions différentes mais leur intégration globale est rarement facile parce que nous restons à un niveau de description relativement abstrait. Nous traitons des points particuliers qui varient d'une méthode à l'autre. Ainsi, Merise nous a permis de bien dégrossir les problèmes de règles de gestion : multi-facturation, ... La méthode met en valeur l'organisation cohérente et non redondante des données. Elle traite globalement le problème, contrairement aux autres méthodes expérimentées ici. OMT pose plus de problèmes de modélisation d'une part parce que les concepts utilisables sont plus nombreux et plus riches et d'autre part parce qu'il faut donner des responsabilités aux objets tant pour les données que pour les traitements. Ce qui n'a rien de trivial dans le type d'application traité ici. Les TAGs nous ont permis de réfléchir précisément sur les comportement des objets et leurs relations de clientèle avec d'autres objets. Préciser la sémantique des méthodes permet de savoir si les informations sont disponibles ou pas.

La démarche varie d'une méthode à l'autre. Merise et OMT ont une approche descendante tandis que Z et TAG ont une approche ascendante. La démarche descendante facilite la compréhension globale du système mais reste souvent imprécise et ambiguë. A l'inverse, la démarche ascendante est précise et correcte mais l'analyste risque de surspécifier le système.

Le résultat de ce travail reste partiel et de nombreuses vérifications et corrections sont encore à faire surtout pour Merise et OMT. Il faudra alors la faire valider par le commanditaire et vérifier la cohérence des modèles par des relecteurs. Une voie intéressante est la formalisation du problème en Z ou TAG à partir des descriptions Merise ou OMT. Tout ceci dépasse le cadre fixé ici.

Références

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

[Abr96] Jean-Raymond Abrial. *The B-Book Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AG91] Antonio J. Alencar and Joseph A. Goguen. OOZE: An Object-Oriented Z Environment. In Pierre America, editor, *Proceedings of ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 180–199, Geneva, Switzerland, 1991. Springer Verlag.

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

[And96] Pascal André. Traduction du Formalisme Entité/Association en Z. Rapport personnel, MSF, Université de Nantes, mars 1996.

[AR95] 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 RR-98, IRIN, octobre 1995. (53 pages).

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

[Bre91] Ruth Breu. *Algebraic Specification Techniques in Object-Oriented Programming Environments*, volume 562 of *Lecture Notes in Computer Science*. Springer Verlag, June 1991. Thesis, Universität Passau, Germany.

[CCR95] Corina Cîrstea, Dan Chiorean, and Jean-Claude Royer. Type checking formal classes: Principles and algorithms. Rapport de Recherche RR-99, IRIN, octobre 1995. (29 pages).

[DD90] D. Duke and Roger Duke. Towards a Semantics for Object-Z. In D. Bjorner, C.A.R. Hoare, and H. Langsmaack, editors, *Proceedings of VDM'90*, pages 244–261. Springer Verlag, 1990.

[Der94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes and Paul Jeremaes. *Object-Oriented Development* THE FUSION METHOD. Object-Oriented Series. Prentice Hall, 1994.

[GH78] J. V. Guttag and J. J. Horning. The algebraic Specification of Abstract Data Types. *Acta informatica*, 10: 27–52, 1978.

[GS96] Andy Galloway and Bill Stoddart. Integrated Formal Methods. Rapport de Recherche RR-128, IRIN, juillet 1996. (26 pages).

[Har87] David Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming 8*, pages 231–274, 1987.

[Hay92] Ian Hayes, editor. *Specification Case Studies*. C.A.R Hoare Series. Prentice-Hall International, 2 edition, 1992.

[JCJO91] Ivar Jacobson, Magnus Christensen, Patrik Jonsson, and Gunmar Overgaard. *Object-Oriented Software Engineering : A Use Case Driven Approach.* Prentice Hall, 1991.

[Jon93] Cliff B Jones. *VDM, une méthode rigoureuse pour le développement du logiciel.* Masson, Paris, 2e edition, 1993.

[Lan94] Kevin Lano. Specification of Real Time Systems in Object-Oriented Formal Methods. Technical report, Imperial College, August 1994.

[Lan96] Kevin Lano. *The B Language and Method: A Guide to Practical.* Springer Verlag London, 1996.

[LG90] Barbara Liskov and John Guttag. *La Maîtrise du Développement de Logiciel : abstraction et spécification.* Collection "Ingénierie des systèmes d'information". Les Editions d'Organisation, 1990.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall International, 1988.

[MNC+90] Gérald Masini, Amedeo Napoli, D. Colnet, D. Léonard, and Karl Tombre. *Les langages à objets.* Collection IIA. InterEditions, 1990.

[Mus80] 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, January 1980.

[RB93] Arnold Rochfeld and Mokrane Bouzeghoub. From Merise to OOM. *Ingénierie des systèmes d'information*, 1(2): 151–176, February 1993.

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

[SBC92] Susan Stepney, Rosalind Barden, and David (Eds) Cooper, editors. *Object Orientation in Z.* Springer Verlag, 1992.

[Smi93] Graeme Smith. A Fully-Abstract Semantics of Classes for Object-Z. *Formal Aspects of Computing*, 3(1), 1993. BCS.

[Som92] Ian Sommerville. *Le génie logiciel.* Addison-Wesley France, 4e edition, October 1992.

[Spi89] Mike Spivey. *The Z notation: a Reference Manual.* C.A.R Hoare Series. Prentice-Hall International, 1989.

[Spi90] Mike Spivey. A guide to the `zed` style option, 1990. Free LaTeXstyle file - ftp `archive-management@comlab.ox.ac.uk`.

[TRC83] Hubert Tardieu, Arnold Rochfeld, and R. Coletti. *La méthode Merise.* Editions d'Organisation, 1983. tomes 1, 2 et 3.

[Vai95] Alain Vailly. Les diagrammes de flot de données (essai de synthèse). Rapport de Recherche RR-103, IRIN, Nouvembre 1995. (45 pages).

[WL88] J. Woodcock and M. Loomes. *Software Engineering Mathematics.* Addison Wesley, 1988.

Table des matières

Liste des figures