UNIVERSITE JOSEPH FOURIER-GRENOBLE 1
SCIENCES - GEOGRAPHIE

THESE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER

Discipline : Informatique

Présentée et soutenue publiquement

par

Christophe Saint-Marcel

Le 8 décembre 1999

TRAÇABILITE & REUTILISATION DES SPECIFICATIONS COMPORTEMENTALES D'OBJET

LE MODELE NCR

Directeurs : M. Philippe Morat & M. Dominique Rieu

Composition du jury

Colette Rolland (Rapporteur)

Michel Léonard

Jean-Pierre Giraudin

Bernard Espinasse (Rapporteur)

Pierre-Yves Cunin

Thèse préparée au sein du

LABORATOIRE LOGICIELS, SYSTEMES ET RESEAUX - IMAG

À Fusion,
avec toute ma sympathie.

Toute ressemblance avec des objets existants, ayant vécu où à venir, enfin je l'espère, est volontaire.

L'auteur

Introduction

> Un patron est une règle tripartite qui exprime une relation entre un certain contexte, un certain problème qui apparaît répétitivement dans ce contexte et une certaine configuration logicielle qui permet la résolution de ce problème.
>
> R. Gabriel [Appleton97]

Les patrons peuvent être vus comme une simple manière de présenter et de structurer des connaissances d'ingénierie. Cette approche oblige pourtant le lecteur à raisonner en termes de problème et pas uniquement en termes de solutions. Nous pensons que l'élaboration d'une thèse suit le même cheminement de pensée, de la définition du problème à la recherche d'une solution qui soit si possible la meilleure. Il est donc intéressant de présenter la thématique de ce mémoire à la manière d'un patron. Pour cela, nous avons retenu quatre champs qui définissent le travail accompli et structurent ce mémoire : contexte, motivation, problème et solution.

- **Contexte**

  Cette thèse est réalisée dans le contexte des méthodes de conception orientée objet.

- **Motivation**

  Le problème est illustré par un exemple de modélisation concernant le système d'information d'une bibliothèque. Cet exemple est affiné dans l'ensemble de l'ouvrage par la définition de modèles de spécification de la dynamique et plus particulièrement des comportements d'objets du système.

- **Problème**

  Lors de la modélisation de systèmes complexes, le concepteur est souvent confronté à la difficulté de spécifier puis d'implanter le comportement individuel des objets du système.

- **Solution**

  La solution adoptée conduit à un équilibre entre les spécifications structurelles d'une part et dynamiques d'autre part. Cette nouvelle structuration doit augmenter de façon significative la traçabilité du processus de conception et la réutilisation de comportements dans les applications.

Nous détaillons ci-dessous ces quatre éléments caractéristiques de notre travail.

Contexte

De nombreux modèles et notations ont été introduits ces dernières années pour répondre aux besoins divers de l'ingénierie des systèmes. La plupart des méthodes de conception, qu'elles soient issues du génie logiciel (OOD, OMT, MECANO, etc), des systèmes d'information (MERISE, REMORA, etc) ou du génie cognitif (KADS) proposent des représentations structurelles et dynamiques du système. Ces méthodes ont choisi la notion d'objets pour regrouper structures de données et traitements. Les diagrammes de type Entité/Relation [Chen76] se sont rapidement imposés comme les piliers de la modélisation structurelle des objets et du système. Le besoin de modéliser le système d'information selon différents éclairages a donné jour à de nombreux modèles (statiques, dynamiques et fonctionnels) qui ont complété ces diagrammes E/R (OMT, MERISE, etc). Aujourd'hui un consensus s'est dégagé sous la forme d'un unique langage de modélisation [UML97]. Celui-ci reprend l'ensemble des modèles découverts ces dernières années et qui ont depuis fait leur preuve. Différents auteurs soulignent pourtant que ces modèles ne constituent pas toujours une solution convaincante pour faire coexister les dimensions statique et dynamique [Vayda95]. Des efforts restent donc à faire pour assurer la cohérence globale de ces modèles. Dans ce mémoire, nous nous proposons d'étudier plus particulièrement l'intégration des diagrammes de classes et des diagrammes d'états.

Les diagrammes de classes sont utilisés comme les "colonnes vertébrales" (backbone) de la modélisation des systèmes d'information. Leur sémantique est souvent étendue par les autres modèles. Ces diagrammes décrivent les structures de données sur lesquelles on va opérer. La dimension dynamique décrit la structure de contrôle des objets. Les diagrammes d'états ont été utilisés pratiquement depuis le début de la modélisation objet pour formaliser ces aspects. L'idée de base étant la définition d'une machine qui a un nombre fini d'états. Celle-ci reçoit des événements de l'extérieur, chaque événement pouvant causer une transition d'un état à un autre. Les diagrammes d'états ont l'avantage de donner une définition explicite et formelle d'un comportement d'objet.

Les méthodes orientées objets proposent des versions plus ou moins complexes des diagrammes d'états. Shlaer & Mellor [Shlaer92] utilisent des diagrammes d'états à plat dits de Moore. Les statecharts, dernière génération de diagrammes d'états introduite par D. Harel [Harel87] réduisent l'explosion combinatoire inhérente à ce type de représentation grâce à la généralisation et à la concurrence d'états. Ils offrent un haut niveau d'expression mais en contrepartie leur implantation et leur utilisation reste délicate. Les statecharts sont adaptés pour la spécification de

systèmes réactifs mais leur emploi pour la spécification des systèmes d'information est peu abordé [Legrand97].

L'un de nos objectifs est donc de favoriser l'usage des statecharts dans les méthodes de conception pour une meilleure prise en compte des comportements des objets. L'accent est mis sur la gestion de comportements complexes d'objets nécessitant en particulier la prise en compte d'évolutions parallèles.

Motivation

Nos propositions sont illustrée à l'aide d'exemples tirés d'une application de gestion de bibliothèques. Nous décrivons en langue naturelle une partie de cette application afin de nous familiariser avec le système complet. Cette description est ensuite associée à un diagramme de classes (figure 1) qui présente la structure informatique d'un réseau de bibliothèques. Cette description est ensuite raffinée tout au long de l'état de l'art par l'ajout de nouveaux modèles et de nouvelles spécifications dynamiques qui se prêtent à notre discussion.

Le système d'information considéré comprend un réseau de bibliothèques municipal. Les bibliothèques se partagent les ouvrages : un système de distribution quotidien est mis en place en fonction des demandes formulées dans chaque bibliothèque. Trois processus caractérisent l'ensemble des bibliothèques :

- La gestion des prêts. Celle-ci englobe trois activités distinctes : la gestion des abonnés, des emprunts et des réservations d'ouvrages. Tout abonnement est valable une année et pour toutes les bibliothèques de la ville. Le prêt est limité à trois ouvrages et ce pour deux semaines. Au-delà, l'abonné est sanctionné et peut faire l'objet de relances par écrit. Chaque abonné peut réserver au plus deux ouvrages qui sont soient déjà sortis, soient disponibles dans une autre bibliothèque de la ville. La demande des abonnés peut être satisfaite de deux manières, soit en consultant un fichier central de toutes les bibliothèques de la ville, soit en essayant de commander l'ouvrage directement auprès des éditeurs.
- La maintenance des ouvrages. Chaque exemplaire doit être examiné au moins une fois dans l'année.
- L'acquisition de nouveaux livres. Les bibliothèques peuvent commander les livres les plus récents auprès des éditeurs avec lesquels elles sont directement en relation. Elles doivent aussi gérer les dons gracieux réalisés par des particuliers. Un tri des ouvrages est réalisé dans ce cas par les services qui s'occupent de la maintenance.

Le diagramme de classes ci-dessous forme un référentiel pour chacun des exemples introduits par la suite. Il fait apparaître deux classes centrales dans l'application, Ouvrage et Livre. La première représente l'ensemble des documents qui sont référencés par toutes les bibliothèques. La seconde représente l'ensemble des exemplaires qui peuvent être empruntés par les abonnés d'une des bibliothèques. L'emprunt et le retour de chaque livre sont comptabilisés au niveau de l'ouvrage afin de savoir à tout moment s'il reste des exemplaires disponibles dans une des bibliothèques. Un ouvrage peut être réservé ; La réservation est effective lors du retour d'un livre dans une des bibliothèques de la ville.

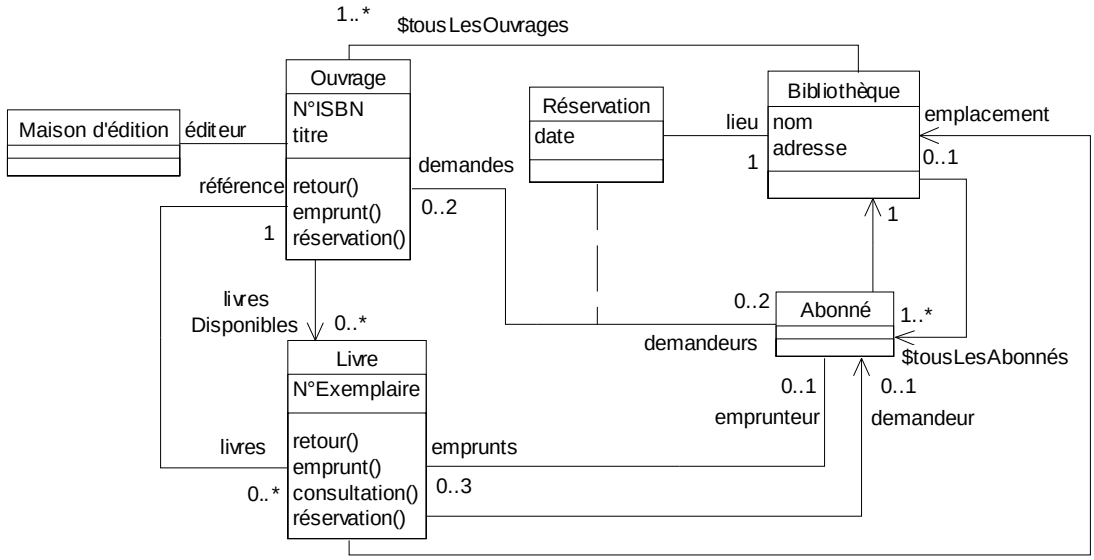


figure 1:Structure de l'application de gestion de bibliothèques

Chaque bibliothèque connaît l'ensemble des abonnés de la ville (**$tousLesAbonnés**) et l'ensemble des **livresDisponibles** correspondants aux ouvrages (**$tousLesOuvrages**) référencés par toutes les bibliothèques.

Problème

La plupart du temps l'expression de comportements d'objets à l'aide de statecharts reste marginale dans le processus de développement. Plusieurs raisons peuvent être invoquées :

- Les spécifications de statecharts sont réservées à des spécialistes. Son importance est souvent peu reconnue. Leur spécification est un exercice difficile à cause de la granularité choisie. En effet, la plupart des concepteurs trouvent souvent plus simple et plus naturel d'exprimer la dynamique globale du système (modèle de traitements de Merise, diagrammes de séquence et de collaboration d'UML) dont ils ont une vision plus intuitive. Cette difficulté est accrue pour

les objets dont l'évolution est complexe et ce malgré qu'ils soient les candidats à privilégier dans l'analyse et la conception du système d'information.

- Les spécifications de statecharts sont peu réutilisables. Un statechart décrit le comportement des objets d'une classe. Ce modèle est donc de fait très spécifique. Il n'apporte pas le niveau de généricité nécessaire à la description de comportements de plusieurs types d'objets. Alors qu'aujourd'hui, on se tourne vers l'usage de patrons et de composants réutilisables, cette limite compromet fortement leur utilisation dans les futures méthodes de conception. De plus, les règles de sous-typage des statecharts, quant elles sont définies, sont souvent draconiennes et ne facilitent pas leur extension.
- Les spécifications des statecharts sont peu exploitables. Alors que les systèmes réactifs ont su parfaitement intégrer ce modèle et l'ont rendu exécutable[1], on ne trouve pas d'équivalent dans les systèmes d'information. L'intégration des dimensions statique et dynamique souvent jugée difficile est repoussée au moment de l'implantation. Elle est alors réalisée de manière implicite sans réelle conceptualisation en amont du codage. Les statecharts sont donc plus utilisés comme outils de documentation pour la conception des objets du système que comme un véritable modèle implantable.

Solution

Notre objectif n'est pas d'introduire un nouveau formalisme de plus à la littérature abondante des méthodes de conception à objet (la première moitié des années 90 a vu fleurir une cinquantaine de méthodes objet [Muller97]) mais plutôt de faciliter la coexistence, la cohérence et la réutilisation des modèles existants. Nos travaux concernant les aspects comportementaux des méthodes de conception se proposent de :

- Décrire les statecharts dans un modèle à objets. Il s'agit ici de coupler l'information détenue par les classes du système à celle obtenue des statecharts ; ces deux concepts offrant une vue complémentaire de l'objet. Nous pensons que leur intégration peut être envisagée de plusieurs manières en brisant le postulat : « une classe, un statechart ». Nous devons pour cela définir les règles qui permettent leur intégration.
- Offrir un modèle de spécification naturel et flexible. Nous voulons avant tout favoriser l'utilisation des statecharts dans les modèles à objets. Or, le formalisme des statecharts tel

[1] De nombreux outils existent sur le marché des systèmes embarqués : BetterState développé par la société Integrated Systems (http://www.isi.com), Baan VisualSTATE (http://www.visualstate.com/products/visual.htm), Statemate et Rhapsody deux outils proposés par I-Logix (http://www.ilogix.com)

qu'il est défini aujourd'hui offre une vision rigide de la dynamique des objets. Nous nous orientons au contraire vers des définitions comportementales plus souples (on renoncera en particulier au déterminisme strict qui est habituellement de rigueur lors de l'utilisation des statecharts).

- Favoriser la réutilisation du comportement d'objet par l'amélioration des tâches de conception et de codage. L'une des voies la plus pertinente aujourd'hui semble celle des patrons d'ingénierie : patrons de conception (design pattern) [Gamma95] [Coad92], patrons de métier (ou objets de métier), etc. Cependant celle-ci ne s'intéresse pas ou peu à la réutilisation de spécifications dynamiques, ceci pour les raisons déjà mentionnées : difficulté de conception, d'implantation et d'intégration des diagrammes d'états. Nous proposons donc une spécification comportementale de composants abstraits qui peuvent être utilisés et réutilisés dans plusieurs contextes.
- Garantir une traçabilité des spécifications. Le suivi de la conception est un enjeu majeur pour la maintenance des logiciels. Quelques approches ont tenté de rationaliser la production de code dynamique (le patron état de E. Gamma [Gamma95] et ses extensions) mais ces traductions offrent peu de souplesse, le code généré perdant souvent en lisibilité. L'effort de spécification que nous nous proposons de réaliser au niveau conceptuel doit permettre une automatisation de la génération du code.

ORGANISATION DU MEMOIRE

CHAPITRE 1 - MODELISATION DES EVOLUTIONS D'OBJETS : LES STATECHARTS **9**

1- Les statecharts, genèse d'un standard pour la modélisation dynamique **13**

Présentation rapide des modèles dynamiques utilisés couramment pour spécifier le comportement individuel des objets. Présentation du formalisme des statecharts.

2- statecharts activistes vs. classificatoires **39**

Etude de l'utilisation des statecharts dans le contexte particulier des modèles à objets. Définition de deux courants complémentaires.

- **mots clé** : Comportement individuel d'objet, statecharts, conformité comportementale.

CHAPITRE 2 - SUIVI DES EVOLUTIONS D'OBJETS : LES ROLES **53**

1- Méthodes de conception « centrées comportement », introduction aux rôles **57**

Description générique des nouvelles méthodes de conception « centrées comportement ». Aperçu des techniques et modèles de réutilisation utilisées dans ces méthodes. Un concept émergeant et fédérateur pour de nombreuses méthodes : le rôle.

2- Diversité comportementale d'objet et patrons d'ingénierie **71**

Discussion sur les concepts de rôle et d'état étayée par de nombreux patrons.

- **mots clé** : Méthode de conception dirigée par le comportement, rôle, patron.

CHAPITRE 1

Modélisation des évolutions d'objets : les statecharts

« Ceci n'est pas une thèse »

Les systèmes d'information sont la plupart du temps constitués d'objets dont l'évolution et le comportement sont difficiles à spécifier car changeant au cours du temps. Il s'agit souvent d'objets dont l'état et de manière plus générale l'évolution sont étroitement liés à l'exécution des processus métiers du système. C'est ainsi que l'évolution d'une commande est liée à l'exécution du processus de gestion des commandes, celle d'un livre au processus de gestion des prêts, etc. L'objet est alors soumis à une évolution lui permettant de passer d'un état à l'autre. Son comportement est lié au processus dans lequel il évolue : dans une gestion de prêt un livre ne peut réagir qu'à une demande d'emprunt ou de retour. De plus son comportement dépend de son état courant : un livre dans l'état emprunté peut prendre en compte un retour mais pas une demande d'emprunt. C'est à ce type de comportement individuel d'objet que nous nous adressons dans ce mémoire. Pour le représenter, les méthodes de conception ont introduit plusieurs modèles et formalismes dont nous faisons état (cf. § 1.1). Parmi ceux-ci, nous choisissons le formalisme graphique des statecharts comme étant le plus adéquat pour nos travaux. La partie 1.2 de ce chapitre décrit les concepts importants qui font le succès des statecharts pour la représentation graphique de machines à états finis. Nous présentons ensuite la notation complète (cf. § 1.1.3) telle qu'elle est décrite dans le langage de modélisation unifié (UML).

Nous nous intéressons à l'utilisation spécifique des statecharts pour la conception de systèmes à objets (cf. § 2). L'accent est mis sur les potentialités d'utilisation et surtout de réutilisation des statecharts au travers de leurs différents usages. L'idée de réutiliser des spécifications comportementales est née d'un simple constat : dans une application nous retrouvons toujours des comportements généraux qui peuvent être appliqués à plusieurs types d'objets. Certaines évolutions sont suffisamment standardisées pour envisager leur capitalisation : un livre est un exemple de ressource à la fois pour le lecteur et pour la bibliothèque au même titre qu'un homme est une ressource au sein d'une entreprise.

1. LES STATECHARTS, GENESE D'UN STANDARD POUR LA MODELISATION DE LA DYNAMIQUE

Les méthodes de conception à objets utilisent différents modèles pour spécifier le comportement des objets : les diagrammes de transitions d'états [Jacobson93, Rumbaugh95, Coad92, Shlaer92, etc], des expressions logiques [Arapis90], des réseaux de Petri [Léonard91, Kappel91, Embley92]. Ces dernières années, les statecharts se sont largement imposés dans la communauté objet et sont aujourd'hui reconnus comme partie intégrante du standard de modélisation [UML97]. Notre travail est bâti sur l'hypothèse que les statecharts constituent aujourd'hui de par leur richesse et de par leur flexibilité la voie la plus prometteuse pour spécifier puis implanter de telles évolutions dans les systèmes d'information. Dans un premier temps (cf. § 1.1), nous justifions ce choix par une comparaison entre les statecharts et les solutions à base d'assertions, de réseaux de Petri et de diagrammes d'états généralement présentes dans la littérature pour décrire les comportements individuels d'objets.

Nous introduisons ensuite l'ensemble des concepts qui constituent le formalisme des statecharts tels qu'il est défini dans le langage de modélisation unifié (UML). Nous présentons de manière synthétique leurs fondements dans le paragraphe 1.2 puis le formalisme complet dans le paragraphe 1.3.

Nous proposons ensuite une étude de la sémantique des statecharts dans le contexte des systèmes réactifs (cf. § 1.4) puis dans celui des systèmes à objets (cf. § 1.5). Ces deux domaines ont contribué en grande partie à donner aux statecharts leurs lettres de noblesse pour la représentation de la dynamique. Les statecharts ont évidemment connu de nombreuses applications et extensions[2] qui ne sont pas reprises ici. Nous nous limitons dans ce chapitre aux concepts et raffinements consensuels qui ont fait le succès de ce formalisme.

[2] Pour n'en citer que deux : les mini-statecharts introduits par P. Scholtz (http://www4.informatik.tu-muenchen.de/~scholzp/publications.html) et les statecharts étendus de A. Suraj(http://powerlips.ece.utexas.edu/~anitha/).

1.1. Dynamique d'objets

Le comportement d'un objet est généralement défini par l'ensemble des méthodes applicables sur cet objet. L'étudier, c'est observer les états pris par l'objet lors de l'exécution de ses méthodes. Pour cela, plusieurs formalismes ont été introduits : les assertions, les réseaux de Petri, les diagrammes d'états, etc. Pour la plupart basés sur des représentations graphiques, ces formalismes se sont imposés pour modéliser la dynamique des objets ou du système car ils véhiculent des concepts simples.

1.1.1. ASSERTIONS

Les assertions spécifient le mode d'emploi, i.e. les contraintes liées à leur emploi et le résultat des méthodes. Leur utilisation dans les modèles à objets a été popularisée par le langage Eiffel [Meyer92]. Chaque assertion définit le contrat entre les clients potentiels d'un service et le fournisseur de ce service :

- pré-conditions : contraintes que doit satisfaire le client pour utiliser une méthode.
- post-conditions : contraintes que l'objet fournisseur du service s'engage à satisfaire pour le client après exécution d'une méthode.

Ces assertions peuvent être exprimées sur les valeurs des paramètres de l'opération, les valeurs des attributs ou des relations de l'objet. Ce sont généralement des expressions quantifiées de la logique du premier ordre qui ne doivent pas modifier l'état du système [OCL97].

Ouvrage : : emprunt (Livre unLivre, Abonné unAbonné)

pre : livresDisponibles->includes(unLivre);

post : livresDisponibles

= livresDisponibles@pre

->excluding(unLivre);

Ouvrage : : retour (Livre unLivre)

post : livresDisponibles

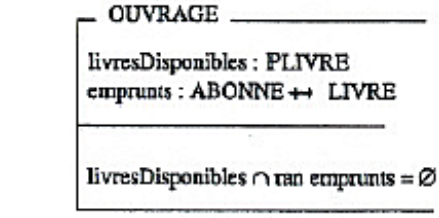
= livresDisponibles@pre

->including(unLivre);

figure 1-1 : Spécification à l'aide d'assertions

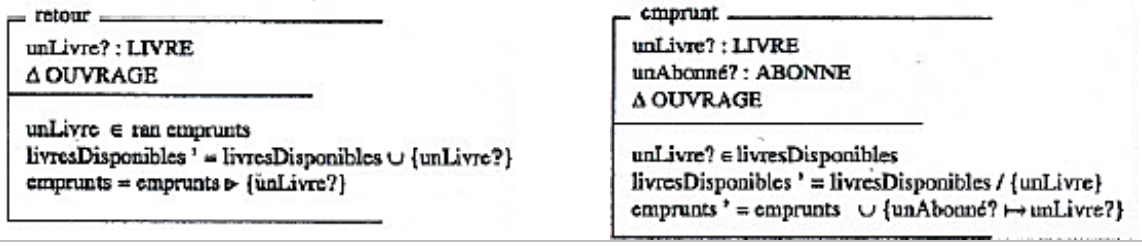
La figure 1-1 exprime en OCL que l'emprunt d'un livre est possible à condition qu'il reste des exemplaires en rayon. Ce livre ne fait alors plus partie des livres disponibles. Il redevient disponible lorsque le retour est signalé : unLivre est inclus dans l'ensemble des livresDisponibles. La notation livresDisponibles@pre fait référence à la valeur de livresDisponibles avant l'exécution du service

Ce type de spécification est largement utilisé par les langages formels tels que VDM, B ou Z. On peut écrire de la même manière une spécification du problème en Z [Habrias93] :



a) Un schéma Z

Le schéma a) permet de connaître l'état d'un ouvrage à tout moment. On peut modifier cet état en utilisant les deux opérations spécifiées ci-dessous (retour et emprunt).



b) Spécification d'opérations en Z

figure 1-2 : spécifications en Z

1.1.2. RESEAUX DE PETRI [PETERSON77]

Les réseaux de Petri ont été d'abord introduits pour la modélisation des systèmes concurrents et parallèles [Reisig85]. Ceux-ci ont introduit deux concepts fondamentaux, les *places* et les *transitions*. Sur chaque place on peut déposer un certain nombre de jetons qui constituent un marquage initial du réseau. Chaque jeton peut être interprété comme une occurrence d'événement ou un état d'un objet. Le marquage modélise donc l'état du système, il évolue en franchissant des transitions. Une transition est franchissable lorsque toutes les places en entrée contiennent au moins un nombre de jetons supérieur où égal à la valuation de l'arc qui les relie à la transition (par défaut, la valuation est de 1). Le franchissement d'une transition enlève à chaque place d'entrée un nombre de jetons égal à la valuation de l'arc d'entrée et ajoute à chaque place de sortie un nombre de jetons égal à la valuation de l'arc de sortie.

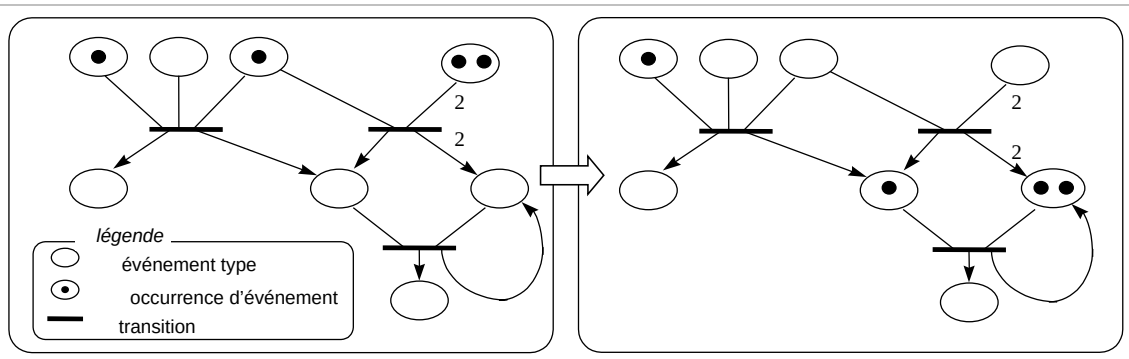


figure 1-3 : Un exemple de réseau de Petri

Les réseaux de Petri ont plusieurs inconvénients :

☹ Ce sont des réseaux « plats » à l'aide desquels on ne peut pas exprimer de notion de profondeur ou de hiérarchie.

☹ Ils proposent une représentation opérationnelle du comportement sans considération pour les données.

Et des avantages :

☺ Ils représentent la dynamique globale du système de manière compréhensible.

☺ Le parallélisme est supporté de facto par le modèle.

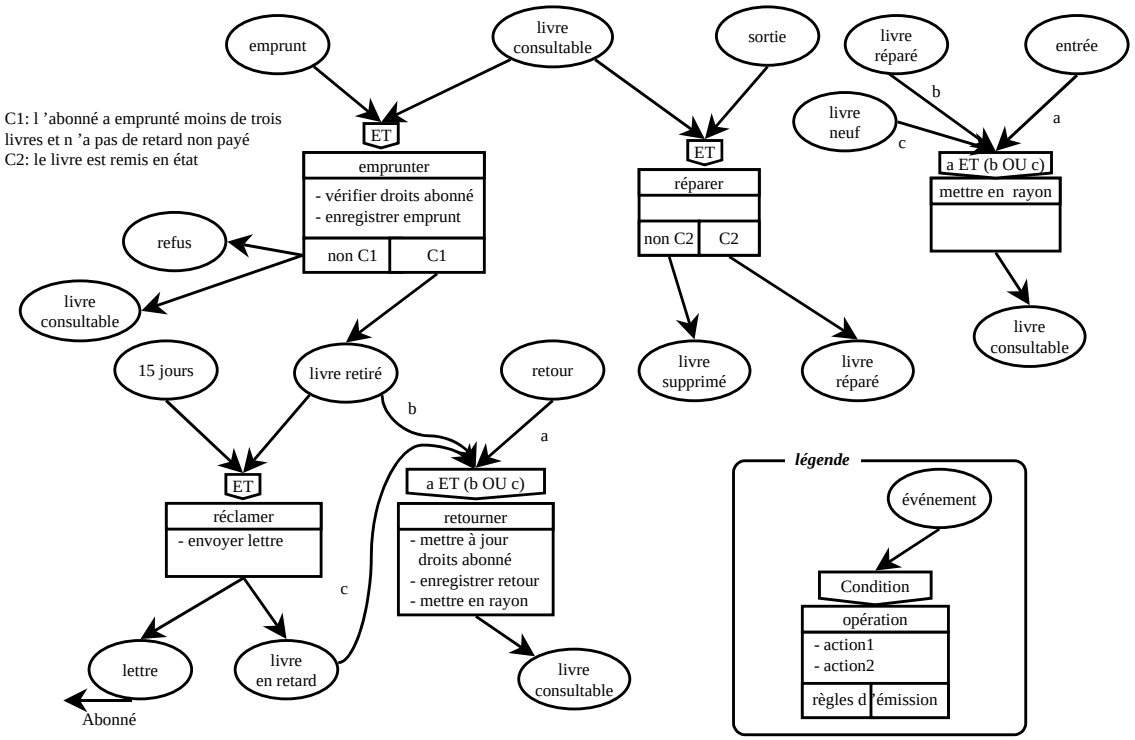


figure 1-4 : Description des processus de prêt et de maintenance d'une bibliothèque

Les réseaux de Petri constituent la base formelle de nombreux modèles disponibles dans les méthodes d'analyse et de conception (M7 [Léonard91], OOSA [Embley92] ou M2PO [Hill93]. Certaines recherches étendent le formalisme de base (les statecharts colorés [Jensen85], utilisation conjointe des réseaux de Petri et des objets [Sibertin-Blanc85, Sibertin-Blanc98]), réduisant ainsi les inconvénients mentionnés ci-dessus. Alors que d'autres méthodes utilisent avec succès le formalisme des réseaux de Petri, il n'existe pas à notre connaissance de méthode commerciale qui tire parti de ces extensions. La méthode Merise par exemple présente ses modèles de traitements conceptuels (MCT) et organisationnels (MOT) comme une extension des réseaux de Petri : la figure 1-4 présente le processus de prêt d'une des bibliothèques de la ville. Le MCT décrit les opérations conceptuelles réalisées en réponse aux événements reçus (événements extérieurs au système) ou émis (événements internes au système). Chaque opération est décrite par une séquence d'actions : par exemple, lors du retour d'un livre, il s'agit de mettre à jour les droits de l'abonné, d'enregistrer le retour et de remettre le livre en rayon.

1.1.3. DIAGRAMMES DE TRANSITIONS D'ETATS

Un diagramme de transitions d'états [Booch91], appelé aussi diagramme d'états [Rumbaugh95], diagramme de l'histoire des objets [Coad90] ou cycle de vie des objets [Panet94], est une représentation graphique d'une machine à états finis. Il s'agit d'un graphe orienté dans lequel les nœuds (représentés graphiquement sur la figure 1-5 par des boîtes aux coins arrondis) représentent des états et les arcs (représentées graphiquement par des flèches sur la même figure) des transitions d'un état à un autre.

La figure 1-5 représente le comportement d'un ouvrage à l'aide d'un diagramme de transitions d'états. Ce dernier contrôle l'emprunt de chaque ouvrage afin de satisfaire la demande des abonnés. L'objectif est de savoir si les livres sont très demandés et si ces demandes sont satisfaites rapidement. On s'intéresse pour cela aux états extrêmes où tous les exemplaires d'un ouvrage sont en rayon (Tous En Rayon) et où ceux-ci sont tous empruntés (Aucun Exemplaire). Les événements emprunt et retour provoquent un changement d'état dans le cycle de vie de l'ouvrage. Ces événements sont « gardés » par une condition : un ouvrage passe de l'état ExemplairesDisponibles à TousEnRayon lors du retour d'un exemplaire si tous les autres exemplaires sont déjà en rayon (condition c2).

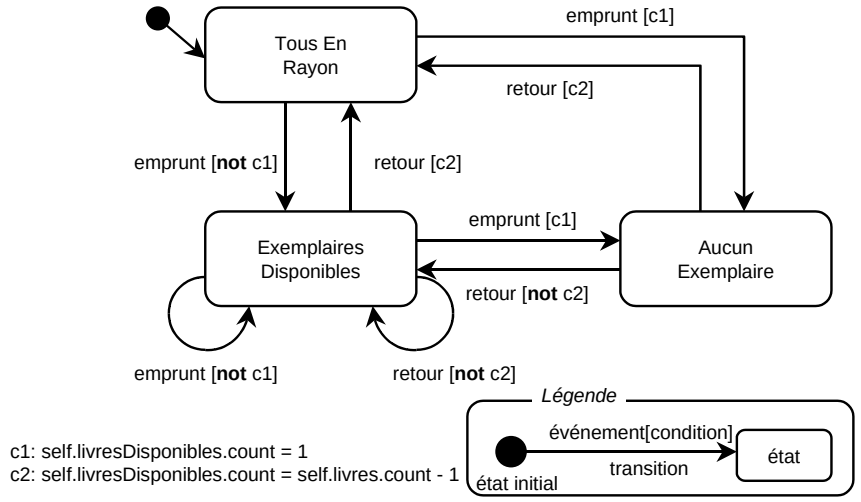


figure 1-5 : Diagramme d'états classique d'un ouvrage

On remarque que dès que le système devient un peu complexe, l'utilisation des diagrammes d'états traditionnels est compromise [Duffy95]. Ce formalisme présente en fait plusieurs inconvénients résumés ainsi :

☹ Les diagrammes d'états standards sont plats. On ne peut pas exprimer de notion de profondeur, hiérarchie ou modularité. Il n'y a donc ni factorisation d'états, ni factorisation de transitions.

☹ Ils ne permettent pas l'expression de la concurrence, étant par nature séquentiels.

☹ Ils sont difficilement extensibles. Le nombre d'états grossit exponentiellement en fonction de la grandeur du système.

Tous ces inconvénients font qu'ils n'offrent que peu, voire pas, de réelle possibilité de réutilisation et qu'ils deviennent rapidement illisibles.

1.1.4. DIAGRAMMES DE TRANSITIONS D'ETATS & RESEAUX DE PETRI

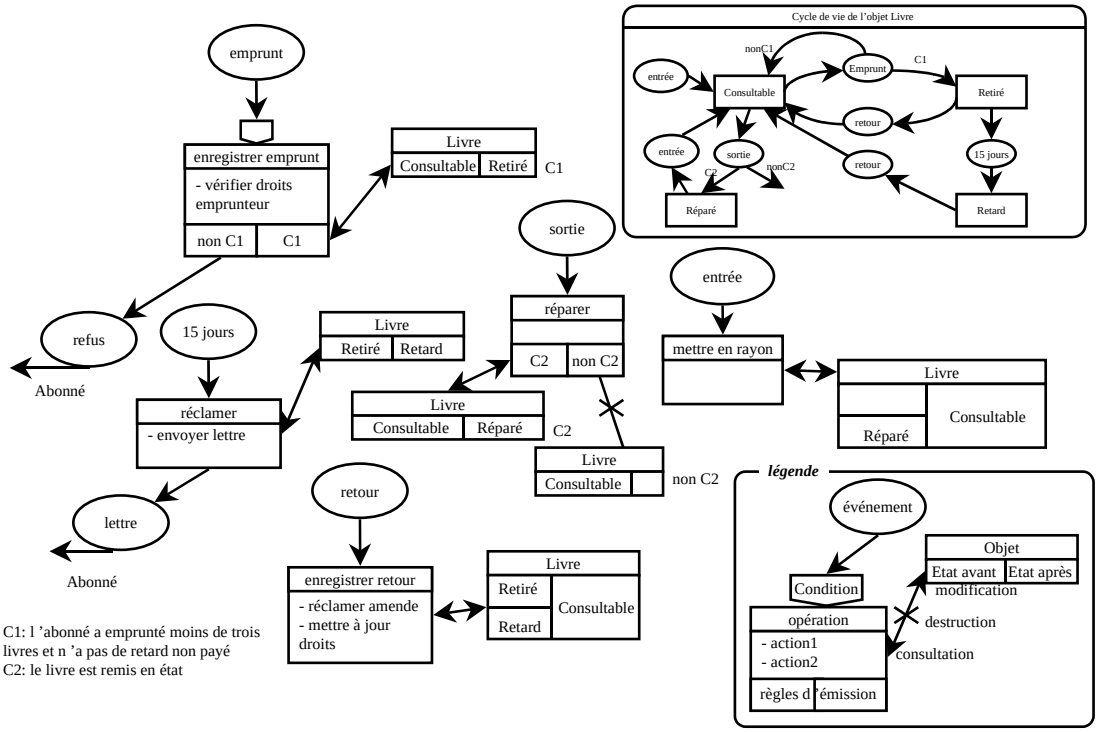


figure 1-6 : MCTA de la gestion des emprunts

Alors que les réseaux de Petri sont la plupart du temps utilisés pour modéliser la dynamique du système, les statecharts sont mieux adaptés pour modéliser le comportement individuel des objets du système d'information. Là encore la méthode Merise dans une de ses extensions (Merise2 [Panet94]) en donne une illustration avec la définition de Modèles de Traitements Analytiques (MCTA et MOTA) qui sont issus du raffinement des modèles de traitement (respectivement MCT et MOT) pour prendre en compte l'évolution des objets du système :

- Les événements externes utilisés pour déclencher les opérations du MCT peuvent déclencher des transitions dans les différents cycles de vie des objets. On spécifie ici (figure 1-6) l'évolution des livres dans le processus de gestion des emprunts.
- Chaque événement interne est remplacé par une consultation ou modification des états des objets du système. La synchronisation entre opérations est alors réalisée à l'aide des états d'objets qui pré et post-conditionnent le traitement. Ainsi, avant d'enregistrer un emprunt, on doit vérifier que le livre se trouve dans l'état initial Consultable ; le livre passe dans l'état final Retiré si la condition C1 est vérifiée.

On peut noter à ce niveau la différence sémantique existant entre les CVO de Merise et les diagrammes d'états. Dans les premiers, la condition sur le franchissement des transitions est vérifiée après l'exécution de l'opération conceptuelle déclenchée par l'événement alors que les diagrammes d'états utilisent des conditions vérifiées avant l'exécution des actions. On peut ainsi obtenir un diagramme d'états équivalent au CVO de la figure 1-6 en rajoutant un état En examen (figure 1-7) qui consiste en la réalisation de l'activité réparer (cf. § 1.3.1).

Les opérations enregistrer emprunt, enregistrer retour et mettre en rayon correspondent ici à des actions (cf. § 1.3.1) du diagramme d'états car leur durée est négligeable. Ceci est possible car la valeur de l'expression c1 n'est pas modifiée par le traitement emprunter au contraire de la condition c2 qui elle dépend du traitement réalisé dans En Examen :

{c1}emprunter{c1} et {non c1}emprunter{non c1}

alors que {non c2} réparer {c2 ou non c2}.

Dans le même esprit, des règles de passage ont été introduites par [Dano97] pour passer des réseaux de Petri décrivant des fonctionnalités générales d'un système à des spécifications de comportements d'objets spécifiés à l'aide de statecharts.

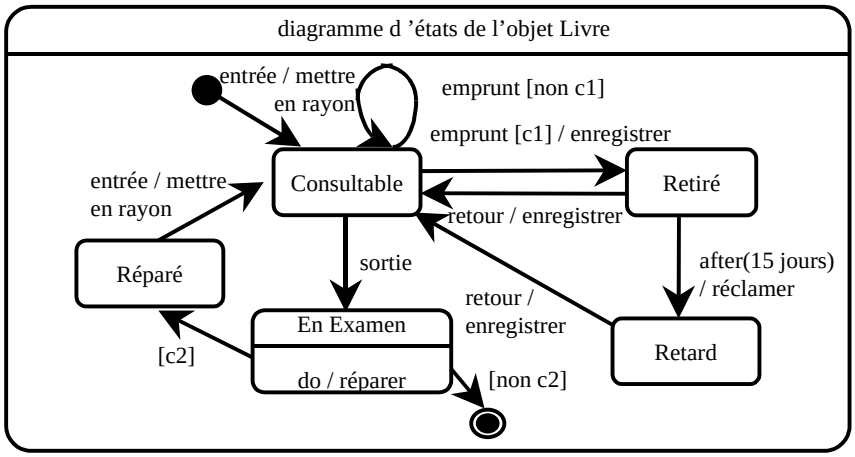


figure 1-7 : diagramme d'états de l'objet Livre

1.2. Statecharts

Pour contourner en partie les inconvénients des diagrammes d'états, D. Harel a introduit les statecharts en 1987 [Harel87], notation qui implante à la fois des machines de Moore et des machines de Mealy [Hopcroft79] (cf. § 1.2.3). Les statecharts sont aujourd'hui reconnus comme un modèle standard [UML97] pour la modélisation de la dynamique.

Les statecharts sont avant tout un formalisme graphique puissant pour modéliser des comportements complexes. Ils se présentent comme une extension des diagrammes de transitions d'états basée sur le formalisme graphique des hygraphs [Harel88]. Un hygraph est

représenté par un rectangle aux bords arrondis appelé parfois « blob ». Les blobs (figure 1-8) sont utilisés pour implanter des principes de la théorie des ensembles tels que l'union (A ⇔ B ∪ C), l'intersection (C ⇔ A ∩ D) et la différence (B ⇔ A - D) entre deux ensembles.

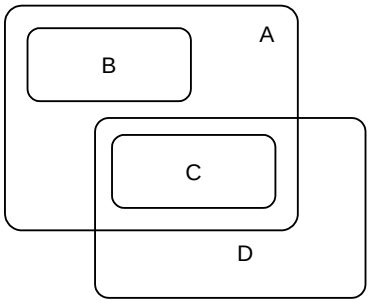
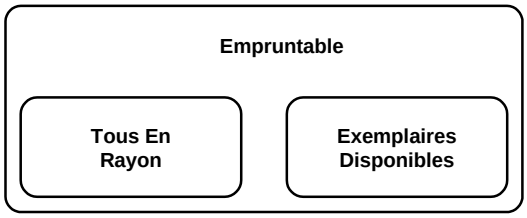


figure 1-8 : Formalisme des Hygraphs

De manière générale, les statecharts sont basés sur les concepts de profondeur, orthogonalité et diffusion.

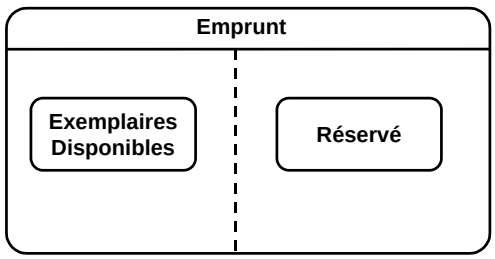
1.2.1. PROFONDEUR

Le premier concept introduit par Harel est la hiérarchie d'états. Il s'agit d'une décomposition OU des états dont les sous-états (enfants) sont exclusifs entre eux, on ne peut donc être que dans l'un ou l'autre des sous-états. De plus les sous-états forment une partition du super état (parent). Dans un sous-état, le prédicat définissant le super-état est vrai ainsi que celui du sous-état.

On a : TousEnRayon ⇒ Empruntable, ExemplairesDisponibles ⇒ Empruntable, TousEnRayon ⇒ **non** ExemplairesDisponibles, Empruntable ⇒ (TousEnRayon **ou** ExemplairesDisponibles)	

1.2.2. ORTHOGONALITE

La seconde amélioration concerne la possibilité de définir des états orthogonaux. Deux composants indépendants peuvent être décrits dans un même diagramme à l'aide d'une décomposition ET des états. Etre dans un état ET veut dire que l'on est dans tous les sous-états immédiats au même moment.

On a : ExemplairesDisponibles ⇒ Emprunt, Réservé ⇒ Emprunt, Emprunt ⇒ (ExemplairesDisponibles **et** Réservé))	

Ce Statechart montre qu'il est aussi possible d'implanter le produit cartésien de deux ensembles à l'aide de hygraphs : l'ensemble A est l'ensemble des paires (b, c) où b ∈ B et c ∈ C.

1.2.3. DIFFUSION

Une machine à états finis traditionnelle sert à reconnaître des chaînes de mots et fournit un signal seulement lorsque l'état final est atteint. Or un système réactif produit des sorties à tout moment de son exécution, un diagramme d'états doit faire de même. C'est par exemple le cas des diagrammes d'états construits sur les machines de Mealy [Hopcroft79] dans lesquels il est possible d'associer une sortie à une transition ou de Moore dans lesquels la sortie est associée à l'entrée dans un état (figure 1-9). Cette sortie est diffusée dans tout le système et peut déclencher d'autres transitions ou actions.

Considérons une machine à états qui prend en entrée une chaîne de caractères composée de 0 et de 1. Cette machine émet des 1 en sortie lorsque la chaîne comprend au moins deux 1 en séquence.

La figure ci-contre montre les deux diagrammes d'états minimaux, respectivement de Moore (a) et de Mealy (b) qui représentent cette machine.

Les machines de Mealy utilisent généralement moins d'états qu'une machine de Moore pour générer la même sortie du fait que les sorties sont associées aux transitions [Katz93].

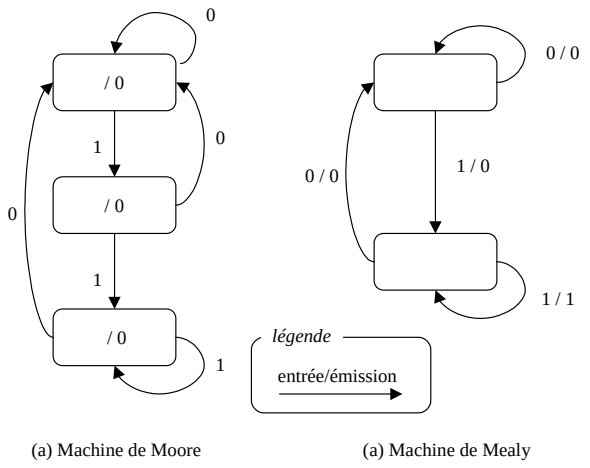


figure 1-9 : Deux diagrammes d'états représentant un même comportement

En résumé, on peut dire que les statecharts se résument à [Huizing89] :

STATECHARTS = MACHINE A ETATS FINIS + PROFONDEUR + ORTHOGONALITE + DIFFUSION

Le statechart ci-dessous (figure 1-10) donne un premier aperçu des spécifications qui peuvent être obtenues par application de ces principes. Nous occultons ici les détails liés au formalisme qui sont présentés dans le § 1.3.

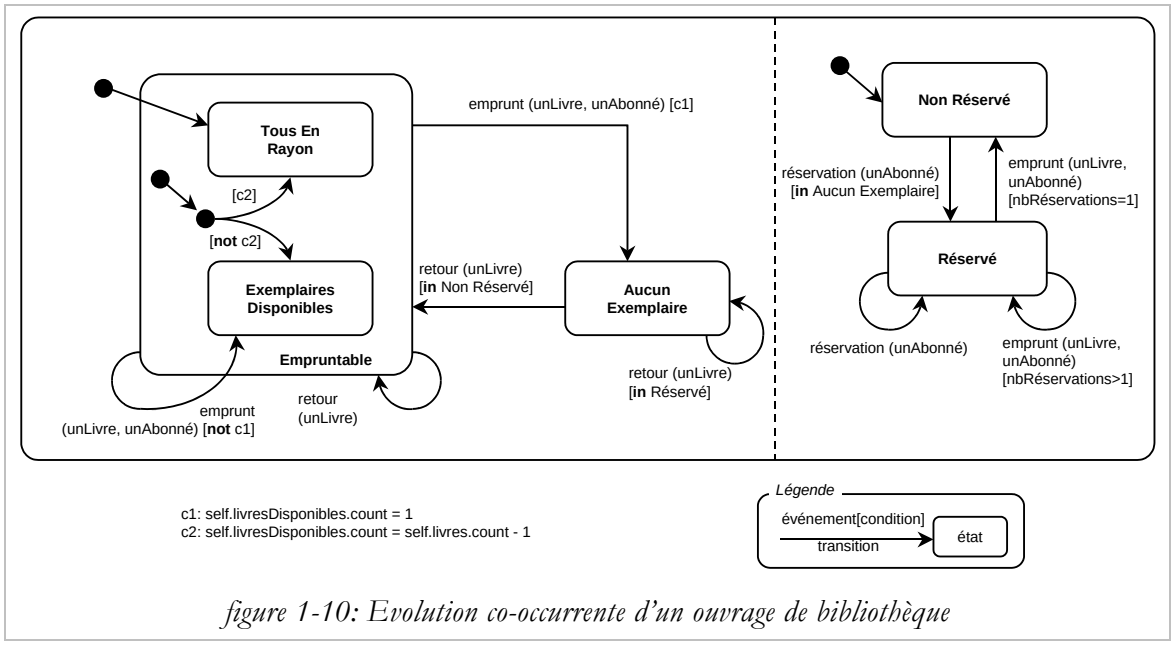


figure 1-10: Evolution co-occurrente d'un ouvrage de bibliothèque

Ce formalisme permet la représentation de comportements plus complexes que dans un simple diagramme d'états : on contrôle à l'aide de ce statechart à la fois le prêt et la réservation des ouvrages. En effet, la bibliothèque ne gère pas la réservation explicite de tel ou tel exemplaire. La réservation porte sur un ouvrage : lorsqu'un exemplaire de cet ouvrage est retourné, il est mis de côté en attendant que l'abonné vienne le retirer. La réservation se fait bien sûr lorsqu'il n'y a plus de livres disponibles et on peut avoir plusieurs réservations portant sur un même ouvrage.

La réservation ajoute une certaine complexité à la figure 1-5, en partie compensée par la factorisation des transitions emprunt et retour. Sur cette figure, la transition emprunt est héritée par les deux sous-états de Empruntable.

	Graphique	Evénementiel	Données	Parallélisme	Modularité	Hiérarchisation	Complexité	Objets multiples
Assertions			☺		☺	☺		
Réseau de Petri	☺	☺		☺	☺		☺	☺
Diagramme d'états	☺	☺	☺					
statecharts	☺	☺	☺	☺	☺	☺	☺	😐

Le tableau ci-dessus reprend chacun des critères qui devraient à notre sens qualifier un formalisme modélisant des comportements d'objet. Seules les assertions ne prennent pas en compte les événements et ne permettent pas la représentation de la séquence d'événements (ou

de l'ensemble des événements synchronisés [Hartmann93]) qui affectent un objet et qui constituent son cycle de vie.

On s'aperçoit que les statecharts répondent à chacun des critères. En effet, il s'agit d'un formalisme graphique et événementiel qui spécifie le comportement de chaque objet en fonction de ses données ; on a vu qu'il s'agissait d'un formalisme modulaire et hiérarchique qui offre la possibilité de spécifier des comportements co-occurrents, avec en plus un bon rapport complexité de l'objet modélisé / complexité du modèle. De plus, la sémantique précise de ce formalisme en fait un bon candidat pour les approches qui s'intéressent au passage semi-formel/formel. On en retrouve par exemple des traductions dans les langages B [Nguyen98] ou Z et Object-Z [Dupuy98].

1.3. Formalisme

Nous décrivons dans ce paragraphe le formalisme complet des statecharts tel qu'il est présenté dans le standard de modélisation UML. Originellement, ce formalisme a été entièrement défini par D. Harel [Harel87].

1.3.1. CONCEPTS

{Etat}

Un état est une situation durant la vie d'un objet ou une interaction pendant laquelle il satisfait une condition, réalise une action atomique ou encore attend un événement. Un état a une durée ; il occupe un intervalle de temps.

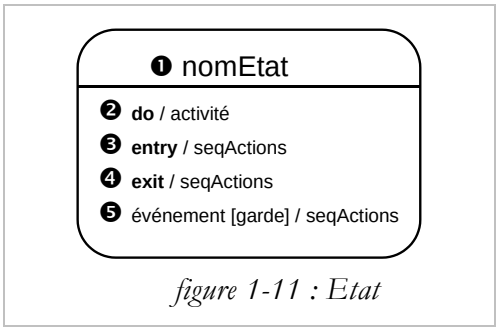


figure 1-11 : Etat

Un état a un nom❶ optionnel, il peut-être associé à une activité continue❷ (sur la figure 1-12, l'état En examen correspond à la restauration du livre et donc à l'activité réparer). Une activité[3] est une opération qui prend un certain temps et dont l'exécution peut être interrompue par un événement extérieur, contrairement à l'exécution d'une action qui est atomique. Une action ou une séquence d'actions est déclenchée par l'entrée❸ dans l'état, par la sortie❹ de l'état ou encore par un événement extérieur❺, dans ce cas uniquement elle peut être « conditionnée ». Les activités et actions n'ont pas d'effet de bord sur l'état de l'objet.

{Evénement}

Dans un diagramme d'états, un événement est le seul élément qui peut déclencher une transition d'états. Un événement peut être de plusieurs types (pas nécessairement exclusifs) :

[3] Dans le langage de modélisation unifié, une activité est forcément une machine à états finis qui a un état d'entrée et un état de sortie uniques.

- Une condition (le plus souvent booléenne) qui devient vraie. L'événement est alors défini à l'aide du mot clé **while** suivi par l'expression booléenne entre parenthèses.
- La réception explicite d'un signal.
- La réception d'un appel de procédure. Signaux et appels de procédure sont des événements paramétrés qui déclenchent tous deux une transition. Ils ne sont pas visuellement différenciés dans un diagramme d'états.
- Une durée écoulée après un événement donné. Celle-ci est notée comme une expression temporelle à l'aide du mot clé **after**. Ainsi, un livre emprunté plus de quinze jours entre automatiquement dans l'état Retard (figure 1-12). Une lettre de réclamation est alors émise au nom de l'abonné.

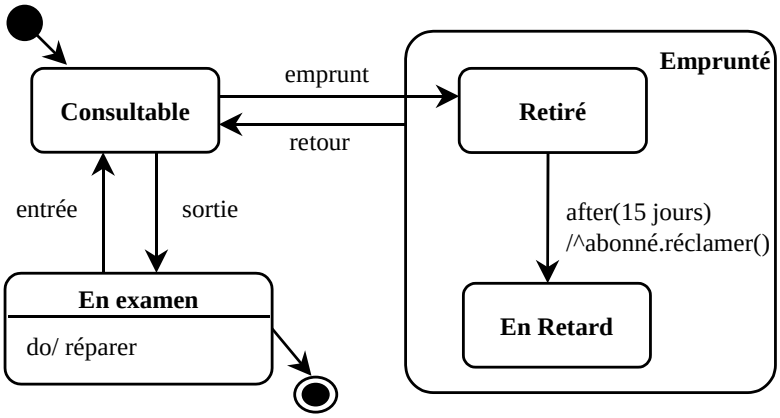


figure 1-12 : Emprunt : gestion du retard à l'aide de statecharts

{Transition}

Une transition est une relation entre deux états indiquant qu'un objet passe d'un état initial à un état final, en réalisant certaines actions spécifiées, à la réception d'un événement particulier et sous réserve de la satisfaction de conditions requises. Un tel changement est appelé franchissement de transition.

Graphiquement, une transition est représentée par une flèche pleine de l'état source vers l'état cible et étiquetée comme suit :

label := événement-signature '[' garde ']' '/' séquenceDActions '^' envoi-clause

où

- événement-signature décrit un événement avec ses paramètres :

  événement-signature := nomEvénement '(' paramètre ',' … ')'

- garde est une expression booléenne sur les paramètres de l'événement et les attributs ou les liens de l'objet. La garde peut aussi porter sur les états co-occurrents de l'objet ou sur les états d'autres objets.
- séquenceDActions est une expression procédurale exécutée lorsque la transition est franchie. Elle peut être écrite à l'aide des paramètres de l'événement ou d'opérations, d'attributs et de liens de l'objet. séquenceDActions est une opération atomique, i.e. non interruptible.
- envoi-clause est un cas spécial d'action spécifiant la communication inter-objets avec le format :

  envoi-clause := destination '.' nomMessage '(' argument ',' ... ')'

  où destination représente un objet ou un ensemble d'objets. La destination et les arguments peuvent être décrits à l'aide des paramètres de l'événement et les attributs et les liens de l'objet.

  Exemple (figure 1-10) :

retour (unLivre : Livre) [**in** Réservé] / self.demandeurs->size = self.demandeurs@pre->size -1 ^ unLivre.réserve(self.demandeurs->first)

événement-signature **garde** **séquenceDActions** **envoi-clause**

1.3.2. DECORATIONS

De nombreuses « décorations » graphiques et sémantiques ont été définies pour modéliser des comportements plus complexes et améliorer la lisibilité des statecharts.

{Transition complexe}

[UML97] distingue deux types de transitions, les transitions simples et complexes. Alors qu'une transition simple a des états source et cible uniques, une transition complexe en admet plusieurs. Une transition complexe représente la synchronisation et/ou le partage du contrôle entre états co-occurrents. Elle est représentée par une barre grasse qui est appelée barre de synchronisation.

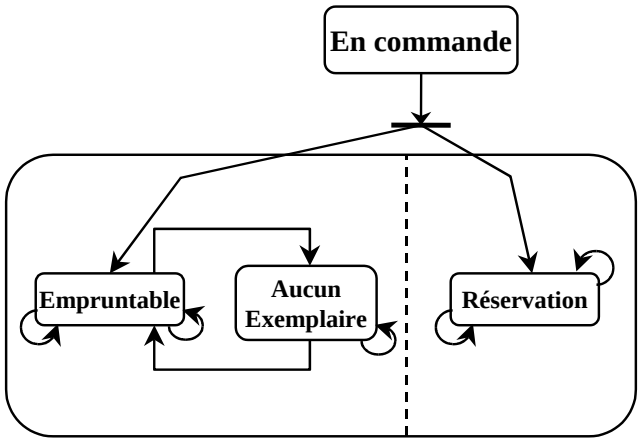


figure 1-13 : transition complexe

Une transition complexe est équivalente à une combinaison de transitions simples (si on considère que la barre de synchronisation est équivalente à un état). On aurait pu représenter l'entrée dans les états Réservation et Empruntable de la figure 1-13 à l'aide de deux transitions simples.

{Transition brisée}

Une transition brisée permet d'abstraire visuellement certains états spécifiques d'un statechart, ici Tous en rayon et Exemplaires disponibles, deux sous-états de Empruntable (cf. § 1.2.1). Les transitions qui partent ou arrivent des états supprimés sont alors liées à leur état parent le plus spécifique. Sur l'exemple, la transition (Empruntable, *emprunt*, Exemplaires disponibles) est coupée et rattachée à l'état Empruntable à l'aide d'une souche (barre pleine). Seuls les événements portés par les transitions d'entrée des états supprimés restent visibles, c'est le cas pour emprunt.

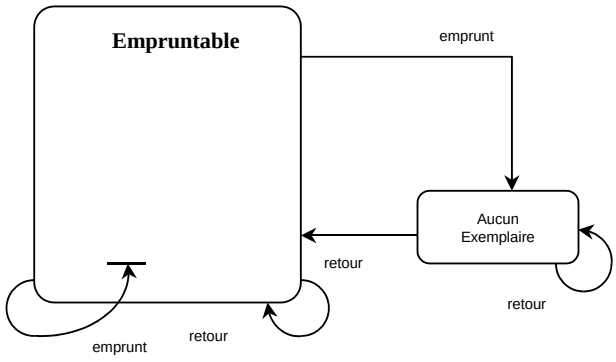


figure 1-14 : Abstraction du Statechart de la figure 1-10

{Historique}

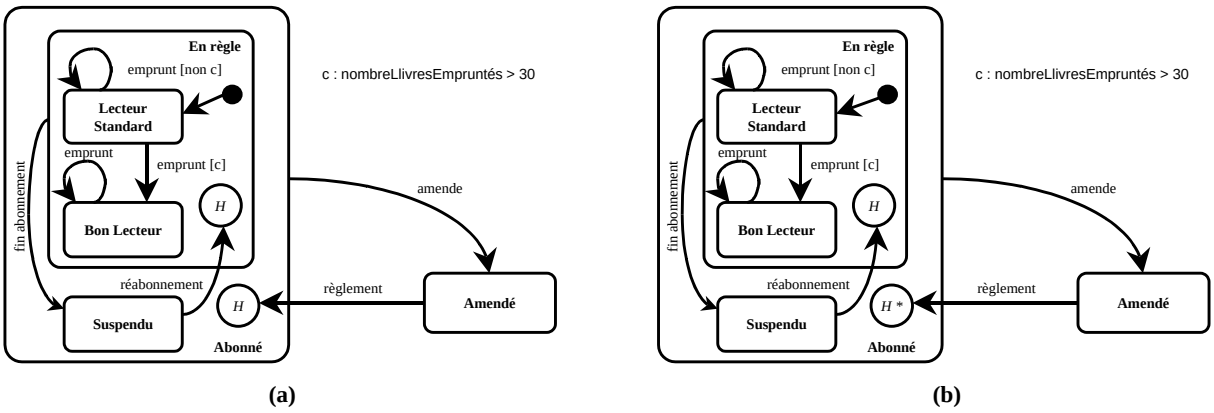


figure 1-15 : statecharts d'un Abonné : gestion de l'historique

L'historique définit l'entrée dans un groupe d'états en surchargeant l'entrée par défaut. Il est représenté par un **H** cerclé.

La manière la plus simple d'utiliser l'historique consiste à entrer dans le dernier état visité, l'historique étant géré au niveau d'abstraction où il apparaît (figure 1-15, statechart (a)). On peut aussi entrer dans les états dernièrement visités au plus bas niveau en marquant le **H** d'un astérisque (cas de la figure 1-15, statechart (b)). Le concepteur peut décrire de nombreuses situations en combinant des gestions d'historique à plusieurs niveaux. Ainsi sur la figure 1-15 (a), un abonné qui a été Amendé et qui règle son amende (événement règlement) revient dans l'état

soit Suspendu, soit Lecteur Standard (qui est l'état d'entrée par défaut de l'état En Règle) alors que sur la figure 1-15 (b), celui-ci peut ré-entrer dans l'état Lecteur Standard, Bon Lecteur ou Suspendu (la transition d'entrée par défaut de l'état En Règle est surchargée du fait de l'astérisque). Autrement dit, le fait d'être amendé dans le premier cas ramène obligatoirement l'abonné En Règle dans l'état par défaut Lecteur Standard alors que dans le second cas celui-ci peut rester Bon Lecteur même lorsqu'il rend ses livres en retard.

{Condition}

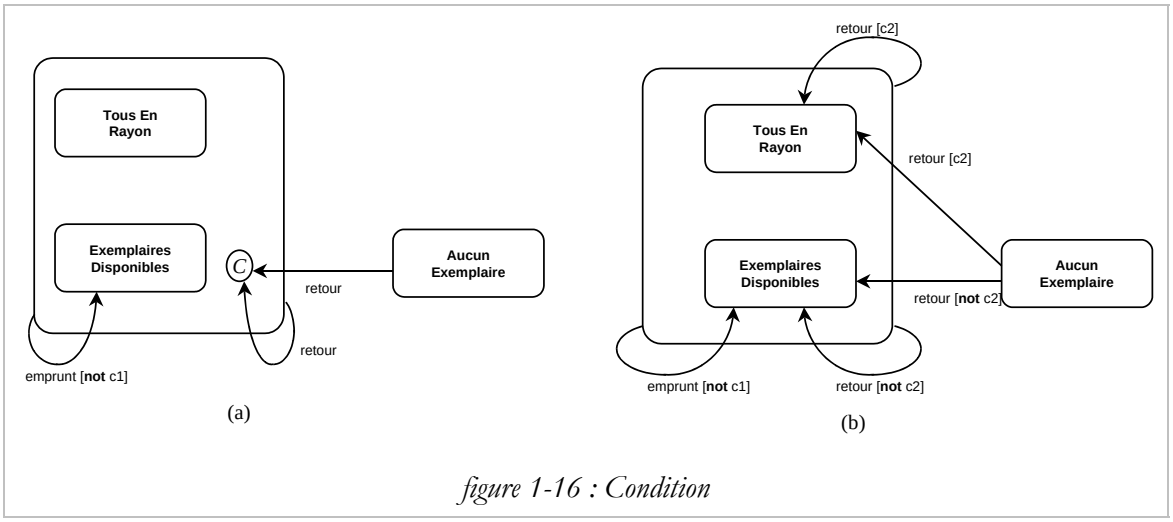


figure 1-16 : Condition

La condition simplifie l'expression de conditions multiples sur une même transition.

Le statechart (a) est équivalent au statechart (b) (*figure 1-16*), les conditions étant omises sur le schéma et décrites séparément. Ainsi les transitions de retour ne sont pas dupliquées, ce qui allège le schéma. Le statechart de la figure 1-10 (partie gauche) où apparaissent directement les conditions est lui aussi une alternative de présentation équivalente à ces deux statecharts.

{Sélection}

La sélection intervient quand l'état entrant est déterminé par un événement externe : le choix d'un bouton sur un tableau de bord, par exemple. Ici, l'état d'un livre retourné à la bibliothèque est soumis au jugement de l'employé qui réceptionne le livre. Celui-ci peut choisir de le sortir du prêt et de le rendre Non Consultable (figure 1-17) ou au contraire de le laisser Consultable.

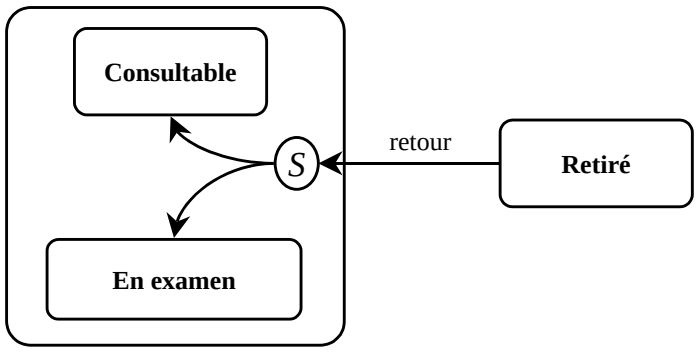


figure 1-17 : Sélection

L'événement retour est spécifié comme la disjonction de deux sous-événements qui conduisent aux états Consultable ou En examen.

1.4. Statecharts et systèmes réactifs

Les statecharts ont été introduits en premier lieu pour la modélisation des systèmes réactifs. Il s'agit sûrement d'un des domaines les plus prolixes concernant ce formalisme. Nous proposons dans la première partie (cf. § 1.4.1) une rapide exploration des spécificités et problèmes de ce domaine. La deuxième partie (cf. § 1.4.2) présente l'intégration spécifique des statecharts dans une approche originale d'aide à l'ingénierie des procédés logiciels.

1.4.1. PRESENTATION

On oppose les systèmes réactifs aux systèmes transformationnels. Dans ces derniers, la spécification des relations entre les entrées et sorties du système est généralement suffisante : un système transformationnel a une structure linéaire, il prend une entrée, produit une sortie et se termine. Un système réactif, par contre, est en continuelle interaction avec son environnement. « Si un système transformationnel peut-être comparé à une boîte noire, un système réactif peut être assimilé à un cactus noir » (Amir Pnueli). La spécification de la relation existant entre les entrées et les sorties n'est plus suffisante : les entrées d'un système réactif sont dépendantes des sorties précédentes : l'état interne du système doit être pris en compte pour spécifier la réaction à une entrée du système. C'est pourquoi le formalisme des statecharts est approprié pour la modélisation des systèmes réactifs et a été sujet à de nombreuses variantes dont il n'est pas fait état ici. M. Beeck par exemple en dénombre plus de vingt variantes dans [Beeck94]. Ce paragraphe recense les aspects qui semblent aujourd'hui consensuels, notamment sur la gestion du temps dans les statecharts.

1.4.1.1. Evénements

Les choix temporels liés au formalisme des statecharts sont justifiés par le besoin de décrire les systèmes réactifs à un haut niveau d'abstraction et ceci de manière discrète. Un système réactif se distingue par deux phases dans lesquels l'événement joue un rôle primordial :

- L'environnement envoie des événements au système pour déclencher des traitements. Chaque événement est un signal discret qui n'a pas de durée : à un instant donné, l'événement surgit, est capturé ou définitivement perdu. Evidemment, la génération d'événements doit répondre à la même contrainte et prendre un temps nul.
- Le système réagit en envoyant ou en générant des nouveaux événements. Il est nécessaire de déterminer le temps de réaction du système lorsqu'il est réactif car les réactions du système peuvent interférer avec les futures entrées. Il faut savoir à tout moment dans quel état se trouve le système. Pour cela, la stratégie consiste à passer d'un état à l'autre de manière instantanée, ceci à l'aide de transitions non durables. Seul le fait de rester dans un

état a alors une durée quantifiée. Ces hypothèses sont compatibles avec l'hypothèse de synchronisation [Berry88] qui assure que le système est infiniment plus rapide que l'environnement.

Le système a donc un temps de réaction nul. Modifier la chaîne de causalité (i.e. la séquence de transitions déclenchées par une entrée) n'affecte pas le temps de réponse du système, puisque 0+0 =0. La figure 1-18 illustre les conséquences de l'introduction d'un temps de réponse nul. La chaîne de causalité peut être représentée par : t1 déclenche t2, t2 déclenche t3. Ces trois transitions sont déclenchées au même instant, de même que les événements α, β, δ, σ. Le passage de la configuration {A, B, C} à la configuration {A', B', C'} se fait de manière instantanée sur capture de l'événement α.

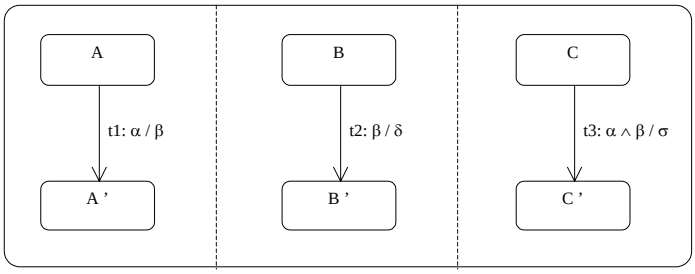


figure 1-18 : Transitions instantanées

1.4.1.2. Problèmes

Les problèmes rencontrés pour la modélisation des systèmes réactifs [Beeck94] sont principalement de deux ordres et tous deux liés à l'hypothèse d'un temps nul :

- L'opérationalisation des spécifications abstraites n'est pas triviale puisque les traitements sur machine prennent du temps et invalident l'hypothèse de synchronisation (cf. § 1.4.1.1). Certaines solutions opérationnelles prennent en compte cette dimension en introduisant deux niveaux de temps [Harel96b]. Le premier niveau compte le temps en macro-intervalles, qui mesurent le temps observable. Chaque macro-intervalle est divisé en un nombre arbitraire de micro-intervalles. La chaîne de causalité dans un macro-intervalle est modélisée à l'aide d'une séquence de micro-intervalles.
- De nombreuses hypothèses doivent être ajoutées pour vérifier les propriétés de causalité des statecharts (le principe de causalité assure que dans un intervalle donné toute transition t ne peut être déclenchée par des événements générés par une transition apparaissant après t [Huizing92]) et gérer l'indéterminisme inhérent au modèle des statecharts. Il est important de noter que les statecharts n'empêchent pas l'indéterminisme, mais donnent les moyens de le lever dans de nombreux cas notamment

grâce à la gestion de priorités, à la possibilité de spécifier la non-occurrence d'un événement, etc.

1.4.2. APEL : UN FORMALISME POUR L'INGENIERIE DES PROCEDES

L'amélioration des procédés de logiciels nécessite des formalismes de haut niveau pour décrire les aspects qualité et organisationnel d'un projet. Ces formalismes doivent être utilisables non seulement pour la capture mais aussi pour l'exécution des spécifications. De ce constat est né le formalisme *APEL*, noyau d'un environnement de procédés de logiciels supportant les tâches d'ingénierie [Dami98].

APEL (Abstract Process Engine Language) est un langage ambitieux qui couvre une grande partie des besoins en ingénierie des logiciels. Il manipule des concepts au premier abord hétérogènes issus de domaines tels que les systèmes temps-réel, les méthodologies orientées objet, l'intégration d'outils, les Systèmes d'Information et le travail coopératif. L'intégration de ces concepts doit aboutir à un tout cohérent, minimal et facile à étendre. Dans *APEL*, les aspects dynamiques des processus sont décrits à l'aide de diagrammes de contrôle, diagrammes de données et diagrammes d'états. Dans ce contexte particulier, nous nous intéressons plus particulièrement à la gestion des événements et des diagrammes d'états.

1.4.2.1. Gestion des événements

Les aspects dynamiques sont basés sur le concept d'événement et de capture d'événement. Les événements sont générés automatiquement chaque fois que :

- Une méthode est appelée, notée « m(producer [, activity]) » où m est la méthode, producer l'objet sur lequel la méthode est exécutée et activity l'activité dans laquelle la méthode est appelée
- L'état d'une instance est modifié, noté « (producer, new-state [, activity]) » où producer est l'objet qui a changé d'état, new-state le nouvel état de l'objet et activity l'activité dans laquelle la méthode est appelée.
- Un signal d'horloge est généré, noté « (producer, event-id,'TMP') » où producer est une entité, event-id un identifiant et TMP une chaîne constante

Tous les événements d'instance générés sont systématiquement diffusés. Ils sont alors soit capturés par des clients potentiels soit ignorés. La capture d'un événement est représentée par une paire (définition de l'événement, expression logique). Pour qu'un événement soit capturé par un client, l'expression logique doit être vraie au moment où cet événement est « reconnu » par le client.

Les événements sont capturés par les entités qui ont besoin de réagir à un événement. Il est intéressant de noter que tout changement d'état dans un diagramme d'états génère un événement qui signale ce changement.

1.4.2.2. Diagrammes d'états et d'activités

Les auteurs concilient les approches centrées produit et les approches centrées activité à l'aide des diagrammes d'états [Estublier95].

Dans *APEL*, les diagrammes d'états sont utilisés pour représenter le cycle de vie d'objets et plus particulièrement celui des produits, des agents et des activités du système. Les activités sont associées à des diagrammes d'états prédéfinis et qui ne peuvent être étendus alors que les produits et agents peuvent spécifier leur propre évolution. Une approche intéressante consiste à spécifier cette dernière par rapport à la réalisation d'une activité. Pour cela, [Estublier95] introduit le concept de diagramme d'états local. Un diagramme d'états local définit les changements d'états d'un type de produit pour une activité particulière. Dans ce diagramme, la décomposition des états suit exactement celle des activités (figure 1-19). De fait, les états composites sont des états abstraits qui indiquent que le produit est dans une activité ou non : un état abstrait est toujours nommé par le nom de l'activité en cours préfixée par « In_ ».

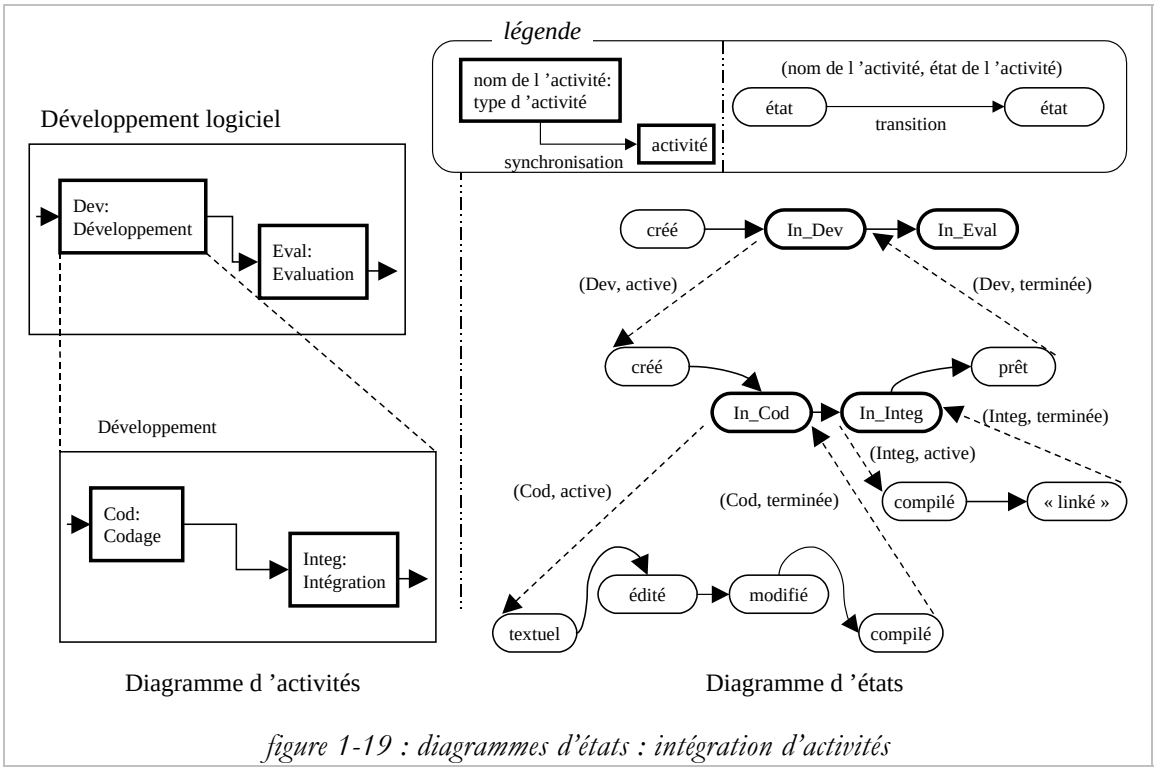


figure 1-19 : diagrammes d'états : intégration d'activités

Les diagrammes d'états locaux font la distinction entre les états locaux (qui ne sont pas partagés par les activités) et les états globaux (qui font partie d'un référentiel unique et sont partagés par les différentes activités).

Tout produit peut avoir un diagramme d'états structuré et modulaire par activité. La capture d'un tel diagramme peut se faire en fonction de différents points de vue (i.e. différentes activités) et à différents niveaux d'abstraction. La réutilisation des activités est favorisée car l'intégration d'une nouvelle activité dans un diagramme d'états n'a pas d'effets de bord sur les parties existantes de ce diagramme.

1.5. Statecharts et modèle objet

La différence entre des statecharts classiques et des statecharts objets est liée au contexte externe de la machine à états représentée. Alors que les statecharts spécifient plutôt la dynamique de processus ou d'activités dans les systèmes réactifs, les statecharts objets représentent le comportement d'un type. Ce paragraphe liste les différences induites par le paradigme objet.

1.5.1. STATECHARTS : CONTROLEURS D'OBJETS

Le comportement d'un objet est spécifié par un statechart associé à sa classe. Cette modélisation n'est pas obligatoire et n'a de sens que pour les objets qui ont un comportement significatif [Rumbaugh95]. Les objets dont on ne décrit pas le comportement sont appelés primitifs par opposition aux objets non primitifs dont le comportement est spécifié dans le modèle.

Le contrôle porte sur la dynamique de l'objet en termes de communication et de liens avec les autres objets. Cela pose deux problèmes distincts :

- Garantir une initialisation cohérente du système. On doit gérer à ce niveau la création des instances et l'initialisation des attributs et relations entre objets. Pour cela, on peut utiliser des scripts d'initialisation [Harel96a] dans chaque statechart associé à un objet.
- Assurer le bon déroulement du programme. Il s'agit de prendre en compte deux types de changements dans le système. Le premier dépend de l'évolution du système et intervient en réponse aux événements générés à la fois par le système et l'environnement. Le second correspond à un changement dans la structure du modèle : quatre types de changement sont possibles, création et destruction d'instances, création ou destruction des relations entre objets.

1.5.2. EVENEMENTS

Les langages de programmation utilisent un mode de communication synchrone basé sur l'envoi de messages. La sémantique des événements est donc largement modifiée pour répondre à

cette contrainte. Dans les statecharts objets, les événements ne sont plus des signaux primitifs[4] mais des concepts paramétrés qui transportent des données. Ces événements peuvent déclencher des opérations, la conjonction d'événements n'étant plus supportée. On introduit volontairement un traitement séquentiel des événements pour éviter l'indéterminisme des systèmes réactifs en faisant de sorte qu'à chaque événement corresponde une méthode unique. Si plusieurs objets peuvent répondre à un même événement par l'activation d'une de leur méthode, l'ordre de réponse est choisi aléatoirement.

Un statechart est stable lorsque toutes les parties d'une transition sont complètement exécutées. Cette exécution ne prend pas un temps nul comme dans les systèmes réactifs. Une transition qui active une méthode par un appel d'opération doit attendre la fin de son exécution pour se terminer. Cette différence entre événements et opérations est importante pour le client : générer un événement n'est pas contraignant pour un objet qui peut ensuite garder le flot de contrôle alors qu'invoquer une opération laisse l'objet « endormi » en attendant son exécution par l'objet appelé. En effet, les opérations sont des entités plus concrètes que les événements dans le sens ou une opération admet un objet cible clairement défini et est donc peu diffusable dans le système. Au contraire, on a vu que les événements peuvent être diffusés largement en utilisant des mécanismes de distribution flexibles (broadcasting) et de délégation (cf. § 1.2.3).

1.5.3. COMMUNICATION ENTRE OBJETS

On distingue deux types d'approches :

- Un objet peut générer un événement qui est placé soit dans une pile partagée en attente du traitement par un serveur [Harel96a] soit dans les piles individuelles des objets concernés [Allen95]. Dans le cas où la pile est partagée, une stratégie de délégation est définie pour chaque événement : un événement émis par un objet peut être redistribué au système tout entier, seulement à certains des ses composants ou simplement en interne. Dans tous les cas, l'objet qui émet un événement est aussi receveur. Cette sémantique additionnelle restreint la portée des événements, de manière limitée dans le cas d'une communication personnelle entre objets ou large si l'événement est diffusé dans tout le système. De la même manière qu'on délègue des événements à certains objets, on peut imaginer que certains objets seraient exclus de la diffusion (broadcasting) par exemple pour des raisons de sécurité.
- Un objet peut invoquer directement une opération d'un autre objet. Dans cette approche, le contrôle est laissé à l'objet appelé qui exécute la méthode sans délai jusqu'à son terme et peut

[4] Les statecharts objets n'ont pas d'équivalent en ce qui concerne l'ensemble prédéfini d'actions, de conditions et d'événements qui est généralement défini pour des statecharts classiques (cf. § 1.4.2.2).

ensuite rendre une réponse. La méthode est terminée lorsque l'objet appelé est dans un état stable.

Alors que le modèle à objets classique est basé sur l'envoi de messages, l'introduction des événements peut être intéressante. Le tableau ci-dessous résume les points forts des deux approches respectives :

Communication à l'aide d'événements :
La communication basée sur les événements permet la création d'applications client/serveur. Le concepteur n'a pas à se préoccuper des problèmes de séquencement liés à l'envoi de messages. L'utilisation des événements est plus appropriée lors de la phase d'analyse [Cook94] puisque plus intuitive et moins contraignante.
Communication à l'aide d'envoi de messages :
Cette approche est intéressante quand le concepteur veut garder le contrôle du séquencement des opérations. L'invocation directe d'opérations est plus efficace et plus simple. Le concepteur n'a pas à se préoccuper des problèmes de queue (overhead) et cette approche semble plus adaptée aux langages orientés objets [Harel96a].

1.6. Relations sur les statecharts

Le standard de modélisation UML propose trois mécanismes d'extension des statecharts, le raffinement, l'héritage et le sous-typage. Chacune de ces relations répond à des besoins particuliers.

1.6.1. RAFFINEMENT

Le raffinement est utilisé pour capturer les relations existantes entre deux statecharts. Il ne spécifie pas de contraintes particulières mais exprime quatre types de relations entre composants : la redéfinition, la substitution, l'ajout et la suppression.

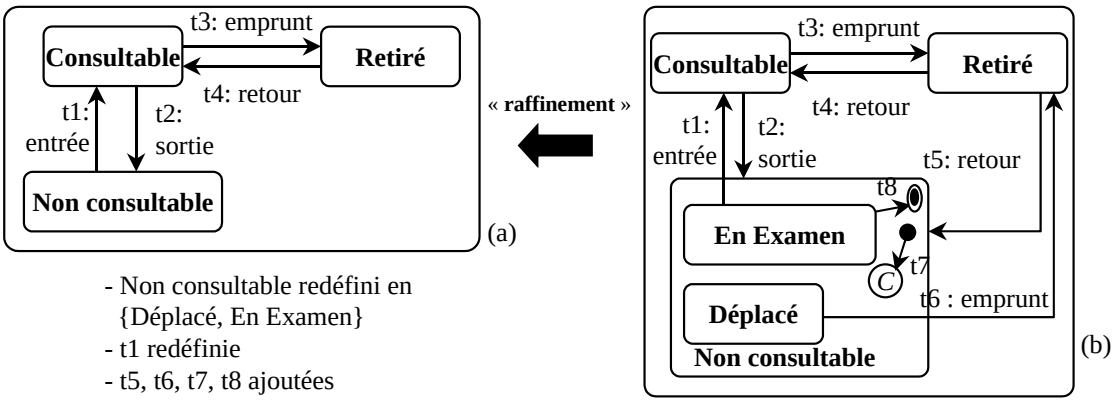


figure 1-20 : Raffinement entre statecharts

Le raffinement fournit un mécanisme flexible et sans contraintes pour modifier un statechart et spécifier les relations entre le statechart initial et le statechart modifié. Sur la figure 1-20, nous considérons maintenant qu'un livre peut être retiré du prêt pour la maintenance ou pour le transport d'une bibliothèque à l'autre. Le statechart (b) est alors obtenu par raffinement du statechart (a) pour prendre en compte les deux situations où l'ouvrage est retiré du prêt.

Les relations d'héritage et de sous-typage sont des relations de raffinement spécifiques au modèle à objets qui concernent aussi les statecharts (cf. § 2.2). Elles utilisent les mêmes opérations sur les composants des statecharts mais rajoutent des contraintes spécifiques.

1.6.2. HERITAGE

L'héritage encourage la réutilisation de code plus qu'il ne préserve le comportement. Les contraintes peuvent se résumer ainsi :

Les états et les transitions ne peuvent être qu'ajoutés ou redéfinis. Les règles de redéfinition sont interprétées comme ceci :

- Un état redéfini peut ajouter ou supprimer des transitions d'entrée mais ne peut qu'ajouter des transitions sortantes. De plus, il peut être décomposé en sous-états, co-occurrents ou non.
- Une transition redéfinie peut avoir un nouvel état cible mais doit conserver son état source.
- Un garde redéfini peut avoir une condition de garde différente.
- Une séquence d'actions peut contenir les mêmes actions dans le même ordre et peut en ajouter de nouvelles.

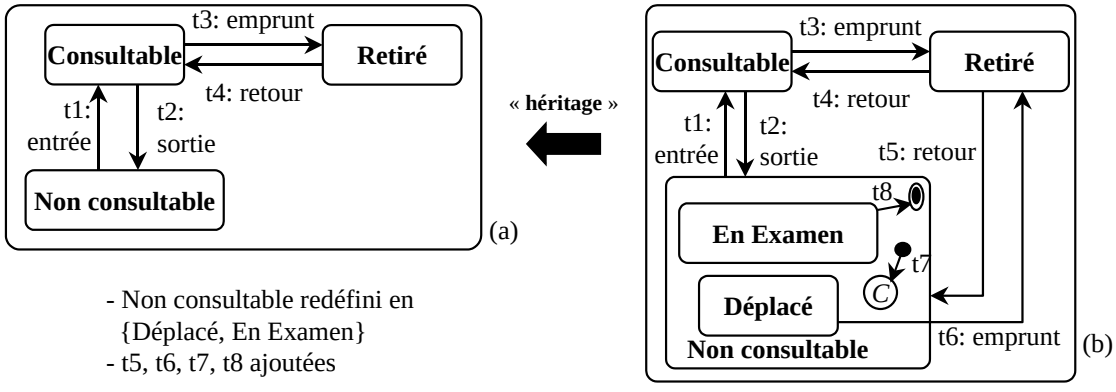


figure 1-21 : Héritage entre statecharts

D'après les règles ci-dessus, le statechart (b) de la figure 1-20 n'est pas une spécialisation du statechart (a) puisque l'état Déplacé n'hérite pas de la transition sortante t1 : l'héritage doit préserver le fait que tout exemplaire dans l'état Déplacé puisse ré-entrer dans le processus de

prêt et ainsi devenir Consultable . La figure 1-21 respecte les règles de redéfinition définies sur la relation d'héritage. Remarquons cependant que le comportement obtenu n'est pas très réaliste car il n'empêche pas la consultation et donc l'emprunt d'un exemplaire réservé (le garde de la transition t1 ne peut être qu'affaibli) .

1.6.3. SOUS-TYPAGE

La relation de sous-typage vise à conserver la conformité comportementale entre un statechart et son raffinement. Elle doit pour cela préserver les relations entre pre et post-conditions des événements/opérations sur le type (les pre et post-conditions sont réalisées par les états et les relations par les transitions). Une étude plus approfondie de la relation de sous-typage entre statecharts est proposée dans le chapitre 2.2. Nous en résumons ici les contraintes.

Les états et les transitions ne peuvent être détruits. Ils sont soit ajoutés, soit redéfinis. Les règles de redéfinition sont interprétées comme ceci :

- Un état redéfini a les mêmes transitions de sortie ; Il peut en ajouter des nouvelles. L'ensemble de ses transitions d'entrée peut être différent. Il peut être décomposé en sous-états co-occurrents ou non.
- Une transition redéfinie peut avoir un nouvel état cible qui est un sous-état de l'état cible dans le type de base de telle manière que la post-condition soit toujours garantie.
- Le nouveau « garde » ne peut qu'affaiblir la condition héritée.
- Une séquence d'actions contient les mêmes actions dans le même ordre. Elle peut en ajouter de nouvelles.

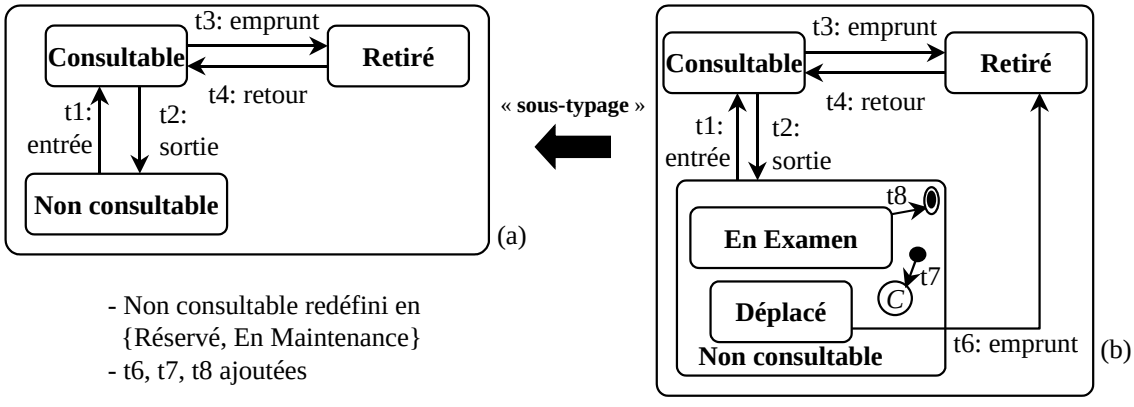


figure 1-22 : Sous-typage de statecharts

D'après ces règles de redéfinition, on s'aperçoit que le statechart (b) de la figure 1-21 ne peut être obtenu par sous-typage du statechart (a) et ce à cause de l'ajout de la transition t5. En effet, cet ajout n'a de sens que si le garde de la transition t4 est renforcé. Or le sous-typage interdit le renforcement de garde afin de conserver le séquencement des événements : d'après le statechart

(a), un livre dans l'état Retiré peut atteindre Non consultable par succession des événements retour et sortie. Cette séquence n'est pas respectée dans le statechart (b) de la figure 1-21 où le livre peut passer directement de l'état Retiré à l'état Non consultable par la transition t5.

Il est à noter que le sous-typage préserve le comportement en conservant le séquencement du statechart parent. Cependant la figure 1-22 montre que l'ajout de la transition t6 peut complètement changer le protocole malgré les contraintes de sous-typage, un ouvrage n'ayant plus besoin d'être Consultable pour être emprunté.

1.7. Conclusion

A la vue de cette partie, les statecharts constituent un bon modèle pour la représentation des comportements d'objets. L'abstraction des états et la factorisation des transitions permettent de conserver un bon rapport entre la complexité graphique des statecharts et celle de l'objet modélisé. De plus, les nombreuses décorations introduites par D. Harel permettent la représentation de comportements suffisamment complexes et proches de la réalité.

Nous avons vu que deux domaines se partagent les principaux articles sur les statecharts : les systèmes réactifs et les méthodes de conception à objets. Dans chacun d'eux, la problématique est pourtant différente. Le premier utilise plutôt les statecharts pour décrire la dynamique des systèmes en termes d'événements alors que le second les utilise pour contrôler l'intégrité des objets du système. Nous nous intéressons par la suite uniquement à l'utilisation des statecharts pour la modélisation de la dynamique des objets dans le contexte des systèmes d'information.

2 STATECHARTS OPERATOIRES VS. CLASSIFICATOIRES

La sémantique des statecharts introduite par D. Harel a été reprise de nombreuses fois et souvent modifiée en fonction du domaine d'application et des besoins spécifiques. Alors que le premier paragraphe présentait une vision assez large et générale de la sémantique des statecharts, nous nous intéressons ici plus particulièrement à l'usage des statecharts pour la modélisation à objets des systèmes d'information. En étudiant la bibliographie, on note deux tendances très nettes : d'un côté une approche " opératoire " et de l'autre une approche " classificatoire ". Ce chapitre a pour but de comparer ces deux tendances en insistant sur le caractère réutilisable des statecharts obtenus dans l'une et dans l'autre.

Nous retrouvons intuitivement la dualité qui existe pour l'objet entre les données et le comportement. Cette dualité a été soulevée dans [D'Souza95] à propos des méthodes de conception. En effet, on distingue généralement les méthodes dirigées par le comportement (CRC, Booch, OBA, etc.) de celles qui sont dirigées par les données (OMT, Shlaer & Mellor). Cette dualité semble créer un schisme dans la communauté objet entre ceux qui mettent les données au second plan sur la base que les services qui constituent le comportement de l'objet sont plus fondamentaux et ceux qui trouvent difficiles de s'abstraire ou des données pour décrire un comportement.

Pour définir un comportement, on peut s'intéresser à la structure interne d'un objet et penser les objets en termes d'états dès les phases d'analyse : un livre est emprunté ou en rayon. Au moment de la conception et de l'implantation, on peut raffiner cette vision en définissant ces états par rapport aux valeurs prises par l'objet. C'est l'approche classificatoire.

D'un autre côté, on peut se demander quels sont les services rendus par un objet et chercher ensuite à contraindre les séquences d'appels qui sont réalisés sur cet objet : une première étude fait apparaître qu'un livre peut se consulter, s'emprunter et se rendre. Son comportement peut

ensuite être décrit plus précisément comme une séquence d'emprunts et de retours successifs. C'est l'approche opératoire.

Ce chapitre tente de démontrer que la perception du problème, selon qu'elle est classificatoire ou opératoire, influe sur l'usage des statecharts.

2.1. Interprétation des états

2.1.1. ETAT CLASSIFICATOIRE

2.1.1.1. Définition

Etat [Panet94] : Un état d'objet est un stade transitoire par lequel passe un objet au cours de son cycle de vie. Cet état est fonction de :

- la valeur des propriétés de l'objet : " Un élève est dit *recalé* si sa note moyenne est inférieure à 10/20 ".
- la valeur des propriétés des objets qui lui sont reliés : " Un client est dit *suspendu* si le solde d'un de ses comptes est négatif depuis plus de cinq jours ".
- la valeur des états des objets qui lui sont reliés : " Un client est dit *inactif* s'il n'existe pas pour cette commande d'occurrence de relation avec une livraison ".
- ses relations ou de son absence de relation avec d'autres objets : " Une commande est dite *en attente* s'il n'existe pas pour cette commande d'occurrence de relation avec une livraison ".
- la valeur des propriétés de ses relations avec d'autres objets : " Un produit est dit *en rupture* de stock dans un dépôt si la quantité en stock est inférieure au seuil critique ".

Cette définition de l'état d'objet a l'avantage d'être formalisable si on considère que les relations entre objets ne sont que des attributs particuliers. Une classe est alors modélisée par le produit cartésien de ses attributs. Un état peut être vu comme un vecteur de valeurs contenant une entrée pour chaque attribut de l'objet. La définition d'un état est bien sûr basée sur le comportement observable d'un objet :

Un état est un ensemble de vecteurs de valeurs observables partagées par des attributs comportementaux intéressants [McGregor93].

En utilisant ce postulat, [LeGrand98] donne une définition formelle de l'interprétation des états d'un Statechart :

Pour toute classe c on définit S_c l'ensemble des états du cycle de vie de c.

Soient les n attributs $A_{c,1}...A_{c,n}$ d'une classe c où le domaine de Ai est Dc,i. L'état d'un objet de c est défini par les valeurs du vecteur $v=(a_1...a_n)$ où $A_i \in D_{c,i}$ et a_i est la valeur courante de $A_{c,i}$.

Soit $\Pi_{c,n}$ le produit cartésien des domaines $D_{c,1}$ x...x $D_{c,n}$, l'interprétation Ic d'un état de Sc est un sous-ensemble de $\Pi_{c,n}$, et est formellement défini par :

$I_c : S_c \rightarrow 2^{\Pi_{c,n}}$

Si s est un état terminal, alors $I_c(s) \subseteq \Pi_{c,n}$

Si s est un état-OU alors $I_c(s) = \bigcup_{x \in \text{enfants(s)}} I_c(x)$,

avec $\forall(x,y) \in \text{enfants(s)} \times \text{enfants(s)} \quad x \neq y \Rightarrow I_c(x) \cap I_c(y) = \varnothing$

Si s est un état-ET alors $I_c(s) = \bigcap_{x \in \text{enfants(s)}} I_c(x)$

2.1.1.2. Recouvrement et complétude

Dans cette approche, l'état r racine du cycle de vie de c est tel que $I_c(r) = \Pi_{c,n}$. Il couvre donc complètement le produit cartésien des valeurs prises par les attributs de la classe c. L'ensemble des enfants (sous-états) de r forme une partition de r (figure 2-1). Dans notre application, un livre peut par exemple avoir de 0 à n ouvrages en rayon et donc avoir potentiellement n+1 états (un état par ajout d'ouvrage). Du point de vue de la conception, nous nous sommes jusqu'à présent uniquement préoccupé du fait qu'il y avait des exemplaires disponibles (0<nbOuvragesDisponibles<n), aucun exemplaire (nbOuvragesDisponibles = 0), ou bien qu'ils étaient tous présents (nbOuvragesDisponibles = 0) en rayon. Plutôt que de représenter les n+1 états d'un livre, les statecharts permettent par abstraction de n'en considérer que trois.

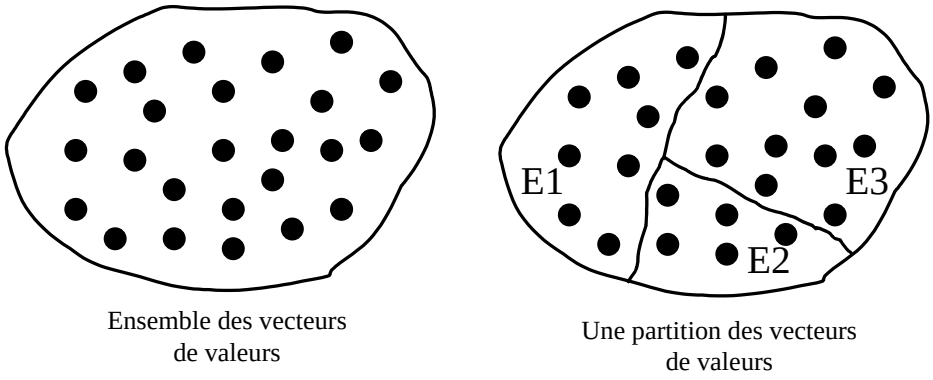


figure 2-1 : Approche classificatoire et représentation des états [McGregor93]

Ici, la description des états est fortement liée à la structure de données et ne prend pas en compte la dynamique réelle de l'objet : nous pensons que les valeurs prises par un objet dépendent évidemment du domaine de chacun des attributs mais aussi des méthodes qui modifient ces valeurs de telle manière qu'on ait en réalité souvent $I_c(r) \subseteq \Pi_{c,n}$.

La distinction entre les états potentiels et les états effectivement atteints ainsi que le cycle de vie de l'objet est à notre sens prise en compte par les états opératoires.

2.1.2. ETAT OPERATOIRE

2.1.2.1. Définition

Etat [UML97] : une condition ou une situation de la vie d'un objet qui satisfait certaines conditions, accomplit certaines activités ou est en attente de certains événements.

- Un état correspond à un intervalle entre deux événements reçus par un objet, " Après le décrochage du combiné et avant la composition du premier chiffre, une ligne de téléphone est dite dans un état de *tonalité* ". L'état d'un objet dépend des séquences d'événements qu'il a reçus.
- Un état a une durée ; il occupe un intervalle de temps. Un état est souvent associé à une activité continue, " le téléphone sonne ", ou à une activité qui prend un certain temps pour se terminer, " voler de Grenoble à Paris ".
- Un état est souvent associé à la valeur d'un objet satisfaisant une condition, " l'eau est dite dans l'état *liquide* si la température de l'eau est comprise entre 0°C et 100°C ".

Lors de la définition des états, l'approche opératoire ignore les attributs qui n'affectent pas le comportement de l'objet. L'objectif est la représentation d'un comportement en faisant abstraction de certaines propriétés de l'objet. La complétude du point de vue des données n'est donc pas ici un critère de qualité du Statechart. En effet, celle-ci semble difficile à maintenir dans un système qui manipule des objets complexes : la solution proposée par les modèles opératoires consiste à ne considérer que les valeurs d'attributs qui semblent judicieuses pour le problème à modéliser.

La tendance opératoire a donné naissance à un nouveau type d'état appelé état-action [UML97]. Ce dernier représente l'exécution d'une action atomique, typiquement l'appel d'une opération. Il est équivalent à un état anonyme dont la spécification est un unique événement d'entrée. Ce type d'état est utile pour représenter la dynamique de l'objet en termes d'activités à réaliser ; il est à la base du modèle d'activités introduit dans le langage de modélisation unifié. Le modèle d'activité se présente comme une variation d'une machine à états où la majorité des états sont liés à la réalisation d'une activité et où la majorité des transitions est déclenchée par la complétion de cette activité [UML97].

A l'aide d'un unique formalisme graphique, UML a donc bâti deux modèles dont la seule différence peut se résumer à une restriction sur les états : un diagramme d'activités est un diagramme d'états qui a une majorité d'états-action. Cette double définition des états était déjà pressentie dans la méthode OMT [Rumbaugh95] où les exemples de diagrammes d'états utilisent indifféremment des états et des états-action.

2.1.2.2. Cohérence

La cohérence entre les états et la structure de l'objet est généralement assurée par des classes-état. Ces dernières constituent un pont entre la structure logique du modèle à objet et la nature dynamique des modèles à états [Icon97a]. L'idée de départ " est en réalité très simple : si le comportement est vraiment différent d'un état à l'autre, alors la structure logique l'est sûrement " [D'Souza92]. On retrouve cette approche dans les méthodes Syntropy [Cook94] et Catalysis [Waldén94].

Tous les états d'un Statechart doivent avoir un équivalent sous la forme d'une classe-état dans le modèle structurel. L'imbrication des états est décrite à l'aide de l'héritage entre classes. La règle est que toutes les classes-état connectées au même super-type sont exclusives entre elles. C'est par exemple le cas pour un livre qui peut se trouver dans l'état En Examen, Consultable, Retiré ou En Retard du point de vue de l'Emprunt (figure 2-2).

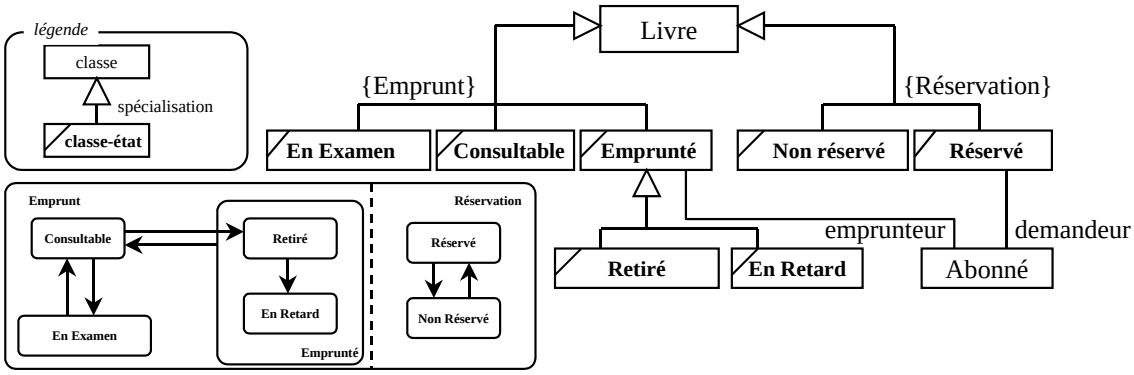


figure 2-2 : statecharts et classes-état

La relation entre les deux modèles est triviale : on associe à un état A une classe-état cA et aux sous-états de A des sous-classes-états de cA . Elle devient intéressante lorsqu'on exprime des contraintes sur les états orthogonaux. Par exemple, {**in** Consultable ⇒ **not in** Réservé**}** est un invariant qui peut être exprimé à la fois sur le Statechart et sur le modèle à objet.

2.1.2.3. Recouvrement et complétude

La complétude est exprimée sur les événements. Un Statechart doit décrire tous les événements invocables sur un objet. Si nous reprenons l'exemple du livre dont nous disposons n ouvrages et que nous considérons que la maintenance d'un livre ne peut se faire que de manière globale, lorsque tous les ouvrages sont en rayon, celui-ci a deux états significatifs, tousEnRayon et aucunExemplaire, qui ne couvrent pas les états potentiellement atteints. L'approche opératoire fait la différence entre les états potentiellement atteints et ceux qui le sont réellement au vu de la dynamique des objets.

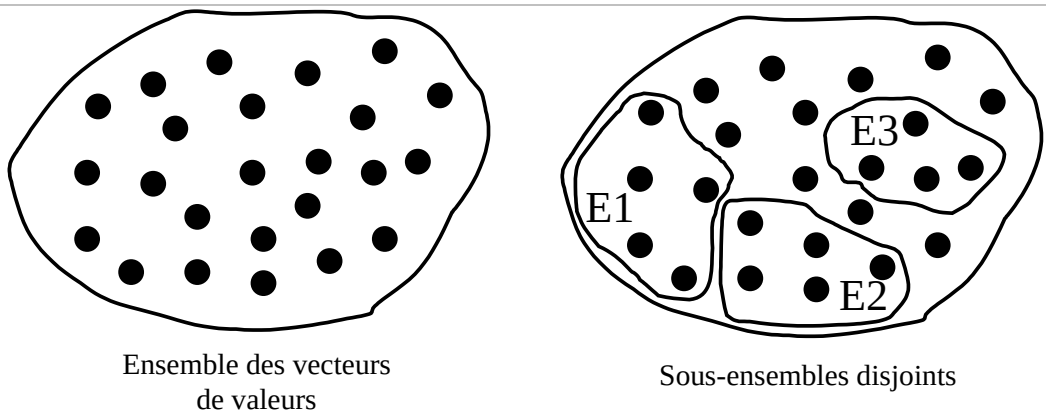


figure 2-3 : Approche opératoire et représentation des états

L'approche classificatoire s'attache à décrire les états potentiellement atteints par un objet alors que l'approche opératoire s'intéresse aux états effectivement atteints.

2.1.3. ETATS ET REUTILISATION

Dans l'ensemble des méthodes de conception, la notion d'états et plus particulièrement de diagramme d'états apparaît au moment de la conception. Leur réutilisation est peu abordée voir inexistante dans les méthodes actuelles. En effet, peu de méthodes intègrent les spécifications à base de diagrammes d'états dans le processus de développement du logiciel et à plus forte raison s'intéressent à leur réutilisation. On sait que les efforts de réutilisation dans les méthodes de conception actuelles ne portent plus uniquement sur l'objet (ou sur la classe) mais s'adressent à des structures plus grosses d'objets collaborants (cf. chapitre 2 § 1.2). La granularité de cette solution constitue donc un frein important pour la réutilisation des statecharts.

Seule l'approche « opératoire » propose une solution pour implanter les états dans les modèles à objets en utilisant des classes-état. De plus, les classes-état sont souvent utilisées pour réutiliser des méthodes. Leur réutilisation se limite à une réutilisation de code alors que les statecharts offrent une vision plus conceptuelle et permettent d'envisager une réutilisation dès la conception.

2.2. Sous-typage de statecharts

Quelle que soit l'approche, classificatoire ou opératoire, les statecharts sont toujours associés à leurs classes respectives. Leur intégration dans un monde à objets pose évidemment le problème de leur coexistence avec les opérateurs sur les classes que sont l'héritage et la composition (figure 2-4). Si la première est largement traitée dans la littérature, on retrouve peu d'articles qui traitent de la composition et des cycles de vie [Brunet98]. C'est pourquoi nous nous intéressons plus aux différentes manières de sous-typer les statecharts. Ce choix s'explique aussi par notre volonté de réutiliser les statecharts, le sous-typage étant un opérateur naturel pour y parvenir.

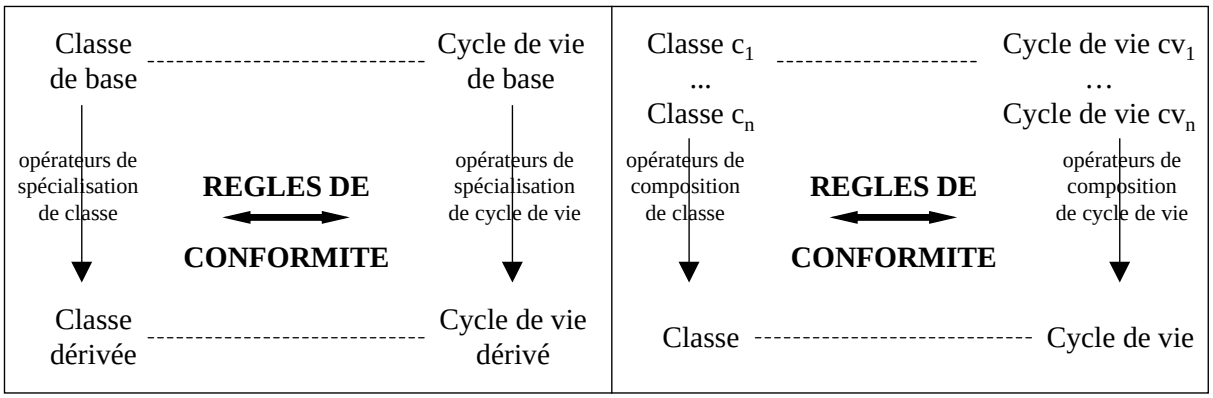


figure 2-4 : Conformité entre modèle de classes et de cycles de vie [Legrand97]

Si la plupart des méthodes insistent sur l'importance de pouvoir substituer un sous-type à son super-type, peu adressent le problème de la conformité comportementale. Alors que les règles de sous-typage sont aujourd'hui consensuelles, il n'en va pas de même en ce qui concerne le sous-typage comportemental. Beaucoup de livres omettent encore les règles qui allient sous-typage et statecharts. Par exemple, Shlear & Mellor évitent le problème et préconisent d'associer les diagrammes d'états soit à un super-type, soit à un sous-type [Shlaer92]. Cependant si tous les sous-types partagent un même comportement, il peut être factorisée dans le super-type pour en améliorer la maintenance. De la même manière, dans [Rumbaugh95], le sous-typage est ramené dans le cas le plus courant à l'ajout de diagrammes d'états orthogonaux au diagramme d'états du super-type.

Nous montrons que là encore, le sous-typage de diagrammes d'états peut être vu de deux manières : dans l'approche classificatoire, le sous-typage préserve les données de l'objet ; dans l'approche opératoire, le cycle de vie est vu comme une séquence d'événements qu'il faut préserver.

2.2.1. SOUS-TYPAGE CLASSIFICATOIRE

2.2.1.1. Conformité ascendante

Contrainte

Le cycle de vie d'un type T est conforme au cycle de vie de son super-type T', en ascendant, si tous ses états peuvent être remplacés dans le cycle de vie de T' par un état plus général.

Autrement dit, cette contrainte forte implique qu'un état introduit dans un Statechart dérivé doit être complètement contenu dans un des états du Statechart parent [McGregor93]. En prenant en compte la définition des états donnée dans le § 2.1.1, [Legrand97] donne une propriété nécessaire pour établir la conformité ascendante, appelée « conservation de la sémantique des états » :

Soit d une classe dérivée de la classe c. La spécialisation permet l'ajout dans d de nouveaux attributs et/ou la restriction du domaine des attributs existants dans c.

La projection $p_{d,c}$: Π_m --> Π_n est une fonction totale qui assigne à chaque vecteur de valeurs, le vecteur de ses n premiers éléments. Cette projection est étendue à un ensemble X de vecteurs d'états : $p_{d,c}(X) = \bigcup_{x \in X} p_{d,c}(x)$.

La propriété de conservation de la sémantique des états implique que pour tout état s de d, on a :

$\forall s \in S_c \quad p_{d,c}(I_d(s)) \subseteq I_c(s)$

Quand une classe dérivée est obtenue sans restriction du domaine des attributs, la propriété devient:

$\forall s \in S_c \quad p_{d,c}(I_d(s)) = I_c(s)$

2.2.1.2. Conformité descendante

Contrainte

Le cycle de vie d'un type T est conforme au cycle de vie de son super-type T', en descendant, si le déclenchement sur un objet de type T, de toute transition héritée du cycle de vie de T', rend conforme à l'interprétation le nouvel état de destination de cette transition dans le cycle de vie de T.

Peu d'approches se sont intéressées à la conformité descendante en termes de données. On en trouve une bonne définition dans [LeGrand98].

Une transition est définie ici par un tuple (X, l, Y) de S_c x L x S_c, où X est l'état source, l le label et Y l'état cible de la transition . Il existe deux fonctions totales $source_c$: S_c x L x S_c et $destination_c$: S_c x L x S_c qui définissent respectivement pour toute transition :

- son état source tel que source((X,l,Y))=X,
- son état cible tel que destination((X,l,Y))=Y.

On peut dire qu'une transition t d'un cycle de vie de la classe c est conforme en descendant dans le cycle de vie de d, une sous-classe de c, si et seulement si, pour toute instance obj de la classe d :

si $v_d(obj) \in I_d(source_c(t))$ et t est franchie, alors $t(v_d(obj)) \in I_d(defaut_d(destination_c(t)))$

Où $v_x(obj)$ est le vecteur de valeurs courant d'une instance obj de classe x, et $t(v_x(obj))$ est le nouveau vecteur après franchissement de la transition t, la fonction totale $defaut_c$: $S_c \rightarrow 2^{S_c}$

définissant pour un état ses états par défaut. L'interprétation I_c (cf. § 2.1.1.1) est étendue pour prendre en compte un ensemble d'états : $I_c(X) = \cup_{x \in X} I_c(x)$.

2.2.2. SOUS-TYPAGE OPERATOIRE

Un Statechart décrit les cycles de vie d'un objet, i.e. toutes les séquences possibles d'appels de méthodes qui peuvent être invoquées sur cet objet. Dans l'approche opératoire, le sous-typage tente de préserver ces séquences d'appels sans se préoccuper de la définition des états. Entre types et sous-types, la conformité comportementale est vue là aussi de deux manières, en ascendant et en descendant.

2.2.2.1. Conformité ascendante

Contrainte

Le cycle de vie d'un type T est conforme au cycle de vie de son super-type T', en ascendant, si la restriction du cycle de vie de T aux opérations définies pour le type T' est un cycle de vie autorisé par T'.

Cette approche reprend l'idée que les objets instances d'une sous-classe doivent aussi se comporter comme les instances de leur super-classe et donc que tout comportement observé dans une sous-classe doit être conforme à celui qu'il aurait en tant qu'instance de sa super-classe :

Si OS(C) désigne l'ensemble des séquences d'événements observables pour une classe C, nous avons pour toutes les super-classes C' de C :

$$\Pi\,(OS(C)) \subseteq OS(C')$$

où Π (OS(C)) est la projection des séquences appartenant à OS(C) sur les événements définis dans C'.

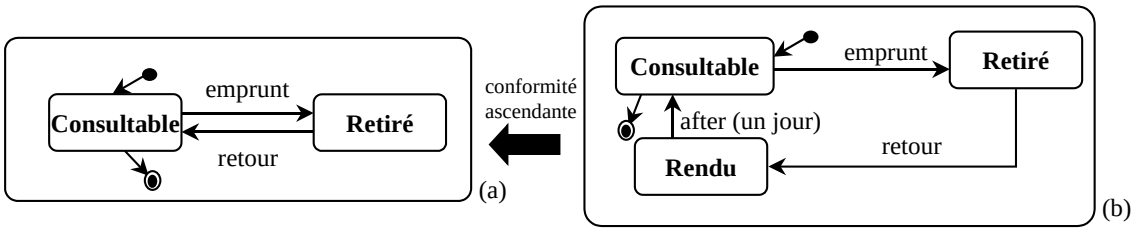


figure 2-5 : Conformité ascendante

Il y a exactement deux événements observables dans le Statechart (figure 2-5 a) dont les séquences autorisées se résument à une suite d'emprunt et de retour qui peut être décrite par le langage L1=(emprunt.retour)*. Le Statechart b) réagit à un nouvel événement qui fixe un délai d'un jour avant que le livre soit à nouveau Consultable. Ce délai prend en compte la mise en rayon qui se fait à la fin de la journée par les bibliothécaires. Les nouvelles séquences pour le Statechart b) peuvent être décrites par le langage L2=(emprunt.retour.unJour)*. La projection de ces séquences

d'événements sur les événements observables dans le Statechart a) peut être décrite à l'aide du langage L1. Le Statechart b) est donc conforme au Statechart a) en ascendant.

2.2.2.2. Conformité descendante

Contrainte

Le cycle de vie d'un type T est conforme au cycle de vie de son super-type T', en descendant, si la séquence d'événements décrite par le cycle de vie de T' est invocable par tout objet de type T.

On peut interpréter un Statechart comme un contrat passé avec des clients. Celui-ci sert alors à décrire tous les services invocables dont le client peut se servir et dont on garantit l'exécution.

Toute séquence invocable dans un Statechart doit l'être dans tous les sous-statecharts.

Comme chaque objet qui appartient à une sous-classe peut-être vu comme appartenant à sa super-classe, il doit aussi avoir les garanties que les méthodes de la super-classe (ainsi que leur séquencement) sont encore invocables :

Si IS(C) représente l'ensemble de tous les événements invocables sur une classe C et si C est sous-classe de C',

$$IS(C') \subseteq IS(C)$$

Les statecharts a) et b) de la figure 2-5 ne sont donc pas conformes en descendant. En effet, une suite d'événements emprunt et retour, reconnue par le Statechart a) n'est pas reconnue par le Statechart b). Par contre, les statecharts c) et d) de la figure 2-6 sont conformes en descendant. Toutes les séquences d'événements invocables dans le Statechart c) le sont aussi dans le Statechart d). En effet, les séquences d'événements autorisées par les statecharts c) et d) peuvent être décrites respectivement par le langage L3 = (emprunt.retour|sortie.entrée)*.sortie et L4 = (emprunt.retour|sortie.entrée|sortie.emprunt.retour)*.sortie.

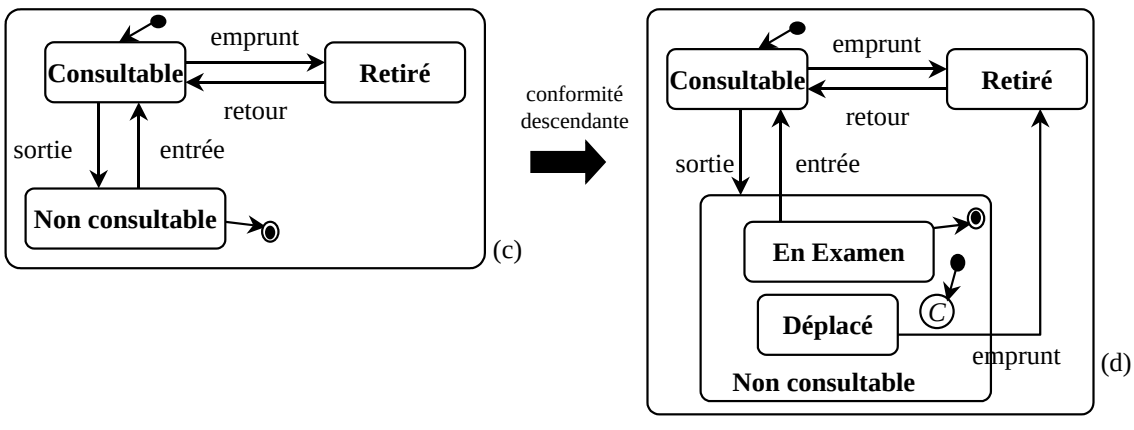


figure 2-6 : Conformité descendante

On remarque cependant que la conformité ascendante n'est pas respectée par le Statechart d). En effet, la séquence d'événements {sortie, emprunt, retour, sortie} autorisée dans d) n'est pas valide dans c). Pour avoir à la fois une conformité ascendante et descendante, il aurait fallu prévoir l'ajout du déplacement et particulièrement la possibilité d'emprunter un livre qui n'est pas consultable (figure 2-7). Les conformités ascendante et descendante offrent donc un cadre rigide pour le sous-typage de statecharts et ne se prête pas à une réutilisation facile des statecharts.

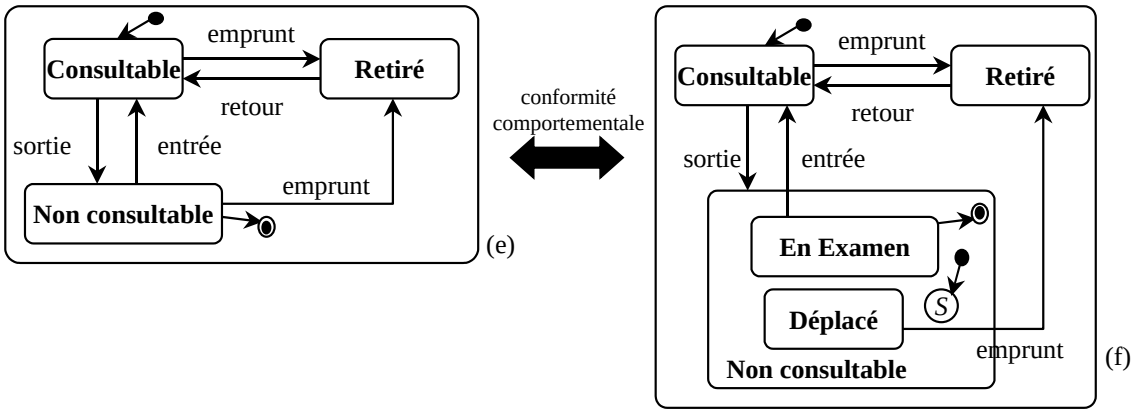


figure 2-7 : Conformité comportementale totale

2.2.3. CONFORMITE ET OPERATEURS DE SPECIALISATION

Dans le paradigme objet, le sous-typage modifie le comportement d'un objet à la fois par extension et par raffinement. La première concerne l'ajout de propriétés (structurelles : attributs et méthodes /comportementales : états et transitions) et la seconde leur redéfinition. Chaque approche doit donc concilier les principes théoriques énoncés dans les parties précédentes sur la conformité comportementale et les possibilités offertes en termes d'héritage d'objets. [Legrand97] distingue dans la littérature deux manières de réaliser cette conciliation, soit en définissant des opérateurs de sous-typage, soit en établissant des règles de dérivations de cycles de vie. Dans tous les cas, il s'agit d'établir un compromis entre la conservation du comportement et la rigidité des contraintes mises en place : il faut trouver les contraintes de sous-typage qui préservent au mieux le comportement et qui ne nuisent pas à l'extension des statecharts entre sous-types. Celles-ci diffèrent selon que l'on a une vision ascendante ou descendante du sous-typage. Le tableau ci-dessous résume les principales recommandations pour chaque approche.

Conformité	**redéfinition des transitions**	**Redéfinition des états**	**Suppression**	**ajout de transitions**	**ajout d'états**
Descendante	pré-condition relâchée post-condition renforcée	Non	Non	Oui	Oui
Ascendante	pré-condition peut être renforcée	Décomposition en sous-états	Des états et des transitions	Possible sous certaines réserves	Uniquement parallèles

Nous détaillons ci-après les recommandations de sous-typage de statecharts introduites plus spécifiquement dans la méthode Syntropy [Cook94]. Cette dernière constitue à notre avis l'approche « opératoire » la plus aboutie actuellement en matière de sous-typage de statecharts. Elle prend notamment en compte l'héritage multiple de statecharts (traité partiellement par J. D. McGregor [McGregor93]). Seules les principales recommandations sont reprises ici (pour des raisons de clarté, nous éludons les règles spécifiques aux concepts de la méthode Syntropy) :

1. Quand un sous-type est défini, les statecharts des super-types sont complètement hérités. Par défaut, le Statechart du sous-type est la combinaison orthogonale des statecharts de ses super-types.

2. L'ajout d'un Statechart qui a des états de noms différents de ceux des super-types est par défaut orthogonal et dispose de sa propre liste d'événements.

3. On peut étendre la définition d'un état en réutilisant les noms d'états des super-types. Les états des super-types qui apparaissent dans le Statechart du sous-type sont appelés états étendus. Dans le sous-type, on peut ajouter aux états étendus de nouveaux états et transitions, redéfinir des transitions existantes, ajouter à l'intérieur des états imbriqués et des machines concurrentes. La cible de l'état de départ ne peut pas être modifiée.

4. Un événement avec une pré-condition ou des gardes incomplets (i.e. un ensemble de transitions gardées dont la somme des gardes n'est pas vraie) peut être redéfini dans un sous-type en appauvrissant la pré-condition ou les gardes. On élargit ainsi l'ensemble des conditions qui font qu'une transition peut être franchie.

5. Les transitions peuvent être redéfinies dans un sous-type par reciblage et par division. Une transition reciblée est redéfinie comme entrant dans un des sous-états de son état d'origine. Une transition peut être divisée en deux ou plus transitions qui sont déclenchées au moins dans les mêmes conditions. On a deux sortes de division, par la source et par la cible des transitions.

6. Les transitions peuvent être redéfinies par renforcement des post-conditions. Si une transition est divisée, la post-condition s'applique à toutes les transitions résultantes. De la même manière, les invariants d'états sont renforcés dans les sous-type par ajout de termes. La plupart du temps, cela est réalisé par insertion d'invariants dans les sous-états ajoutés au sous-type. L'invariant d'un état doit être consistant avec les gardes et post-conditions de toutes les transitions entrantes.

8. Les générations d'événements (actions) portées par les transitions peuvent être redéfinies dans le sous-type pour générer des séquences d'événements différents des transitions du super-type. On peut faire appel aux générations du super-type par un appel direct du super.

9. Toutes les générations d'événements d'entrée et de sortie d'un état s'appliquent automatiquement dans le sous-type. Elles peuvent être redéfinies en suivant les mêmes règles que précédemment.

10. Les états finaux ne sont pas des vrais états au sens de l'héritage. La cible de transitions qui se terminent par un état final peut-être redéfinie. On ne conserve que la post-condition qui est héritée et doit être consistante avec la nouvelle cible.

En résumé, pour sous-typer on laisse par défaut les statecharts des super-types inchangés ou on les remplace par une description complète des états et transitions, en s'assurant que la structure d'états est conforme à celle du super-type : une réorganisation des états dans le sous-type est conforme structurellement si les états du super-type sont encore mutuellement exclusifs dans la nouvelle configuration.

2.3. Conclusion

La conformité ascendante garantit que tout cycle de vie d'une instance du sous-type peut être vu par projection (des événements ou des états) comme un cycle de vie du super-type. La conformité descendante garantit que toutes les séquences d'événements invocables (respectivement tous les états atteignables) par les instances d'un super-type peuvent être invoquées (respectivement atteints) par les instances d'un sous-type.

De nombreux articles insistent sur la nécessité de préserver l'une ou l'autre de ces conformités et proposent leurs propres règles et opérateurs théoriques pour y parvenir. Nous avons volontairement distingué dans ce chapitre deux approches complémentaires de la conformité et en avons donc donné plusieurs définitions, orientées données d'une part (approche classificatoire) et événements de l'autre (approche opératoire). Cependant, et pour reprendre les paroles de [Cook94], la conformité reste un idéal : il est facile de montrer qu'aucune recommandation ne peut prévenir un changement radical de comportement d'un statechart à l'autre et ainsi garantir une conformité totale.

De plus, le respect strict de la conformité limite les possibilités de réutilisation des statecharts, leur conception étant rendue plus difficile. De par les contraintes qu'il impose, la définition de statecharts en conformité reste un art. Dans [Fowler97a], l'auteur insiste sur le fait qu'il est parfois nécessaire de modifier un Statechart pour pouvoir l'étendre en conformité. Or , la nécessité de prendre en compte les futures extensions va à l'encontre même des principes objets. C'est pourquoi le standard de modélisation propose une relation d'héritage entre statecharts (cf. § 1.6.2) qui encourage la réutilisation de code plus qu'elle ne préserve le comportement. Cette relation a été préférée par exemple dans [Harel96a] pour réutiliser des parties de code générées en C++ à partir de statecharts.

Aujourd'hui et ce malgré les recommandations données dans le langage de modélisation unifié (UML), il n'existe pas d'approches formelles qui soit consensuelle et qui traite complètement le sujet.

CHAPITRE 2

Suivi des évolutions d’objets :les Rôles

- C’est une victoire, n’est-ce pas ?
- C’est une thèse.

Pyrus187

Les comportements d'objets sont généralement plus difficiles à spécifier lorsqu'ils sont dépendants du rôle joué par ces mêmes objets. C'est ainsi qu'une personne peut jouer simultanément différents rôles (appelés aussi perspectives ou points de vue) dans un système ; rôles auxquels elle peut librement adhérer sans pour autant changer d'identité. Ainsi, une personne peut jouer plusieurs rôles simultanément selon qu'elle est employée, étudiante, cliente, etc. De manière générale, le rôle est utilisé pour ne parler que d'une partie des propriétés statiques, dynamiques et comportementales des objets. Ce concept a l'avantage d'être intuitif pour les utilisateurs qui ne voient souvent les objets qu'au travers des rôles pris dans les processus du système : dans une bibliothèque, un livre joue plusieurs rôles correspondant à l'emprunt , la réservation, la maintenance, etc. Certaines méthodes de conception ont bien compris l'intérêt des rôles et l'utilisent comme pivot de leur démarche. La partie 1 présente ces méthodes en insistant sur les deux critères qui sont leurs points forts et que nous voulons préserver : la réutilisation et la traçabilité des spécifications. Nous voyons que ces dernières ne portent pas ou peu sur le type de comportement individuel qui nous a intéressés dans le premier chapitre.

C'est pourquoi dans la partie 2 nous ciblons notre étude sur le domaine particulier des patrons de conception. Nous voyons que ce dernier constitue un lien tangible entre toutes les parties de cet état de l'art. En effet, il est le réceptacle de tous les problèmes fréquemment soulevés lors de l'utilisation des états et des rôles pour la modélisation des systèmes d'information. Nous pensons que ces deux concepts peuvent être utilisés conjointement pour modéliser des comportements multiples et évolutifs. En effet, lorsqu'un objet intervient et évolue dans plusieurs processus, il admet des comportements mais aussi des évolutions différentes en fonction du rôle joué dans ces processus. C'est ainsi qu'un objet livre joue un rôle et a une évolution différente dans un processus de prêt, dans celui de réservation et dans celui de maintenance. De la même manière, une personne a une évolution différente dans son rôle familial, dans son rôle professionnel, etc. Dans ce cas, il s'agit de gérer des évolutions multiples, on parle également d'évolutions co-occurrentes (ou concurrentes) d'objets.

1. METHODES DE CONCEPTION « CENTREES COMPORTEMENT », INTRODUCTION AUX ROLES

Deux caractéristiques semblent faire défaut aux spécifications comportementales obtenues à l'aide de statecharts : d'une part, la modélisation du comportement d'objet est tardive dans la démarche et d'autre part elle se fait pratiquement toujours « à partir de rien ». Or les méthodes de conception modernes s'orientent aujourd'hui vers une conception « sans couture » où la traçabilité entre modèles est préservée et où la réutilisation est encouragée à toutes les étapes du cycle de développement du logiciel. Ces deux aspects sont devenus le cheval de bataille de nombreuses méthodes (Catalysis [D'Souza98], B.O.N [Waldén94], OORAM [Reenskaugh92]) dans lesquelles le concept de rôle se présente souvent comme un dénominateur commun. Ces approches sont dites « dirigées par le comportement » par opposition à celles qui sont centrées sur les données, i.e. sur la modélisation statique des objets et de leurs relations (OMT, Schlear&Mellor). Au vu de notre problématique, nous avons été amenés à privilégier les approches dirigées par le comportement.

Ces méthodes utilisent des modèles comportementaux basés sur la description des interactions du système. Les interactions offrent une granularité plus grosse que les statecharts ainsi qu'une vision de la dynamique plus proche des utilisateurs, ce qui les rend facilement appréhendables par les utilisateurs et les concepteurs du système. Elles décrivent en premier lieu les fonctionnalités du système à la manière de la méthode Merise (cf. chapitre 1 § 1.1.4) mais en tirant pleinement partie du paradigme objet. Par raffinements successifs, la description des interactions permet d'obtenir un schéma de classes qui correspond au mieux aux besoins fonctionnels.

Le premier paragraphe présente les modèles qui sont devenus le standard pour représenter de telles interactions. Il présente par la même la démarche CSO inspirée d'Objectory [Rational97]. Celle-ci a été affinée pour les besoins d'un cours de conception des systèmes à objets donné en première année de cycle universitaire [CSO99]. L'objectif n'est pas ici de couvrir complètement

l'ensemble des méthodes « centrées comportement » mais plutôt d'introduire les principes qui dirigent ce type démarche.

Le paragraphe 1.2 présente plus spécifiquement les dernières avancées en termes de réutilisation ainsi que leur intégration dans les méthodes dirigées par le comportement.

1.1. Une démarche dirigée par le comportement

La démarche CSO repose sur un enchaînement classique de trois étapes fondamentales : l'expression des besoins dont l'objectif est une mise en accord entre les concepteurs et les futurs utilisateurs du système, l'analyse qui permet d'exprimer les connaissances du domaine, les contraintes et objectifs du système indépendamment de toute technologie et enfin la conception dernière étape de spécification qui propose des solutions en fonction d'une technologie cible en l'occurrence la technologie objet. La figure 1-1 illustre ces trois étapes ainsi que les modèles UML préconisés dans chacune d'elle.

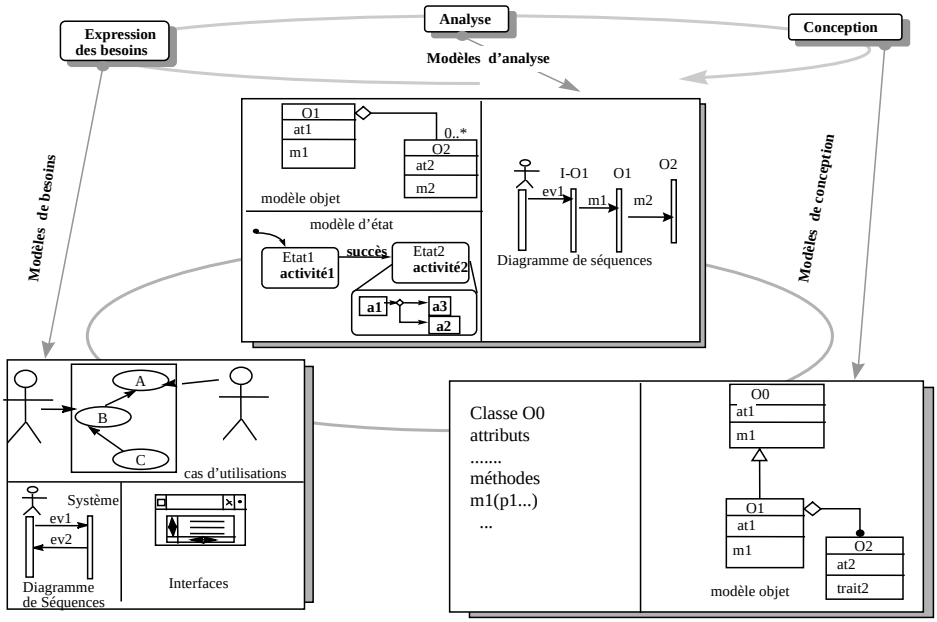


figure 1-1: Démarche "dirigée par le comportement"

La première phase, avant même la phase d'analyse, est couramment appelée phase de capture ou d'expression des besoins utilisateurs. La technique la plus utilisée pour capturer ces besoins est celle des cas d'utilisation [Jacobson93]. Cette dernière est envisagée selon deux angles, celui des experts du domaine, souvent non spécialistes en informatique, et celui des ingénieurs des besoins dont le but est de faire évoluer les besoins acquis vers des besoins compris et formalisés. Un reproche souvent formulé à propos des représentations utilisant les cas d'utilisation est leur manque de caractère formel [Övergaar98]. Il semble difficile en effet de privilégier à la fois le

dialogue entre utilisateurs et concepteurs du système et une formalisation accrue. Des recherches récentes proposent néanmoins des formalisations du modèle des cas d'utilisation [Dano97][D'Souza98] qui se paient souvent au prix fort en ce qui concerne la lisibilité du résultat.

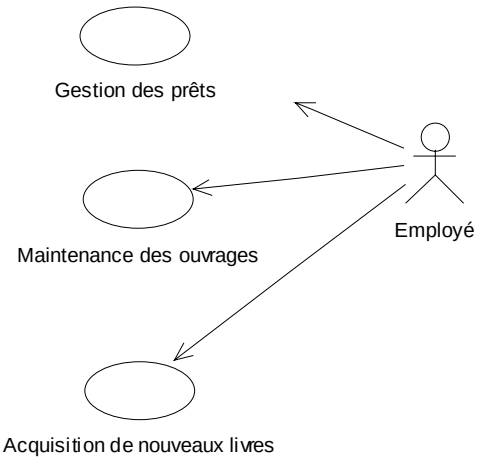


figure 1-2 : Capture des besoins, diagramme de cas d'utilisation

La figure 1-2 présente un diagramme de cas d'utilisation utilisant le formalisme originel présenté dans le langage unifié [UML97]. Celui-ci utilise deux concepts, les cas d'utilisation qui représentent les fonctions du système et les acteurs qui symbolisent les rôles des intervenants interagissant avec le système. Chaque cas d'utilisation est souvent associé à un ou plusieurs scénarios correspondant à des descriptions textuelles, structurées ou non du contexte, du cas d'utilisation.

Au premier abord, notre problème fait intervenir un seul type d'acteur, l'employé de la bibliothèque. Nous nous intéressons ici uniquement au système informatique et ne sont donc représentés que les acteurs qui interagissent directement avec ce dernier. Nous ne présentons dans cette partie que deux scénarios choisis parmi l'ensemble des scénarios qui décrivent le système de la bibliothèque :

Cas d'utilisation	Scénario	Description
Gestion des prêts	Réservation d'un ouvrage	Un employé réserve un ouvrage à partir de son N°ISBN. L'ouvrage est réservé pour une bibliothèque et un abonné donnés. **Contraintes** : Un abonné peut réserver au plus deux ouvrages différents. Un ouvrage ne peut pas être réservé plus de deux fois.
Acquisition de nouveaux livres	Ajout d'un livre	Un employé entre de nouveaux exemplaires d'un ouvrage dans une bibliothèque. **Contraintes** : Le livre doit avoir un N°Exemplaire unique. L'ouvrage est déjà répertorié.

La phase suivante détermine comment les cas d'utilisation et plus particulièrement les scénarios sont réalisés en termes d'objets collaborants. Pour cela, les diagrammes d'interaction

sont communément utilisés. Ces diagrammes sont divisés en deux vues équivalentes : les diagrammes de séquence présentent le séquencement des messages échangés entre objets pour réaliser un scénario et les diagrammes de collaborations introduisent une vue spatiale de ces interactions.

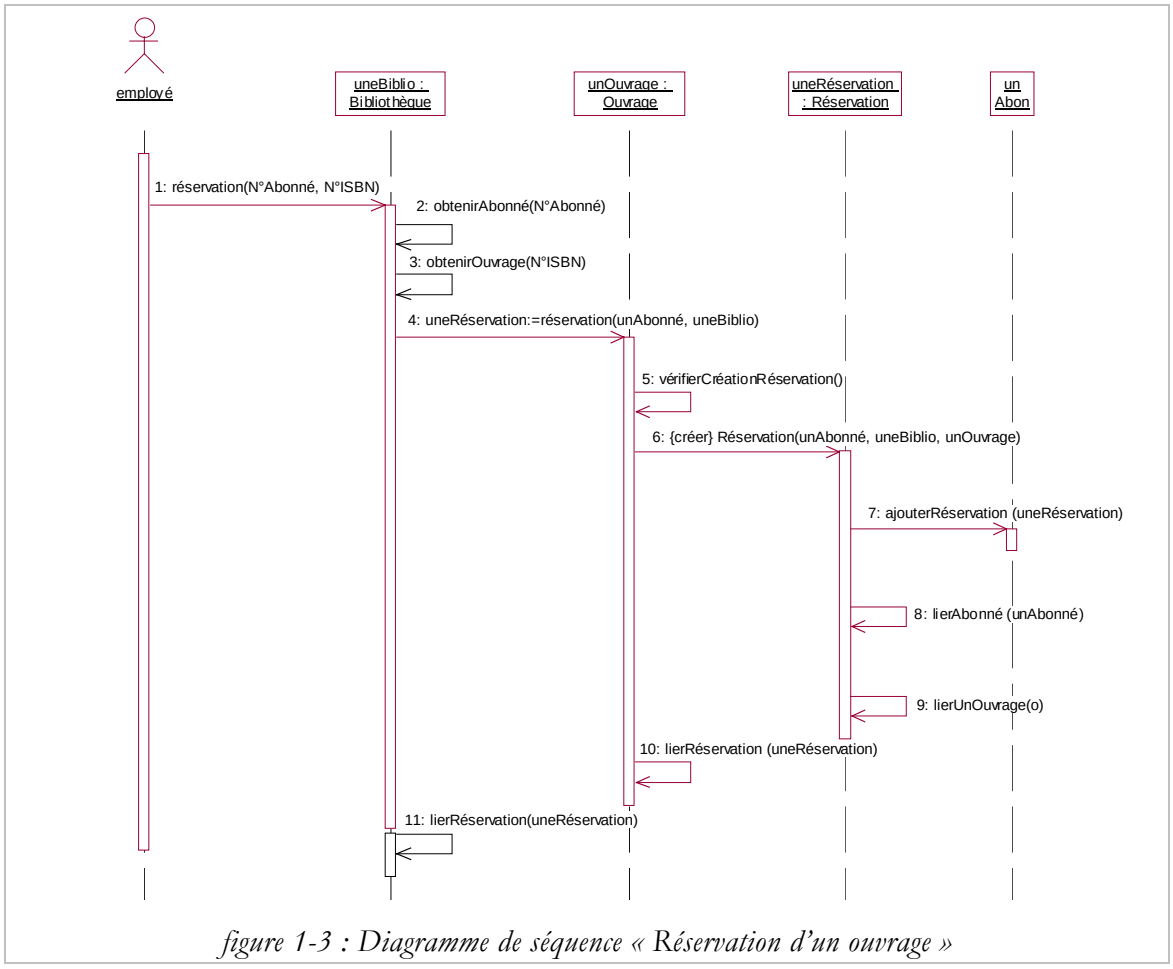


figure 1-3 : Diagramme de séquence « Réservation d'un ouvrage »

Un diagramme de séquence est en deux dimensions : l'axe vertical correspond au temps tandis que l'axe horizontal recense les objets qui interagissent dans la séquence en question. La ligne verticale attachée à chaque objet montre la ligne de vie de l'objet en question. Le diagramme de séquence ci-dessus (figure 1-3) décrit la réalisation du scénario « Réservation d'un ouvrage » qui fait intervenir un acteur, l'employé qui joue le rôle d'intermédiaire entre l'abonné et le système informatique :

La réservation est prise pour un abonné et un ouvrage donnés (1). L'abonné et l'ouvrage sont recherchés respectivement en fonction du N°Abonné (2) et du N°ISBN (3). Un message de réservation est envoyé directement à l'ouvrage (4). L'ouvrage vérifie s'il peut être réservé (5), i.e. s'il n'est pas déjà réservé par deux abonnés. La réservation est alors créée pour un abonné, une bibliothèque et un ouvrage donné (6). Les liens inter-objets sont ensuite mis à jour (7..11).

Un diagramme de collaboration montre les relations parmi les rôles d'objets. Les relations sont induites par les messages échangés entre objets de la collaboration. Dans le diagramme de collaborations « ajout d'un exemplaire » (figure 1-4), seul l'employé intervient.

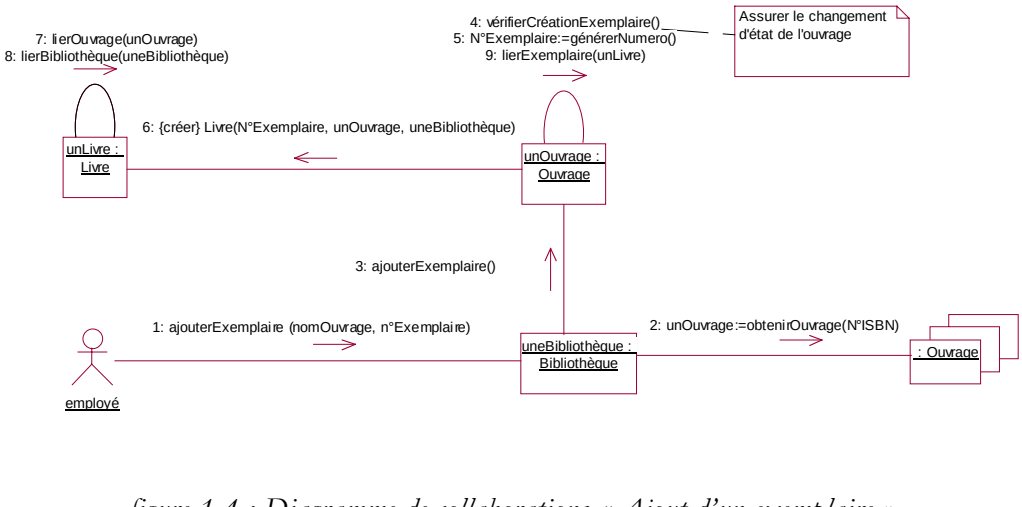


figure 1-4 : Diagramme de collaborations « Ajout d'un exemplaire »

L'ajout d'un exemplaire est réalisé en prenant en paramètre le numéro de l'ouvrage (1). Ce dernier permet de retrouver de manière unique l'ouvrage dans l'ensemble des ouvrages référencés dans la bibliothèque (2). Un message d'ajout d'exemplaire est envoyé à l'ouvrage. Celui-ci génère automatiquement le numéro d'exemplaire (3) puis le crée (4). Les liens inter-objets sont ensuite mis à jour (7..9) : un livre est associé à son ouvrage et à la bibliothèque où il est mis à disposition.

La dernière étape consiste à déduire la structure de l'application à partir des fonctionnalités du système décrites ci avant en termes d'interactions. A partir de chaque diagramme de séquence est associé un diagramme de classes partiels. La génération de diagrammes de classes est plus ou moins cablée; elle permet de garantir qu'il n'existe pas de classe dans le modèle qui ne participe à aucune collaboration et par là même à aucun cas d'utilisation. Elle introduit la traçabilité entre le besoin exprimé sous forme de cas d'utilisation et son implantation exprimée sous forme de classes [Kettani98]. La figure 1-5 présente les diagrammes de classes partiels générés à partir des deux scénarios en utilisant les règles spécifiées dans [S^{t}-Marcel98].

La démarche esquissée dans ce paragraphe est dirigée par les besoins; elle préconise donc une approche du système descendante (top-down). Parallèlement à cette tendance, se développent des approches ascendantes (bottom-up) qui elles privilégient la réutilisation de composants existants. Elles offrent une vision complémentaire de celle donnée jusqu'à présent dans cette partie. De plus, elles permettent elles aussi la génération de schémas de classes et de modèles dynamiques appropriés. Nous voyons que ces approches s'appuient sur la définition précise des différents rôles joués par les objets du système.

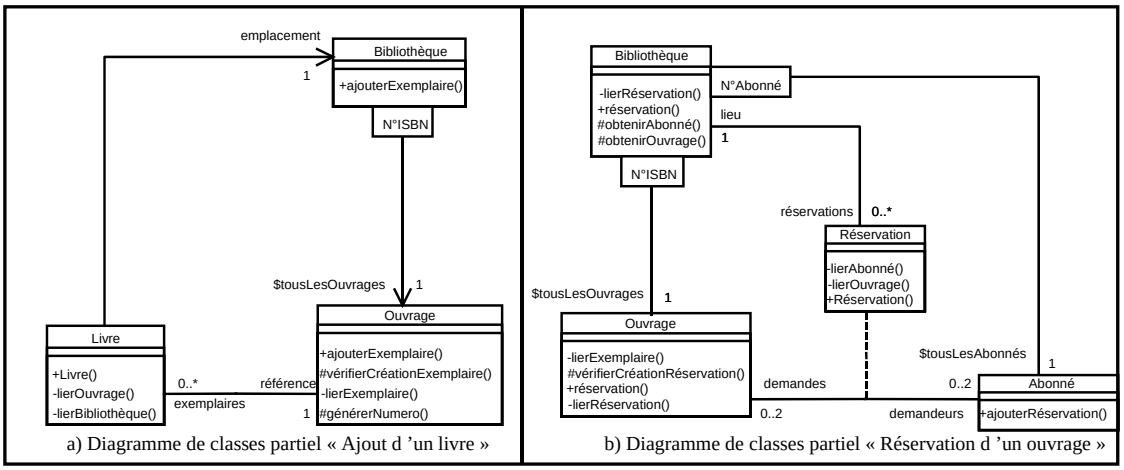


figure 1-5 : Diagrammes de classes partiels déduits des fonctionnalités du système

1.2. Méthodes et réutilisation

L'un des objectifs des technologies objets est l'amélioration en productivité et en qualité des tâches de conception et de codage grâce à la réutilisation de composants. En effet, de nos jours, toute véritable industrialisation se caractérise par la conception et le développement d'un ensemble de produits à partir d'éléments réutilisables. Ceci est désormais vrai dans plusieurs domaines industriels, par exemple la construction automobile ou l'électronique. Dans le secteur du logiciel, les informaticiens des banques ont été les premiers à découvrir et à appliquer ces principes de réutilisation, en raison de la centralisation et de l'homogénéité des systèmes d'information dans les banques. Ainsi ont-ils été des pionniers dans l'usage professionnel d'un langage comme SmallTalk. Aujourd'hui, cette démarche se généralise dans tous les secteurs d'activité de la production de logiciels où, d'une manière générale, la réutilisation lors de la production d'un logiciel vise trois objectifs principaux : diminuer les coûts de développement et de maintenance, réduire les délais et améliorer la qualité du logiciel [Morel96].

Le besoin de réutilisation est l'une des bases du succès de l'approche objet des dernières années. Ainsi, les langages à objets SmallTalk [Goldberg84], Eiffel [Meyer87] ou C++ [Badouel93] permettent de répondre à ce besoin en mettant à la disposition du développeur des bibliothèques de composants réutilisables. Cependant, le plus souvent, de telles bibliothèques n'apparaissent qu'au niveau du codage et permettent uniquement la réutilisation de quelques classes ou fonctions. De nombreux progrès restent à faire pour réellement intégrer la réutilisation dans tout le processus de développement d'une application, en particulier lors des phases d'analyse et de conception où les mécanismes permettant de réutiliser des connaissances acquises lors de ces phases restent encore pratiquement inexistants.

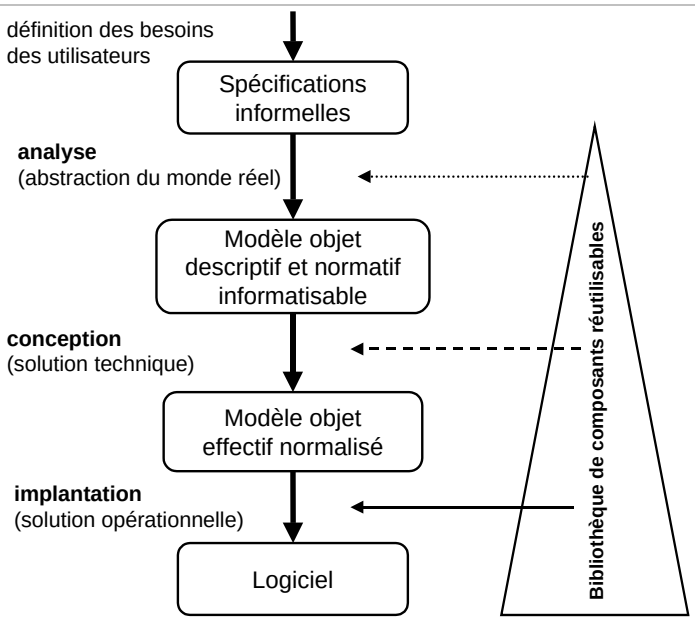


figure 1-6: Réutilisation de composants dans le cycle de vie [Nerson92]

C'est le but des approches à base de patrons ou de frameworks [GeO99] que de pallier à ce manque en proposant des composants et des connaissances acquises par des développeurs et réutilisables dès le niveau de l'analyse des besoins (figure 1-6).

1.2.1. COMPOSANTS CONTRACTUELS

Un composant est un terme générique qui définit l'unité de réutilisation dans une méthode. Alors que l'unité de réutilisation se limite souvent à la notion de classe dans les méthodes de conception à objets classiques, ce terme regroupe ici les notions de framework, de patron et plus généralement d'interactions d'objets. Le grain de réutilisation est donc plus gros que la classe : chaque composant est une brique qui peut utiliser d'autres composants.

Un contrat définit précisément les obligations qu'ont à la fois le fournisseur et l'utilisateur d'un composant. La première obligation du fournisseur est de définir comment un composant peut être réutilisé. L'utilisateur doit documenter en retour l'utilisation ou l'évolution qu'il fait du composant. Cette idée a été utilisée par B. Meyer [Meyer92] pour mettre en œuvre le langage EIFFEL. Elle a depuis connu de nombreuses variantes. On la retrouve dans les méthodes CRC (Classes-Responsabilités-Collaborations) [Wirfs-Brock90], B.O.N. [Waldén94] Catalysis [D'Souza98] et dans certaines approches [Steyaert96, Mens98]. Dans [D'Souza98] par exemple, les auteurs utilisent la notion de contrat au cœur de la méthode Catalysis. Cette méthode repose sur trois principes (figure 1-7), l'abstraction, la précision et l'assemblage qui constituent les éléments nécessaires à toute réutilisation.

L'**abstraction** permet d'omettre certains détails dans une description pour ne considérer que les choses importantes. Elle est indispensable pour représenter des éléments complexes d'un

système. Cependant elle implique souvent une forte ambiguïté, les spécifications de haut niveau étant souvent exprimées en langue naturelle ou à l'aide de diagrammes ad hoc.

Principe	Objectif
Abstraction	Ne tenir compte que des éléments importants en omettant les détails sur les besoins utilisateurs et l'architecture.
Précision	Soulever rapidement les inconsistances, Supprimer les imprécisions habituellement liées aux modèles abstraits.
Assemblage	Limiter l'activité à l'adaptation et à la connexion de composants déjà existants.

figure 1-7: Trois principes de modélisation

C'est pourquoi le principe d'abstraction doit se combiner à celui de **précision**. On obtient ainsi des spécifications assez abstraites pour être réutilisées et assez précises pour lever rapidement les inconsistances.

L'objectif est une réutilisation des spécifications par **assemblage de collaborations**. Une collaboration est une brique de base à partir de laquelle est construite l'architecture du système. Elle comprend une partie structurelle qui représente les classes, interfaces et autres éléments qui composent la collaboration et une partie dynamique qui décrit comment ces éléments interagissent. Le patron Opération-bancaire est un exemple de collaboration dont la spécification est donnée à l'aide du langage unifié (UML) : la partie structurelle est décrite à l'aide d'un diagramme de classes et la partie dynamique utilise des diagrammes de séquences. De même, les diagrammes de séquence « Réservation d'un ouvrage » et « Ajout d'un exemplaire » sont des collaborations qui décrivent des opérations du système.

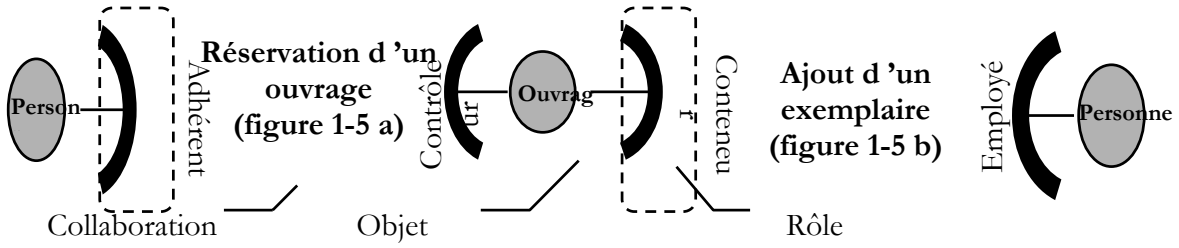


figure 1-8: Deux rôles d'un ouvrage

C'est à ce niveau qu'interviennent les rôles. Ceux-ci sont utilisés pour représenter la position tenue par chaque objet dans chaque collaboration ainsi que leurs responsabilités : un ouvrage joue des rôles différents pour la réservation et pour la mise à jour des exemplaires (figure 1-8). Les objets jouant ainsi plusieurs rôles dans l'application assurent le « liant » entre les collaborations. L'architecture du système est alors construite par synthèse[5] des collaborations et des rôles de l'application. La figure 1-8 inspirée de[D'Souza98] illustre cette approche : à partir de deux

[5] Synthèse : composition ou combinaison de parties ou d'éléments pour former un tout [Webster77]

diagrammes de séquences dans lesquels l'ouvrage joue un rôle, on peut construire une nouvelle collaboration, la nouvelle structure de l'ouvrage étant obtenue par union [GeO99] de ses rôles (figure 1-9).

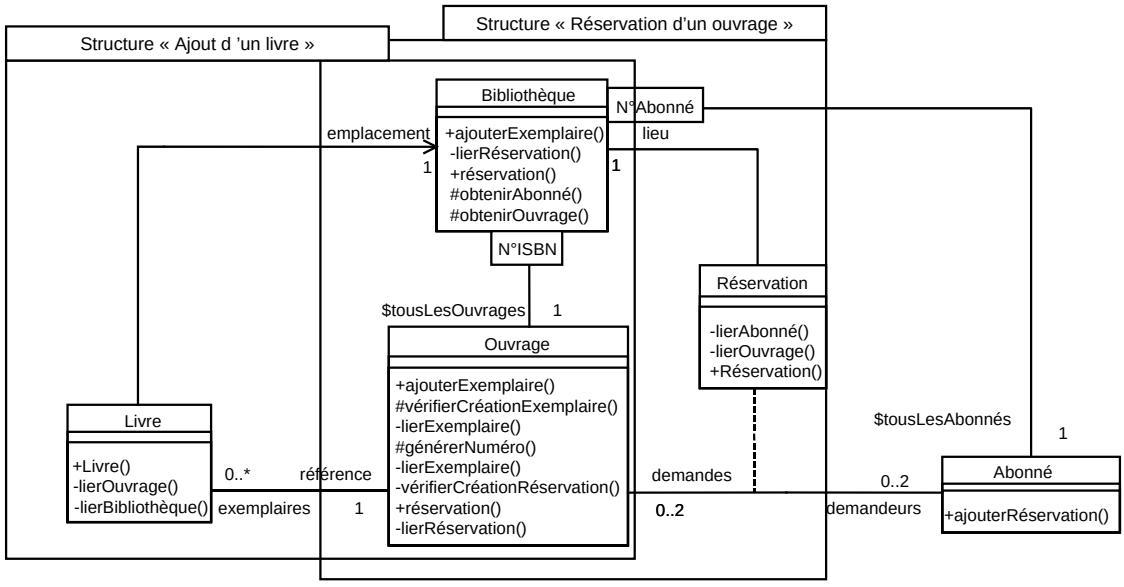


figure 1-9 : Diagramme de classes partiel déduit des fonctionnalités du système

1.2.2. PATRONS

1.2.2.1. Définitions

Le terme patron de conception (*design pattern*) a été introduit dans le domaine de l'architecture des bâtiments par C. Alexander dans deux ouvrages : «A pattern language » [Alexander77] suivi de «A timeless way of building » [Alexander79]. C. Alexander pensait que les grands bâtisseurs, au fil des siècles, n'avaient pas suivi un modèle préétabli avec des règles rigoureuses mais que les architectures avaient été adaptées les unes aux autres et à leur environnement. Cette idée est concrétisée au travers de 253 patrons propres aux thèmes récurrents en architecture

Des approches similaires à celle d'Alexander sont apparues en parallèle dans d'autres disciplines. En effet, quelle que soit la discipline, il se pose toujours des problèmes récurrents à résoudre. Dans la plupart des cas, des personnes plus expérimentées ont déjà rencontré le même type de problème et ont su le résoudre avec brio. C'est le principe de base du concept de patron qui consiste à capitaliser un problème récurrent du domaine et sa solution de manière à faciliter la réutilisation et l'adaptation de cette solution lors d'une nouvelle occurrence du problème. Ainsi, Alexander assimile le patron à un savoir-faire formalisé :

« Chaque patron décrit à la fois un problème qui se produit très fréquemment dans votre environnement et l'architecture de la solution à ce problème de telle façon que vous puissiez utiliser cette solution des millions de fois sans jamais l'adapter deux fois de la même manière ».

Dans le domaine de l'informatique, les premiers patrons orientés objets ont été présentés par K. Beck et W. Cunninghan [Beck87] lors de la conférence OOPSLA de 1987. Il s'agissait d'une première adaptation du langage de patrons d'Alexander à la conception et à la programmation objet. Quelques années plus tard, dans le cadre de l'ingénierie des systèmes d'information, P. Coad [Coad92] propose de faciliter l'analyse d'un système en identifiant les besoins selon sept patrons pré-établis. Il définit un patron orienté objet comme une abstraction d'un doublet, d'un triplet ou d'un ensemble de classes qui peut être réutilisée encore et encore pour le développement d'applications. Sa définition reste donc très proche de celle proposée par le dictionnaire : « un patron est une forme finie, originale, un modèle proposé ou accepté pour imitation ; quelque chose considérée comme un exemple destiné à être copié ».

De manière générale, un patron constitue une base de savoir-faire pour résoudre un problème récurrent dans un domaine particulier. L'expression de ce savoir-faire :

- permet d'identifier le problème à résoudre par capitalisation et organisation d'expériences,
- propose une solution possible et normalement correcte pour y répondre,
- offre les moyens d'adapter cette solution au contexte spécifique.

Tout patron doit présenter ses trois principaux composants (le problème, la solution et le contexte) dans un formalisme de représentation. Plusieurs formalismes peuvent être utilisés : la «Portland Form[6]», le formalisme de P. Coad [Coad92] ou le formalisme du groupe des quatre [Gamma95]. Si tous ces formalismes sont globalement équivalents, ils se distinguent par le nombre de rubriques plus ou moins détaillées qu'ils proposent. Ainsi, le formalisme de P. Coad est composé de 7 rubriques alors que celui de Gamma en comporte 13.

1.2.2.2. Exemple

Dans un contexte de développement objet, il s'agit de capitaliser des savoir-faire métier (par exemple la gestion des opérations bancaires) ou généraux (par exemple la gestion des ressources). Les patrons sont généralement classifiés en fonction de l'étape d'ingénierie à laquelle ils s'adressent : on parle de patrons d'analyse, de patrons de conception ou de patrons d'implantation. Nous ne donnons ici qu'un exemple basé sur quatre rubriques (nom, intention, motivation et description) qui suffisent à définir le patron d'analyse Nouvelle-Opération-Bancaire [GeO99]. Nous avons choisi ce patron comportemental car la solution proposée est de deux natures ; la partie dynamique est formalisée à l'aide de diagrammes de séquence et la partie statique par un diagramme de classes. Nous voyons par la suite que ce patron peut s'apparenter à une collaboration à partir de laquelle nous avons pu produire les figure 1-3 et figure 1-4 par

6 http://www.c2.com:80/ppr/about/portland.html

adaptation [GeO99] du patron à notre domaine de gestion de bibliothèque. Des exemples plus complets de patrons d'analyse, de conception et d'implantation sont donnés dans la partie 2.

- **Nom :** Patron Nouvelle-Opération-Bancaire [GeO99]
- **Intention :**

// problème auquel s'adresse le patron, son point fort.

Ce patron permet de traiter uniformément toute demande de prise en compte d'une nouvelle opération bancaire.

- **Motivation :**

// solution textuelle et graphique obtenue en appliquant le patron sur un exemple.

La prise en compte d'une nouvelle opération bancaire doit être traitée de manière uniforme de manière à faciliter la maintenance et l'évolution du système. Prenons deux exemples : une demande d'ouverture d'un nouveau compte client et une opération de retrait sur un compte. Ces deux exemples sont traités (figure 1-10) par deux cas d'utilisation dont les scénarios illustrent bien les analogies entre ces deux processus métiers. La partie droite de la figure propose un cas d'utilisation générique conforme aux deux exemples. Une analyse plus détaillée de ces deux processus métier confirme leur similitude. Elle est illustrée par deux diagrammes de séquences (cf. figure 1-11) mettant en évidence les collaborations entre objets.

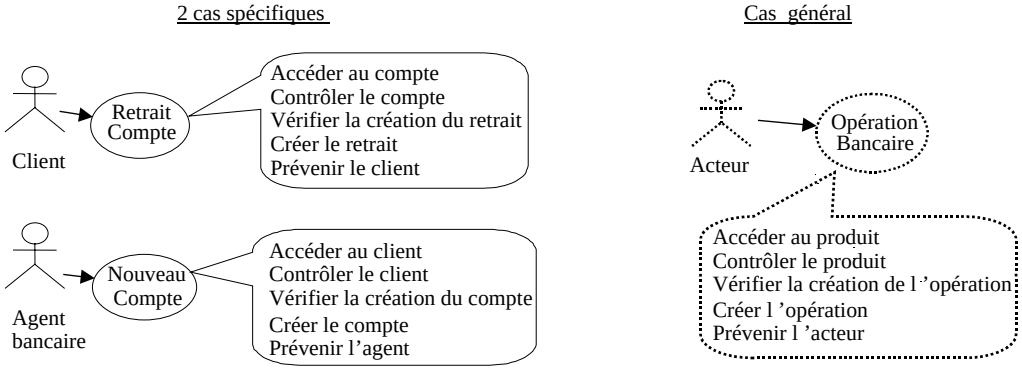
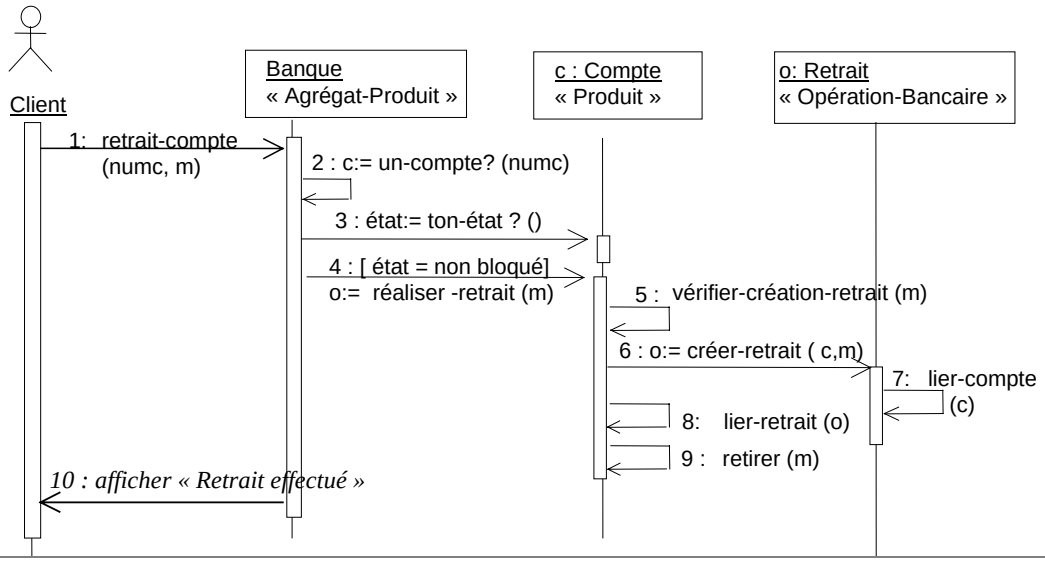


figure 1-10 : Trois cas d'utilisation des opérations bancaires



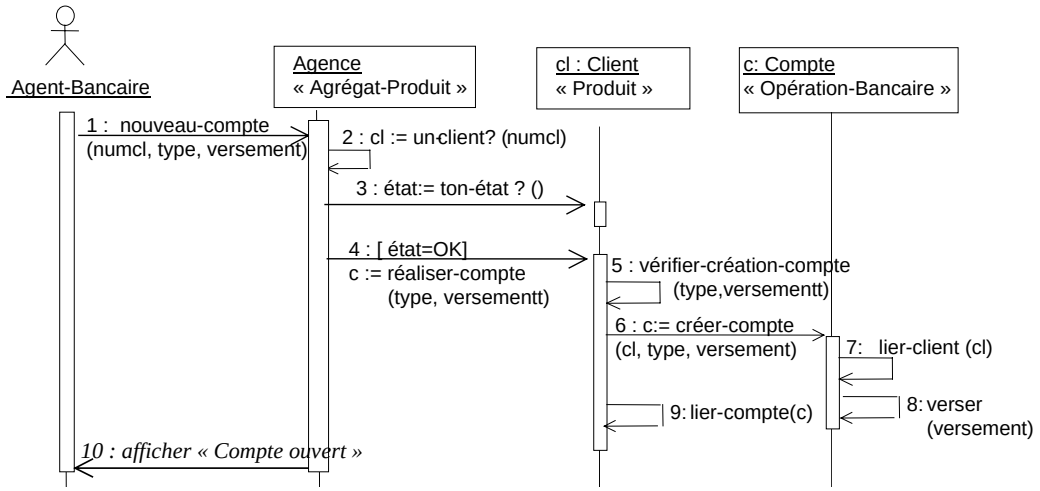


figure 1-11 : Deux diagrammes de séquence spécifiques

- **Description :**

// solution semi-formelle proposée par le patron originel, exprimée à l'aide de diagrammes UML.

Ce patron fait intervenir trois classes. La classe Agrégat-Produit détient une méthode permettant la réalisation de la demande d'opération ; elle connaît et contrôle ses produits. La classe Produit connaît et contrôle la création de ses opérations bancaires. La classe Opération-Bancaire maintient un lien vers son produit. La figure 1-12 illustre par un diagramme de séquence général les collaborations entre ces trois classes pour la prise en compte d'une nouvelle opération bancaire. Le diagramme de classes de la figure 1-13 précise les associations entre ces classes et leurs méthodes déduites du diagramme de séquence.

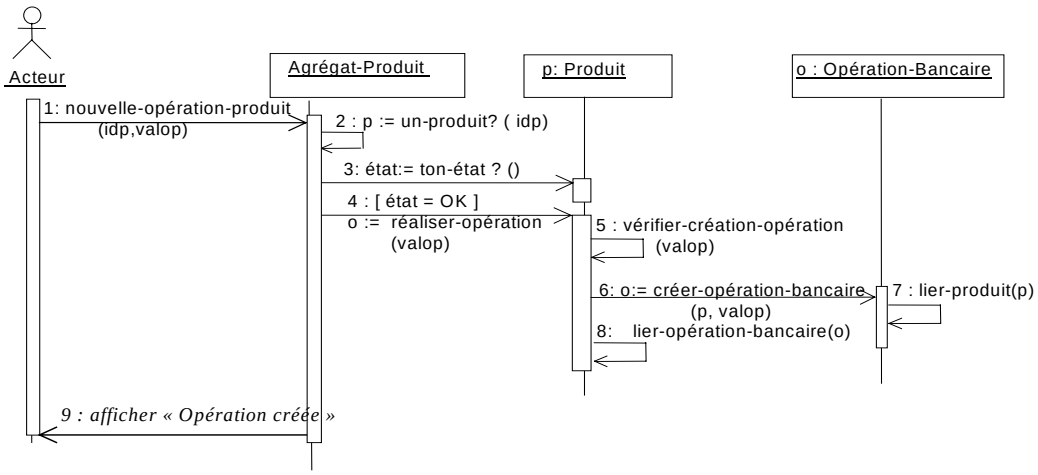


figure 1-12 : Diagramme de séquence général pour une nouvelle opération bancaire

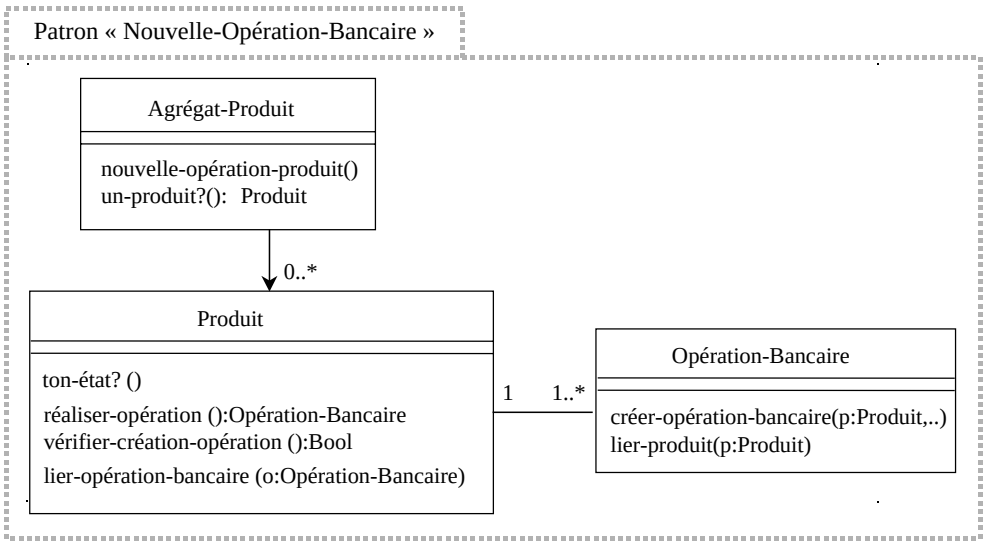


figure 1-13: Diagramme de classes pour une nouvelle opération bancaire

Chaque jour voit naître des dizaines de patrons dans tous les domaines de l'informatique (certaines conférences leur sont consacrées : la première conférence internationale, PLOP, dédiée à la notion de patrons de conception date de 1994). Et malgré la définition de catalogues de patrons [Coad92, Gamma95] et de langages de patrons [Beck87, Fowler97b, Front-Conte99] qui permettent une meilleure structuration de l'information, la définition d'opérateurs et de relations sur les patrons fait cruellement défaut pour envisager une réutilisation à grande échelle des patrons. Des recherches débutent dans ce sens [GeO99].

1.3. Conclusion

Les standards de modélisation ont été fixés par le langage de modélisation unifié (UML). La normalisation du processus de développement du logiciel est aujourd'hui à l'étude [Kruchten99]. Dans l'ensemble des méthodes de conception, des tendances se dégagent et nous avons essayé d'en extraire ce qui nous paraît aujourd'hui la quintessence. Nous avons vu dans un premier temps comment à partir de l'étude des fonctions du système, on pouvait obtenir un schéma de classes qui réponde précisément aux besoins des utilisateurs finaux du système. Dans un second temps nous avons présenté une approche basée sur la réutilisation de composants. La présentation d'un patron de conception spécifique a suggéré qu'il était possible d'obtenir le même schéma de classes toujours par l'expression correcte du problème mais cette fois en adaptant une solution existante. L'intégration d'une approche dirigée par le comportement et d'une approche par réutilisation, déjà amorcée dans la méthode Catalysis [D'Souza98], constituera à coup sûr les bases de la future méthode unifiée.

Cette intégration passe actuellement par la définition précise des rôles des acteurs dans les cas d'utilisation puis celle des rôles d'objets dans les différentes interactions du système. Au vu des parties précédentes, l'utilisation conjointe des rôles et des états nous paraît porteuse, la

première pour spécifier la visibilité externe des comportements d'objets, la seconde pour contrôler leur comportement interne. De par l'étude de plusieurs patrons de conception faisant référence dans le domaine, la partie suivante précise comment cette utilisation peut être réalisée et nous permet de répondre à la question : rôles et états, un mariage de raison?

2. DIVERSITE COMPORTEMENTALE D'OBJET ET PATRONS D'INGENIERIE

On s'aperçoit après étude des différents courants que les modèles à base de rôles et d'états sont complémentaires. Ils offrent pour les premiers la traçabilité, la réutilisation et une couverture large du cycle de développement et pour les seconds le contrôle de la dynamique.

De fait, leur conceptualisation et intégration a été abordée dans plusieurs domaines de l'informatique, au niveau des langages orientés objet [Kristensen96b], des systèmes de gestion de bases de données orientés objet [Albano93, Rieu91] et des systèmes de représentation de connaissances par objets [Marino93, Dekker92]. Cependant, on se retrouve toujours confronté au même problème : un comportement variable d'objet ne peut être complètement saisi à l'aide des concepts de classe et d'héritage offerts par les langages de programmation objet classiques [Schoenfeld96]. En effet, l'héritage entre classes implique une partition des services : un objet est instance d'une unique classe, or une classe offre un ensemble statique de services ; les services ne peuvent donc évoluer sans changement de classe, autrement dit sans perte d'identité. Face à cette insuffisance plusieurs types de propositions sont envisagés.

La première consiste à doter le langage de nouvelles fonctionnalités lui permettant de prendre en compte implicitement ou explicitement les concepts de rôle ou d'état. C'est l'approche suivie par de nombreux travaux en représentation des connaissances [Marino93, Dekker92]. Dans une moindre mesure c'est également le cas de Java [Coplien96] où le concept d'interface permet de décrire des comportements abstraits d'une classe d'objets. Chaque rôle peut ainsi être représenté par une interface de cette classe qui spécifie le comportement des objets de la classe dans ce rôle. L'évolution et le contrôle de l'évolution de l'objet dans ses différents rôles sont programmés dans la classe. Cette « recette de cuisine » permet d'implanter des évolutions multiples d'objets en Java.

Soigneusement documentée, elle peut constituer un patron d'implantation ou idiome[7] [Coplien96].

La seconde approche consiste, une fois le problème récurrent clairement identifié, à proposer une solution générique conceptuelle (et donc indépendante d'un langage donné) mais aussi à illustrer cette solution par des exemples de code et des astuces de programmation dans des langages cibles. C'est l'approche des patrons d'analyse et de conception (cf. § 1.2). Un des grands avantages des patrons d'ingénierie est d'apporter des solutions tant conceptuelles que techniques pour des problèmes récurrents et clairement identifiés. C'est pourquoi, nous avons choisi de présenter dans cette partie uniquement des patrons comportementaux. Nous restreignons notre champ d'investigation aux patrons qui répondent à notre intention de départ :

Intention : Comment représenter la diversité comportementale[8] d'un objet ?

Nous retrouvons dans les différents patrons étudiés une certaine complémentarité et similitude entre les solutions utilisant les états et les rôles et ce dans toutes les phases du développement, de l'analyse à l'implantation. Pour comparer ces solutions, nous introduisons une définition précise des notions d'états et de rôles dans le paragraphe 2.1. Celle-ci est le résultat de l'étude réalisée dans les parties précédentes. Un ensemble de propriétés partiellement illustré par des diagrammes de classes et d'objets est défini sur les relations entre l'objet, ses rôles et ses états. Ces propriétés et diagrammes aboutissent à un modèle d'analyse.

Le paragraphe 2.2 décrit et compare les patrons existants avec notre modèle d'analyse. Nous insistons particulièrement sur les patrons « Etat » de E. Gamma [Gamma95] et « Rôle » de P. Coad [Coad92] qui sont fédérateurs dans la littérature abordant ces concepts.

2.1. Rôle et Etat

La plupart des travaux de recherche traitant des évolutions multiples d'objets proposent d'enrichir la vision monolithique des approches objets traditionnelles où un objet appartient à une et une seule classe. Généralement deux concepts sont introduits (parfois sous des appellations différentes) : les concepts de rôle et d'état.

Le concept d'état permet de modéliser des objets dont la réponse à certains événements varie au cours du temps. Un objet qui réagit toujours de la même manière est dit état-indépendant [Fowler97a]. Au contraire un objet état-dépendant réagit différemment aux événements selon son état. La définition d'état d'UML [UML97] nous semble tout à fait pertinente :

---

[7] Un patron d'implantation ou idiome est destiné à montrer comment réaliser, dans un langage donné, un trait (feature) absent de ce langage. Par exemple comment réaliser l'héritage multiple en Java, comment représenter des évolutions multiples d'objets...

[8] On entend généralement par diversité la multiplicité des comportements liés à un objet unique.

Etat : une condition ou une situation de la vie d'un objet qui satisfait certaines conditions, accomplit certaines activités ou est en attente de certains événements.

Le rôle est un concept largement usité pour ne parler que d'une partie des propriétés statiques, dynamiques et comportementales des objets : les rôles familial, professionnel sont des rôles courants de personnes. Le rôle est caractérisé par le fait qu'il n'a pas d'existence propre. En conséquence, il est toujours associé à un objet auquel il est ajouté ou retiré dynamiquement, ses propriétés étant visibles pour certains clients seulement.

Six caractéristiques définissent de manière unique le rôle [Kristensen95] :

- **Visibilité** : L'accès à un objet est restreint aux méthodes du rôle, incluant les méthodes de l'objet intrinsèque mais excluant celles des autres rôles.
- **Dépendance** : Le rôle ne peut exister sans objet intrinsèque.
- **Identité** : Un objet et ses rôles forment une seule identité qui doit être manipulée en tant que telle.
- **Dynamique** : Un rôle peut être ajouté et ôté durant le cycle de vie de l'objet.
- **Multiplicité** : Plusieurs instances de rôle peuvent être associées en même temps à un objet.
- **Abstraction** : Les rôles peuvent être classifiés et organisés en hiérarchies d'agrégation et de composition.

L'état est un concept difficile à appréhender au moment de l'analyse, il est généralement introduit lors de la phase de conception. Nous avons vu qu'au contraire le rôle est un concept intuitif pour les utilisateurs qui peut apparaître dès les phases de capture des besoins et d'analyse (cf. § 1.1). De nombreuse méthodes ont étendu leur notation pour intégrer le concept de rôle [Maughan94, Renouf94].

Les relations entre les rôles et les états d'un même objet peuvent être vues de plusieurs manières. Cette complexité est abordée dans le patron « Délégation par les rôles » d'E. D'Souza [D'Souza98]. Pour notre part nous l'abordons en intégrant l'évolution de l'objet dans ses rôles. Nous retiendrons la définition du modèle NCR [St-Marcel98] qui sera défini dans le chapitre 3 :

Rôle : une évolution possible d'un objet suivant une ou plusieurs caractéristiques identifiées et calculables.

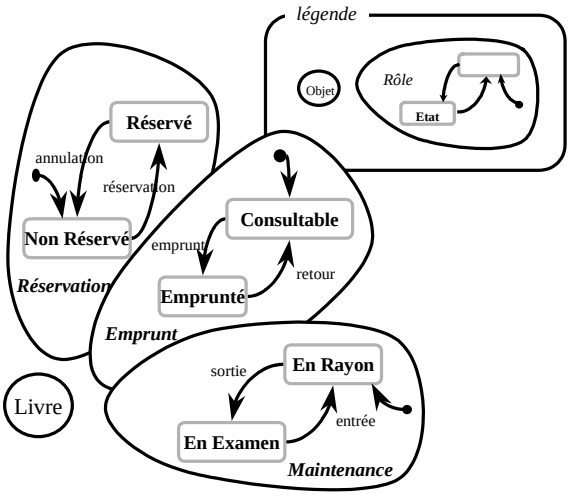


figure 2-1 : Rôles et états des livres

Remarquons que dès lors, la notion d'objet à état-dépendant (respectivement état indépendant) s'applique à présent à l'objet dans un rôle. Un objet dans un rôle est dit état-dépendant si cet objet dans ce rôle réagit différemment aux évènements selon son état. On parle également de rôle dynamique. La figure 2 illustre trois rôles dynamiques de l'objet livre, chacun d'eux modélisant une évolution possible d'un livre de bibliothèque. Dans chacun de ses rôles, l'objet peut passer par différents états. Il est à noter que ces trois rôles sont dépendants les uns des autres et qu'il est possible d'exprimer des contraintes existentielles sur les rôles et états du type « un livre En Maintenance ne peut être emprunté ». Par rapport aux modèles précédents, la maintenance est isolée de l'emprunt ; elle est considérée ici comme un rôle à part du livre.

Par la suite et pour simplifier, nous considérons que tous les rôles sont dynamiques. En effet, un rôle statique peut être traité comme un cas particulier d'évolution à un seul état.

En nous appuyant sur cette définition et sur certaines caractéristiques de [Kristensen95], nous proposons un ensemble de propriétés exprimant les relations entre un objet, ses rôles et ses états :

- **Multiplicité** : Un objet peut jouer simultanément plusieurs rôles
- **Dynamique** : Un rôle peut être ajouté et ôté durant le cycle de vie de l'objet. Nous parlerons des rôles actifs de l'objet. Le livre l1 (figure 2-2) a pour rôles actifs l'emprunt et la maintenance.
- **Comportement** : le comportement d'un objet est restreint à ses méthodes intrinsèques et celles de ses rôles actifs (en excluant les méthodes des rôles auxquels il n'a pas adhéré). Sur la figure 2-2 le comportement du livre l1 ne détient pas les méthodes d'évolution dans le rôle prêt.
- **Visibilité** : Un objet client ne perçoit qu'un sous-ensemble des rôles actifs de l'objet.
- **Evolution** : L'objet évolue dans ses rôles actifs. Dans chaque rôle, l'objet ne peut être que dans un et un seul état. Un livre évolue dans ses rôles de maintenance, de prêt, etc. Un livre en maintenance ne peut être que « retiré » ou « en rayon ».
- **Dépendance** : Un rôle ne peut exister sans objet. Un état associé à un rôle ne peut exister sans ce rôle. Le concept de livre « emprunté » n'a de sens que dans le rôle emprunt.
- **Identité** : Un objet, ses rôles et ses états forment une seule identité qui doit être manipulée en tant que telle.
- **Etat Courant** : L'état courant d'un objet est qualifié en fonction de ses états dans ses différents rôles. Un livre peut être dans l'état : « libre » et « disponible » et « en rayon ».

La figure ci-dessous illustre en partie ces énoncés. La partie gauche est un diagramme de classes résultat d'une analyse du problème. La partie droite illustre ce diagramme de classes par un diagramme d'objets.

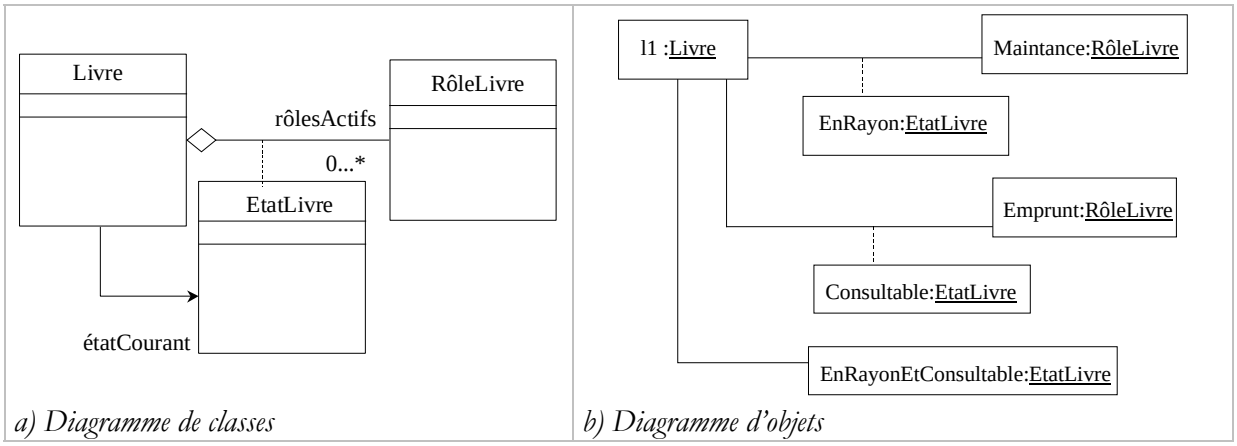


figure 2-2 : Modèle d'analyse « Objet-Rôle-Etat »

2.2. Patrons Rôle et Etat

La littérature sur les patrons est à l'origine de nombreuses discussions sur les rôles et les états. L'étude de deux patrons représentatifs, le patron Etat d' E. Gamma [Gamma95] et le patron Rôle de P. Coad [Coad92], nous permet de comprendre comment et surtout dans quelle mesure nous pouvons actuellement modéliser la diversité comportementale[9] dans un modèle objet. Les deux patrons sont présentés dans ce paragraphe avec le formalisme simplifié qui a été présenté dans le paragraphe 0 augmenté de cinq champs dont nous donnons la définition ci-dessous. Chaque champ est obtenu après une étude complète du patron original et de ses patrons apparentés :

- **Nom** : nom du patron.
- **Intention** : problème auquel s'adresse le patron, son point fort.
- **Motivation** : solution textuelle et graphique obtenue en appliquant le patron pour la modélisation des livres de bibliothèque.
- **Description** : solution semi-formelle proposée par le patron originel, exprimée à l'aide de diagrammes UML.
- **Extension** : introduction aux patrons qui proposent des alternatives de conception ou d'implantation de la solution.
- **Décision** : les choix de conception sont exprimés à l'aide d'un arbre de décision.
- **Implantation** : évaluation des techniques d'implantation du patron.

[9] On entend généralement par diversité comportementale la multiplicité des comportements liés à un objet unique.

- **Discussion** : évaluation de la solution par rapport à notre modèle d'analyse (figure 2-2).
- **Patrons apparentés** : d'autres patrons utilisés pour documenter les champs précédents.

2.2.1. PATRON ETAT

- **Nom :** Patron Etat
- **Intention :** Le patron Etat donne la possibilité à un objet de modifier son comportement quand son état interne change.

- **Motivation :**

Le patron Etat permet de gérer une évolution simple d'objet qui peut être décrite par un diagramme de transitions d'états où chaque état est modélisé par une classe. L'objet à un instant donné ne peut être que dans un unique état.

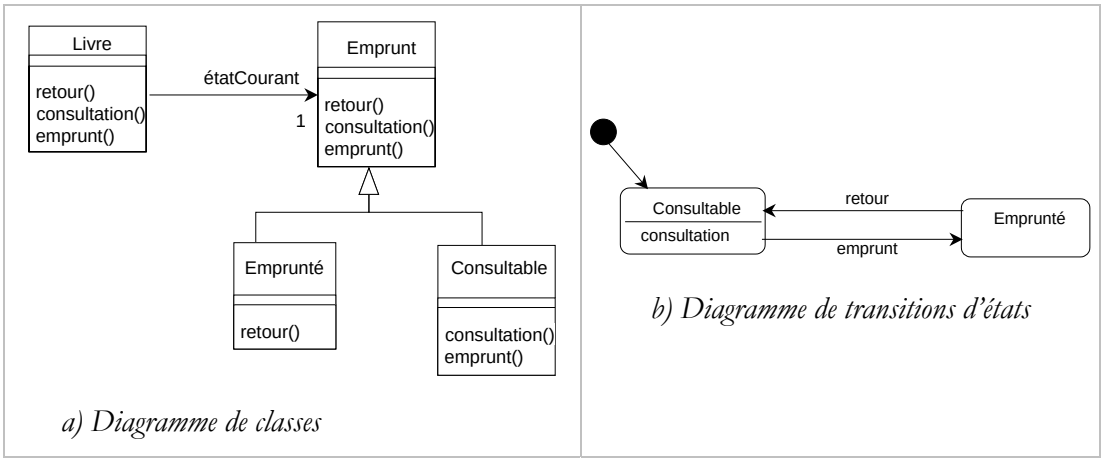


figure 2-3: Modélisation d'un rôle à l'aide du patron Etat

L'idée du patron Etat est d'introduire une classe abstraite (Emprunt) pour représenter les différents états du Livre. La classe Emprunt définit une interface commune à toutes les classes qui représentent les états opérationnels. Les sous-classes d'Emprunt implantent chaque comportement spécifique. Les méthodes retour et emprunt, par exemple, implantent les transitions d'états. De même chaque livre peut être consulté sur place lorsqu'il est Disponible. La classe Livre doit maintenir le lien avec l'état courant « étatCourant » (qui est instance d'une sous-classe d'Emprunt).

- **Description :**

Le Contexte définit l'interface pour les clients ; les états sont donc transparents pour l'utilisateur. La méthode changeEtat() n'est visible que par les états du contexte. Ce dernier doit maintenir une instance qui définit son état courant. La classe Etat encapsule le comportement associé à un état du contexte, il s'agit d'une classe abstraite. Les sous-classes concrètes d'Etat

(Emprunté et Disponible dans l'exemple) implantent le comportement spécifique associé à un état du contexte.

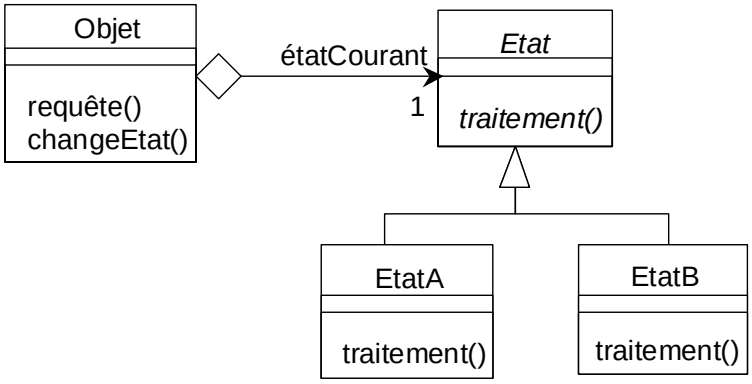


figure 2-4 : Structure du patron Etat

Les interactions peuvent être décrites à l'aide d'un diagramme de séquence. (1) `o1` reçoit une requête, (2) si la requête est liée aux états, il la délègue à l' « état courant ». `o1` est alors passé en arguments à l'état qui traite la requête. Ce dernier peut ainsi accéder au données de l'objet et réaliser le traitement. (3) L' « état courant » notifie ensuite à l'objet `o1` le changement d'état (si le traitement est lié au franchissement d'une transition) en passant en paramètre le nouvel état courant. Les transitions sont donc spécifiées par les états dans cet exemple. Dans le cas général, elles peuvent être spécifiées indifféremment par le contexte ou par les états concrets.

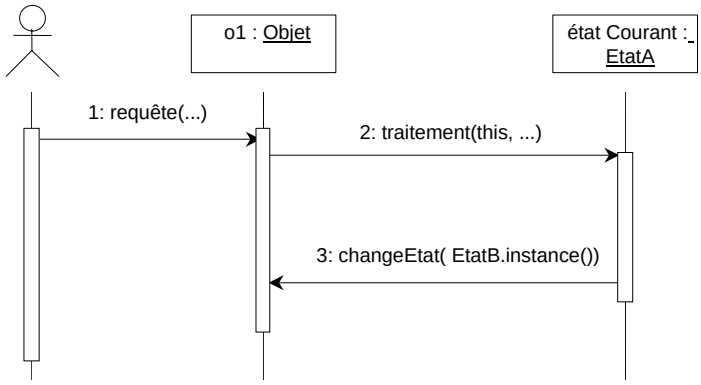


figure 2-5 : Dynamique du patron Etat

La méthode « instance » d'EtatB est une méthode statique qui permet l'obtention d'un singleton partagé de la classe EtatB. Le patron « Flyweight » d'E. Gamma [Gamma95] explique quand et comment les objets de type état peuvent être partagés.

- Extension :

[Dyson96] présente sept extensions ou raffinements du patron Etat de Gamma. Les extensions consistent en des nouveaux conseils pour implanter ce dernier alors que les raffinements formalisent des aspects existants mais non détaillés dans le patron initial. Les auteurs ont ainsi défini un langage de patrons centré sur le patron Etat (figure 2-6).

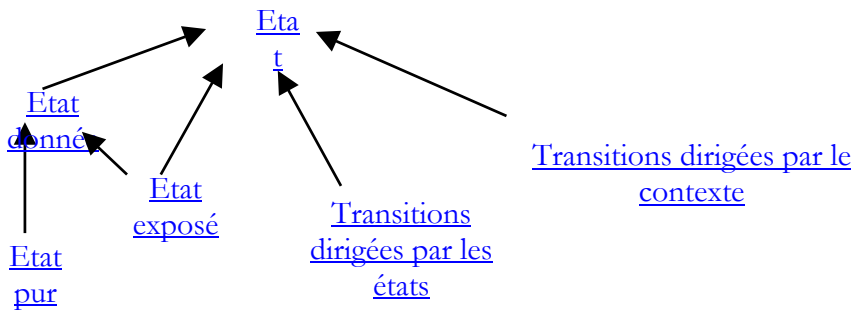


figure 2-6 : Langage de patrons centré états

Ces patrons répondent chacun à un problème précis et en donnent une solution consensuelle :

Problème	Solution	Nom
Comment modéliser les différents comportements d'un objet selon son état ?	Réifier les états de l'objet en leur déléguant les services spécifiques aux états. C'est la solution choisie par E. Gamma.	Etat
Où placer les données membres ?	Si une donnée membre est utilisée dans un unique état, alors la placer dans cet état. On peut créer artificiellement une super-classe qui regroupe les données liées à plusieurs états. Si la donnée est indépendante des états, la placer dans l'objet.	Etat donnée
Comment limiter le nombre de méthodes spécifiques à un petit nombre des états de l'objet ?	Exposer directement l'état aux utilisateurs qui peuvent lui adresser directement leurs requêtes[10].	Etat exposé
Comment réaliser les transitions ?	L'état assure lui-même la transition	Transitions dirigées par

[10] Cette solution rend les états visibles pour l'utilisateur.

	de lui-même vers un autre état. L'atomicité de la transition est donc garantie, l'objet n'en assurant plus le franchissement.	les états
Comment réutiliser les états pour plusieurs objets?	Utiliser des états purs qui sont des états sans données membres. Ils ont l'avantage d'être partagés par plusieurs objets.	Etat pur
	L'objet assure les transitions entre états. Ces derniers peuvent être partagés par plusieurs objets qui ont des machines à états différentes.	Transitions dirigées par l'objet

- **Décision**

[Ran96] propose une famille de patrons qui peut être utilisée pour simplifier la conception et l'implantation des objets dont la représentation dépend fortement des états (« moody objects »). Celle-ci est représentée à l'aide d'un arbre de décision. Ce dernier factorise les propriétés communes de chaque patron en fonction des choix de conception. L'arbre de décision donne une solution en fonction des besoins et des contraintes exprimées sur chaque objet dont le comportement est complexe.

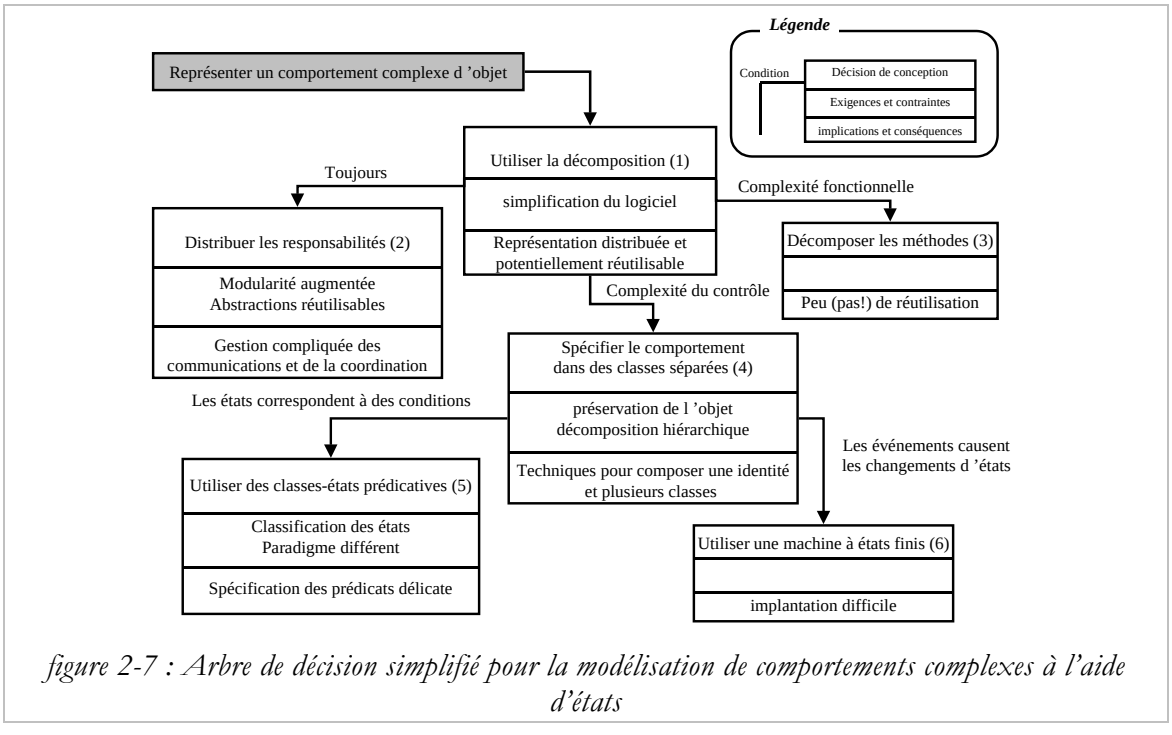


figure 2-7 : Arbre de décision simplifié pour la modélisation de comportements complexes à l'aide d'états

Pour réduire la complexité d'un objet on utilise la décomposition (1). Celle-ci peut être de trois types : délégation des services à d'autres objets (2), décomposition des méthodes de l'objet (3), spécification des états de l'objet (4). L'introduction des états répond donc uniquement au besoin de simplifier le contrôle de l'objet. La nature de ces états est liée au type d'application (cf. paragraphe deux paradigmes) :

(5) si les états correspondent à des conditions, on utilise des classes prédicatives,

(6) si les événements causent le changement d'état, le comportement est représenté à l'aide d'une machine à états finis.

- Implantation :

L'implantation des spécifications comportementales peut être réalisée de différentes manières :

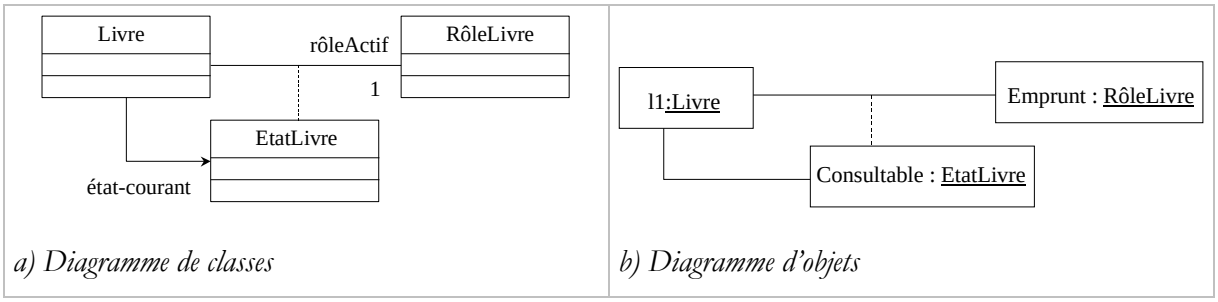
- A l'aide d'expressions conditionnelles (de type « if » et « switch »). Cependant elles alourdissent le code qui est alors difficile à maintenir et à modifier. Ainsi, l'ajout de nouveaux états peut nécessiter la modification de plusieurs tests disséminés dans le code. Deux problèmes majeurs de l'approche à base de tests conditionnels ont été soulevé par [Beck95]. D'une part l'ajout d'états requiert de nouveaux tests conditionnels. D'autre part, ce type de logique doit être répété dans toutes les méthodes. La maintenance est donc compliquée car elle concerne toujours l'ensemble des méthodes de la classe.
- En modélisant les états abstraits par des classes. Cette représentation répond à plusieurs objectifs [Ran96] :
    - Les classes-états permettent la décomposition de comportements complexes ; Chaque état spécifie uniquement le comportement qui lui est spécifique.
    - Les états abstraits qui sont d'importants concepts du domaine du problème sont explicitement représentés. La traçabilité entre les différentes étapes du développement et les vues logicielles s'en trouve améliorée. Ainsi, les états assurent le « pont » logique entre la dimension statique et dynamique dans les méthodes Syntropy [Cook94] ou Catalysis [D'Souza98].
    - La visibilité et l'accès aux attributs sont contrôlés par l'état.

  L'inconvénient de cette solution est la génération de plusieurs hiérarchies de classes et de classes-états. Son implantation nécessite deux types d'objets alors qu'un seul serait nécessaire [Dyson96]. La complexité est donc augmentée et le code non optimisé en termes de mémoire. Il évite pourtant le problème délicat de la migration et de l'identité

d'objets, un objet ne pouvant appartenir qu'à une unique classe dans la plupart des langages à objets.

- A l'aide de matrices. Cette alternative introduite par Cargill [Cargill92] convertit les entrées dans chaque état en transitions d'états vers l'état suivant. Alors que le patron Etat modélise plutôt un comportement spécifique à l'état, l'approche avec matrices s'intéresse d'avantage à la spécification de transitions d'états.

- Discussion :



a) Diagramme de classes *b) Diagramme d'objets*

figure 2-8 : Modèle d'analyse du patron Etat

Le patron Etat ne répond pas tout à fait à l'intention que nous nous sommes fixée. Seule une partie de notre intention est traduite dans la solution objet proposée par E. Gamma. En effet, celle-ci ne modélise qu'un seul rôle d'objet (le rôle où l'état courant de l'objet est défini comme identique à l'état de l'objet dans le rôle actif considéré (figure 2-8a). La classe Etat de la figure 2-8b représente en termes d'interface un rôle actif puisqu'elle factorise l'ensemble des services de ce dernier.

- **Patrons apparentés :** Un langage de patrons centré état [Dyson96], un arbre de décision pour la conception (MOODS [Ran96]), State Action Mapper [Palfinger97].

2.2.2. PATRON ROLE

- Nom : Patron Rôle

- Intention : A un moment donné, un objet peut jouer plusieurs rôles. Le patron rôle donne la possibilité à l'objet d'adhérer et/ou perdre ces rôles.

- Motivation :

Le patron rôle est utilisé lorsqu'un objet a des valeurs d'attributs et des services variables selon le rôle joué. Il prend en compte les différentes perspectives d'objets.

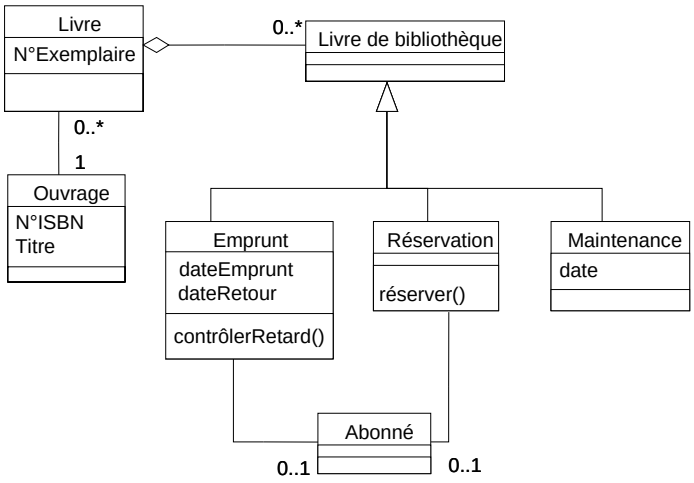


figure 2-9 :Livre &Rôles

Un livre a par exemple des caractéristiques intrinsèques telles qu'un N°ISBN, un titre, etc. (propriétés qui peuvent être factorisées dans un objet, ici l'Ouvrage). Le livre connaît aussi les rôles joués par ses instances correspondant à l'Emprunt, la Réservation et la Maintenance.

- Description :

Ce patron permet la représentation des rôles, l'adhésion et la perte de ces derniers. Cette approche a l'avantage d'être plus concise et flexible que l'utilisation de l'héritage multiple. La solution de conception choisie est similaire à celle d' E. Gamma : le rôle (respectivement l'état) est modélisé par une classe et associé à l'objet.

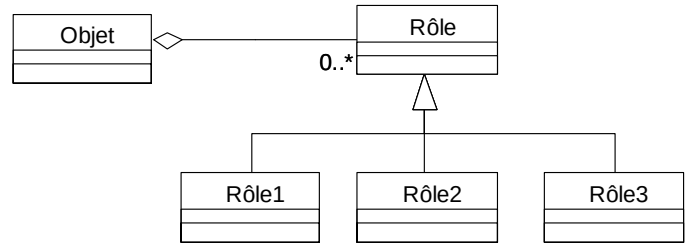


figure 2-10 : Structure du patron Rôle

- Extension :

[Baümer97] proposent une discussion sur l'implantation du patron Rôle. Ils utilisent pour cela un autre patron de [Gamma95], le décorateur. Ce dernier permet le masquage de la structure objet/rôles aux clients, l'objet apparaissant comme un tout. Cette approche a déjà été abordée dans [Fowler97b] en même temps qu'une comparaison entre les concepts de rôles et de patrons. La figure 2-11 présente une version simplifiée de ce patron.

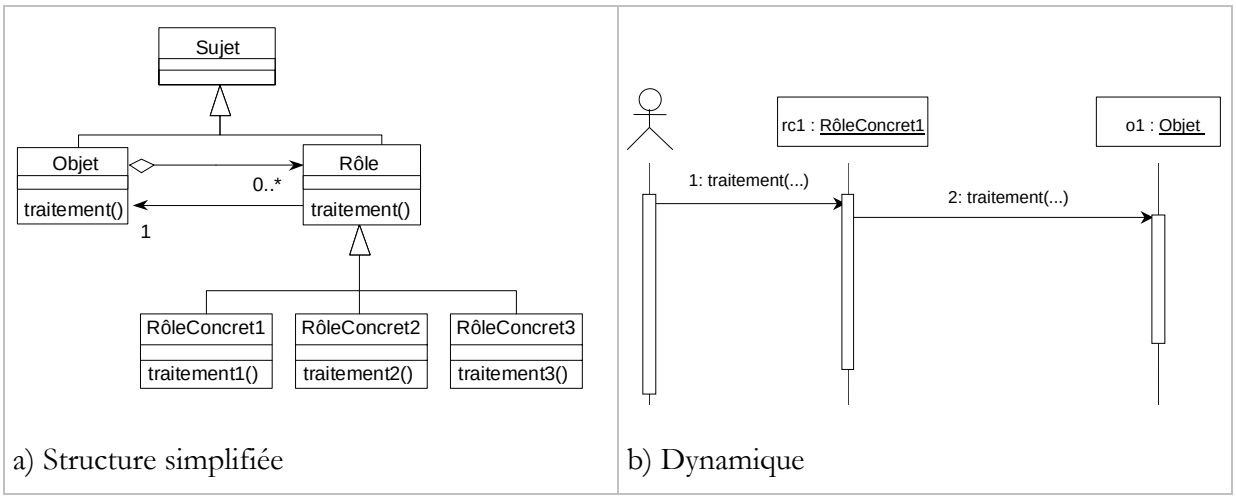


figure 2-11 : Le patron « Objet Rôle »

Le patron « Objet Rôle » donne une solution objet qui respecte bien le critère de visibilité. En effet, chaque client accède à l'objet par ses rôles alors que ce choix était non explicité dans la solution de P. Coad. La délégation se fait ici du rôle vers l'objet (figure 2-11 b). Les traitements spécifiques (traitement$_i$()) des rôles concrets sont eux directement implantés dans les classes concrètes de rôles.

Un autre aspect n'est pas abordé dans le patron de P. Coad. Il s'agit de la dynamique entre rôles. Or, il est souvent intéressant de spécifier l'exclusion entre rôles, le séquencement de rôles, etc. Certaines approches vont dans ce sens ; Elles utilisent pour cela :

- des invariants (cf. figure 2-12 inspirée de la notation M.O.N. [Maughan94]) ou des règles logiques [Pernici90].

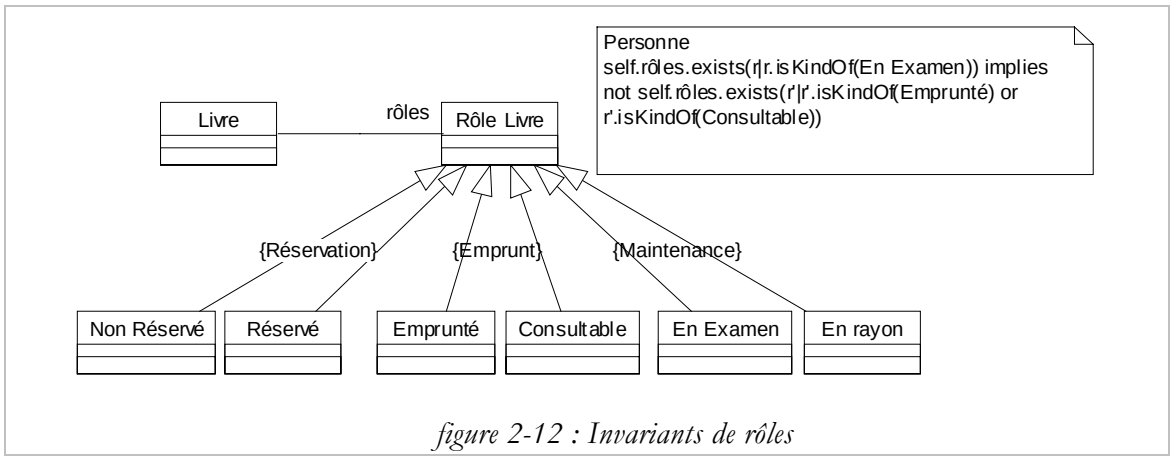


figure 2-12 : Invariants de rôles

Un objet peut jouer plusieurs rôles simultanément dans plusieurs relations[11]. On peut exprimer textuellement des contraintes simples sur l'existence de ces rôles pour une instance de Livre (figure 2-12) : « un livre retiré pour maintenance ne peut être ni consulté, ni emprunté ».

[11] Il est important de noter que cette propriété n'est pas prise en compte par les patrons Etat : un objet a toujours un unique état à un moment donné, ceux-ci étant exclusifs entre eux. Il s'agit d'une limitation par rapport à la sémantique des statecharts qui admettent des états concurrents.

- A l'aide d'expressions régulières [Kristensen95], où chaque expression régulière définit le séquencement légal des occurrences de rôles d'une classe donnée. Le comportement ainsi spécifié peut être modélisé à l'aide d'un formalisme graphique plus détaillé que dans l'approche précédente. Nous pouvons exprimer :

- L'atomicité : une instance du rôle existe.
- Le séquencement : des instances de rôle existent dans le séquencement donné. Les flèches indiquent s'il s'agit d'un séquencement strict (la terminaison du rôle implique le début du suivant) ou non.
- Le recouvrement : des instances de rôle existent concurremment.
- L'itération et la duplication : il s'agit respectivement d'un cas spécial de séquencement et de recouvrement des instances d'un même rôle.

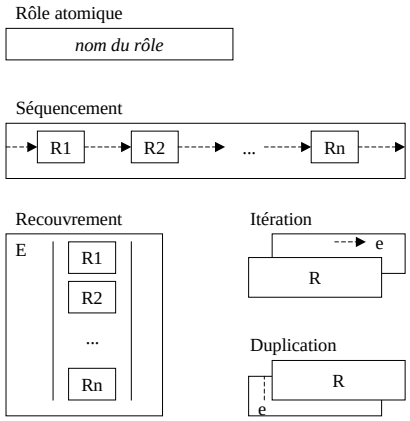


figure 2-13 :Dynamique des rôles

- Décision :

[Fowler97b] définit un langage de patrons de Rôle à partir duquel on peut déduire un arbre de décision qui décrit les choix de conception relatifs à l'utilisation de rôles (figure 2-14).

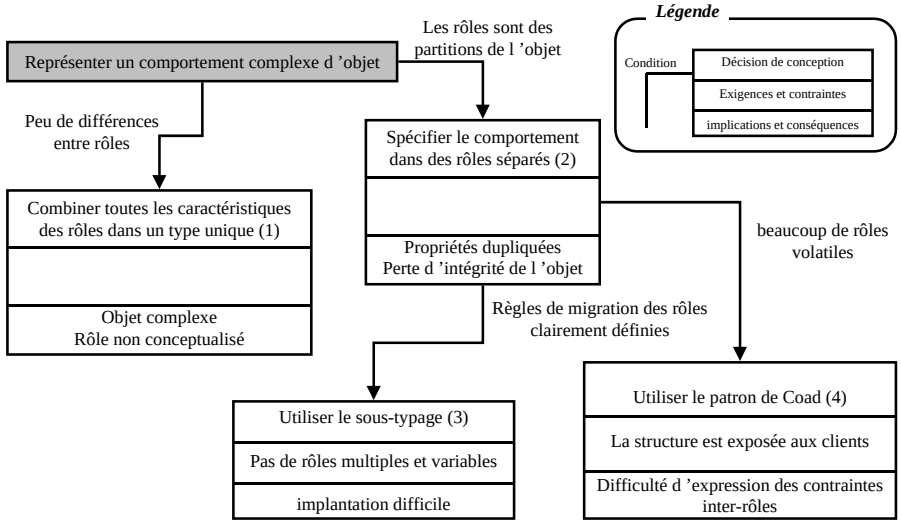


figure 2-14 : Arbre de décision pour la modélisation de comportements complexes à l'aide de rôles, inspiré de [Fowler97b]

Une solution triviale consiste à combiner dans un premier temps toutes les caractéristiques des rôles dans un seul type (1). Cette solution a peu d'intérêt car elle ne conceptualise pas le rôle et mène à la spécification d'un type complexe. Les autres approches tirent avantage du couple

objet/rôle et séparent le comportement dans plusieurs rôles. Lorsque les rôles sont des partitions avec peu de propriétés partagées, ils peuvent être définis comme des objets indépendants (2). Cependant, lorsqu'il y a un fort recouvrement entre rôles, le partage des données est réalisé grâce à l'héritage, soit par sous-typage direct de l'objet (3) ou par association d'une classification de rôles à l'objet (4) (cf. § 2.2.2 Patron Rôle).

- Implantation :

L'implantation n'est pas considérée dans le patron originel qui est plutôt vu comme un patron d'analyse. Cependant, M. Fowler donne plusieurs variantes d'implantation du patron rôle [Fowler97b]. De fait, nombreux sont les problèmes posés par l'implantation du patron Etat qui restent valides dans le patron Rôle à cause de solutions de conception relativement similaires.

- Discussion :

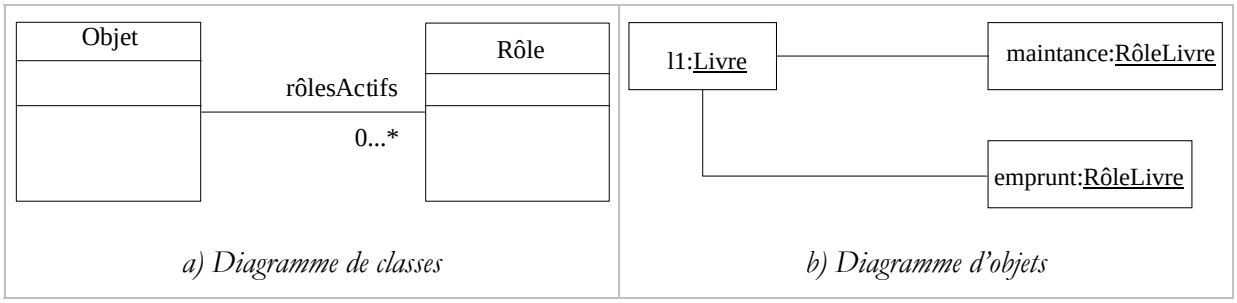


figure 2-15 : Modèle d'analyse du patron Rôle

Le patron Rôle de P. Coad prend en compte les différentes perspectives d'un objet. Il est en ce sens plus proche de notre intention. Cependant, ces perspectives sont toujours représentées de manière statique ; La notion d'état d'objet dans un rôle est perdue. Là aussi l'exemple ne peut être traité que partiellement.

- **Patrons apparentés :** Rôles et processus métiers [Schoenfeld96], Langage de patrons de rôles [Fowler97b].

2.3. Conclusion

[Fowler97b] déduit de l'utilisation du patron Etat de Gamma et de celle du patron Rôle de Coad que : « leur implantation est relativement similaire, seules leurs interfaces diffèrent ». En définissant des critères plus précis, nous avons montré que la différence conceptuelle entre les deux patrons est plus grande. Le tableau ci-dessous reprend les deux patrons et vérifie leur adéquation avec chaque critère défini dans le paragraphe 2.1.

Propriétés / Patrons	Multiplicité	Dynamique	Comportement	Visibilité	Evolution	Dépendance	Identité	Etat-Courant
Etat E. Gamma	non un objet n'a qu'un seul rôle	n'a pas de sens	oui car un seul rôle	n'a pas de sens car un seul rôle	oui	oui pour les états	oui	confondu avec l'unique état actif
Rôle P. Coad	oui un objet joue plusieurs rôles	oui un rôle peut être ajouté ou supprimé	oui car l'objet n'est lié qu'à ses rôles actifs	non	non géré explicitement	pas spécifié	non	pas de concept d'état

figure 2-16 : Patrons & Critères des objets évolutifs

Nous avons vu que l'ensemble des problématiques sur le comportement individuel des objets n'est pas couvert par les patrons existants : les patrons Rôle et Etat ne traitent à notre sens que partiellement le problème des évolutions multiples d'objets. La possibilité de réutiliser des évolutions d'objets n'est pratiquement pas envisagée sinon sous la forme de code alors que les statecharts, de par leur pouvoir d'abstraction et donc d'extension fournissent à notre sens un formalisme intéressant pour capitaliser ces évolutions.

CHAPITRE 3

Vers une plus grande (ré)utilisation de comportements d’objets

« Effectivement, je me rappelle maintenant qu’il avait un faux air emprunté en le réservant. Mais quelques jours plus tard, je vis que ce que j’avais d’abord pris pour un emprunt n’était qu’une réserve bien naturelle. Et nous en fumes tous deux retournés. »

Pourquoi les statecharts restent-ils le parent pauvre des méthodes actuelles? Comment pouvons nous y remédier ? C'est à ces deux questions que nous allons essayer de répondre dans ce chapitre. Les deux premiers chapitres nous ont donné une vision globale à la fois des modèles et des démarches comportementaux. Nous en donnons dans ce chapitre une vision plus personnelle en mettant en exergue leurs avantages et leurs manques.

Nous illustrons notre discours par un exemple de modélisation (cf. § 1.1). Celui-ci est proposé sous la forme d'une interrogation à laquelle nous essayons de répondre dans un premier temps en proposant des représentations tirées de la connaissance de l'état de l'art puis en proposant notre propre solution de modélisation (cf. § 1.4). Nous voyons comment ces solutions répondent aux critères de qualité que nous nous sommes fixés : flexibilité, réutilisation et traçabilité. Leurs faiblesses nous aident à définir précisément le « cahier des charges » du modèle NCR ainsi que les prémisses d'une méthode centrée sur la réutilisation des comportements individuels d'objets. Nous montrons ainsi que le maillon faible des statecharts ne tient pas dans le modèle lui-même mais plutôt dans l'approche méthodologique qui en est faite. Ce constat nous amène à présenter une nouvelle démarche pour la réutilisation de comportements individuels d'objets (cf. § 1.4.1).

1. NCR, UN MODELE QUI A DE LA RESSOURCE ?

« Il m'arrive parfois de me sentir comme un objet, comme prisonnier de mon comportement. »

1.1. Contexte

Les méthodes de conception orientées objet ont intégré très tôt de nombreuses vues du système d'information, conscientes en cela qu'un unique modèle ne pourrait suffire. Ces vues sont souvent classées en fonction de trois critères selon qu'elles sont statique, dynamique ou fonctionnelle [Bouzeghoub95] :

- La vue statique décrit les objets du système et les relations qu'ils entretiennent entre eux. Elle se préoccupe de la structure du système d'information.
- La vue dynamique contrôle le comportement des objets et plus particulièrement leurs changements d'états.
- La vue fonctionnelle décrit les processus de transformation qui ont lieu dans le système d'information.

C'est plus particulièrement aux deux premiers critères que nous nous adressons dans ce mémoire. Nous utilisons pour cela deux diagrammes présents dans le langage de modélisation unifié, les diagrammes de classes pour la représentation de la vue statique et les statecharts pour la représentation de la vue dynamique. Alors que les diagrammes de classes constituent aujourd'hui le modèle incontournable autour duquel s'articulent les méthodes de conception (sous une forme ou une autre : E/R [Chen76], modèle Classe/Relation [Desfray94], diagramme de classes de l'UML [UML97]) , le choix des statecharts peut paraître discutable. En effet, à l'ère des patrons ou des frameworks (collaborations réutilisables de classes), la granularité des statecharts, de l'ordre de l'objet, semble petite. Bien que ce choix puisse se positionner comme un postulat à ce

travail, nous avançons ci-dessous les raisons qui nous ont amenées d'une part à choisir le formalisme des statecharts pour représenter ces comportements d'objets et d'autre part à favoriser leur réutilisation pour les systèmes d'information.

Nous représentons le comportement d'objets à l'aide de statecharts car :

- Ils constituent la représentation la plus aboutie des machines à états finis, donnent une double vision du comportement à la fois descriptive (observation du comportement) et prescriptive (invocation du comportement) alors que les autres modèles privilégient souvent une seule vision, descriptive pour les assertions, prescriptive pour les réseaux de Petri ou les diagrammes d'interactions.
- Ils offrent dans un même modèle la précision et l'abstraction. Ils permettent l'expression de comportements formalisés, sémantiquement « riches » grâce à l'utilisation de gardes, d'invariants d'états, d'activités, etc.
- Ils donnent une définition graphique de comportements complexes. On sait aujourd'hui que le contrôle des systèmes complexes passe par le contrôle de ses objets complexes. La modélisation du comportement de ces derniers reste un enjeu d'actualité. Comme le disent Cook&Daniels, « plutôt que de spécifier globalement le comportement du système, l'orientation objet favorise la localisation des descriptions comportementales dans les types » [Cook94]. En effet, pendant de nombreuses années les méthodes ont mis l'accent sur l'expression du comportement global du système avant de s'intéresser au comportements locaux poussées en cela par les technologies à objet. Ce fut le cas dans la méthode Merise [Nanci96] où les modèles de traitements (cf. chapitre 1 § 1.1.2), basés initialement sur les réseaux de Petri, ont été étendu dans la seconde version pour prendre en compte le comportement (cycle de vie) des objets pivots [Panet94].

Nous voulons réutiliser le comportement d'objets car :

- La réutilisation des systèmes complexes passe par la réutilisation de ses objets : on parle de plus en plus souvent d'objets métiers, qui sont des objets spécifiques à un métier et qui offrent un fort potentiel de réutilisation. Leur réutilisation ne peut passer que par une bonne compréhension de leur comportement. Cette idée est déjà appliquée pour la réutilisation des composants dont le comportement est spécifié à l'aide de collaborations [D'Souza95].
- Il existe de très nombreux comportements « type » qui restent suffisamment abstraits pour être adaptés et réutilisés dans des contextes particuliers. Le comportement de ressources que nous traiterons par la suite en est l'exemple.

- Une expression complète et formalisée des comportements individuels d'objets requiert un énorme travail de spécification . De plus, la qualité des spécifications obtenue est difficile à mesurer.

1.2. Problème

Nous l'avons vu, le formalisme des statecharts est arrivé aujourd'hui à maturité. De fait, il devrait être de plus en plus utilisé par les méthodes de conception pour représenter la dynamique des objets du système. Or, nous remarquons que son utilisation est effective uniquement dans des contextes particuliers (lorsque les contraintes temporelles sont fortes par exemple), mais qu'il reste absent des systèmes de gestion classiques. L'état des lieux réalisé dans les deux premiers chapitres nous amène aux constats suivants :

- Enseignée dans les cours universitaires et professionnels, la spécification de statecharts reste un exercice réservé à des spécialistes. La difficulté ne réside pas tant dans le modèle lui même qui est basé sur des concepts simples (états et transitions) mais dans sa mise en œuvre dans les systèmes à objets. De plus, il existe peu de critères permettant de jauger la qualité des spécifications à base de statecharts.
- Il n'existe pas aujourd'hui de corpus de spécifications de statecharts susceptible d'en favoriser la réutilisation. Les spécifications de statecharts sont peu utilisées, peu réutilisables et donc à plus forte raison peu réutilisées.
- « La plupart des méthodes qui fournissent des vues structurelles et comportementales du système… ne fournissent pas les indications pour faire cohabiter ces vues et encore moins pour en assurer la cohérence » [Vayda95]. Ce constat s'applique dans une certaine mesure aux modèle à objets vis-à-vis des statecharts. Nous avons vu dans le premier chapitre qu'il était difficile de concilier conformité structurelle et comportementale des classes du système.
- Il existe une réelle dissymétrie entre la puissance de modélisation proposée par les statecharts d'une part et les possibilités d'implantation dans les langages à objets actuels. La difficulté à assurer le passage de diagrammes de transitions d'états à une éventuelle implantation a déjà été mise en avant dans la méthode BON [Waldén94].

Parce qu'aujourd'hui ils ne garantissent ni réutilisation, ni traçabilité, les statecharts sont souvent utilisés comme des compléments de spécification par les concepteurs mais ne sont jamais véritablement intégrés dans le processus de développement. Nous avons pourtant identifié dans la littérature deux courants méthodologiques pour intégrer des statecharts dans les méthodes de conception à objets. Nous faisons état ci-après de ces deux courants.

1.2.1. DEUX APPROCHES

Nous avons vu dans le premier chapitre que la dualité présente dans l'objet entre la structure (les attributs) et le comportement (les méthodes) se retrouvait de manière symétrique dans la définition et l'utilisation des statecharts. Nous avons montré que cette dualité s'exprimait en premier lieu par une caractérisation différente des états et par des règles de conformité duales. Nous allons montrer ici qu'elle influe aussi sur le mécanisme de construction des statecharts dans les systèmes d'information. Nous essayons pour cela de définir le Statechart de la classe Ouvrage de notre bibliothèque.

Approche classificatoire (figure 1-1**)**

Le processus de construction d'un Statechart suivant l'approche classificatoire se déroule en trois étapes :

1. Identification structurelle

Le concepteur détermine la structure de l'objet, i.e. les attributs et les relations de l'objet. Il détermine leur type et donc leur domaine de valeurs.

Ici, un ouvrage a deux attributs, le nombre d'exemplaires disponibles de cet ouvrage nbExDispo *de type Entier et le nombre d'exemplaires dans la bibliothèque* nbEx *qui détermine le domaine de* nbExDispo.

2. Identification des états

La découverte des états se fait à partir des attributs de la classe. L'ensemble des états forme une partition de l'ensemble des vecteurs de valeurs prises par les attributs.

Sur notre exemple, on peut définir de manière triviale trois états significatifs pour un ouvrage, l'état TousEnRayon *où* nbExDispo = nbEx, Aucun Exemplaire *où* nbExDispo = 0 *et* ExemplairesDisponibles *qui abstrait les autres valeurs de* nbExDispo *strictement comprises entre* 0 *et* nbEx.

On a alors les relations fondamentales :

$$nbExDispo = nbEx \Rightarrow TousEnRayon \textbf{ et}$$

$$nbExDispo = nbEx \Rightarrow Aucun\ Exemplaire \textbf{ et}$$

$$0 < nbExDispo < nbEx \Rightarrow ExemplairesDisponibles$$

3. Identification des transitions

Il ne reste alors qu'à déterminer les transitions en fonctions des valeurs prises avant et après franchissement des transitions. Chaque état doit être atteignable, i.e. avoir au moins une transition d'entrée.

Si l'emprunt et le retour portent sur un seul exemplaire, on se rend compte par exemple qu'il est impossible d'atteindre l'état TousEnRayon *à partir de l'état* Aucun Exemplaire *avec un* emprunt.

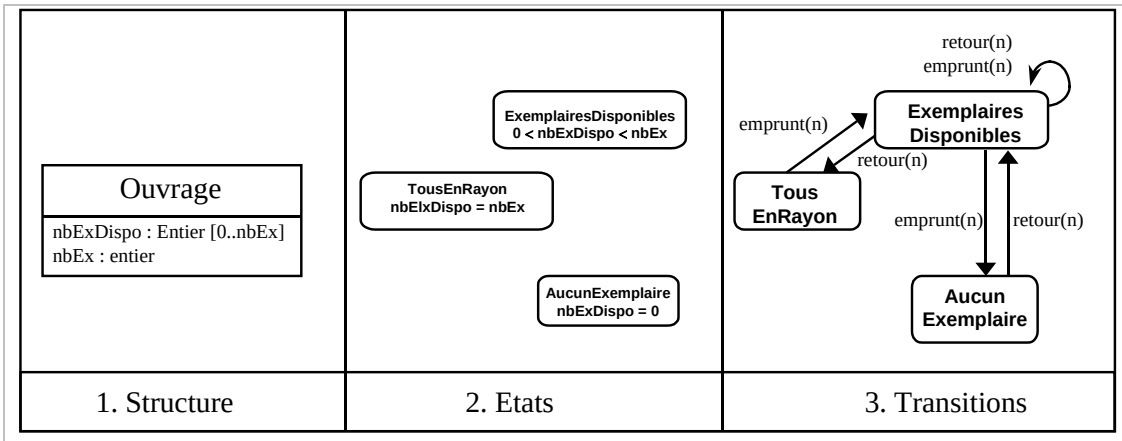


figure 1-1: Génération d'un Statechart à partir de la structure

Nous pouvons remarquer que les transitions sont secondaires puisque dans cette approche l'état d'un objet peut être déterminé à tout moment à partir de ses valeurs d'attributs (sans se soucier de son cycle de vie, i.e. la séquence d'événements qui l'a conduit dans cet état).

Cette vision du comportement est très proche de celle que l'on retrouve dans les mécanismes de classification d'objets en représentation des connaissances (modèle TROPES [Marino93], SHOOD [Escamilla93]) et c'est en cela que nous avons nommé cette approche « classificatoire ». La spécification de statecharts nécessite ici une bonne connaissance de la structure des objets du système. Elle est donc efficace pour des types de données « simples » que l'on retrouve dans le système informatique (pile, …) mais est difficile à mettre en œuvre pour les objets complexes du système d'information. En effet, alors que nous avons considéré uniquement les valeurs de deux attributs pour spécifier la structure d'un ouvrage, ne perdons pas de vue que la structure des objets dépend des propriétés mais aussi des relations qu'ils entretiennent entre eux, relations qui restent en général difficiles à appréhender dans leur globalité.

Approche opératoire (figure 1-2**)**

Dans cette approche, la découverte des états est réalisée à partir de l'étude du comportement de l'objet. La construction se résume en trois étapes :

1. Identification des services

   Dans notre exemple, l'identification des services est réalisée arbitrairement. Nous avons vu que le concepteur peut les obtenir à partir de l'analyse des besoins utilisateurs en suivant des méthodes dirigées par le comportement (cf. chapitre 2 § 1.1).

   *Dans le processus de prêt, l'*emprunt *et le* retour *sont des services bien identifiés pour un ouvrage de la bibliothèque.*

2. Identification incrémentale des transitions et des états

La connaissance des états et des transitions d'un objet est obtenue de manière incrémentale en « traçant » chaque service rendu par cet objet (état de départ et état d'arrivée). Cette identification peut aussi se faire à partir de la connaissance du séquencement. Elle se termine lorsque tous les services sont pris en compte. Les transitions peuvent être gardées afin de lever l'indéterminisme. Ainsi, l'emprunt sur la figure 1-2.a est conditionné par le nombre d'exemplaires en rayon nbExDispo.

L'emprunt, par exemple, ne peut se faire que si l'on dispose d'exemplaires. Il précède forcément un retour. Il est donc préférable de commencer par identifier les transitions franchies lors d'un emprunt. De la même manière un retour ne peut pas avoir lieu lorsque tous les exemplaires sont disponibles.

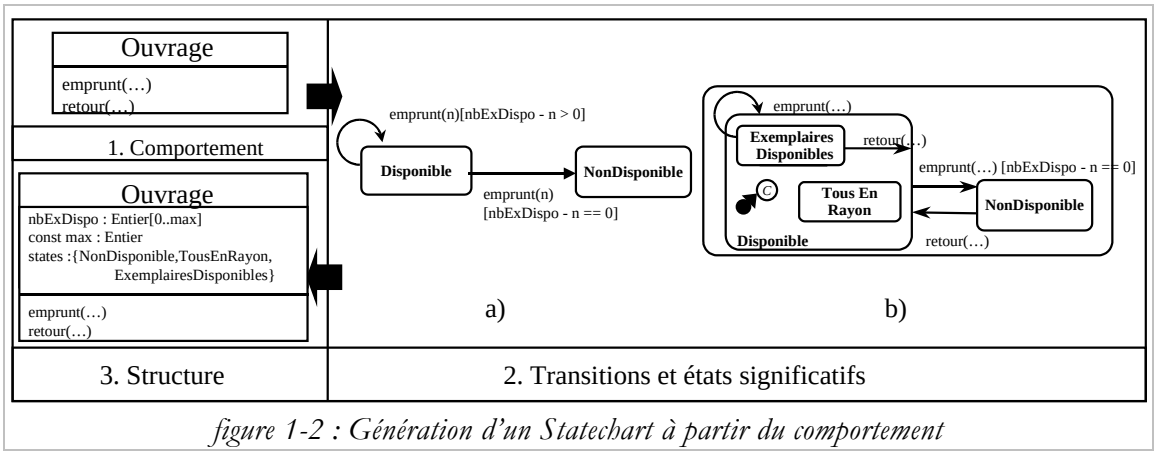


figure 1-2 : Génération d'un Statechart à partir du comportement

3. Adaptation de la structure

La structure est déterminée en fonction des conditions exprimées dans les gardes. De manière classique, la partition d'états est exprimée à l'aide d'un type énuméré.

Sur la figure 1-2*, on peut facilement déduire de l'étape 2.a) que l'ouvrage a un attribut nbExDispo dont le domaine est complètement défini dans l'étape 2.b).*

La variable states est utilisée pour représenter les trois états finaux. Elle peut prendre les valeurs TousEnRayon, NonDisponible et ExemplairesDisponibles. On obtient dans ce cas des relations inverses par rapport à la première approche :

$$TousEnRayon \Rightarrow nbExDispo = max \textbf{ et}$$

$$NonDisponible \Rightarrow nbExDispo = 0 \textbf{ et}$$

$$ExemplairesDisponibles \Rightarrow 0 < nbExDispo < max$$

Cette approche fait appel à une connaissance à priori du comportement des objets. L'objectif est ici de fournir une partition des services de l'objet, i.e. une définition de l'ensemble des services autorisés pour chaque état. Cette vision est donc plus actuelle car adaptée aux méthodes dirigées

par le comportement (cf. chapitre 2 § 1.1). Elle suit en cela le postulat que « c'est de l'étude des comportements des objets que l'on en déduit leurs propriétés structurelles… appliquant ainsi le bon principe : Dis-moi comment tu te comportes, je te dirai comment tu es. » [CSO99].

Ces descriptions ont l'avantage d'intervenir très tôt dans le processus de développement et ce dès les phases d'analyse si on préconise une démarche dirigée par le comportement.

1.2.2. UNE MEME DEMARCHE

Nous avons vu que la spécification des statecharts peut être réalisée de deux manières. Leur intégration dans le processus de développement reste par contre similaire quel que soit le type d'identification. On commence par identifier les objets qui ont un comportement « intéressant » [Rumbaugh95] puis on leur associe un Statechart unique. La cohérence entre modèles est assurée du modèle dynamique vers le modèle statique dans l'approche opératoire et inversement du modèle statique vers le modèle dynamique dans l'approche classificatoire (figure 1-3).

Les spécifications graphiques d'automates à états finis permettent la génération des squelettes de classes. Le concepteur doit associer les transitions du modèle dynamique aux méthodes du modèle statique pour l'implantation. La spécification des classes obtenues se limite le plus souvent à une définition de l'ensemble de ses méthodes. Cette spécification est plus ou moins précise selon que l'on emploie ou non des assertions. De manière générale, on retrouve deux types de traduction :

- Soit une traduction réalisée "à la main" par les programmeurs dont on ne peut vérifier ni la qualité, ni la traçabilité. Pour éviter cela, nous avons vu que l'utilisation du patron Etat de E. Gamma apportait une première solution pour « standardiser » la traduction des diagrammes d'états (cf. chapitre 2 § 2.2.1).
- Soit une traduction câblée du modèle graphique. Cette traduction a l'avantage de rendre un résultat traçable et dont on peut vérifier la qualité. On en trouve de nombreuses propositions dans la littérature ou dans les outils existants. Nous donnons ci-dessous un exemple classique de traduction automatique en C++ du comportement d'une ressource mono-exemplaire.

```
enum State {Indisponible, Disponible}
enum event {retour, emprunt}
void retour() ;
void emprunt() ;
void transition(Event e){
  static State s = Disponible ;
  switch(s){
case Indisponible :
      switch(e){
        case retour :
          s = Indisponible ;
          retour() ;
            break ;
        }
        break ;
        case Disponible :
          switch(e){
            case emprunt :
              s = Indisponible ;
              emprunt() ;
             break ;
         }
         break ;
   }
}
```

Cette traduction a l'inconvénient d'utiliser des tests conditionnels qui sont difficiles à maintenir. Dans tous les cas, la réutilisation des traductions obtenues, notamment par spécialisation, reste un problème non résolu.

L'approche classique[12] établit un faible couplage sémantique entre les modèles statique et dynamique au niveau conceptuel : la cohérence est vérifiée au moyen de règles élémentaires : il s'agit d'associer chaque diagramme d'états à une classe et de vérifier qu'à chaque événement/transition correspond une méthode de la classe. Ces règles ont l'avantage d'être facilement (automatiquement) vérifiables mais offrent peu de flexibilité au concepteur lors de la phase de traduction des modèles de conception.

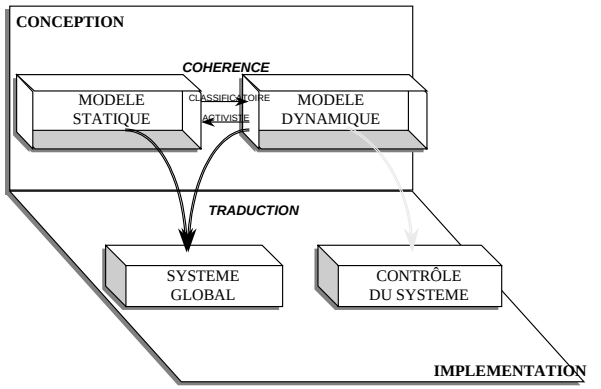


figure 1-3 : Démarche classique intégrant des spécifications structurelles et comportementales

1.3. Motivation

Dans ce paragraphe, il s'agit de confronter les différentes approches exposées dans ce mémoire sur un exemple significatif : les multiples ressources intervenant dans les systèmes d'information. Nous avons choisi cet exemple pour son apparente simplicité mais aussi pour sa richesse et ses potentialités de réutilisation. En effet, la gestion de ressources est un thème récurrent des systèmes d'information. Ce thème a été repris de nombreuses fois dans la littérature sous diverses formes : patron allocation de ressources[D'Souza95], comportement type de ressource et méta-schéma [Rolland93], etc. Nous verrons qu'il illustre parfaitement le problème de la dualité de l'objet.

Enoncé du problème

Nous considérons deux spécifications de classes représentant respectivement une ressource mono-exemplaire (un livre, un CD dans notre bibiothèque, une machine, un homme dans une entreprise,…) et une ressource multi-exemplaires[13] (un ouvrage de bibliothèque, une équipe, un stock d'entreprise,…).

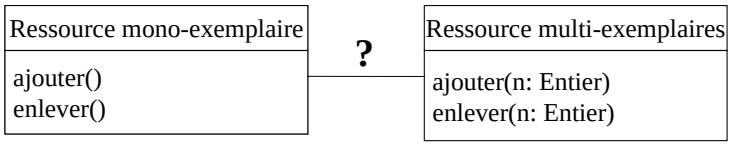


figure 1-4 : Deux types de ressource

12 Les aspects fonctionnels qui sont introduits dans la plupart des méthodes ne sont pas pris en compte. Nous les avons volontairement isolés, considérant qu'il s'agit d'un apport orthogonal dont l'influence sur le modèle global peut être étudiée ultérieurement.

13 On considère que la ressource multi-exemplaires est bornée; il est en effet assez rare de rencontrer des ressources infinies dans les systèmes d'information.

En considérant uniquement la partie comportementale de ces classes, il est aisé d'imaginer qu'il existe une relation forte entre ces dernières qui permettrait de réutiliser la spécification de l'une pour définir l'autre. Notre objectif est de découvrir la nature de cette relation au jour des différents points de vue précédemment introduits dans ce mémoire.

Quatre points de vue sont utilisées pour établir des relations classiques d'héritage ou de délégation entre ces deux ressources :

- Intuitif : L'approche intuitive consiste à définir les états de ces deux ressources puis à les comparer à l'aide des propriétés intuitives qui lient un état à son état parent (cf. § 1.3.1). On cherche à établir une relation d'héritage.
- Opératoire : Il s'agit d'élaborer deux statecharts en utilisant l'approche opératoire introduite au chapitre 1 § 2.1.2 puis en comparant leur conformité (cf. § 1.3.2). On cherche à établir une relation d'héritage.
- Classificatoire : Il s'agit à nouveau d'élaborer les statecharts des deux ressources. Nous utilisons l'approche classificatoire introduite au chapitre 1 § 2.1.1 pour comparer leur conformité (cf. § 1.3.3). On cherche à établir une relation d'héritage.
- Patrons : L'objectif est la réutilisation du savoir-faire détenu par les patrons de conception (cf. chapitre 2 § 1.2.2) pour modéliser le problème. On ne privilégie pas une relation plutôt qu'une autre (héritage et/ou délégation).

1.3.1. APPROCHE INTUITIVE

L'approche que nous qualifions ici d'intuitive fait appel à la connaissance à priori que l'on a des états significatifs d'un objet. Ces états sont souvent extraits lors d'une première lecture du cahier des charges et donc identifiés et spécifiés au plus tôt dans le processus de développement. L'état matrimonial d'une personne est par exemple bien connu de tout le monde : célibataire, concubin, marié, divorcé, veuf, etc. Ainsi, une ressource mono-exemplaire et une ressource multi-exemplaires ont respectivement deux états significatifs, « Disponible » et « Indisponible » (le livre de n°ISBN 0-13-203860-9 est disponible dans la bibliothèque ou est emprunté, i.e. indisponible), et trois états « Pleine », « Vide » et un état intermédiaire « Provisionnée » (tous les exemplaires de l'ouvrage de Cook & Daniels sont empruntés, i.e. l'ensemble des exemplaires est vide).

A partir de ces définitions partielles, qu'on pourrait qualifier de définitions d'analyse [D'Souza96], il est possible de comparer les états découverts en « voyant » une ressource mono-exemplaire au travers des états d'une ressource multi-exemplaires et inversement (figure 1-5).

Vide ⇔ Indisponible

et

Pleine ⇒ Disponible,

Provisionnée ⇒ Disponible,

Disponible ⇒ (Pleine ∨ Provisionnée),

Pleine ⇒ ¬ Provisionnée

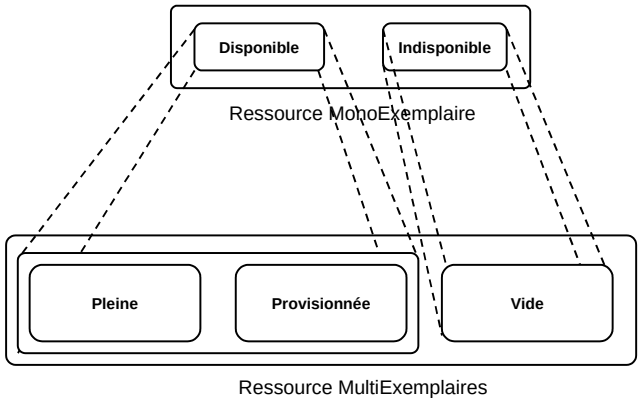


figure 1-5 : Ressources et décomposition d'états

En combinant ces relations non formelles avec le mécanisme de décomposition-OU [Duffy95] des états (cf. chapitre 1 § 1.2.1), on en déduit facilement que l'état Disponible peut être décomposé en deux sous-états Pleine et Provisionnée. La décomposition d'états étant le mécanisme privilégié pour dériver des statecharts [Legrand97], la vision intuitive suggère que la ressource multi-exemplaires soit une sorte de ressource mono-exemplaire dont le statechart est obtenu par décomposition de l'état Disponible.

1.3.2. APPROCHE OPERATOIRE

L'approche opératoire (cf. chapitre 2) définit le cycle de vie d'un objet comme une séquence d'événements autorisée par celui-ci. Chacune des deux ressources autorise certaines séquences d'ajout et de retrait. En étudiant ces séquences et en fonction de la conformité désirée entre ces ressources, on peut en déduire le sens de la relation qui les unit. Pour comparer les ressources, il est nécessaire de réaliser certaines substitutions.

On a

$ajouter_1$: Ressource * Entier -> Ressource

$ajouter_2$: Ressource -> Ressource

En termes de séquencement, nous pouvons écrire les substitutions suivantes :

Soient R ∈ Ressource et n ∈ Entier :

$ajouter_1$ (R, n) → $ajouter_2$ (R) ^{n} et

$ajouter_2$ (R) → $ajouter_1$ (R, 1)

Une ressource mono-exemplaire ne peut subir que des séquences d'ajout et de retrait successives. Nous donnons une trace possible des services rendus par une ressource mono-exemplaire : {$ajouter_2$ (R), $enlever_2$ (R), $ajouter_2$ (R),…} cycle de vie qui peut être décrit par le langage L1=($ajouter_2$ (R).$enlever_2$ (R))* ou après substitution par L2=($ajouter_1$ (R, 1).$enlever_1$ (R, 1))*.

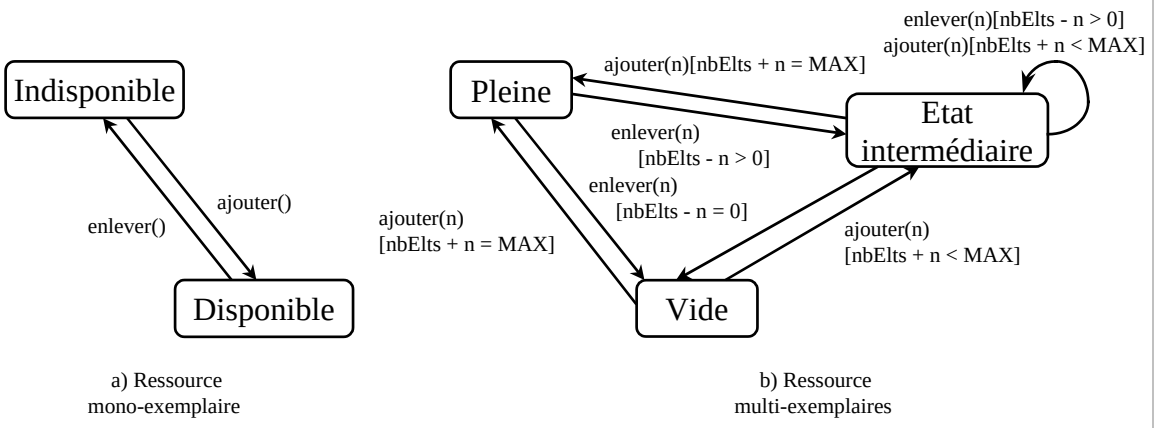


figure 1-6 : Diagrammes d'états de deux ressources

Une ressource multi-exemplaires a au moins autant d'ajouts que de retraits. Nous donnons une trace possible des services rendus par une ressource multi-exemplaires : {ajouter$_1$ (R, 50), enlever$_1$ (R, 20), enlever$_1$ (R, 10),…}, séquence qui peut être décrite par le langage L3 = (ajouter$_1$ (R, n) + enlever$_1$ (R, m))* avec $0 \leq n\text{-}m \leq MAX$.

On a donc L2 = (ajouter$_1$ (R, n).enlever$_1$ (R, m))* avec n=m=1. L'ensemble des cycles de vie décrits par L2 est donc inclus dans l'ensemble des cycles de vie de L3.

D'après les définitions données au chapitre 1 § 2 :

- Une ressource mono-exemplaire est conforme en ascendant avec une ressource multi-exemplaires. En effet, le comportement d'une ressource mono-exemplaire peut être observé au travers du comportement d'une ressource multi-exemplaires (cf. figure 1-6)). Remarquons que cette solution offre peu d'intérêt.
- Une ressource multi-exemplaires est conforme en descendant avec une ressource mono-exemplaire. En effet, l'invocation du comportement d'une ressource mono-exemplaire par un objet qui a le comportement d'une ressource multi-exemplaires est valide mais pas le contraire.

La conformité comportementale « opératoire » ne privilégie donc pas un sens de relation entre ces deux types de ressource (figure 1-7).

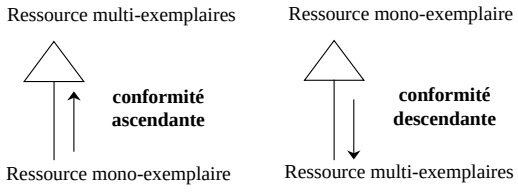


figure 1-7 : Relations et approche « opératoire »

1.3.3. APPROCHE CLASSIFICATOIRE

L'approche classificatoire (cf. chapitre 1 § 2), on l'a vu, travaille plus sur la caractérisation des états par rapport à la structure de la classe que sur les séquences d'événements. Il est donc nécessaire de préciser les attributs des deux ressources. Nous choisissons de les représenter classiquement à l'aide d'un attribut booléen pour la ressource mono-exemplaire et de deux attributs nbElts et max pour la ressource multi-exemplaires. Remarquons que cette spécification ne privilégie pas à priori un sens d'héritage plutôt qu'un autre.

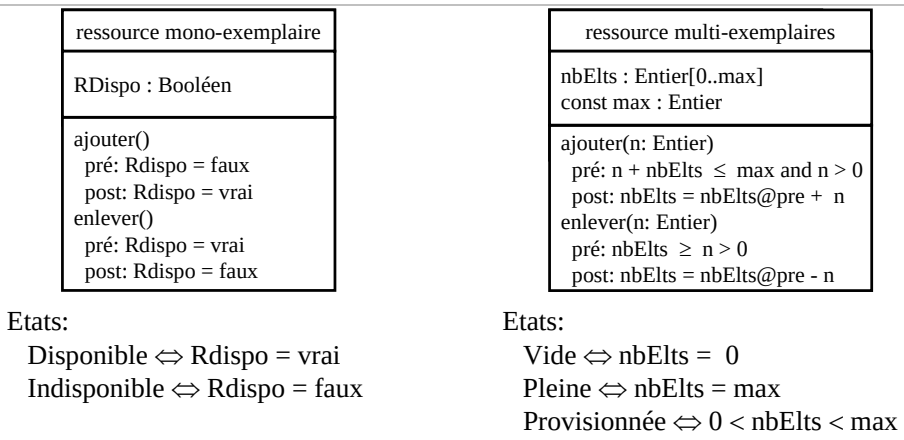


figure 1-8 : Définition de la structure des ressources

D'après les définitions données au chapitre 1 § 2 :

- Une ressource multi-exemplaires est conforme en ascendant avec une ressource mono-exemplaire. En effet, nous avons montré dans l'approche intuitive que tous les états d'une ressource multi-exemplaires peuvent être remplacés par un état plus général dans une ressource mono-exemplaire (figure 1-9).

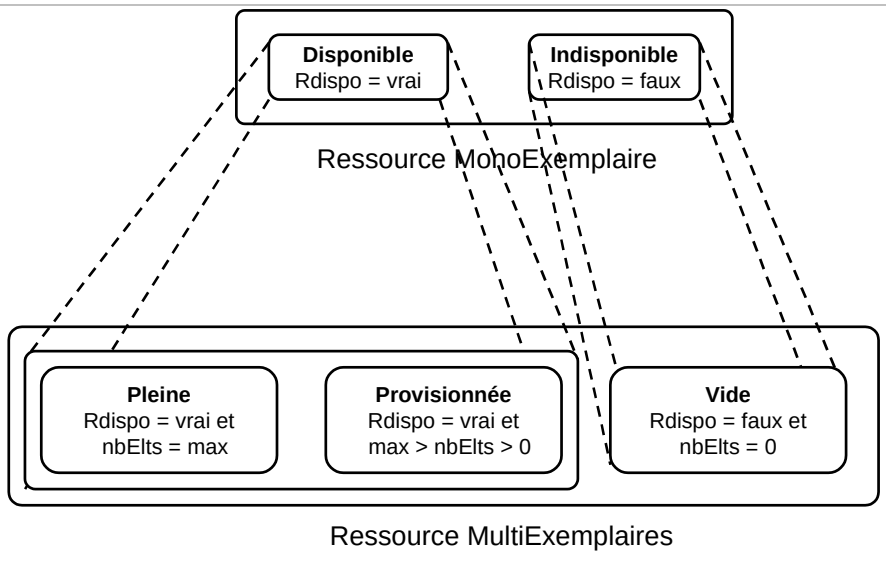


figure 1-9 : Ressources et conformité ascendante

- Une ressource multi-exemplaires est conforme en descendant avec une ressource mono-exemplaire. En effet, l'invocation du comportement d'une ressource mono-exemplaire par un objet qui a le comportement d'une ressource multi-exemplaires est valide et pas le contraire.

On a {Vide **ou** Etat Intermédiaire} ajouter$_1$ {Etat Intermédiaire **ou** Plein}

et {Indisponible} ajouter$_2$ {Disponible}

ajouter$_1$ ne peut être réalisé par composition de ajouter$_2$ alors que ajouter$_2$ peut l'être par ajouter$_1$:

{Indisponible} ajouter$_2$ (R) {Disponible} → {Vide} ajouter$_1$ (R, 1) {Plein}

La conformité comportementale « classificatoire » privilégie donc l'héritage de la ressource mono-exemplaire par la ressource multi-exemplaires (figure 1-10).

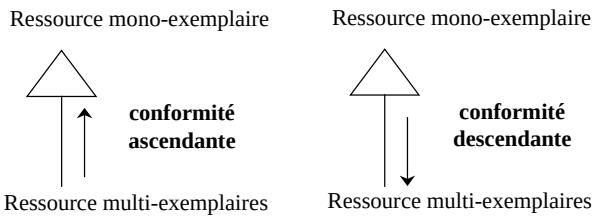


figure 1-10 : Relations et approche « classificatoire »

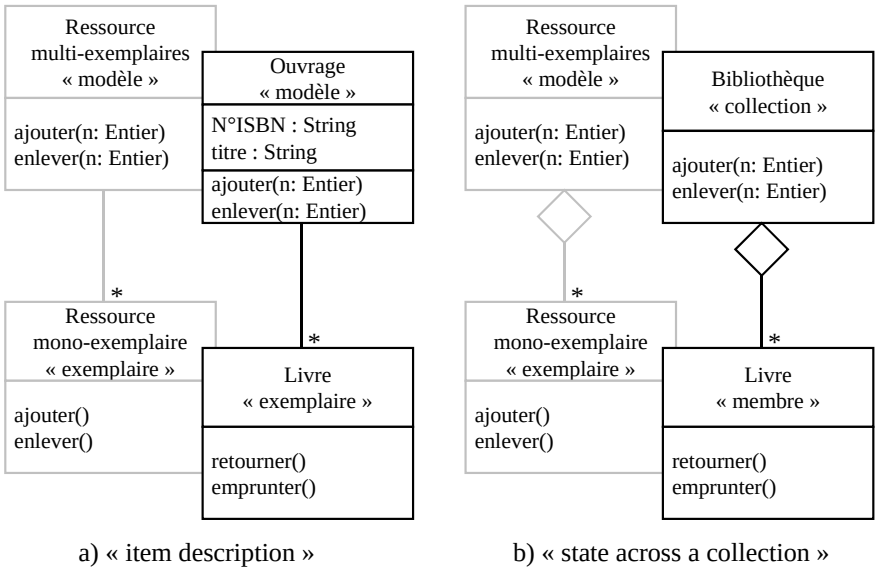
1.3.4. APPROCHE PATRONS

Nous avons vu que les méthodes s'orientent vers plus de réutilisation, les patrons (cf. chapitre 2 § 1.2.2) constituant un formalisme prometteur pour cela. Pour trouver la solution à notre problème, il semble donc naturel de chercher une réponse dans les bibliothèques de patrons existantes. Une première réponse est donnée sous la forme de deux patrons nommés « item description » et « state across a collection » [Coad92]. Ces patrons proposent une solution basée sur la délégation : dans les deux cas, la ressource multi-exemplaires représente l'ensemble des ressources mono-exemplaire (figure 1-11).

P. Coad préconise l'utilisation du patron « item description » (figure 1-11a) pour simuler l'héritage de données entre classes et méta-classes. Il peut être utilisé dans notre cas lorsque des valeurs d'attributs de ressource mono-exemplaire sont applicables à plusieurs instances de cette classe. La ressource multi-exemplaires est utilisée pour représenter une partition de l'ensemble des ressources mono-exemplaires. Par exemple, la classe Ouvrage (figure 1-11a) dans notre application factorise les informations communes aux exemplaires ; Elle détient entre autre le numéro ISBN de l'ouvrage, son titre, etc.

L'auteur préconise l'utilisation du patron « state across a collection » (figure 1-11b) pour représenter des classes conteneurs (i.e. collections). L'idée de partition est absente dans ce patron.

Ainsi, le lien qui existe entre la bibliothèque et ses ouvrages, ses abonnés ou ses livres (figure 1-11b) peut être modélisé à l'aide de ce patron.



a) « item description » b) « state across a collection »

figure 1-11 : Deux adaptations de patrons

La délégation semble séduisante structurellement car elle nous ramène à des solutions connues. Cependant, dans ce cas, mais elle ne favorise pas la réutilisation car le comportement de la ressource multi-exemplaires n'est pas exprimable en fonction de celui de la ressource mono-exemplaire (et encore moins le contraire). Pour l'améliorer, nous utilisons le patron composite de E. Gamma [Gamma95]. Celui-ci consiste à introduire une classe abstraite ressource factorisant à la fois l'attribut nbElts et les deux opérations abstraites ajouter et enlever. La ressource mono-exemplaire redéfinit l'attribut nbElts en restreignant son domaine. Les deux sous-classes réalisent chacune les deux opérations abstraites.

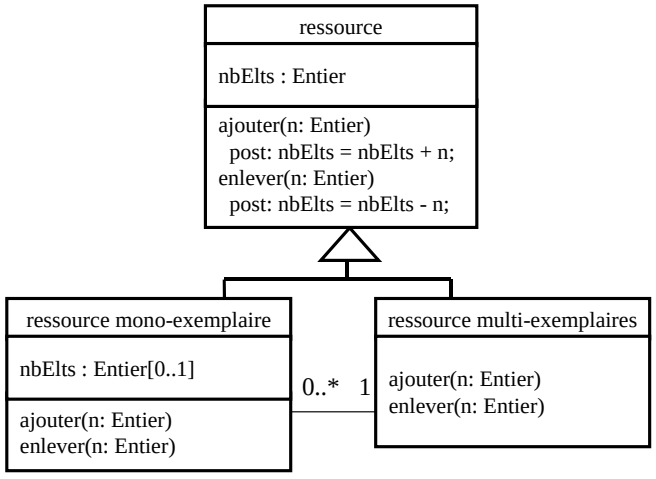


figure 1-12 : Application du patron Composite [Gamma95]

Là encore cette solution n'est pas satisfaisante car elle ne correspond pas à ce que l'on voulait exprimer au début. Le problème initial a été modifié sur plusieurs points : ajout d'un attribut nbElts pour une ressource mono-exemplaire, ajout de paramètres aux deux opérations de la ressource mono-exemplaire pour conserver le polymorphisme. Nous pouvons d'ailleurs remarquer que ces ajouts ne sont pas naturels et que la réutilisation ainsi obtenue perd de son intérêt.

1.3.5. DISSERTATION

Pour définir la nature de la relation qui unit une ressource mono-exemplaire à une ressource multi-exemplaires, nous avons utilisé des solutions proposées dans la littérature sur le comportement d'objet. Comme le montre le tableau de synthèse ci-dessous, il n'y a pas de solution tranchée à notre problème : relation de délégation, d'héritage, sens de cette relation équivoque.

Approche	**Type de relation**	**Sens de la relation**	**Conformité obtenue**	**Caractéristiques**	**Intégration dans le processus de développement**
Approche intuitive	Héritage	mono hérite de multi*	ascendante	non formelle	Analyse
Approche opératoire	Héritage	mono hérite de multi	ascendante	événementielle adaptée aux méthodes dirigées par le comportement	Analyse (-) & Conception (+)
		multi hérite de mono	descendante		
Approche classificatoire	Héritage	multi hérite de mono	ascendante & descendante[14]	structurelle formalisation de la cohérence données/états	Conception
Approche patrons	Délégation	multi délègue à mono		solution consensuelle	Analyse & Conception

* mono = Ressource Mono-exemplaires, multi = Ressource multi-exemplaires

Tableau 1-1 : quatre approches du comportement

Il n'y a donc pas une solution unique à notre problème de départ et encore moins une meilleure solution. Le choix de l'une ou l'autre dépend des caractéristiques recherchées (cf. Tableau 1-1 Caractéristiques). Avant de présenter notre solution, il est nécessaire de faire un retour sur nos choix méthodologiques :

- Ne pas privilégier un modèle.

  Toutes les approches étudiées « mélangent » spécifications comportementales et spécifications structurelles pour garantir une meilleure cohérence. L'approche opératoire par

[14] Ce résultat est en apparence contradictoire avec le résultat obtenu en vérifiant la conformité descendante à l'aide de la définition opératoire. Ceci illustre le fait qu'on peut obtenir une conformité comportementale « classificatoire » entre deux statecharts sans pour autant préserver le séquencement. ; ce qui n'est bien sûr pas possible avec la conformité comportementale « opératoire ».

exemple modifie les hiérarchies de classes pour prendre en compte les aspects dynamiques. Inversement, l'approche classificatoire construit la dynamique d'un objet à partir de la connaissance structurelle de cet objet. Ces approches favorisent un « style » de spécification au détriment de l'autre. C'est ainsi que l'utilisation du patron composite de E. Gamma produit une solution structurelle qui n'est pas naturelle (cf. figure 1-12). Au contraire, notre solution doit donner parts égales aux représentations structurelles et comportementales des objets. Pour ce faire, nous préconisons de construire les classes et les statecharts séparément et ce afin de « casser » la dépendance des statecharts vis-à-vis des classes du système.

- Offrir un modèle de spécification naturel et flexible.

  De nombreux auteurs insistent sur la nécessité de prévenir au plus vite les incohérences dans le cycle de vie du logiciel. C'est pourquoi l'idée de contrôler très tôt la dynamique et même de l'utiliser pour construire les objets du système est séduisante. Pour ce faire, la démarche doit être la plus naturelle possible. L'approche opératoire qui est une démarche incrémentale pour la construction des statecharts nous semble toute indiquée. Couplée à une vision intuitive de la redéfinition des statecharts (cf. § 1.3.1), elle permet l'intégration de ces derniers dans le processus de développement dès la phase d'analyse du système.

- Favoriser la réutilisation du comportement d'objet.

L'approche intuitive est intéressante en termes de découverte des états. Le mécanisme de décomposition-OU des états est assez simple et puissant pour qu'on ne retienne que lui lors de la dérivation de statecharts. Cependant, elle n'est pas suffisante lorsqu'il s'agit de réutiliser les statecharts dans un modèle à objets car porteuse de peu d'informations sur les propriétés que doit vérifier l'objet dans tel ou tel état. En caractérisant les états à la manière de l'approche classificatoire, il est possible de construire des hiérarchies de statecharts réutilisables au même titre que des hiérarchies de classes.

1.4. La solution NCR

1.4.1. UNE DEMARCHE ALTERNATIVE

Nous proposons une spécification dissociée des aspects structurels et comportementaux. Pour cela, nous introduisons deux dimensions reposant respectivement sur l'expression de notions, sortes de classes précisant les propriétés consensuelles des objets du système, et de comportements utilisant le formalisme des statecharts pour exprimer le contrôle des objets. L'intégration est réalisée dès le niveau conceptuel par une troisième dimension qui assure l'intégrité globale du système (figure 1-13). Cette dimension est décrite à l'aide des rôles, concept que l'on peut intuitivement rapprocher de celui d'interface. Chaque rôle peut spécifier une intégration à la fois précise et différente pour des spécifications structurelles et comportementales

données. L'implantation des modèles peut être réalisée de diverses manières (figure 1-13), à partir :

- Des notions vers un modèle exécutable classique.
- Des comportements vers l'implantation directe d'une machine à états finis.
- Des rôles. On obtient alors un modèle exécutable global qui prend pleinement en compte les aspects structurels (les données) et comportementaux (le contrôle).

Nous proposons une nouvelle approche qui consiste à construire les spécifications structurelles et comportementales indépendamment. L'intégration est réalisée à l'aide de règles conceptuelles exprimées dans les rôles et qui assurent la cohérence. La démarche est supportée tout le long du processus de développement par le modèle NCR (cf. § 1.4.2). En autorisant une formalisation de l'intégration des modèles statique et dynamique, le modèle doit offrir une meilleure traçabilité du processus de conception ainsi qu'une démarche plus rigoureuse orientée réutilisation des comportements.

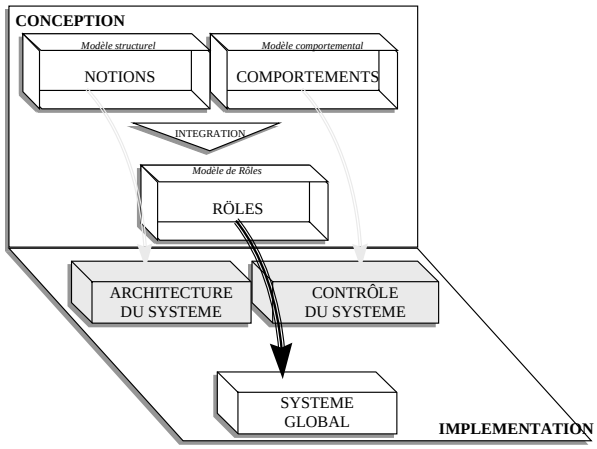


figure 1-13 : démarche proposée

1.4.2. LE MODELE NCR

1.4.2.1. Concepts

NCR est un acronyme pour désigner les trois composants de base de notre modèle que sont la Notion, le Comportement et le Rôle. Chacun d'eux est un paradigme de modélisation spécialisable. Ils déterminent trois dimensions qui sont respectivement structurelle, comportementale et phénoménale.

Notion

Une notion ou classe notion est l'unité modulaire de notre dimension structurelle, elle définit les attributs et méthodes d'un objet du domaine de l'application. Chaque notion ne représente que les propriétés « intrinsèques » de l'objet indépendamment des applications dans lesquelles il évolue. Le terme « intrinsèque » doit être précisé. Il peut s'agir de propriétés très générales telles que le numéro ISBN d'un ouvrage, son auteur, etc. ou de propriétés spécifiques à un métier voir à une organisation. Par exemple une bibliothèque peut imposer un nombre maximal d'emprunts par lecteur, des conditions de maintenance particulières pour ces ouvrages, etc.

Pour favoriser la réutilisation des notions en évitant la création de classes creuses [Rieu91], les notions ne sont pas limitées à la description de concepts abstraits. La spécialisation de classes notions est plutôt vue comme une spécialisation catégorielle qui permet la classification d'objets,

par opposition à la spécialisation par états traitée par les comportements. Le concepteur a donc le libre choix des attributs des notions.

Les notions font partie intégrante d'un diagramme de classes notions qui décrit les relations qu'elles entretiennent. Celui-ci est spécifié actuellement à l'aide des diagrammes de classes UML. Les diagrammes de séquences ou de collaboration peuvent également être utilisés pour la spécification des fonctionnalités des notions.

Exemples de notions d'un système d'information d'entreprise : Employé, Commande, Article, Document, etc. Une notion correspond la plupart du temps à un objet métier du domaine d'application.

Comportement

Un comportement est l'unité modulaire de la dimension comportementale, il représente une évolution réutilisable d'objet à l'aide d'un statechart. Notons aussi qu'ici il peut s'agir d'un comportement très général adaptable à tout domaine. Alors que les notions expriment des propriétés statiques de l'entreprise, les comportements expriment des propriétés comportementales : une règle de gestion d'une bibliothèque peut interdire la remise en rayon d'un ouvrage si celui ci a été réservé par un abonné.

Un comportement d'objet est défini de manière générale comme :

Un Comportement d'objet est une représentation d'une évolution possible d'un objet selon une ou plusieurs caractéristiques identifiables et calculables [S^t-Marcel98].

Intuitivement nous disons qu'un comportement est une "manière" de faire le tri dans l'ensemble des informations concernant un objet, chaque comportement isolant les propriétés essentielles pour décrire une évolution possible de cet objet. Les Cycles de Vie des Objets (CVO) proposés dans Merise 2 [Panet94] constituent une très bonne illustration de cette définition. Un CVO (un **C**omportement) est constitué de plusieurs structures parallèles, chacune d'elle modélisant une évolution possible de l'objet. A chaque structure parallèle est associée une variable d'état (l'ensemble des caractéristiques d'évolution de NCR).

Exemples de comportements réutilisables : l'ingénierie des produits, l'occupation d'une ressource, la maintenance d'un produit, etc.

Sur la figure 1-14, est représenté graphiquement un comportement de « Ressource Limitée ».

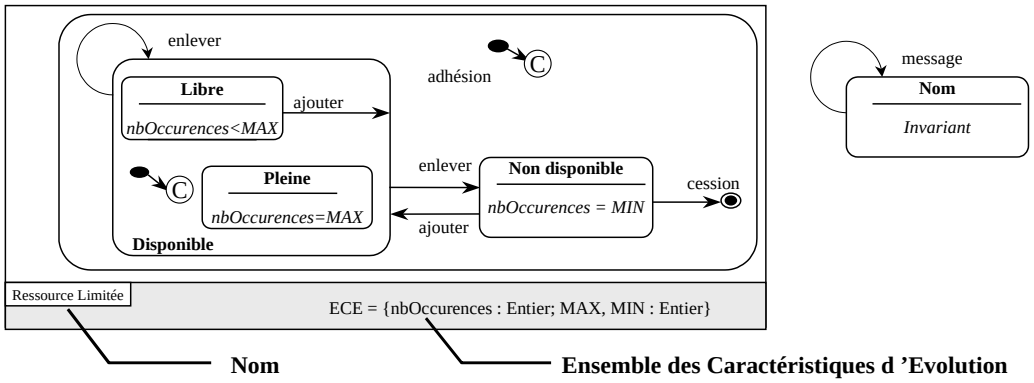


figure 1-14 : Comportement « Ressource Limitée »

Chaque comportement est associé à un Ensemble de Caractéristiques d'Evolutions (ECE) dont les valeurs caractérisent les états. Dans NCR, un objet qui veut rester dans un état doit vérifier l'invariant de cet état. C'est ainsi qu'un objet adhèrant au comportement RessourceLimitée (figure 1-14) doit vérifier nbOccurences = MIN pour se rendre Non Disponible.

Rôle

Un rôle est l'unité modulaire de la dimension phénomale[15]. C'est au sein d'un rôle que l'on combine une notion et un comportement pour obtenir un modèle objet complet pour une application donnée. C'est ainsi que dans une application de gestion de prêts d'une bibliothèque la notion de livre est animée par un comportement de ressource pour donner un LivreEmpruntable (figure 1-15).

Le rôle est le seul générateur d'instances dans NCR, c'est à lui que revient la charge de faire évoluer les objets en contrôlant leur comportement. Une instance d'un rôle est appelée phénobjet[16]. Le rôle sert de " charnière " conceptuelle entre les dimensions statique et dynamique : en général un rôle anime une et une seule notion selon un unique comportement qu'il concrétise. Il s'agit d'un véritable connecteur entre ces deux paradigmes auxquels il est lié par les relations "anime" et "concrétise".

Un rôle peut admettre des propriétés spécifiques qui ne sont alors ni une utilisation des propriétés de sa notion ni une réalisation des propriétés de son comportement. Il s'agit de propriétés liées à la mis en œuvre de l'application.

Exemples de rôles : l'employé en tant que ressource de l'entreprise, les produits en cours de développement, etc.

---

[15] Phénoménal : (Philos.) Qui concerne les apparences des choses : le monde phénoménal s'oppose au monde nouménal des choses en soi.

[16] Phénobjet est la contraction syntaxique de " Phénomène Objet ", ce mot établit une analogie avec le domaine génétique où un phénotype désigne l'ensemble des caractères qui se manifestent visiblement chez un individu [Petit Larousse]

Pour illustrer NCR, nous instancions ses concepts dans le contexte particulier de notre gestion de bibliothèque. Un premier exemple d'instanciation du triptyque NCR est donné pour la notion Livre.

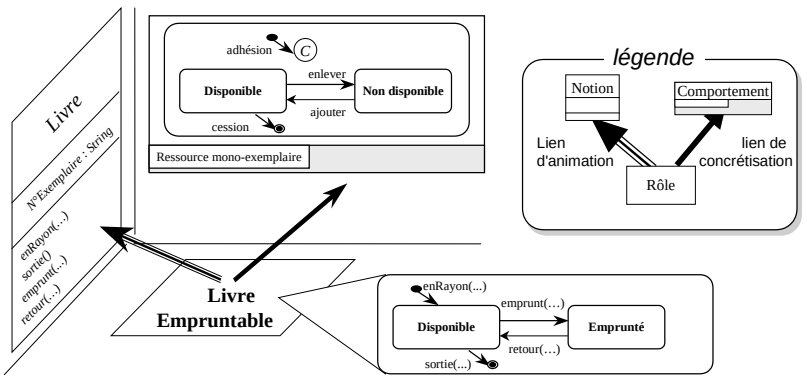


figure 1-15: Instanciation du triptyque NCR

La notion Livre, qui fait partie de la dimension structurelle, donne une spécification générale d'un livre. Nous voulons exploiter cette spécification dans le cadre de la gestion des prêts de la bibliothèque. La dimension comportementale fournit, en l'occurrence, un comportement RessourceMono-Exemplaire. La correspondance entre la notion Livre et le comportement Ressource Mono-exemplaire qu'on veut lui donner est établie par le rôle LivreEmpruntable dont on donne un aperçu graphique sur la figure 1-15. Celui-ci assure par exemple le renommage de l'état Non Disponible, la fusion des méthodes emprunt() et retour() respectivement avec les transitions enlever et ajouter. Nous pouvons remarquer qu'à la création d'une instance de rôle, l'état d'entrée par défaut est Disponible. Il s'agit là d'une restriction par rapport au choix laissé dans le comportement (marqué par un ©).

On retrouve des variantes de « triptyques de modélisation » dans les méthodes actuelles (voir Syntropy [Cook94] ou Classe-Relation [Desfray94]) avec des sémantiques différentes portées par les trois dimensions. Ici, la dimension comportementale représente des évolutions génériques que l'on peut retrouver « dans la nature » pour un ensemble d'objets alors que la dimension de rôles décrit l'évolution effective d'un objet. Ce modèle établit clairement le distinguo entre les évolutions possibles (comportements) et les évolutions effectives (rôles).

L'originalité de notre approche est liée à la « priorité » qui est donnée aux modèles. Ici, la structure n'est plus favorisée par rapport au comportement, chacun étant dissocié et réutilisé indépendamment. Cependant, on ne force pas à une approche comportementale systématique, on la favorise seulement. Le concepteur n'a donc pas à décrire de manière exhaustive le comportement d'un objet mais seulement les parties qu'il veut contrôler : sur l'exemple de la figure 1-15 est spécifié le comportement du livre dans le processus de prêt et, volontairement abstrait, celui lié aux processus de réservation et de maintenance.

1.4.2.2. Principes

Avant de préciser les aspects sémantiques du modèle (cf. § 2), nous proposons dans ce paragraphe un aperçu des potentialités offertes part NCR en termes de modélisation des systèmes d'information. Nous utilisons pour cela les trois principes de base que sont l'animation, la concrétisation et la fusion.

Animation (Notion → Rôle)

La dimension comportementale est par nature abstraite. Par construction, la dimension des notions l'est aussi. Seule la dimension phénoménale est instanciable. Le premier objectif d'un rôle est donc l'instanciation des notions qui sont utilisées dans l'application. Celle-ci est rendue possible par le lien d'animation qui existe entre une notion et un rôle.

Nous laissons au concepteur la possibilité de spécifier une application en utilisant uniquement l'animation de notions : cela a pour effet de construire un schéma de rôles isomorphe à celui des notions. Cette solution va évidemment à l'encontre de notre intérêt qu'est l'utilisation de descriptions comportementales.

Concrétisation (Comportement → Rôle)

La concrétisation associe des propriétés héritées du comportement et donc par nature abstraites à des propriétés concrètes manipulées dans le modèle de rôles. Ce type de mécanisme est utilisé lorsque l'héritage des propriétés du comportement est orthogonal à celui des propriétés de la notion.

Supposons que LivreEmpruntable de la figure 1-15 n'implante pas les méthodes retour(...) et emprunt(...) spécifiques au prêt. L'étude du processus de prêt dans la bibliothèque nous amène inévitablement à enrichir cette Notion. Dans une application classique, cet ajout serait réalisé par sous-classement de la notion. Ici il peut être réalisé par « ajout » comportemental au niveau de la dimension des rôles. Le rôle est alors utilisé comme une extension comportementale dont il doit définir sa réalisation et son intégration avec la structure existante.

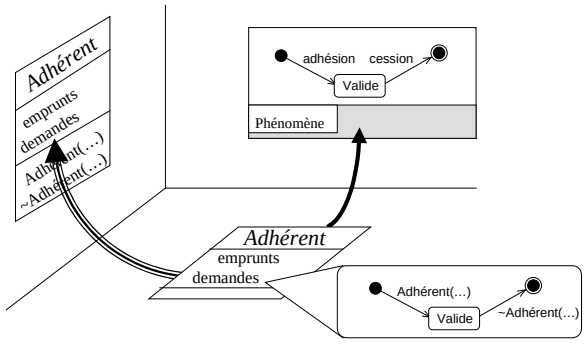


figure 1-16: Concrétisation de Phénomène

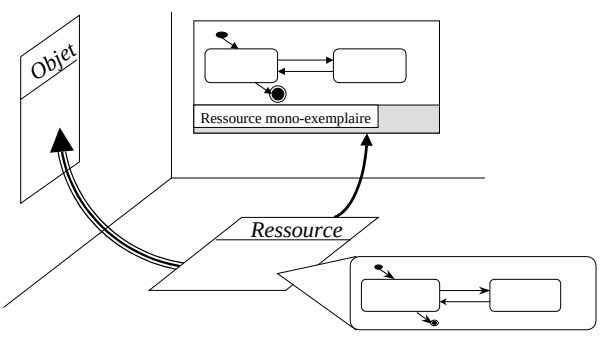


figure 1-17 : Animation d'Objet

Les figures figure 1-16 et figure 1-17 présentent les deux visions obtenues respectivement

par concrétisation de la racine comportementale Phénomène et par animation de la racine structurelle Objet. Ces deux racines sont neutres pour l'intégration, i.e. qu'elles ne modifient pas la sémantique de la notion ou du comportement lors de leur intégration respective.

La première figure montre qu'il est possible de s'abstraire du comportement d'un objet, soit parce que ce comportement n'est pas intéressant ou simplement parce qu'il n'est pas encore connu. On peut par exemple spécifier le rôle d'un adhérent de la bibliothèque à l'aide de ce comportement minimal si seules les propriétés de la notion sont pertinentes.

La racine des notions permet de manière duale d'abstraire des propriétés structurelles pour ne considérer que des aspects comportementaux. Elle est souvent utilisée pour tester la dynamique des comportements, i.e. la validité des automates décrits par chaque comportement.

Intégration (Notion, Comportement → Rôle)

La pleine richesse du modèle NCR est dans l'intégration des concepts manipulés dans la dimension structurelle avec ceux manipulés dans la dimension comportementale. Celle-ci est rendue possible par l'introduction de véritables connecteurs qui assurent le « pont logique » entre les propriétés des notions et des comportements . De manière simplifiée, on peut dire qu'ils définissent comment et sous quelles conditions peuvent être associés dans un rôle les attributs et les méthodes de la notion animée avec les caractéristiques d'évolution, les états et les transitions définies dans le comportement concrétisé par ce rôle. La figure 1-18 présente la vue graphique d'un rôle après intégration d'une notion et d'un rôle.

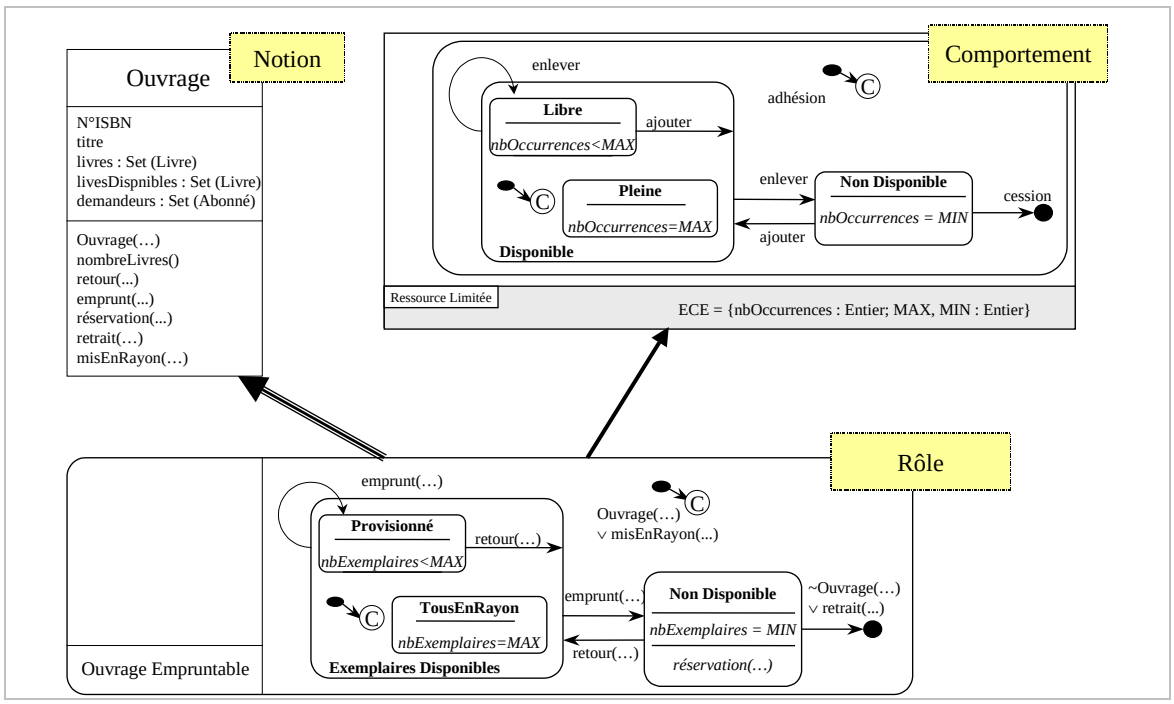


figure 1-18: Intégration graphique d'une notion et d'un comportement

L'intégration permet l'obtention d'une véritable vision duale des objets du système, soit par la conservation du paradigme objet classique avec envois de messages qui assurent la sélection des méthodes de notions, soit en utilisant un modèle événementiel propre aux statecharts avec déclenchement d'actions (cf. chapitre 4 § 2.3).

1.5. Conclusion

L'idée d'une réutilisation par la fonction a toujours été d'actualité dans les méthodes de conception. Aujourd'hui, on sait que les données sont plus « stables » que les traitements. La réutilisation s'est naturellement orientée vers les objets. Or, nous pensons que la fonction est toujours d'actualité dans le monde à objets. Tout d'abord nous savons qu'il est parfois nécessaire de partitionner les services d'un objet sans pour autant qu'il ait à changer d'identité. La nombreuse littérature sur les vues, points de vue, rôles et états en fait foi. Ensuite, nous avons vu qu'il existe des objets qui réalisent les mêmes services, une porte s'ouvre et se ferme comme un coffre ou un compte bancaire sans pour autant qu'une porte soit un coffre ou inversement qu'un coffre soit une porte. C'est lors de la modélisation de telles situations qu'un modèle monolithique montre ses limites en termes de représentation et de réutilisation. Le modèle NCR va au-delà en proposant une vision duale et intégrée de l'objet.

Nous avons vu dans cet « exercice de l'art » que la réutilisation des spécifications comportementales dans le monde à objets est un problème délicat et pour l'instant non consensuel. Les statecharts restent un modèle difficile à appréhender et à concilier avec la spécialisation et la délégation de classes. Bien que certains auteurs proposent aujourd'hui des solutions élégantes pour marier les statecharts (ou diagrammes d'états) avec le paradigme objet, ils maintiennent une relation de dépendance forte entre eux qui nuit, à notre avis, à leur réutilisation. Aujourd'hui, les méthodes de conception nous ont montré qu'il faut « partager pour mieux utiliser », alors pourquoi ne pas « partager pour mieux réutiliser » ?

2. SPECIFICATION DU MODELE NCR

« Et si Platon avait été informaticien ! »

Pré-requis

Dans ce chapitre, nous utilisons la notation unifiée [UML97] pour décrire les concepts du modèle NCR associée au langage d'expression navigationnel OCL[17] (Object Constraint Language [OCL97]) pour enrichir la description graphique. Nous faisons ici l'hypothèse que le lecteur est familier avec les techniques de méta-modélisation qui vont être utilisées pour formaliser le modèle NCR.

Plutôt que de présenter un métamodèle global du modèle NCR, nous introduisons au fur et à mesure du discours les concepts nécessaires. Lorsque cela est possible, nous donnons un exemple d'instanciation du méta-modèle en utilisant cette fois la notation propre au modèle NCR.

La figure ci-contre donne une vision macroscopique des trois concepts du modèle NCR, tous issus d'une racine commune appelée *Paradigme*. Elle rappelle que seuls les rôles sont instanciables dans le modèle NCR et que leurs instances sont appelées phénobjets.

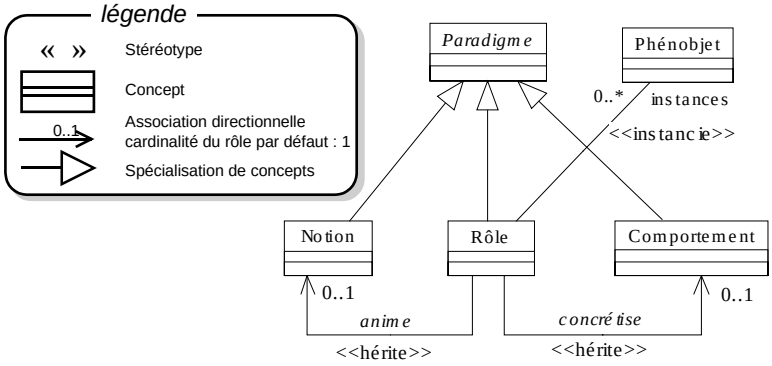


figure 2-1: Les paradigmes NCR

Les associations « anime » du rôle vers la notion et « concrétise » du rôle vers le comportement sont typées à l'aide du stéréotype « hérite » marquant ainsi la nature des relations qui lieront les instances de ces concepts. Nous conservons le lien d'héritage d'UML pour la

[17] Nous utilisons pour vérifier la validité de type des expressions OCL le parser écrit par J. Warmer (http://www.software.ibm.com/ad/standards/ocl.html/)

classification des concepts du métamodèle. Ainsi Notion, Comportement et Rôle sont des concepts qui héritent du concept de Paradigme.

Le langage OCL est un langage navigationnel basé sur des expressions qui sont garanties sans effet de bord sur le modèle. Les expressions OCL sont utilisées pour :

- Spécifier les invariants de classes.

  Le contexte de l'expression, i.e. la classe dont on veut décrire l'invariant, est souligné et précède toujours l'expression. On utilise le mot clé « self » pour désigner une instance du contexte. L'invariant d'une classe représente ce que doit toujours vérifier une instance de cette classe. Nous donnons ci-dessous un exemple arbitraire de contrainte OCL sur le schéma conceptuel de la figure 2-1.

  Rôle -- *Contexte*

  [1] Un rôle a au plus 99 instances -- *Contrainte exprimée en langage naturel.*

  self.instances->size < 100 ; -- *Expression OCL*

- Décrire les assertions des méthodes de classe.

  On utilise ici deux mots clés optionnels « pre » et « post » pour déclarer les assertions. Le « self » désigne là encore l'instance de la classe qui définit la méthode. « result » est un mot clé optionnel qui définit le nom de l'objet retourné.

  Rôle : : getInstances : Set(Phénobjet) -- *Signature de la méthode*

  pre : … -- *pré-condition*

  post : result = self.instances ; -- *post-condition*

2.1. Domaines

Les trois dimensions du modèle NCR présentent des similitudes. Nous avons donc structuré l'information de manière similaire dans les différentes dimensions. De manière analogue, on retrouve une symétrie ternaire dans la méthode Syntropy [Cook94].

Les domaines sont des unités logiques de modélisation qui structurent l'information dans chaque dimension. Par analogie, on peut comparer un domaine à un paquetage (UML, java) à la différence qu'un paquetage peut généralement contenir plusieurs modèles ou concepts de nature sémantique différente, ce qui n'est pas le cas ici.

Il n'y a pas de règles précises pour l'élaboration des domaines. Tout dépend de l'application cible et de son architecture.

Exemples de domaines : Nous avons déterminé deux grands domaines comportementaux qui offrent une grande généricité et sont facilement réutilisables. Le premier concerne la gestion

de ressources et repose sur des critères de disponibilité et de partage (une pièce est disponible pour l'usinage). Le second repose sur la description des cycles de vie d'objet (une pièce est en cours d'usinage).

Domaine

[1] Un domaine est composé d'objets de même nature.

```
self.paradigmes->forAll(p1, p2|p1.oclType[18] = p2.oclType) ;
```

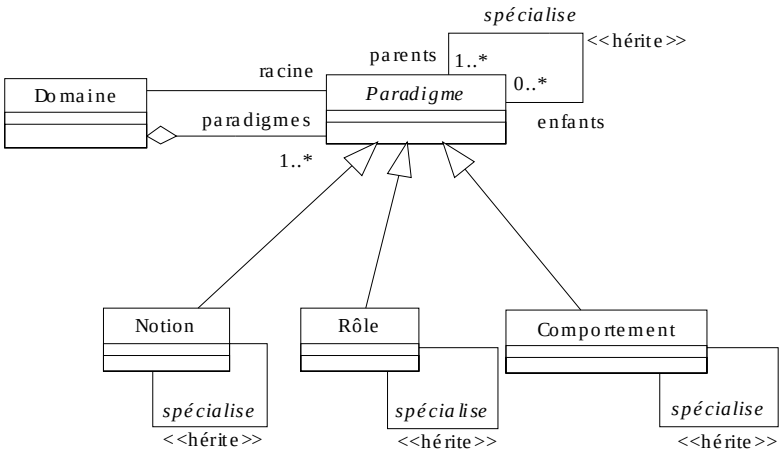


figure 2-2 : domaines NCR

Un domaine est une hiérarchie de paradigmes qui admet une unique racine à partir de laquelle sont spécialisés l'ensemble des paradigmes du domaine. L'ensemble des parents d'un paradigme est obtenu à l'aide de l'opération "tousLesParents". Le résultat est toujours un ensemble de paradigmes contenant le paradigme lui-même.

Domaine

[2] Un domaine admet une unique racine dont tous les comportements du domaine héritent. Cette racine fait partie du domaine.

```
self.paradigmes->forAll(p : Paradigme | p.tousLesParents->includes(self.racine)) ;
```

où tousLesParents est défini par :

Paradigme : :tousLesParents() : Set(Paradigme)

```
post : result = self.parents->iterate(p : Paradigme ; acc : Set{} |
                                        acc->union(p.tousLesParents())->including(self) ;
```

[18] oclType est un opérateur prédéfini dans OCL qui renvoie le type statique de l'instance.

Paradigme

[1] Tout paradigme du domaine, à l'exception de la racine a des parents de même domaine et donc par transitivité avec la contrainte Domaine [1] des parents de même type.

```
(self <> self.domaine.racine) implies self.parents->forAll(d | d.domaine = self.domaine) ;
```

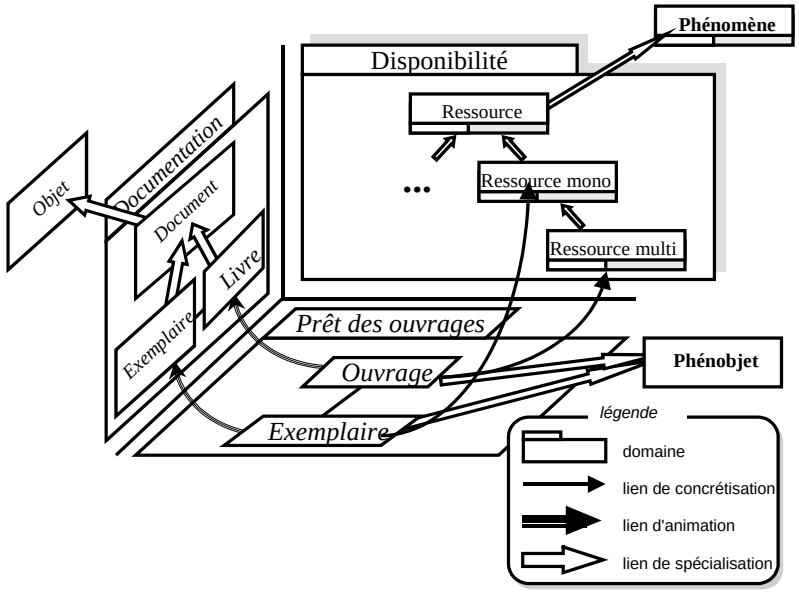


figure 2-3 : Domaines de modélisation NCR

La définition complète des domaines utilisés pour la bibliothèque n'est pas présentée ici. Nous utilisons principalement le domaine Documentation pour structurer l'ensemble des documents manipulés par la bibliothèque (livres, revues, bandes dessinées, etc.) et le domaine Disponibilité pour la définition comportementale de ces documents : celui-ci définit des comportements génériques basés sur l'évolution de ressources mono ou multi-exemplaires, partagées, etc.

Cas particulier des éléments neutres

Chaque élément neutre est la racine d'héritage de sa dimension à partir de laquelle sont définies toutes les racines de domaines. Un élément neutre ne fait partie d'aucun domaine.

Notion

```
self.domaine.racine = self implies (self.parents->exists(Objet) and self.parents->size = 1) ;
```

Comportement

```
self.domaine.racine = self implies (self.parents->exists(Phénomène) and self.parents->size = 1) ;
```

Role

```
self.domaine.racine = self implies (self.parents->exists(Phénobjet) and self.parents->size = 1) ;
```

Par construction, Phénobjet admet pour parents Objet et Phénomène :

Rôle

Phénobjet.parents->includes({Objet, Phénomène}) :

2.2. Propriétés

2.2.1 CLASSIFICATION DES PROPRIETES

Tout paradigme (une notion, un rôle, un comportement) admet des propriétés typées qui peuvent être classifiées en fonction de leur nature structurelle, opérationnelle ou contractuelle (figure 2-4) :

- Une propriété structurelle stocke une information. Un attribut de classe, un paramètre de méthode sont des exemples intuitifs de propriétés structurelles rencontrées dans les modèles à objets classiques.

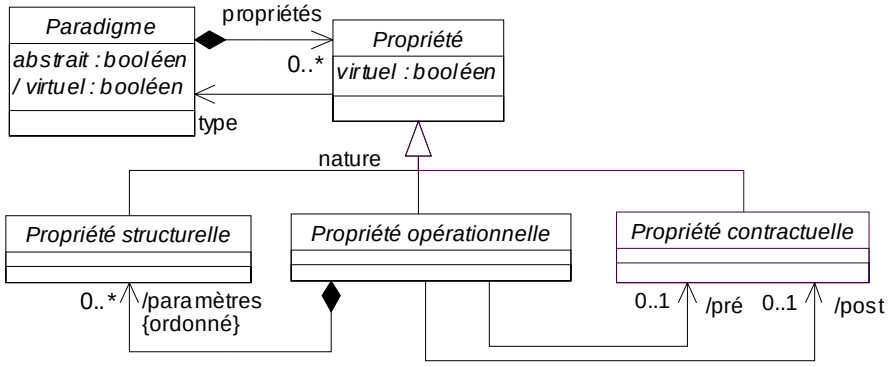


figure 2-4: Propriétés d'un paradigme

- Une propriété opérationnelle traite une information. Elle a une signature qui décrit un ensemble ordonné de paramètres typés. Deux assertions (pré et post-conditions) peuvent être utilisées pour décrire la sémantique des propriétés opérationnelles. Une méthode de classe est un exemple de propriété opérationnelle.
- Une propriété contractuelle[19] vérifie la validité d'une information. Elle est composée d'une expression logique sur des propriétés.

2.2.2. PROPRIETES & ABSTRACTION

Une propriété spécifiée à l'aide du modèle NCR est dite virtuelle si on connaît son domaine de valuation (le type) mais pas la manière de la valuer. Une propriété non implantée est par définition virtuelle. Le concept de virtualité est opposé à celui de réalité. On parlera plutôt de la virtualité d'une propriété d'un paradigme que de la virtualité du paradigme lui-même. En effet la

[19] La notion de contrat a été définie par B. Meyer et mise en œuvre dans le langage EIFFEL. On retrouve cette notion dans la méthode CRC (Classes - Responsabilités - Collaborations) [Wirfs-Brock90].

seconde peut être déduite de la première : un paradigme est virtuel si et seulement si au moins une de ses propriétés est virtuelle.

On dit qu'un paradigme est abstrait s'il ne peut avoir d'occurrences. Les notions et les comportements sont abstraits dans NCR ; seuls les rôles sont des paradigmes concrets générateurs d'instances.

Nous avons la relation fondamentale virtuel ⇒ abstrait qui fait qu'un Rôle non complètement défini est abstrait. Pour être concret, un rôle doit réaliser (rendre réelles) toutes les propriétés du comportement concrétisé et ainsi que les propriétés virtuelles de la notion animée. Nous détaillons cet aspect dans le chapitre concepts avancés (cf. chapitre 4 § 1.1).

Paradigme : :virtuel() : Booléen
post : result = self.toutesLesPropriétés()[20]->exists (prop : Propriété | prop.virtuel) ;

Domaine

[3] Seuls les rôles peuvent être instanciés. Les dimensions structurelles et comportementales restent abstraites.
self.paradigmes->forAll(p|(p.oclType = Notion **or** p.oclType = Comportement) **implies** p.abstrait) ;

2.3. Modèle structurel

La description du modèle structurel est volontairement courte, ceci pour deux raisons.

- Dès le départ, nous avons choisi d'axer notre travail plutôt sur les modèles qui offrent une vision dynamique du système d'information. Le lecteur ne trouvera donc rien d'innovant dans cette partie. Le modèle présenté ici est né du soucis d'homogénéiser les dimensions du modèle NCR en ne gardant que les concepts essentiels et fédérateurs dans chaque dimension.
- Autant il nous parait indispensable d'introduire un nouveau modèle dynamique ou plutôt d'en donner une nouvelle sémantique en fonction des critères de réutilisation et de simplicité que nous nous sommes fixés. Autant ce choix ne se justifie pas pour le modèle statique. En choisissant un sous-ensemble minimal de concepts, nous pensons qu'il sera plus facile d'intégrer l'architecture NCR et plus particulièrement le modèle dynamique avec des modèles existants (diagrammes de classes [UML97], modèle E/R [Chen76], modèle Classe/Relation [Desfray94], etc).

Sur la figure 2-4, nous avons établi une première classification des propriétés selon leur nature. Nous établissons ici une seconde classification en fonction de la dimension d'origine de la

[20] Paradigme : : toutesLesPropriétés() : Set(Propriété)
post : result = self.tousLesParents()->iterate(p : Paradigme ; acc : Set{} | acc-> union(p.propriétés))

propriété (figure 2-5(a)). Les propriétés des notions sont appelées motifs, ce sont des propriétés habituelles de classe : attribut, méthode et invariant. Nous retrouvons là les concepts communs aux modèles à objets classiques[21].

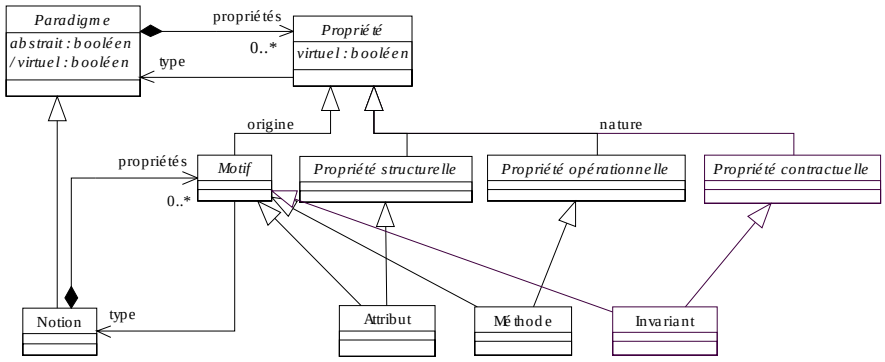
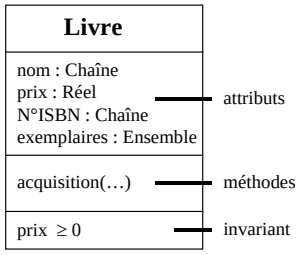


figure 2-5: (a) Notions & propriétés



(b) Instanciation d'une notion

Le concept Attribut (respectivement Méthode et Invariant) est une spécialisation des concepts Motif et Propriété structurelle (respectivement Propriété opérationnelle et Propriété contractuelle). Les attributs, méthodes et paramètres de méthodes sont de type Notion ; ces derniers sont omis sur la figure 2-5 pour des raisons de clarté.

2.4. Modèle comportemental

Nous utilisons le formalisme des machines à états finies pour spécifier la dynamique du système. Chaque domaine comportemental est un ensemble de comportements hiérarchiques dont nous nous proposons de définir l'architecture dans ce paragraphe. La sémantique des statecharts définie dans [Harel88] a été modifiée pour obtenir des spécifications comportementales plus génériques.

2.4.1. DOMAINES COMPORTEMENTAUX

On distingue deux types de domaines comportementaux (figure 2-6) :

DomaineSimple

[1] Un domaine simple est une hiérarchie de comportements simples.

self.comportements->forAll(c|c.oclIsTypeOf(ComportementSimple)) ;

ComportementSimple

[1] Un comportement est simple lorsqu'il est représenté à l'aide d'états non co-occurrents[22].

self.états->forAll(e1, e2|**not** e1.estCooccurrent[23](e2)) ;

[21] Seule l'introduction des domaines complète la sémantique habituellement utilisée.

[22] Phénomène est donc un comportement simple.

DomaineComplexe

[1] Un domaine complexe est une hiérarchie de comportements complexes.

self.comportements->forAll(c|c.oclIsTypeOf(ComportementComplexe)) ;

ComportementComplexe

[1] Un comportement est complexe lorsqu'il est représenté à l'aide d'au moins deux états co-occurrents.

self.états->exists(e1, e2|e1.estCooccurrent(e2)) ;

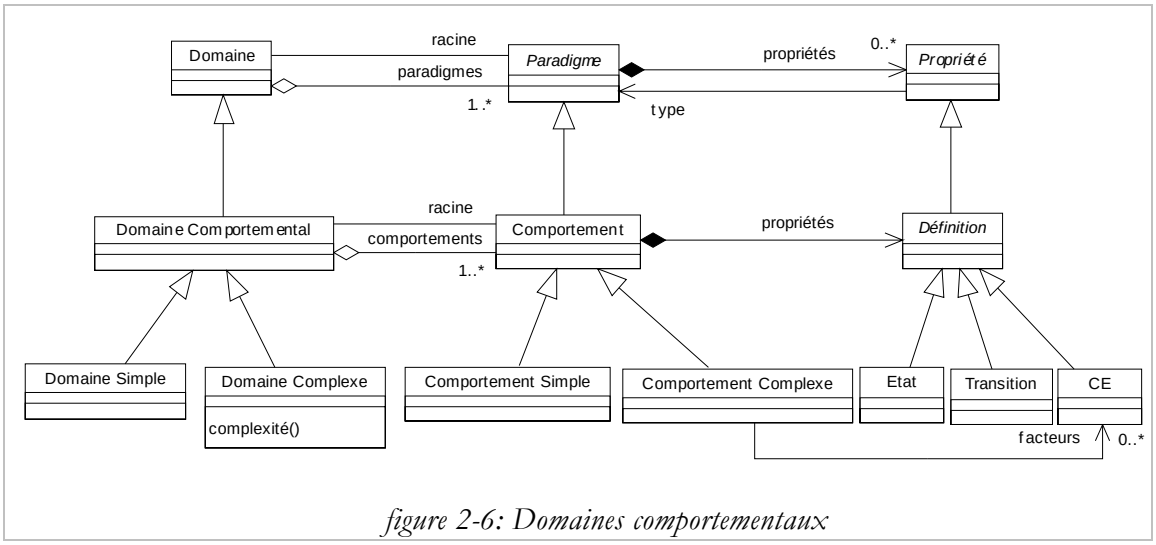


figure 2-6: Domaines comportementaux

Seuls les états de la racine d'un domaine sont déterminés par décomposition-ET (au sens de D. Harel (cf. § 2.4.2.2, A)) de l'unique état Valide de Phénomène en sous-états co-occurrents (figure 2-7). Nous verrons plus loin que nous autorisons au sein d'un même domaine uniquement la décomposition-OU des états.

Dans un comportement complexe, chaque région délimitée par un trait hachuré est un facteur[24] du comportement. Elle peut être nommée pour exprimer explicitement les synchronisations qu'un facteur entretient avec les autres facteurs. Sur la figure 2-7 par exemple, nous avons deux facteurs nommés facteur1 et facteur2. La complexité d'un domaine est égale au nombre de facteurs de la racine de ce domaine, elle représente le nombre d'états co-occurrents qu'un comportement de ce domaine peut avoir à un instant donné.

---

[23] Par construction, il est facile d'en déduire la propriété de co-occurrence entre deux états : deux états sont co-occurrents si ce sont des sous-états directs ou indirects respectivement de deux états distincts de la racine du domaine. On en déduit que deux états co-occurents ont un seul état parent commun, l'état Valide de Phénomène.
Etat : :estCooccurrent(e : Etat) :Booléen ;
post : result = self.tousLesParents()->intersection(e.tousLesParents())->size = 1 ;

[24] Le trait hachuré étant la représentation graphique du produit cartésien des états du comportement, le terme de facteur semblait tout indiqué.

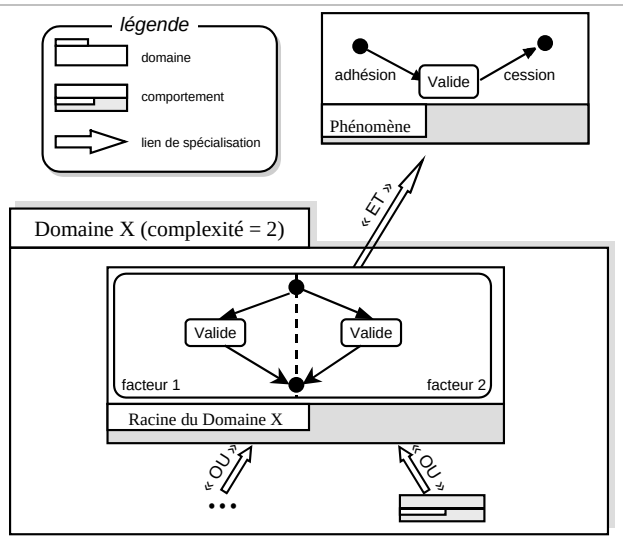


figure 2-7 : Génération de la racine d'un domaine comportemental à partir de Phénomène

Domaine Complexe : :complexité() : Entier ;

post : result = self.racine.facteurs->size ;

ComportementComplexe

[2] Un comportement complexe possède de 0 à n facteurs propres de type ComportementSimple. Les facteurs sont des Caractéristiques d'Evolution particulières du comportement complexe qui permettent de décrire son évolution globale.

self.facteurs->forAll(f|f.type.oclIsKindOf(ComportementSimple) ;

2.4.2. COMPORTEMENTS

2.4.2.1. Concepts

Pour décrire le formalisme des statecharts tel qu'il se présente dans le modèle NCR, nous avons choisi un comportement complexe appelé Ressource à Maintenir (figure 2-8). En effet, plutôt que de cataloguer tous les concepts tel que cela a été réalisé dans le chapitre 1 § 1.1, nous présentons ici les singularités de la notation à l'aide d'un exemple. Les choix réalisés peuvent surprendre un utilisateur averti des statecharts. Notre approche a l'originalité de s'intéresser en premier lieu au contrôle des flux de données et non pas au contrôle des séquences d'événements, ce pourquoi les statecharts sont souvent utilisés.

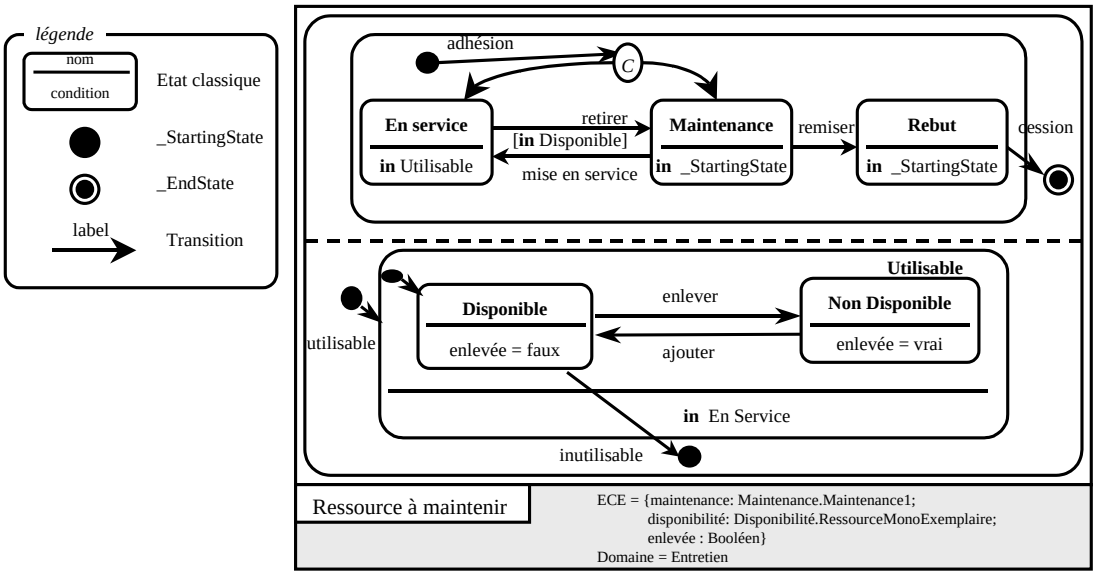


figure 2-8 : Un comportement complexe : Ressource à Maintenir

La ressource que nous avons vue comme un objet consommable pour le domaine de la Disponibilité(figure 2-3) suit également un cycle de vie du point de vue de la maintenance (figure 2-8). Bien sûr, ce cycle de vie influe sur la disponibilité de la ressource. Régulièrement la ressource En Service est retirée pour être examinée En Maintenance. A partir de là, elle peut être remise En service ou jetée au Rebut.

Nous avons là, deux types d'évolutions :

- Du point de vue de la maintenance, la ressource a un comportement une fois auquel elle adhère en entrant dans un état _StartingState et qu'elle quitte définitivement en entrant dans un état _EndState.

- Du point de vue de la disponibilité, la ressource a un comportement plusieurs fois. L'adhésion se fait toujours par un état _StartingState dans lequel elle peut retourner en franchissant la transition (Disponible, *inutilisable*, _StartingState) et y rester.

Nous verrons qu'il s'agit d'une modification importante de la sémantique habituelle des statecharts (cf. § 2.4.2.1, C).

Les propriétés des comportements sont appelées définitions. Il s'agit des propriétés usuelles des statecharts (états et transitions) auxquelles nous avons rajouté le concept de Caractéristique d'Evolution (CE) [Sˡ-Marcel98]. Les CE et les transitions sont des propriétés virtuelles qui seront réalisées dans les rôles (cf. § 2.5.2.1).

A- {CE}

Une caractéristique d'évolution est utilisée pour spécifier l'évolution d'un comportement. Elle est de type Paradigme, ce qui permet l'expression de conditions sur tous les éléments du modèle quelle que soit leur dimension d'origine. Ainsi, le comportement Ressource à maintenir (figure *2-8*) est spécifié à l'aide de trois caractéristiques d'évolution, deux de type Maintenance1 et RessourceMonoExemplaire issues respectivement des domaines comportementaux de la Maintenance et de la Disponibilité, et une de type booléen issue de la dimension structurelle. Ces CE ne peuvent être classifiées selon leur nature car il s'agit de « définitions pures ». Seule leur réalisation dans un rôle est classifiable en fonction de leur nature (cf. § 2.5.2.1). Par définition le fait qu'on ne connaisse pas la nature des CE est une condition suffisante pour que les occurrences de CE soient toujours virtuelles.

CE

[1] Une caractéristique d'Evolution est une propriété virtuelle

self.virtuel ;

L'ensemble des caractéristiques d'évolution nécessaires pour spécifier un comportement est appelé ECE.

Comportement : : ECE() : Set(CE);

post : result = self.propriétés.select(p|p.isKindOf(CE))

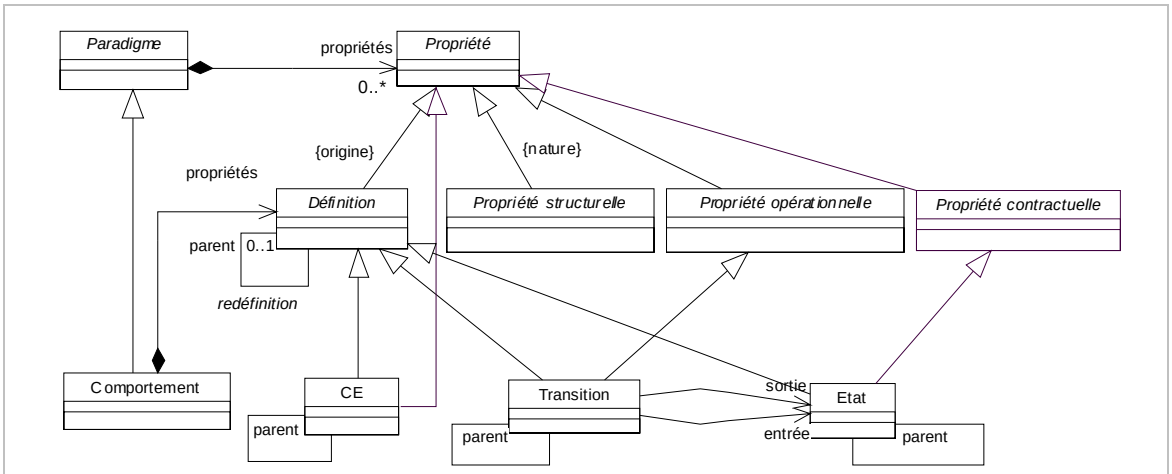
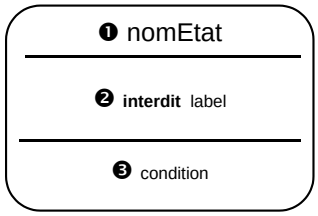


figure 2-9 : Comportement & propriétés

B- {Etat}

Définition

Un état est une abstraction des valeurs de l'ECE, une condition durant la vie d'un phénobjet.

Un état a un nom non optionnel$^{\supset}$, un ensemble d'interdictions$^{\not\subset}$ et une condition$^{\subset}$ (ramenée à vraie par défaut).	 *figure 2-10 : Etat NCR*

Interdiction$^{\not\subset}$

La sémantique des états est moins riche ici que dans la norme (cf. chapitre 1 § 1.1.1) mais elle introduit plus de flexibilité : plutôt que de spécifier de manière exhaustive l'ensemble des événements autorisés dans un état (en entrée, en sortie et dans l'état), nous spécifions l'« interdiction de franchir » certaines transitions lorsqu'on se trouve dans l'état.

La figure 2-11 illustre le principe d'interdiction. Elle représente le comportement d'un distributeur (de café, de sucreries, etc). Une contrainte est exprimée sur ce type de machine par l'ajout de l'interdiction : il n'est pas possible d'ajouter des pièces dans l'état En Attente. Ceci oblige le consommateur à sélectionner son produit avant de régler le montant demandé. L'interdiction est aussi utilisée pour la synchronisation des comportements complexes (cf. {Co-occurrence} figure 2-13). Structurellement elle implique que :

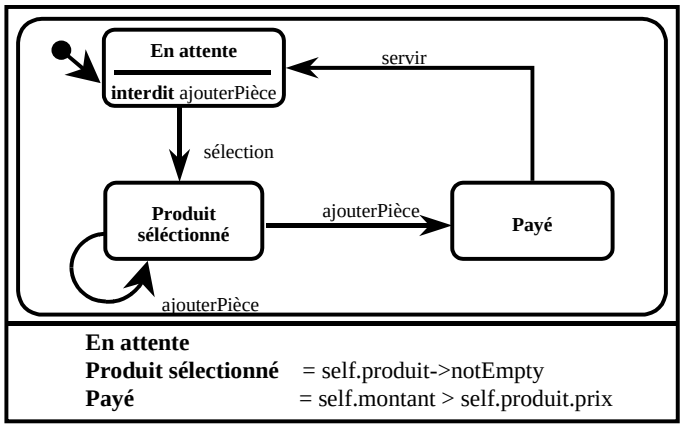


figure 2-11 : Etats et interdiction

Transition

[1] Une transition ne peut avoir pour origine un état dont les parents ou l'état lui-même interdit le label.

not self.entrée.interdiction(self.label) ;

Où la méthode interdiction renvoie un booléen qui prend pour valeur vraie lorsque le label passé en paramètre est interdit par l'état ou par un de ses parents :

Etat : :interdiction(label : String) : Boolean
Post : result = self.tousLesParents()->iterate(p ; acc : Boolean = false |
acc or p.interdictions->includes(label))

Condition$^{\subset}$

Un état est une sorte de propriété contractuelle (figure 2-9) : un phénobjet doit vérifier la condition spécifiée par l'état pour y rester. Cette condition est une expression logique sur les CE :

- Une Ressource à maintenir peut être dans l'état En Service si elle est en même temps dans l'état Disponible du point de vue de sa disponibilité.

Chaque état est une Propriété Contractuelle et hérite donc de la méthode « condition(o : Objet) : Booléen » qu'il redéfinit de telle manière que la condition de son état parent soit renforcée :

Etat : : condition(Phénobjet p) : Booléen ;
post : result **implies** self.parent.condition(p) ;

Cette assertion est réalisée dans le modèle NCR par défaut lorsqu'on génère un nouvel état. La condition de ce dernier est construite en réalisant un ET-logique entre le nouveau prédicat et le prédicat parent : une Ressource à maintenir (figure 2-8) peut être dans l'état Disponible si elle vérifie la condition « retirée = vrai » **et** qu'elle est en même temps dans l'état En Service du point de vue de sa maintenance, condition qu'elle hérite de l'état Utilisable.

Dans NCR, aucun contrôle est réalisé sur les problèmes de complétude et de recouvrement des conditions d'états. En effet, l'idée de former une partition des valeurs d'objets à l'aide des états est séduisante mais ne semble pas réaliste lorsqu'on manipule des objets complexes. Nous privilégions plutôt une conception incrémentale des états, chaque condition étant élaborée en fonction de la connaissance de l'objet. La définition des invariants d'état suit en cela l'approche opératoire (cf. chapitre 1 § 2.1.2) :

La condition est nécessaire mais pas suffisante pour être dans l'état !

Nous pouvons noter à ce stade que le concept d'activité (cf. chapitre 1 § 1.3.1) est absent du modèle comportemental NCR. En effet, l'association d'activités à des états telle qu'elle est réalisée dans l'UML semble contradictoire avec notre caractérisation des états : une activité est habituellement considérée comme une sorte d'opération qui modifie les données du système d'information. L'aspect transformationnel prévalant, les pré et post-conditions d'une activité sont, à notre avis, plus significatives que son invariant, rendant ainsi la conciliation état/activité peu pertinente dans le modèle NCR. Nous verrons plus loin que le concept d'activité est utilisé dans NCR avec une autre acception (cf. § 2.5.2.3).

C- {Etat d'entrée : _StartingState, Etat de sortie : _EndState}

L'objectif de notre approche n'est pas de spécifier des diagrammes d'états complets contrairement à l'ensemble des approches basées sur les cycles de vie mais seulement des états significatifs du cycle de vie global. La sémantique classique des statecharts est donc modifiée : on peut adhérer à un comportement (adhésion) et en sortir (cession) sans que cela corresponde respectivement à la vie et à la mort de l'objet. L'état d'entrée et de sortie ont une sémantique particulière. Il s'agit d'états à part entière (et non pas des pseudo-états comme décrits dans la norme[UML97]). Un phénobjet est dans l'état _StartingState à sa création. Il peut y revenir (on parle de comportement plusieurs fois) ou quitter définitivement le comportement correspondant (on parle alors de comportement une fois). Pour mieux comprendre, nous décrivons ci-dessous deux racines de domaines dont nous donnons le statechart équivalent avec la notation classique (figure 2-12).

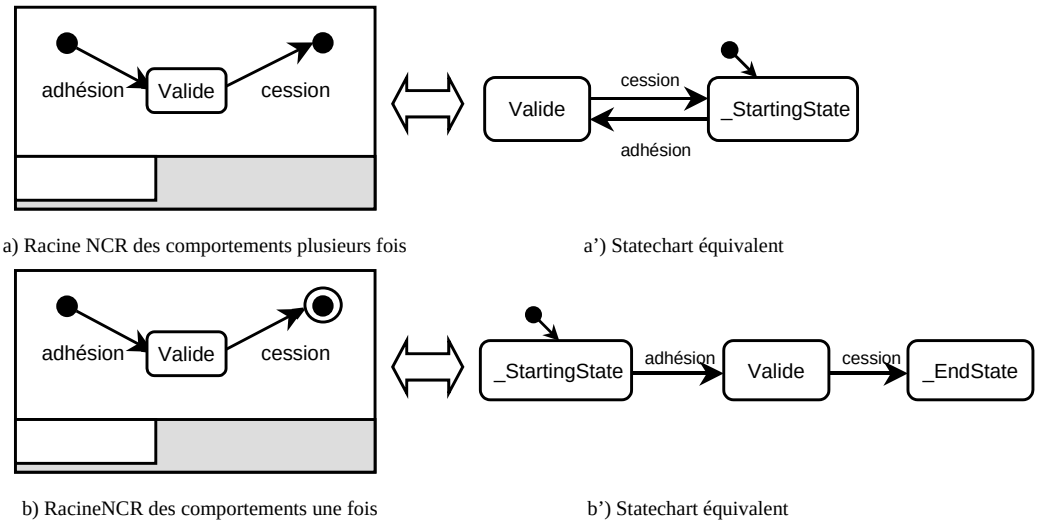


figure 2-12 : Sémantique de _StartingState et de _EndState

1- L'objet entre dans l'état _StartingState à sa création.

2- Il franchit la transition adhésion : le comportement est Valide.

3- L'objet franchit la transition cession et retourne soit dans l'état _StartingState (figure 2-12, comportement a), soit dans l'état _EndState (figure 2-12, comportement b). La mort d'un phénobjet n'est pas spécifiée explicitement dans un comportement. Nous verrons qu'elle ne peut survenir que lorsque le phénobjet est dans l'état _StartingState pour le premier cas ou _EndState pour le second.

Nous voyons sur le comportement de la figure 2-12 a) que l'état _StartingState est dupliqué (comme peut l'être l'état _EndState lorsqu'il existe plusieurs sorties). Il s'agit d'une simplification graphique car il existe un unique _StartingState par comportement hérité de Phénomène.

Nous résumons ci-dessous les différences importantes qui existent entre un statechart [UML97] et un comportement.

- La relation qui lie un statechart à un objet est bijective et est définie statiquement au moment de la conception du statechart et de l'objet. Le statechart décrit le cycle de vie complet de l'objet.
- Un comportement peut être spécifié indépendamment de tout objet. La liaison d'un objet à son comportement est réalisée de manière statique dans la dimension phénoménale. Par contre, L'adhésion et la cession à un comportement sont déterminées dynamiquement. **La création et la destruction de l'objet ne sont jamais explicitées dans un comportement** mais seulement dans la dimension phénoménale (cf. § 2.5.2.2).

D- {Co-occurrence}

Le nombre d'états co-occurrents[25] d'un comportement, i.e. sa complexité, est constant. Ceci est dû à la sémantique particulière des états _StartingState et _EndState. Sur la figure 2-8 par exemple, le fait de retirer un objet le rend inutilisable (transition (En Service, *retirer*, Maintenance)). En adoptant la sémantique classique on passerait de deux états co-occurrents à un unique état Maintenance. Or, le fait que l'état d'entrée (_StartingState) soit un état à part entière fait que le nombre d'états co-occurrents est constant dans le temps.

D'après la figure 2-8, il existe une seule situation dans laquelle on peut retirer un objet, celle-ci peut se spécifier ainsi :

{**in** En Service ∧ **in** Disponible} retirer {**in** Maintenance ∧ **in** _StartingState}

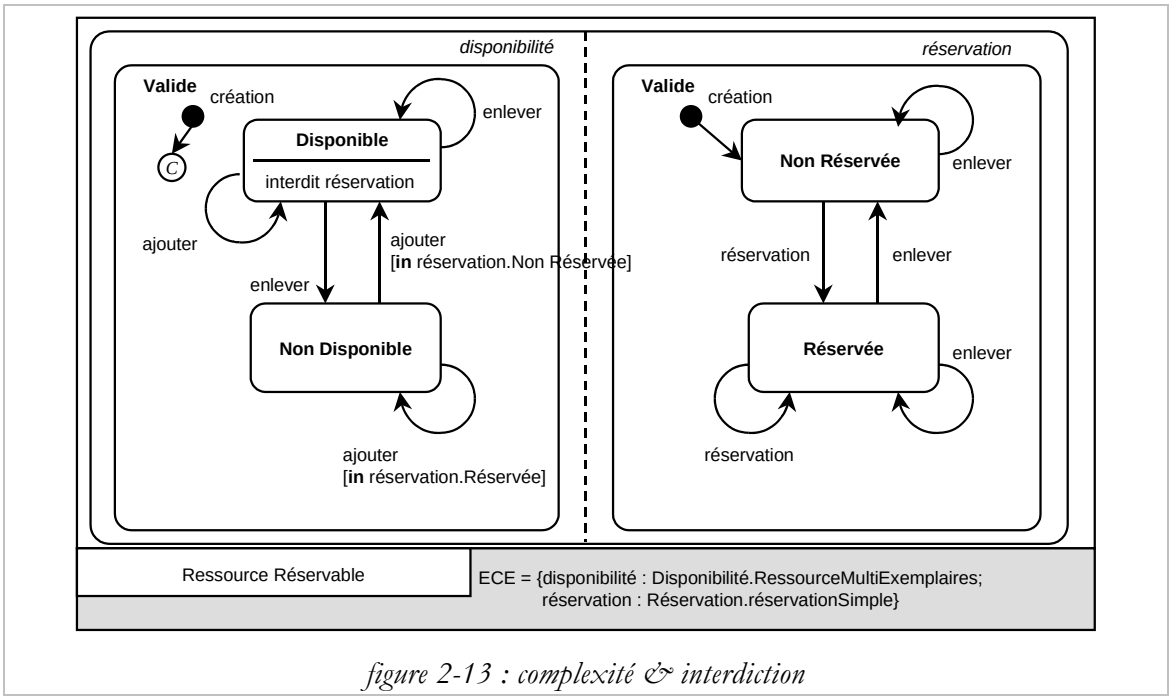


figure 2-13 : complexité & interdiction

Les différentes évolutions spécifiées dans un comportement complexe sont indépendantes. Par défaut, le franchissement d'une transition peut se faire alors que l'objet est dans n'importe quel état co-occurrent ; tout se passe comme si à chacun de ces états était associée une transition réflexive de même label. L'interdiction ajoutée sur la figure 2-13 permet d'aller à l'encontre de ce défaut et traduit le fait qu'un objet qui est Disponible n'a pas de raison d'être réservé : la transition réservation ne peut alors être franchie si l'objet est dans l'état Disponible.

E- {Transition}

Transition

[2] Une transition est une propriété virtuelle

```
self.virtuel ;
```

Comportement

[1] Une transition associe deux états (figure 2-9) de même comportement. Ces états ne sont pas co-occurrents.

```
self.transitions->forAll(t|self.etats.includes(t.entrée) and self.etats.includes(t.sortie) and
                                                    not t.entree.estCooccurrent(t.sortie))
```

La forme générale d'une transition est : « label[pré-condition]/[post-condition] » où pré et post-conditions sont des expressions sur :

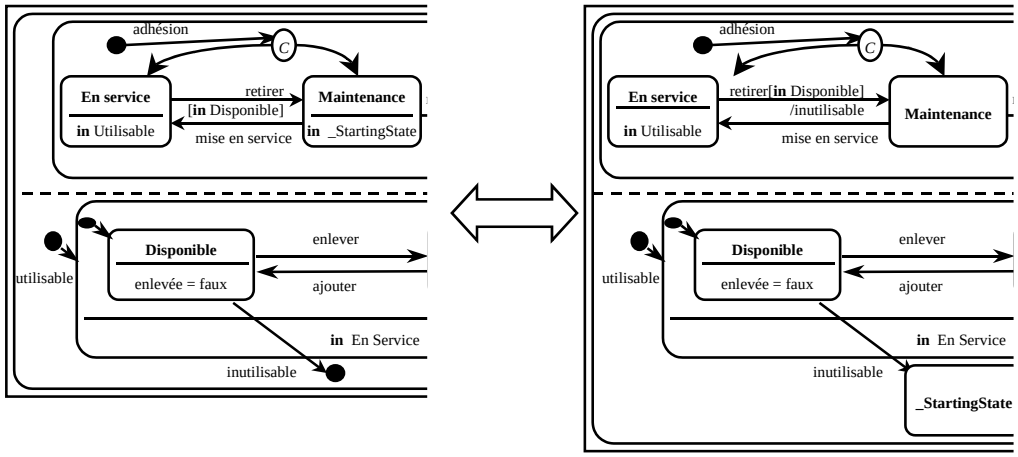
- les CE,
- les états. Dans le comportement Ressource Réservable par exemple, on ne peut ajouter dans l'état Non Disponible du facteur *disponibilité* que si l'objet est dans l'état Réservée pour le facteur *réservation*. On utilise dans ce cas le mot-clé **in** puis le nom de l'état pour marquer l'appartenance à l'état. Ce nom peut être préfixé par le nom du facteur en cas d'ambiguïté (exemple de la figure 2-13 : ajouter[**in** Réservation.Réservée]).

Le formalisme retenu pour décrire les transitions est essentiellement motivé par la volonté de rendre les statecharts plus génériques :

- La synchronisation explicite des statecharts à l'aide d'actions portées par les transitions (cf. chapitre 1 § 1.3.1) n'est pas intégrée dans le modèle NCR. Alors que les actions permettent la description précise des séquences d'événements autorisées, nous portons notre attention sur les propriétés vérifiées avant et après franchissement. Ces propriétés permettent d'exprimer

[25] Un état est co-occurent s'il peut être validé en même temps qu'un autre état.

implicitement (i.e. de manière déclarative) une synchronisation alors que les actions décrivent le séquencement de manière procédurale [UML97] (i.e. impérative). Dans NCR, le concept d'action n'apparaît que dans la dimension phénoménale lorsque le rôle doit réaliser le séquencement :



a) Synchronisation par les états en NCR

b) Synchronisation par les actions en UML

figure 2-14 : Deux synchronisations équivalentes

Sur l'exemple de la figure 2-14 qui représente partiellement le comportement de la figure *2-8*, il existe une seule situation dans laquelle on peut retirer un objet :

{**in** En Service ∧ **in** Disponible} retirer {**in** Maintenance ∧ **in** _StartingState}

Dans un Statechart classique, nous pouvons utiliser une action pour spécifier la même chose plus simplement. Cependant cette solution tout en étant plus précise offre moins de réutilisation :

{ **in** En Service} retirer [**in** Disponible]/inutilisable { **in** Maintenance}

Il est nécessaire de souligner ici que l'équivalence entre deux synchronisations, l'une par les états et l'autre par les actions, n'est pas systématique : des synchronisations exprimées habituellement à l'aide d'actions ne pourront pas être exprimées de manière déclarative.

- Une transition NCR est une propriété opérationnelle particulière qui a un ensemble de paramètres vide. En effet, les transitions ne sont pas paramétrées car les choix de transmission de l'information ne sont pas réalisés dans le modèle comportemental mais par les rôles. Pour la même raison, le concept d'événement est volontairement exclu du modèle comportemental abstrait. Nous rappelons qu'il s'agit ici de spécifier l'évolution générique de l'objet or le concept d'événement, usuellement défini comme un stimulus extérieur à l'objet, ne se prête pas à de telles descriptions.

Transition

[3] Une transition est une propriété opérationnelle non paramétrée

self.paramètres->isEmpty ;

- Il n'y a pas d'impératif d'instantanéité de la transition comme dans OMT ou dans UML, chaque transition est considérée comme un intervalle de temps discret entre deux états.

2.4.2.2. Relations intra-dimension de la dimension comportementale

Dans la dimension comportementale, nous distinguons deux types de relations répondant chacune à un besoin précis : la première est une relation intra-domaine basée sur l'abstraction d'états, la seconde est une relation inter-domaines qui offre la co-occurrence d'états. Nous pouvons d'ores et déjà remarquer que ces deux relations sont basées respectivement sur les deux fondements des statecharts que sont la profondeur et l'orthogonalité(cf. chapitre 1 § 1.2).

A- Spécialisation de comportements (décomposition)

Nous avons vu qu'il existait un mode unique de raffinement des comportements : ceux-ci sont obtenus par décomposition de comportements de même domaine (cf. Paradigme [1]). Les figures 2-15 et 2-16 présentent deux spécialisations issues du domaine de la Disponibilité : une RessourceLimitée est une spécialisation de RessourceMulitExemplaires qui elle-même est une spécialisation de RessourceMonoExemplaire. La décomposition permet le raffinement de tous les éléments comportementaux : les caractéristiques d'évolution, les états et les transitions.

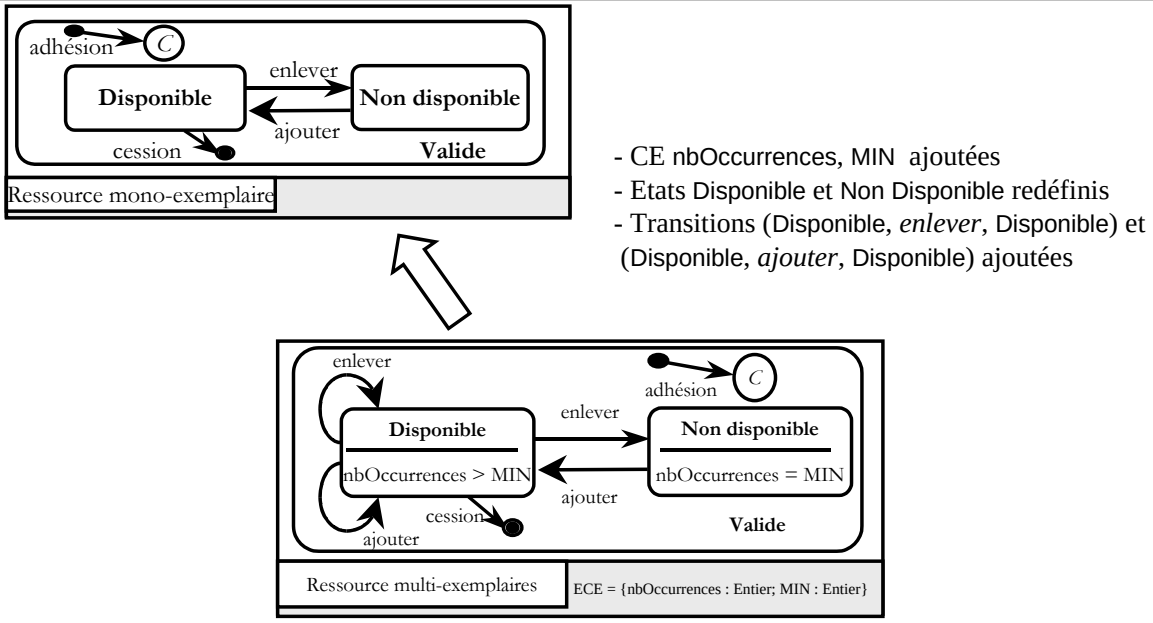


figure 2-15 : Décomposition de comportements(1)

Comportement

[2] La décomposition est un mécanisme de redéfinition des propriétés de comportements parents.

```
self.propriétés->forAll(p|p.parent->notEmpty implies self.toutesLesPropriétés()->includes(p.parent)) ;
```

Nous détaillons ci-après les opérations qui sont autorisées lors de la décomposition en reprenant chacune des propriétés des comportements :

{ECE}

Lors de la décomposition d'un comportement, l'ensemble de ses Caractéristiques d'Evolution est hérité en totalité. Chaque CE peut être renommée et redéfinie en suivant les règles de sous-typage. De nouvelles caractéristiques d'évolution peuvent être ajoutées dans l'ECE. Les CE MIN et nbOccurrences par exemple sont ajoutées sur la figure 2-15.

CE

[1] Le type d'une CE après redéfinition est un sous-type de sa CE parent

```
self.parent->notEmpty implies self.type.oclIsKindOf(self.parent.type)
```

{Etat}

Dans NCR, nous n'autorisons pas la surcharge d'état : les interdictions ainsi que la condition d'état sont héritées en totalité et ne peuvent être surchargées.

Trois opérations sont autorisées sur les états :

- L'ajout d'une clause au prédicat d'un état hérité. Cet état restreint les conditions d'appartenance à l'état. C'est ainsi que sur la figure 2-15 les deux conditions d'états par défaut à vrai sont renforcées afin de prendre en compte les valeurs de nbOccurrences.
- L'ajout d'une interdiction.
- La décomposition d'un état en sous-états. Cette opération est autorisée pour les états feuilles d'un comportement et on ne peut décomposer qu'un unique état à la fois lors d'une spécialisation. Ainsi, sur la figure 2-16, l'état Disponible de la RessourceMultiExemplaires est décomposé en deux sous-états Pleine et Provisionnée. Une spécialisation de RessourceLimitée ne pourrait pas proposer une autre décomposition de l'état Disponible.

Comportement

[3] Une seule décomposition d'états est autorisée lors de la décomposition d'un comportement. Cet état doit être un état qui n'a pas déjà été redéfini dans un des parents de ce comportement.

self.états->forAll(e1, e2|e1.parent->notEmpty and e2.parent->notEmpty implies (self.estUneFeuille(e1.parent) and e1.parent = e2.parent)) ;

Où

Comportement : : estUneFeuille(Etat e) : Booléen

post : result = self.tousLesEtats[26]->includes(e) **and** not self.tousLesEtats->exists(p|p.parent = e) ;

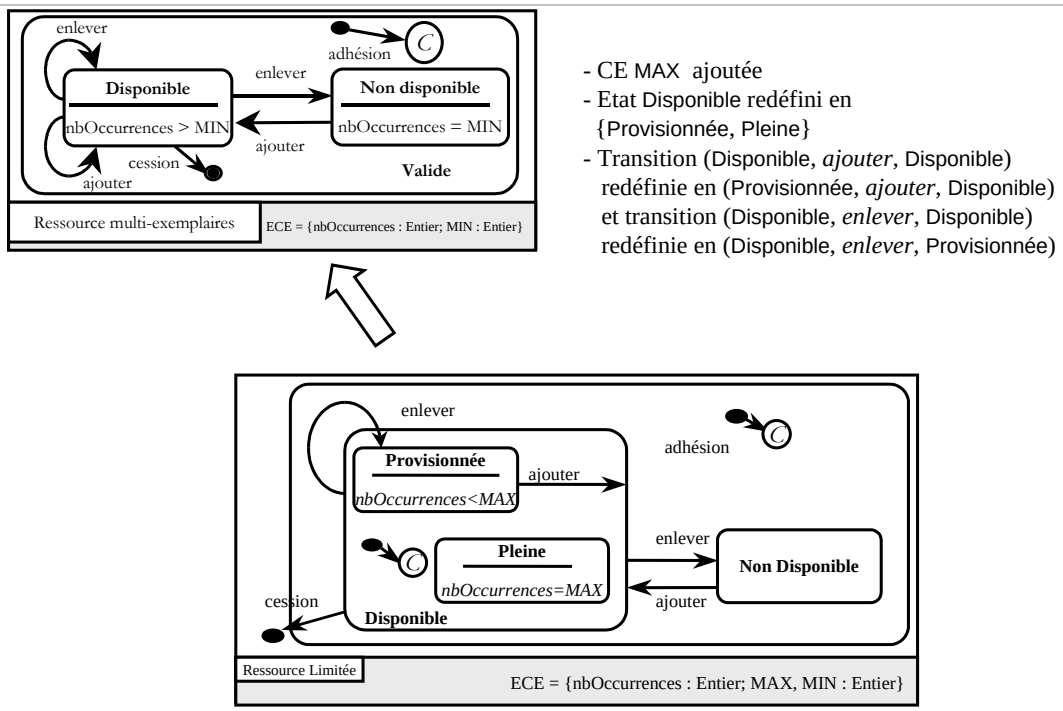


figure 2-16 : Décomposition de comportements(2)

{Transition}

De la même manière que pour les états, on peut redéfinir les transitions de deux manières :

- Par restriction de la pré-condition ou de la post-condition d'une transition héritée,
- En modifiant l'état source ou cible afin que le nouvel état soit un sous-état de l'état original (figure 2-16).

Transition

[4] Une transition peut redéfinir l'état d'entrée et de sortie d'une transition héritée par un sous-état respectivement de l'état d'entrée et de sortie de cette dernière.

self.parent->notEmpty implies (self.entrée.tousLesParents()->includes(self.parent.entrée) and self.sortie.tousLesParents()[27]->includes(self.parent.sortie)) ;

[26] Comportement : : tousLesEtats() : Set(Etat)
post : result = self.toutesLesPropriétés()->select(p|p.isKindOf(Etat)) ;

Contrairement à l'ajout d'états qui lui est prohibé, la définition de nouvelles transitions à tous les niveaux d'abstraction est autorisée et n'est pas limitée (sur la figure 2-15, deux transitions réflexives enlever et ajouter sont définies dans le nouveau comportement). Il est évident qu'elle doit être manipulée avec précaution pour ne pas pervertir les spécifications du comportement décomposé.

{Transition conditionnée}

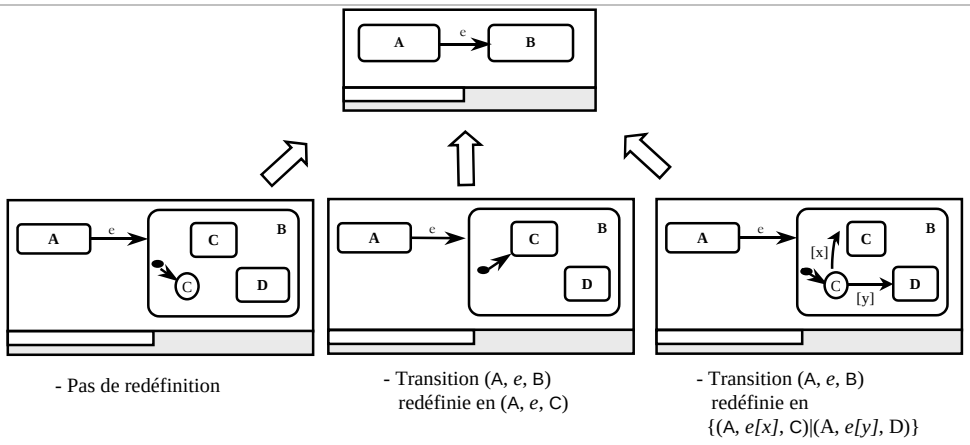


figure 2-17 : Décomposition de comportement & redéfinition de transitions

Nous avons vu que les transitions conditionnées ont une sémantique particulière dans notre modèle. L'exemple de la figure 2-16 montre que celle-ci simplifie la redéfinition des transitions conditionnées. En effet, pas une seule de ces transitions est redéfinie dans notre exemple puisque l'indéterminisme lié au branchement est levé par les post-conditions. Il est toujours possible de revenir à une écriture classique[UML97] en explicitant les conditions pour chaque branche, ce qui revient à redéfinir la transition en deux transitions avec des pré-conditions différentes (figure 2-17).

B - Fusion de comportements

La fusion permet la représentation d'évolutions concurrentes en faisant varier le nombre d'états co-occurents. En termes d'états, il s'agit d'une décomposition-ET [Duffy95] qui permet la représentation de cycles d'évolution parallèles à la manière de la méthode Merise [Panet94].

La fusion permet de réutiliser les domaines simples pour construire des domaines complexes. Nous avons vu sur la figure 2-7 comment créer ex-nihilo un comportement complexe en décomposant Phénomène. Nous voyons maintenant comment enrichir ce mécanisme par fusion de comportements existants. Le domaine Entretien (figure 2-18) est créé à partir de deux domaines simples que sont la Disponibilité et la Maintenance. L'état Valide de Phénomène est décomposé dans RessourceAMaintenir en deux sous-états co-occurrents qui « contiennent »

---

[27] Définition : :tousLesParents() : Set(Définition)
post : result = self.parent.tousLesParents()->including(self) ;

chacun les états et transitions issus de la fusion de Maintenance1 et de RessourceMonoExemplaire. Il est à noter qu'un comportement complexe est toujours obtenu par fusion de comportements simples. La figure 2-7 est donc une simplification ; la racine du domaine X est en fait créée par « décomposition-ET » de Phénomène et création de deux sous-états issus de la « fusion » répétée de Phénomène.

Comportement

[1] La fusion est toujours réalisée à partir de comportements simples. Nous rappelons que Phénomène est un comportement simple qui n'a pas de domaine d'appartenance.

```
self.facteurs->forAll(f|f.oclIsKindOf(ComportementSimple))
```

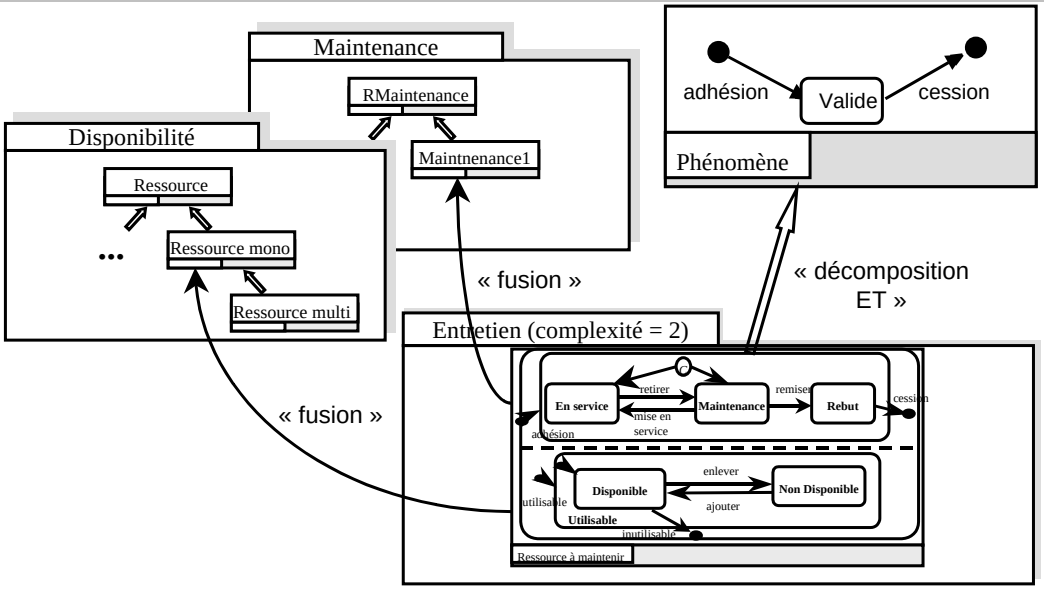


figure 2-18 : Génération d'un comportement complexe par fusion comportementale

La fusion est un mécanisme de redéfinition très proche de la décomposition. Une seule opération est interdite : il n'est pas possible de **décomposer un état hérité en sous-états** lors d'une fusion.

Dans le modèle NCR, il existe deux manières de spécialiser des comportements complexes :

- Soit en commençant par fusionner plusieurs comportements dans un comportement complexe qui peut être redéfini ensuite indépendamment en utilisant les opérateurs de spécialisation.
- Soit en commençant par fusionner plusieurs comportements simples dans un comportement complexe puis en réalisant des fusions successives dans les descendants du comportement complexe.

ComportementComplexe

[3] Quelque soit le facteur de type T d'un comportement complexe, il existe un facteur de type T' pour tout parent de ce comportement tel que T' est parent de T.

```
self.facteurs->forAll(f1|self.parents->forAll(p|p.facteurs->exists(f2| f1.type.tousLesParents()->includes(f2.type))
```

La fusion et la spécialisation sont pour l'instant traitées sans collisions. Nous nous assurons que lorsqu'un comportement complexe est spécialisé indépendamment, le comportement résultat n'a pas de facteurs et ne peut être redéfini à l'aide de la fusion.

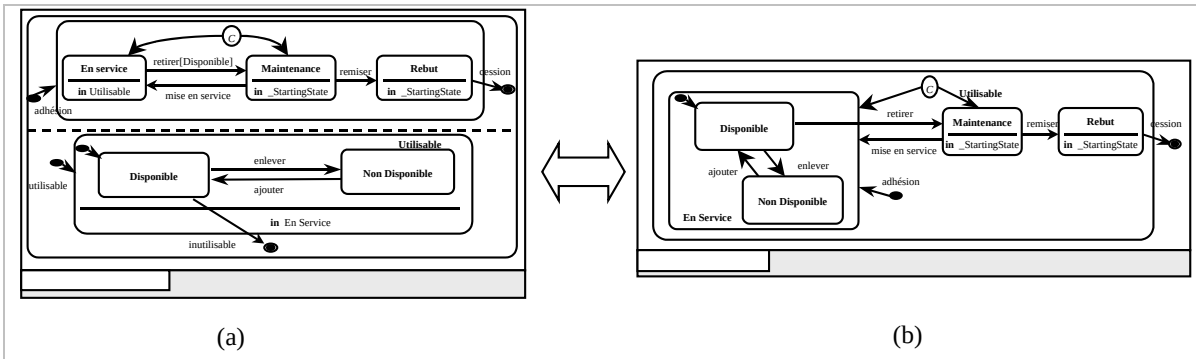
ComportementComplexe

[4] Un comportement complexe a 0 ou n facteurs où n est le nombre de facteurs de son comportement parent.

```
(self.facteurs->size > 0 implies self.parents->forAll(p|p.facteurs->size = self.facteurs->size)) or
self.facteurs->size = 0
```

Les restrictions imposées sur ces deux relations par rapport à un Statechart classique[28] sont justifiées par l'approche du problème prise ici : la profondeur et l'orthogonalité sont utilisées afin d'offrir le plus d'abstractions possibles d'un même comportement. En interdisant l'imbrication d'états co-occurrents, nous privilégions la simplicité des spécifications au détriment du pouvoir d'expression.

Un domaine représente une sorte de « point de vue » [St-Marcel96] comportemental (point de vue sur la disponibilité, la maintenance, etc.) qui peut évoluer librement. La co-occurrence d'états est seulement utilisée ici pour synchroniser ces différents points de vue. Elle offre de plus une vision parcellaire des comportements : tel employé ne s'intéresse qu'à la maintenance des livres, tel utilisateur est concerné uniquement par le prêt, etc. La figure 2-19(a) en donne une bonne illustration.



(a) (b)

figure 2-19 : Deux comportements équivalents

[28] Les statecharts autorisent toutes les imbrications d'états possibles. Un état peut être indifféremment décomposé en états-ET ou états-OU [Harel88].

Il est possible d'écrire le comportement de la figure 2-19(a) de manière équivalente en utilisant seulement une dimension d'évolution : le Statechart équivalent (figure 2-19(b)) est souvent plus compliqué comme l'illustrent Cook&Daniels ([Cook94], pp. 124). Ce n'est pourtant pas le cas ici car il est nécessaire d'exprimer des synchronisations supplémentaires sur les états dans le comportement complexe (a). Ce dernier a pourtant deux avantages : premièrement il réutilise deux comportements existants et surtout, il offre deux points de vue qu'il n'est pas aussi facile de dégager dans le comportement simple (b).

2.5. Modèle phénoménal

2.5.1. ROLES : APPROCHE GLOBALE

Le rôle est un connecteur logique chargé de décrire les interactions qui peuvent exister entre une notion et un comportement. Il hérite sa structure et son comportement respectivement de la notion animée et du comportement concrétisé. La représentation graphique d'un rôle est un raffinement de la représentation graphique héritée de son comportement (figure 2-20). Celle-ci est générée automatiquement à partir de la spécification textuelle des rôles qui exprime la fusion des motifs de la notion et des définitions du comportement. Cette fusion est présentée en détail par la suite. Nous voyons par exemple comment les descriptions textuelles du rôle OuvrageEmpruntable établissent les liens entre les motifs de la notion Ouvrage et les définitions du comportement RessourceLimitée.

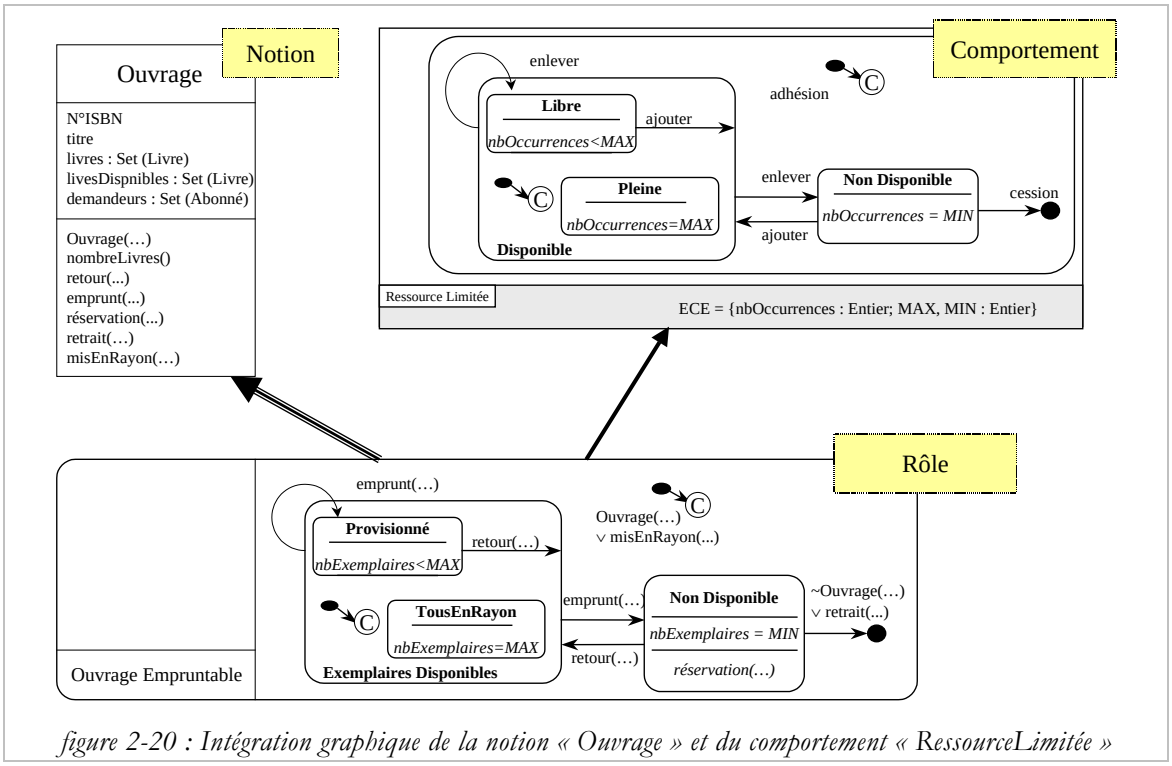


figure 2-20 : Intégration graphique de la notion « Ouvrage » et du comportement « RessourceLimitée »

2.5.2. PROPRIETES DE ROLE

Toute propriété de rôle est une sorte de connecteur. Un connecteur peut utiliser plusieurs motifs afin de réaliser une ou plusieurs définitions qui sont par nature virtuelles. La connexion telle qu'elle est spécifiée dans la figure 2-21 n'est pas réellement symétrique. Un connecteur réalise obligatoirement une définition, c'est sa raison d'être. Cependant il est possible qu'aucun motif ne puisse être utilisé directement pour réaliser la définition. Cette dernière est alors réalisée directement par le rôle. Nous distinguons par la suite ces deux cas, le premier que nous appelons fusion de propriétés structurelles et comportementales et le second concrétisation de propriétés comportementales.

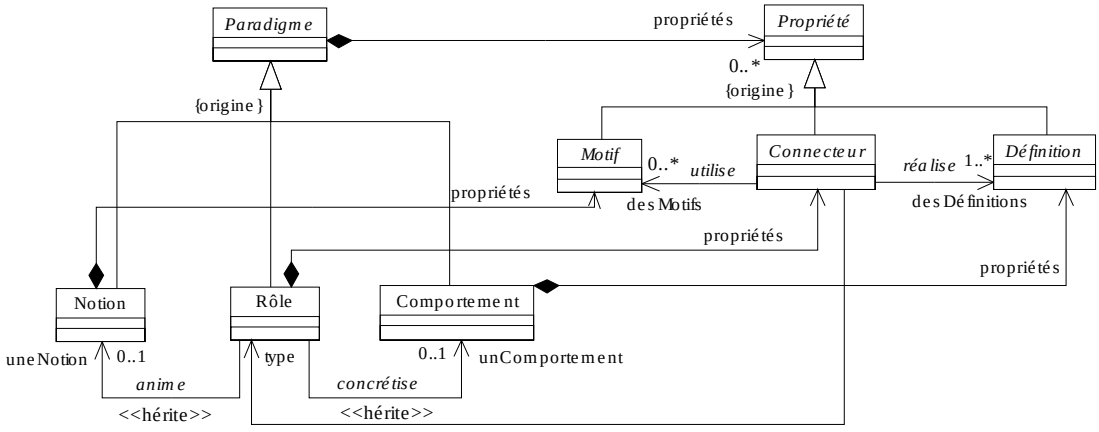


figure 2-21 : Classification selon l'origine du concept

On a :

Rôle

[1] Un rôle définit ses connecteurs en utilisant les motifs de la notion qu'il anime.

self.propriétés->forAll(p|self.uneNotion.toutesLesPropriétés()->intersection(p.desMotifs) = p.desMotifs) ;

et

Rôle

[1Bis] Un rôle définit ses connecteurs en réalisant une définition du comportement qu'il concrétise.

self.propriétés->forAll(p|self.unComportement.toutesLesPropriétés()->intersection(p.desDéfinitions) = p.desDéfinitions) ;

La connexion des propriétés de comportements est complète :

Rôle

[2] Un rôle réalise toutes les définitions du comportement qu'il concrétise.

self.unComportement.toutesLesPropriétés()->forAll(p1|self.toutesLesPropriétés()->exists(p2| p2.desDéfinitions->includes(p1))) ;

Un rôle a exactement quatre propriétés distinctes, les variables, les fonctions, les actions et activités. Elles décrivent chacune une partie de l'intégration notion/comportement. Dans les paragraphes suivants, nous découvrons leur véritable nature en même temps que le processus de conception qui nous a mené aux rôles de la figure 2-20.

2.5.2.1. Variable et fonction, réalisations des CE

La figure 2-22 représente la spécification textuelle de l'intégration des CE de notre exemple (figure 2-20). La variable MIN ainsi que les deux fonctions MAX et nbExemplaires obtenues définissent la relation qui existe entre la structure de l'ouvrage et le comportement de la ressource. Nous retrouvons là une idée commune avec la méthode Merise2 : les états sont définis dans le comportement en fonction des CE, elles-mêmes réalisées dans le rôle par les attributs et relations de la structure animée. Remarquons que la valeur de la variable MIN est fixée à 0. Celle-ci dépend en fait de la cardinalité min de la relation qui lie la classe Ouvrage à la classe Livre. Nous avons là un exemple intéressant d'expression de la cohérence structure/comportement. En supposant que nous ayons accès au méta-modèle NCR, il serait possible de spécifier en OCL la valeur de MIN et ainsi de s'assurer que tout changement de cardinalité dans le modèle structurel soit répercuté dans le modèle dynamique.

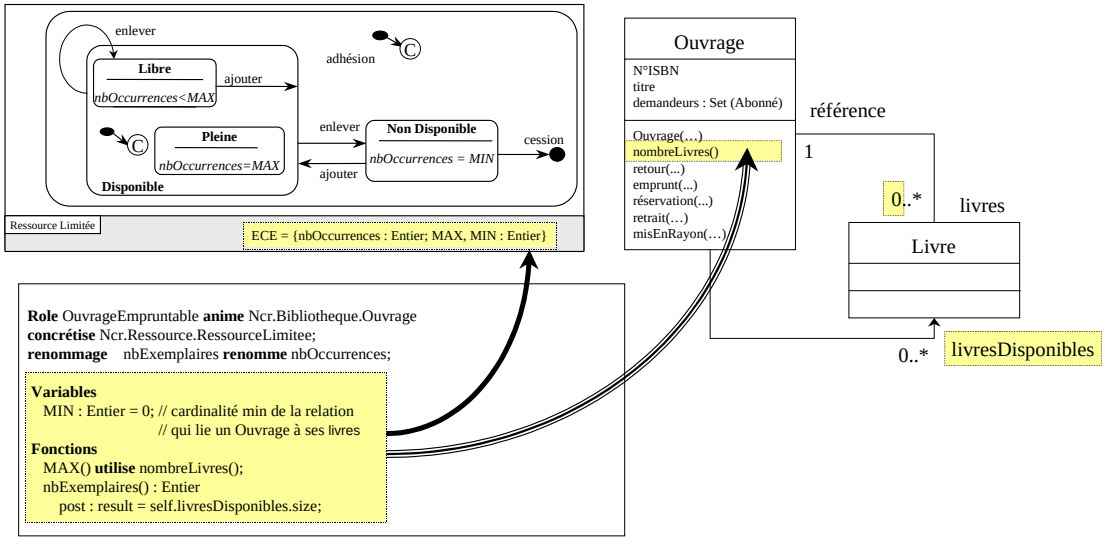


figure 2-22 : Spécification des variables et des fonctions

Variables et fonctions de rôle sont deux manières de concrétiser la caractéristique d'évolution. Seule leur nature diffère, la première étant de nature structurelle et la seconde de nature opérationnelle. On a donc :

- Une variable est une propriété structurelle réalisant une CE soit en utilisant directement un attribut de la notion, soit en définissant sa propre réalisation, étant elle-même une *propriété structurelle* qui peut stocker une information (figure 2-23). La variable MIN par

exemple est directement valuée à 0 sans qu'il y ait fusion avec un attribut de la notion Ouvrage.

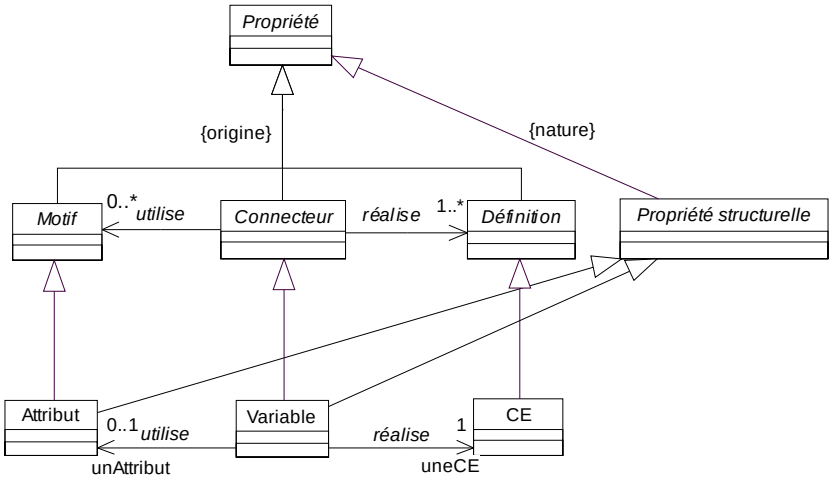


figure 2-23 : Variable de rôle

- Une fonction est une propriété opérationnelle (figure 2-23) réalisant une CE soit en utilisant directement une méthode de la notion, c'est le cas de la fonction MAX, soit en définissant sa propre réalisation, c'est le cas de la fonction nbExemplaires. Dans le second cas, la réalisation est spécifiée textuellement à partir de la structure de la notion (ou de la structure ajoutée dans le rôle) en utilisant le langage OCL (figure 2-22).

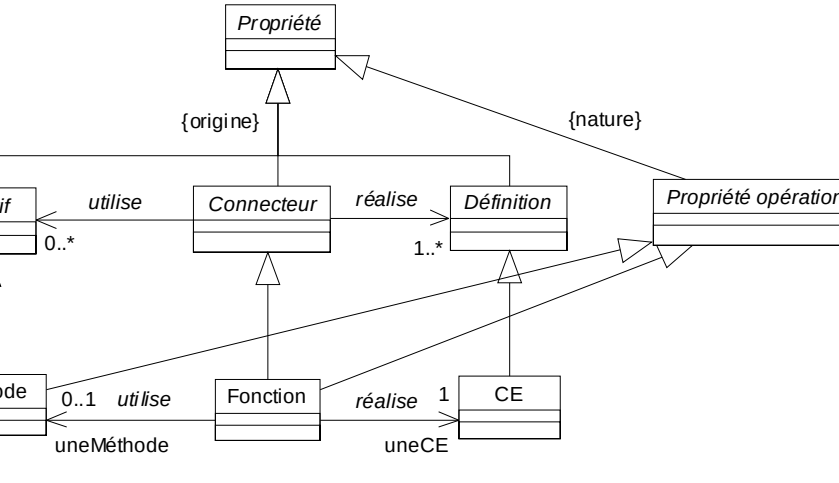


figure 2-24 : Fonction de rôle

{Concrétisation de CE}

La réalisation de caractéristiques d'évolution est triviale. Elle peut être réalisée sous la forme d'une variable, on parlera alors d'ajout comportemental, ou d'une fonction dont la spécification est donnée en OCL en fonction de la structure existante de la notion. Nous devons simplement vérifier que :

Variable

[1] Une variable est du type de la CE qu'elle réalise.

```
self.type = self.uneCE.type;
```

{Fusion d'attribut ou de méthode avec une CE}

La connexion des CE avec des attributs de notions pose un problème de typage. Nous avons identifié trois cas possibles :

- Le cas trivial où le type de la CE est le même que celui de l'attribut ou celui de la méthode. Le type de la variable (resp. fonction) est alors le même.
- Le cas où le type de l'attribut (resp. de la méthode) est un sous-type du type de la CE. L'intégration n'est pas possible dans ce cas là, le domaine de valeur des attributs (resp. des méthodes) étant plus restreint que celui de la CE. Cette interdiction garantie que tous les états du comportement sont potentiellement atteignables après intégration.
- Le cas où le type de la CE est un sous-type de celui de l'attribut (resp. de la méthode). Ce cas correspond à une volonté de restreindre l'évolution d'une notion à l'aide d'un comportement. Dans ce cas, la variable (resp. la fonction) est du type de la CE.

Variable

[2] Une variable qui utilise un attribut a pour type un sous-type de celui de l'attribut utilisé.

```
self.unAttribut->notEmpty implies self.type.oclIsKindOf(self.unAttribut);
```

2.5.2.2. Action, réalisation des transitions

Dans le modèle NCR, le concept d'action est utilisé pour réaliser les transitions du comportement à l'aide de méthodes de la notion. La figure ci-dessous illustre cette intégration pour les transitions du comportement RessourceLimitée et pour les méthodes de la notion Ouvrage. Pour intégrer l'ensemble des transitions de nom ajouter avec la méthode retour, nous utilisons le mot clé all. Lorsqu'il y a ambiguïté sur la transition ou sur la méthode à intégrer, il est possible d'utiliser la notation étendue :

```
Action retour réalise (Provisionnée, ajouter, Disponible), (Vide, ajouter, Disponible)

utilise retour(Livre unLivre) ;
```

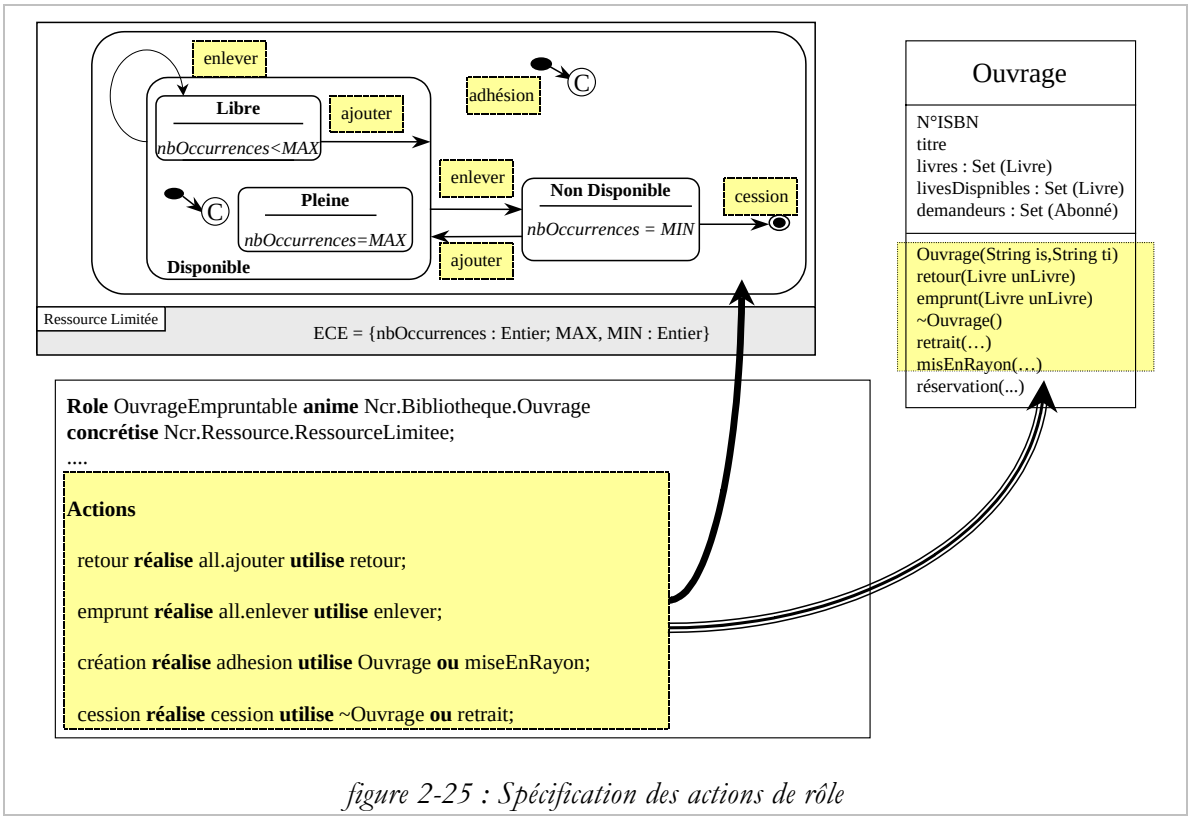


figure 2-25 : Spécification des actions de rôle

Du point de vue comportemental, tout se passe comme si une action réalisait une transition multiple avec plusieurs états d'entrée et autant d'états de sortie. L'ensemble des actions forme une partition de l'ensemble des transitions. Lorsqu'elle s'exécute, une action conduit d'un état à un autre état (non obligatoirement différent) en suivant les spécifications des transitions qu'elle réalise. Le franchissement d'une action correspond à la sélection d'une transition franchissable dans l'ensemble de ses transitions concrétisées.

Du point de vue structurel, une action utilise l'ensemble des méthodes nécessaires à la réalisation des transitions concrétisées L'exécution d'une action peut correspondre à une exécution en séquence des méthodes animées ; il s'agit du cas classique présenté dans la norme [UML97]. Il peut aussi s'agir d'une exécution sélective : une action utilise plusieurs méthodes mais une et une seule est exécutée lors de l'exécution d'une action. Ce type d'intégration est noté dans notre langage à l'aide du mot clé **ou**, nous parlons alors d'utilisation-OU des méthodes (sur la figure 2-25 le franchissement de la transition cession correspond à l'exécution de la méthode retrait ou à celle du destructeur de la notion Ouvrage). Nous parlons d'utilisation-ET des méthodes lorsque celles-ci sont séparées par des **et**.. Nous retrouvons cette distinction dans le métamodèle de la figure 2-26 avec la définition de deux types d'actions, les actions-ET et les actions-OU.

La figure ci-dessous présente le noyau NCR définissant les actions. Les états d'entrée d'une action sont calculés à partir des transitions réalisées :

Action : :entrée()

Post : result = self.desTransitions->collect(entrée)->asSet

De même pour l'état de sortie :

Action : :sortie()

Post : result = self.desTransitions->collect(sortie)->asSet

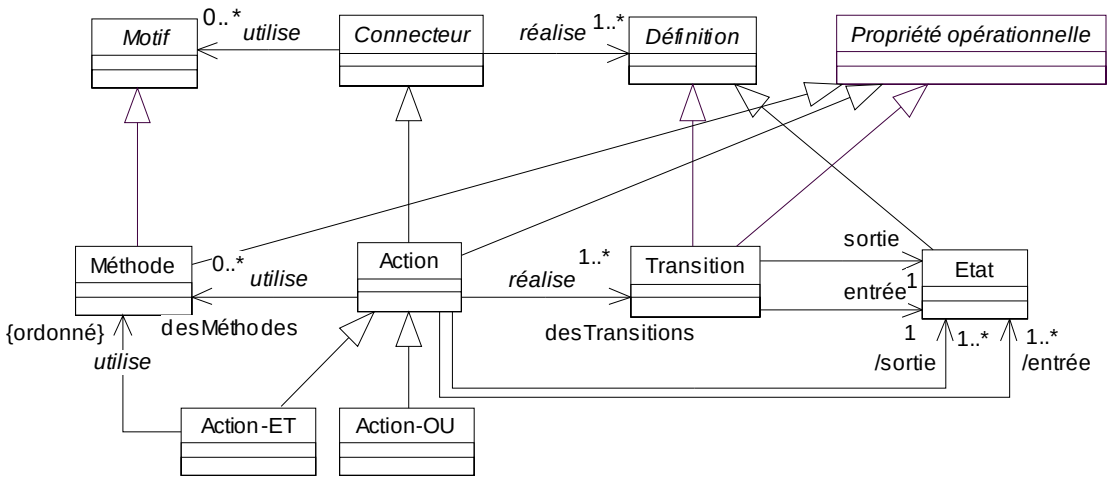


figure 2-26 : Action de rôle

Un action est une sorte de Propriété opérationnelle (figure 2-26) dont nous devons déterminer le type, les paramètres et les assertions. Ceux-ci dépendent uniquement de la relation d'utilisation.

{Fusion de méthodes avec des transitions}

Action-ET

[1] Le type de l'action est le même que celui de la première méthode utilisée :

self.utilise->notEmpty **implies** (self.type = self.utilise->first.type) ;

Les paramètres d'une action sont fixés lorsqu'il s'agit d'une utilisation-ET des méthodes.

Action-ET

[2] Le type de l'action est le même que celui de la première méthode utilisée :

non exprimable en OCL

Lors d'une utilisation-OU la méthode est séléctionnée à l'exécution en fonction des données passées en paramètres (cf. § 2.5.3.2).

{Vie et mort d'un phénobjet}

Nous avons vu que la vie et la mort de l'objet n'étaient pas spécifiés dans les comportements (cf. § 2.4.2.1, C). Ils le sont en fait dans les notions de par l'utilisation de constructeurs et de destructeurs (Les méthodes Ouvrage et ~Ouvrage sur notre exemple). Leur intégration avec des transitions se fait selon des règles précises :

- Un constructeur est intégré avec une transition dont l'état d'entrée est _StartingState. Par convention, une telle transition est appelée adhésion. Lorsque c'est le cas, la construction de l'objet (après création) correspond au franchissement de la transition adhésion. Sur notre exemple (figure 2-27), toute instance d'Ouvrage est dans l'état _StartingState à sa création et « tombe » immédiatement dans un des trois états significatifs après construction (i.e. exécution de la méthode Ouvrage()).
- Un destructeur est intégré avec une transition dont l'état d'arrivée est _StartingState ou _EndState. Par convention, une telle transition est appelée cession. Lorsque c'est le cas, la sortie du comportement correspond à la mort du phénobjet. Ainsi, le comportement de RessourceLimitée couvre complètement le cycle de vie d'un Ouvrage : la construction et la destruction des instances d'un Ouvrage correspondent respectivement à l'adhésion et à la cession du comportement RessourceLimitée (figure 2-27). Par contre, une instance d'ouvrage peut adhérer et céder au comportement en exécutant les méthodes respectivement misEnRayon() et retrait(). En effet, lorsqu'une instance est retirée du prêt, pour maintenance par exemple, ce comportement n'a plus de sens. La mise en rayon notifie le retour de l'instance dans le processus de prêt.

Ces règles d'intégration garantissent qu'à l'exécution d'un phénobjet, nous aurons :

Phénobjet

Un phénobjet ne peut mourir que s'il se trouve dans un état _StartingState ou _EndState.

non exprimable en OCL.

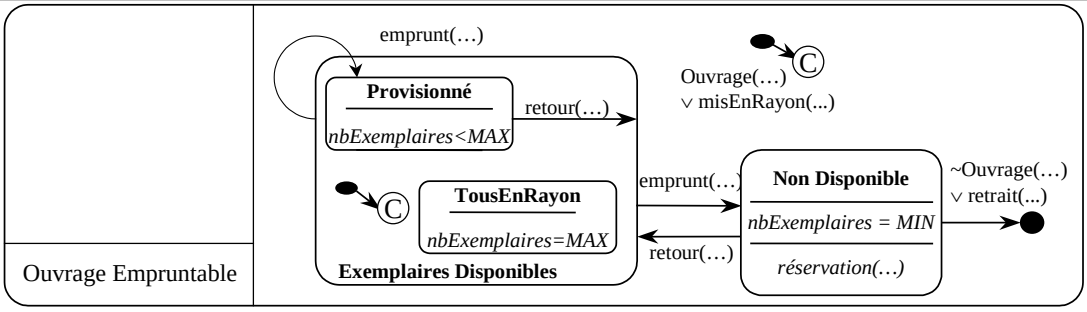


figure 2-27 : Rôle graphique

2.5.2.3. Activité, réalisation des états

Les descriptions comportementales modélisent l'évolution en utilisant uniquement des transitions. Il existe cependant des méthodes de notions qui ne font pas évoluer un phénobjet. Nous avons appelé ces méthodes des activités (en référence à la norme mais avec une acception légèrement différente). Le concept d'activité est le résultat de l'intégration d'une méthode et d'un ensemble d'états (figure 2-28).

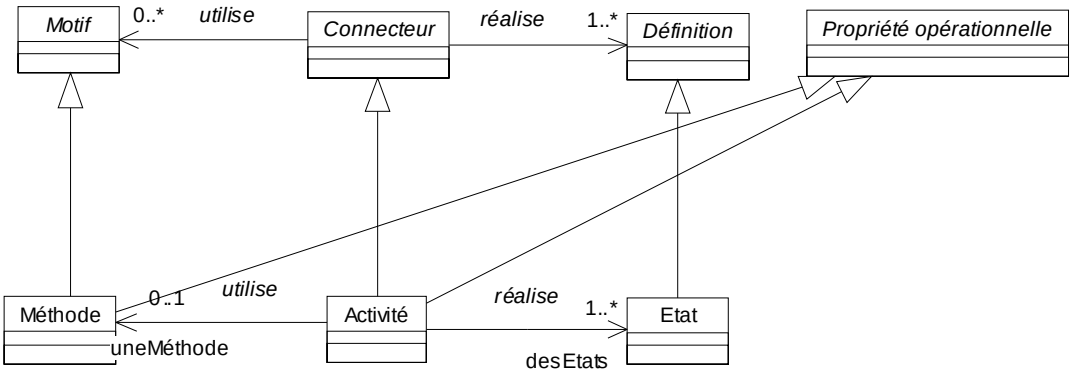


figure 2-28 : Activité de rôle

La figure ci-dessous donne un exemple de déclaration d'une activité de réservation. Celle-ci n'est pas prise en compte dans le comportement d'une RessourceLimitée. En l'intégrant de cette manière, nous reprenons implicitement la contrainte exprimée dans le § 2.4.2.1, D : un ouvrage ne peut être réservé que dans l'état AucunExemplaire.

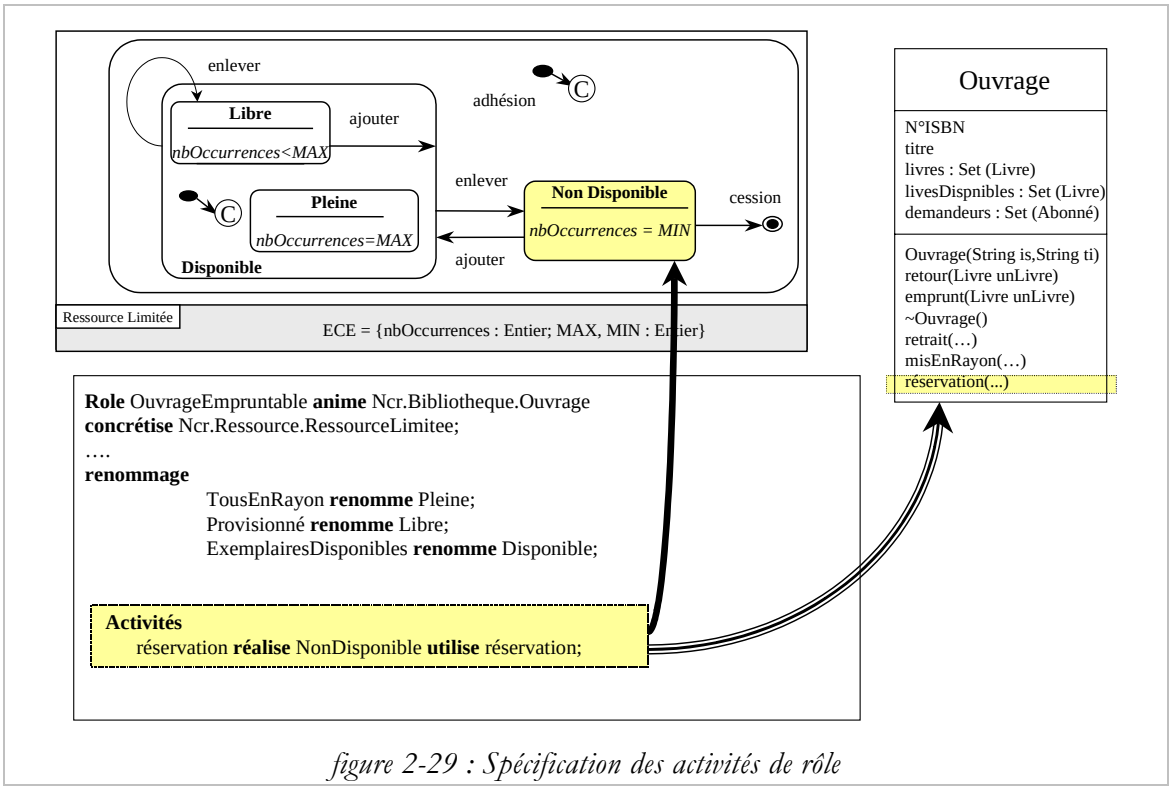


figure 2-29 : Spécification des activités de rôle

Une activité peut s'exécuter dans tous les états qu'elle réalise ainsi que dans leurs sous-états. Implicitement, une méthode dont l'intégration en tant qu'action ou activité n'est pas spécifiée est définie comme une activité exécutable dans l'état Valide de Phénomène lors de la génération d'un rôle. Par construction, cette méthode est donc exécutable dans tous les états du comportement.

Ce défaut permet de s'assurer que :

Rôle

[3] Un rôle utilise toutes les méthodes de la notion qu'il anime.

```
self.uneNotion.toutesLesPropriétés()->select(p|p.oclIsKindOf(Méthode))
->forAll(p1|self.toutesLesPropriétés()->select(p2|p2.oclIsKindOf(Action) or p2.oclIsKindOf(Activité))
->exists(p3|p3.desMotifs->includes(p1)) ;
```

A ce stade, les déclarations liées à l'intégration de la notion Ouvrage et du comportement RessourceLimitée sont terminées. Nous pouvons déduire automatiquement le rôle graphique de la figure 2-20 à partir des fusions et concrétisations déclarées.

De plus, il est possible de réaliser des contrôles sémantiques à partir des spécifications obtenues. Il y a deux sortes de contrôles, ceux qui sont statiques et portent sur la cohérence de l'intégration et ceux qui sont réalisés à l'exécution. Nous en donnons un aperçu dans les deux prochains paragraphes.

2.5.3. CONTROLE DE COHERENCE

2.5.3.1. Cohérence des spécifications

Notre objectif de départ n'était pas de fournir un outil qui vérifie statiquement la cohérence des spécifications de rôles. Ce travail a déjà été réalisé à l'aide de langages formels tels que Z ou B [Nguyen98] et nous pensons réutiliser ces résultats à moyen terme. Nous pouvons d'ores et déjà dire qu'il est possible de réaliser :

- Pour les variables et fonctions, un simple contrôle de type sur les CE réalisées et sur les méthodes ou les attributs utilisés.
- Pour les actions, un contrôle sur les conditions de franchissement des transitions réalisées et sur les assertions des méthodes.

  On doit s'assurer que toute transition est franchissable en vérifiant qu'il n'y a pas contradiction entre la pré-condition, l'invariant de l'état d'entrée des transitions et les pré-conditions des méthodes utilisées.

On doit s'assurer que tout état peut-être atteint en vérifiant qu'il n'y a pas contradiction entre la post-condition, l'invariant de l'état de sortie des transitions et les post-conditions des méthodes utilisées.

- Pour les activités, un contrôle sur les invariants des états réalisées et sur les assertions de la méthode utilisée. On doit s'assurer qu'il n'y a pas contradiction entre ces deux spécifications.

2.5.3.2. Animation des phénobjets

Certaines propriétés ne peuvent être vérifiées statiquement, notamment celles liées à l'implantation. L'animation est un moyen de vérifier dynamiquement les propriétés énoncées lors de l'intégration. Ces propriétés liées à l'exécution portent sur les instances de rôles dont phénobjet est la racine (figure 2-30).

La contrainte la plus élémentaire sur un phénobjet peut s'énoncer ainsi :

Phénobjet

[1] Un phénobjet est toujours dans un état dont il vérifie la condition (l'invariant).

```
self.étatCourant.condition(self) ;
```

Où

Phénobjet

[2] L'état courant d'un phénobjet est un des états du comportement concrétisé par le rôle qui a instancié le Phénobjet, cet état est une feuille du comportement.

```
self.type.unComportement.tousLesEtats()->includes(self.étatCourant) and
self.étatCourant.estUneFeuille(self.type.unComportement);
```

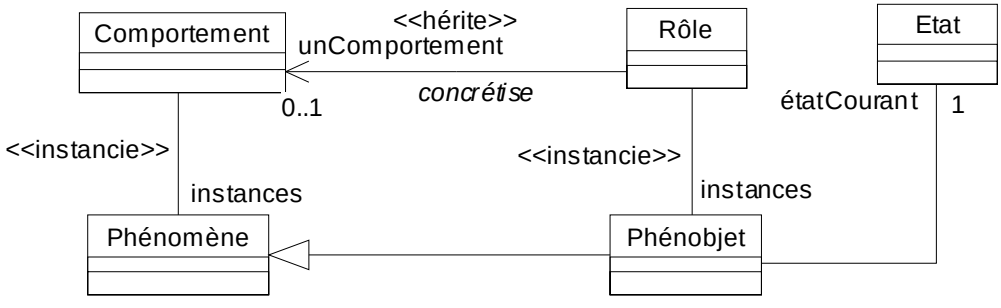


figure 2-30 : Phénobjet, racine des rôles

Les actions et les activités sont des propriétés opérationnelles dont l'exécution doit être contrôlée vis-à-vis des spécifications structurelles et comportementales. Nous énonçons ci-dessous ces contrôles.

A- {Action}

Lors de l'exécution d'une action, nous vérifions à la fois la cohérence structurelle et comportementale :

Phénobjet : :exécuter(Action a, Set(Phénobjet) params)
Pre : a.utilisable(self, params)->notEmpty **and** a.preLink(self)->notEmpty ;
// On vérifie que l'action est exécutable et que le phénobjet est bien dans un état pour lequel il existe des transitions réalisées franchissables.
Post : a.utilisée(a.utilisable@pre(), self) **and** a.postLink(a.preLink@pre(), self) ;
// On vérifie que la post-condition s'est correctement exécutée et qu'une des transitions franchissables a mené le phénobjet dans un état cohérent.

Où la méthode preLink retourne un ensemble de transitions franchissables. En effet, dans le modèle NCR, l'indéterminisme est levé par les invariants d'états (cf. chapitre 4 § 2.4). Il peut donc y avoir plusieurs transitions franchissables à partir d'un même état :

Action : :preLink(Phénobjet obj) :Set(Transition)
Post : result = a.desTransitions->select(t|self.étatCourant.tousLesParents->includes(t.entrée) **and** t.pré.condition(obj)) ;

La méthode postLink retourne vrai lorsqu'une des transitions franchissables conduit le phénobjet dans un état cohérent. On vérifie que la post-condition de la transition ainsi que l'invariant de l'état sont vrais.

Action : :postLink(Set(Transition) transitions, Phénobjet obj) :Boolean
Post : result = transitions->exists(t|(t.post->notEmpty implies t.post.condition(obj)) and t.sortie.invariant(obj)) ;

Les méthodes utilisable et utilisée sont le pendant structurel des méthodes respectivement preLink et postLink pour la classe Action. La méthode utilisable est redéfinie dans les deux sous-classes Action-ET et Action-OU pour prendre en compte le séquencement et la sélection.

Action-ET : :utilisable(Phénobjet obj, Set(Phénobjet) params) :Set(Méthode)
Pre : a.desMéthodes->first.pré->notEmpty **implies** a.desMéthodes->first.pré.condition(obj)) ;

// On vérifie que la précondition de la première méthode intégrée est vraie. Le test sur les paramètres est occulté.

Post : result = a.desMéthodes ;

// On retourne l'ensemble des méthodes utilisables.

Action-OU : :utilisable(Phénobjet obj, Set(Phénobjet) params) :Set(Méthode)

Pre : a.select(params)->notEmpty **implies** a.select(params).condition(obj) ;

// On vérifie que la précondition de la méthode séléctionnée à partir des paramètres (avec la méthode select non décrite ici) est vraie.

Post : result = a.select(params) ;

// On retourne la méthode utilisable sous la forme d'un singleton.

La méthode utilisée est héritée sans redéfinition par les classes Action-ET et Action-OU.

Action : :utilisée(Set(Méthode) méthodes, Phénobjet obj) :Boolean

Post : result = méthodes->last.post.condition(obj) ;

// On vérifie la post-condition de la dernière méthode.

B- {Activité}

Au moment de l'exécution d'une activité, nous devons nous assurer de deux choses :

- Elle peut s'exécuter dans l'état en cours.
- Elle n'a pas d'effet de bord sur le comportement, i.e. elle ne change pas l'état en cours de l'objet.

D'où

Phénobjet : :exécuter(Activité a)

Pre : a.uneMéthode.pré.condition(self)) **and** a.preLink(self)->notEmpty;

// On vérifie que la précondition de la méthode intégrée est vraie et que le phénobjet est bien dans un état où l'activité est autorisée

Post : a.uneMéthode.post.condition(self) **and** a.postLink(a.preLink@pre(), self) ;

// On vérifie la post-condition de la méthode et la condition de l'état, condition nécessaire et suffisante pour que le phénobjet reste dans l'état.

Où la méthode preLink retourne un ensemble d'états dans lequel l'activité peut s'exécuter, cet ensemble est un singleton qui contient l'état parent de l'état courant du phénobjet :

Activité : :preLink(Phénobjet obj) : Set(Etat)

Post : result = a.desEtats->select(e|obj.étatCourant.tousLesParents->includes(e)) ;

La méthode postLink retourne vrai lorsqu'il existe un état dont le phénobjet vérifie l'invariant :

Activité : :postLink(Set(Etat) états, Phénobjet obj) :Boolean

Post : result = états->exists(e|e.condition(obj)) ;

2.5.4. ROLES COMPLEXES

Le modèle de rôles est isomorphe avec le modèle de comportements. Nous avons donc des rôles simples et des rôles complexes qui sont construits à partir de ces rôles simples. Phénobjet est à ce titre une instance particulière d'un rôle simple comme Phénomène est une instance particulière d'un comportement simple (figure 2-31). Les règles sur les instances de rôles énoncées dans les paragraphes précédents sont valables pour les instances de rôles simples et complexes. Nous exprimons dans ce paragraphe uniquement les règles élémentaires de construction et d'animation des rôles complexes, notamment vis-à-vis de la fusion de comportements.

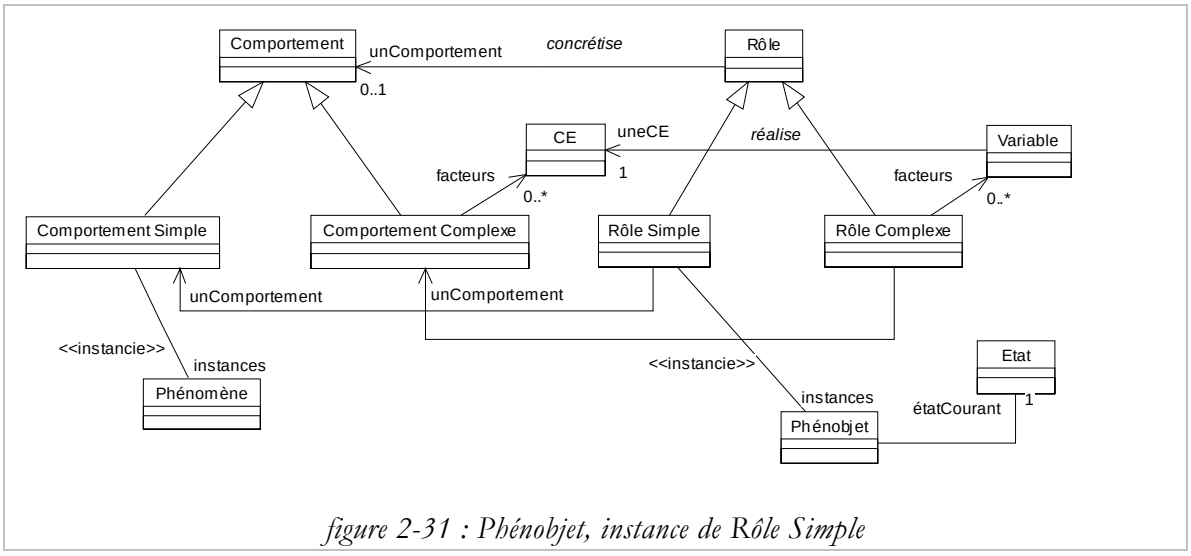


figure 2-31 : Phénobjet, instance de Rôle Simple

2.5.4.1. Construction

Nous donnons ci-dessous l'intégration graphique et textuelle complète du comportement complexe introduit dans le paragraphe 2.4.2. Celui-ci est intégré avec la notion Livre de notre bibliothèque (figure 2-32).

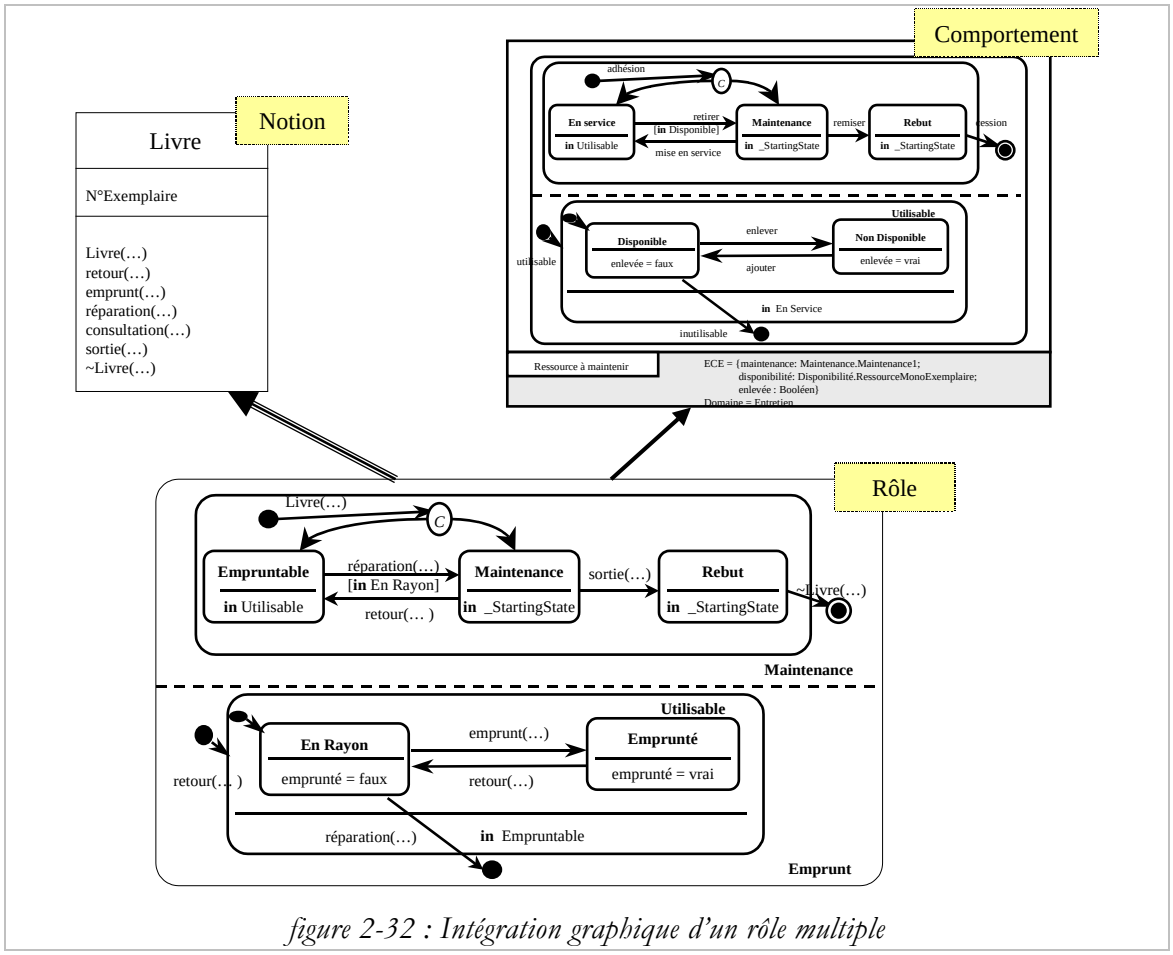


figure 2-32 : Intégration graphique d'un rôle multiple

La spécification textuelle de ce rôle est donnée ci-dessous. Nous voyons que chaque facteur est spécifié séparément à l'aide des concepts déjà définis pour un comportement simple que sont les variables, fonctions, actions et activités.

Role Livre **anime** Ncr.Bibliotheque.Livre
concrétise Ncr.Ressource.RessourceAMaintenir;

renommage emprunté **renomme** enlevée;

Variables // Définition des variables globales du comportement complexe

Maintenance **réalise** maintenance ; // Premier facteur

Emprunt **réalise** disponibilité ; // Deuxième facteur

Fonctions // Définition des fonctions globales du comportement complexe

enlevée() : Booléen
post : result = self.emprunteur->notEmpty;

FACTEUR Maintenance **concrétise** Maintenance.Maintenance1 // Intégration du premier facteur

renommage Empruntable **renomme** EnService;

Actions

création **réalise** adhésion **utilise** Livre;

réparation **réalise** retirer **utilise** réparation;

miseEnService **réalise** miseEnService **utilise** retour;

remiser **réalise** remiser **utilise** sortie;

cession **réalise** cession **utilise** ~Livre;

FACTEUR Emprunt **concrétise** Disponibilité.RessourceMonoExemplaire // Intégration du deuxième facteur

renommage EnRayon **renomme** Disponible;
Emprunté **renomme** NonDisponible;

Actions

retour **réalise** utilisable, ajouter **utilise** retour;

emprunt **réalise** enlever **utilise** emprunt;

réparation **réalise** inutilisable **utilise** réparation;

Activités

consultation **réalise** EnRayon **utilise** consultation;

Cette spécification fait apparaître une propriété intéressante des comportements et des rôles complexes : les facteurs d'un comportement complexe font partie de l'ensemble des caractéristiques d'évolution de ce comportement. En effet, chaque facteur est une propriété nécessaire pour en décrire l'évolution globale :

ComportementComplexe

[5] Les facteurs d'un comportement complexe font partie de l'ensemble des caractéristiques d'évolution de ce comportement.

```
self.facteurs->forAll(f|self.ECE()->includes(f))
```

Les facteurs d'un comportement complexe sont concrétisés dans le rôle complexe et affectés aux variables qui réalisent les caractéristiques d'évolution de l'ECE :

RôleComplexe

[1] Un rôle complexe concrétise les facteurs de son comportement complexe

```
self.unComportement.facteurs->forAll(f1|self.facteurs->exists(f2|f2.uneCE = f1))
```

2.5.4.2. Animation

Pour chaque facteur d'un rôle complexe sont définies indépendamment les actions et activités exécutables. L'exécution d'un rôle complexe est donc déléguée à ses facteurs en suivant les règles définies dans le paragraphe 2.5.3.2.

RôleComplexe

[2] Une exécution dans un rôle complexe correspond à l'exécution d'une action ou d'une activité pour chacun des facteurs de ce rôle.

2.6. Conclusion

Nous avons présenté dans ce chapitre la spécification qui est à l'origine du noyau NCR actuel. Cette spécification est le résultat de l'étude que nous avons menée tout au long de ce mémoire sur les différentes approches du comportement d'objet. A partir des forces et des faiblesses de chaque approche, nous avons pu définir notre propre démarche pour intégrer la structure et le comportement des objets dans le processus de développement des logiciels. Celle-ci a fortement conditionné la sémantique retenue dans chaque dimension. Pour favoriser la réutilisation et la traçabilité des spécifications, nous sommes partis d'un modèle de notions et de comportements ramené à sa plus simple expression. Les notions et les comportements que nous avons décrits sont loin d'atteindre la richesse d'expression et la complexité présentes respectivement dans les classes et les statecharts de la norme [UML97]. Loin de cet objectif, nous avons privilégié les caractéristiques d'abstraction et de déclarativité (cf. chapitre 4 § 2.5) qui sont depuis toujours à la base de la réutilisation dans les modèles à objets.

CHAPITRE 4

Au delà du modèle…

« Ton projet Christophe, c'est une grosse patate qui merge … »

Dominique Rieu

[St-Marcel96]

« Mon projet, c'est le germe fragile d'une rose … »

Christophe Saint-Marcel

[St-Marcel2000]

Les bases du modèle NCR sont aujourd'hui jetées. Cependant, le cœur de notre travail ne réside pas uniquement dans la spécification du modèle. Pour arriver à cette solution, nous avons dû définir les caractéristiques que nous voulions retrouver en termes de modèle et de démarche. Ce chapitre présente les aspects finalisés du modèle en même temps qu'il ouvre des perspectives intéressantes sur l'avenir de NCR.

Une perspective majeure concerne la possibilité d'apprentissage offerte par la simulation des comportements d'objets. Nous proposons deux modes d'exécution dits « fort » et « faible ». Le premier décrit de manière complète et déterministe les comportements. Il permet une modélisation de type classique dans laquelle le comportement (statechart) agit comme un contrôleur et signale les incohérences. Le second mode d'exécution lève certains « verrous » tout en conservant les contraintes que nous jugeons minimales. Nous avons alors des évolutions incomplètes et non obligatoirement déterministes à partir desquelles nous pouvons intuiter dynamiquement des évolutions possibles. Ce mode d'exécution doit favoriser la compréhension des objets complexes du système dont on ne peut à priori appréhender les évolutions dans leur globalité.

Nous présentons dans la deuxième partie les fragments du processus NCR, un processus pour et par la réutilisation de composants structurels et comportementaux. Ce processus intègre complètement les spécifications comportementales et en assure la traçabilité.

1. NCR, CONCEPTS AVANCES

Dans le chapitre 3, nous nous sommes contentés de présenter le modèle à l'aide du langage de modélisation unifié tout en précisant les contraintes à respecter en OCL. Dans cette partie, nous revenons en détail sur des particularités du modèle. Nous fournissons ici un complément d'information sur certaines propriétés comme la délégation de comportements (§ 1.1), l'exécution « faible » et « forte » (§ 1.2), l'interdiction (§ 1.3) et le polymorphisme (§ 1.4). Plus qu'une formalisation poussée du modèle, cette partie présente et situe les fondements qui « gouvernent » aujourd'hui le modèle NCR.

1.1. Relations comportementales

L'ajout d'états co-occurrents est parfois utilisé pour la modélisation de l'évolution d'objets composites [Rumbaugh95]. Dans notre modèle, la composition et de manière plus générale toute relation de type délégation est exprimée à l'aide des caractéristiques d'évolution. La nature transversale des caractéristiques d'évolution permet l'expression de synchronisations réutilisables et qui dépassent l'évolution d'un simple objet. La figure 1-1 présente un exemple de délégation entre comportements.

Nous définissons deux comportements utilisés pour l'assemblage d'objets composites. Le comportement Commande est utilisé pour prendre en compte les livraisons. Une livraison fait passer un objet de l'état En Commande à l'état Livré. L'assemblage des composants est modélisé dans le comportement Assemblage. Celui-ci est réalisé dans un ordre donné et composant par composant en fonction de leur livraison.

La définition d'une relation inter-comportements entre Assemblage et Commande (figure 1-1) semble naturelle et donne graphiquement des contraintes sur le lien qui existe entre le comportement Assemblage et l'ensemble de ses composants de type Commande : le comportement Assemblage a de un à n composants qui sont ordonnés. Les états du

comportement Assemblage sont définis par rapport aux états de ses composants en suivant le patron « Comportement distribué » présenté dans la deuxième partie (cf. § 2.4). C'est ainsi que :

```
// Si un assemblage est En Commande, alors le composant courant (indice i) est En Commande.
En Commande = composants->at(indice).in(En Commande) ;
// Si un assemblage est En Construction, alors tous les composants à assembler sont dans l'état Livré.
En Construction = composants->subsequence(0, indice)->forAll(c|c.in(Livré)) ;
// Si un assemblage est Achevé, alors tous ses composants sont dans l'état Livré.
Achevé = composants->forAll(c|c.in(Livré)) ;
```

La définition de tels comportements permet de dépasser le cadre limité de la réutilisation mono-objet, les deux comportements étant réutilisés en bloc dans la dimension phénoménale.

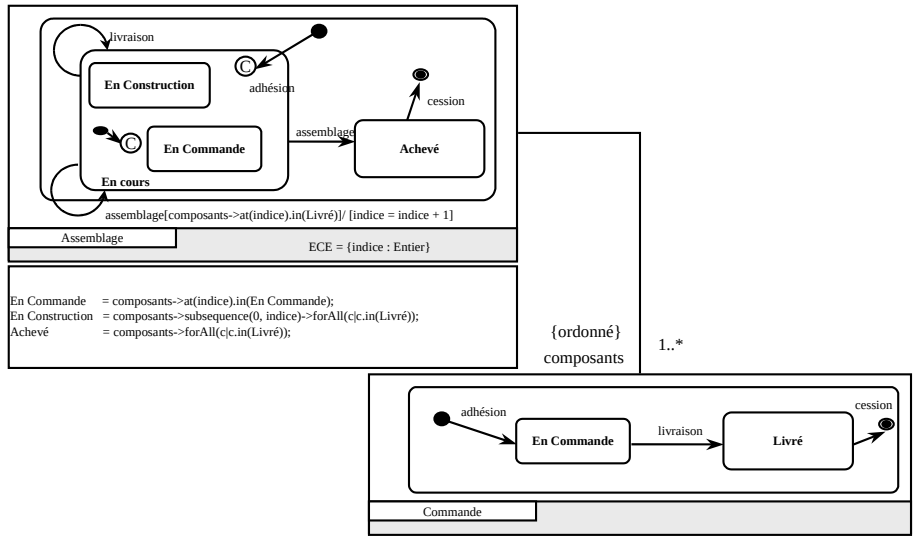


figure 1-1 : Délégation entre comportements

1.2. « Fort » vs. « Faible »

L'objectif principal du modèle NCR est la réutilisation par l'intégration de composants structurels et comportementaux. Nous avons vu que le rôle est le vecteur de cette réutilisation et qu'il répond parfaitement à notre objectif. Cependant, un rôle n'en reste pas moins difficile à spécifier : pour décrire correctement un rôle il est nécessaire d'avoir une bonne compréhension de la notion et du comportement associé, le fait qu'ils soient déjà spécifiés ne résolvant pas tous les problèmes. Lorsque nous avons spécifié le modèle NCR, notre objectif était d'offrir un modèle de contrôle qui ne se limite pas à la détection d'erreurs mais qui permettre d' « emmagasiner » la connaissance structurelle et comportementale. Nous sommes persuadés aujourd'hui que le mariage d'une notion et d'un comportement dans un outil de test peut aller plus loin qu'un simple contrôle des incohérences.

Nous avons introduit deux sémantiques pour le modèle d'exécution des rôles, que nous avons nommées exécution « faible » et exécution « forte ». Tout ce qui a été dit jusqu'à présent sur le modèle de rôles correspond à une sémantique « forte ». Nous allons définir l'exécution « faible » tout en précisant son intérêt. Notons toutefois que cette approche n'est pas validée dans le noyau NCR développé en java et qu'elle constitue à ce jour une piste de recherche significative.

Dans une exécution « forte » toute modification doit avoir été spécifiée. Toute modification non spécifiée est une erreur qui doit être interprétée et corrigée par le concepteur. Lors d'une exécution « faible » nous prenons le contre-pied de cette proposition et considérons des spécifications incomplètes qu'il est possible de raffiner dynamiquement en faisant évoluer le comportement des objets du système.

Le raffinement doit nous amener à une exécution « forte » pour laquelle la spécification comportementale est figée. L'idée de faire évoluer des spécifications comportementales, cette fois basées sur des réseaux de Petri, est aussi présente dans la méthode M7 [DRG99]. Dans cette approche, le concepteur peut librement modifier le réseau en ajoutant ou en modifiant des places et des transitions. Dans l'approche NCR, le concepteur n'intervient que pour valider ou non les évolutions comportementales des objets du système qui lui sont proposées à partir de leur animation. Nous en donnons un exemple intuitif ci-dessous.

1.2.1. EXEMPLE D'EXECUTION « FAIBLE »

Prenons un exemple simple illustrant l'exécution « faible » : supposons que nous voulons spécifier le comportement d'une MachineACafé à l'aide d'un comportement à deux états En Marche et Payé (figure 1-2). Nous réalisons pour cela une intégration du comportement Règlement et de la notion MachineACafé. Cette intégration est volontairement partielle (cf. § 1.2.2) : nous supposons que nous n'avons pas une connaissance suffisante de l'évolution du rôle MachineACafé.

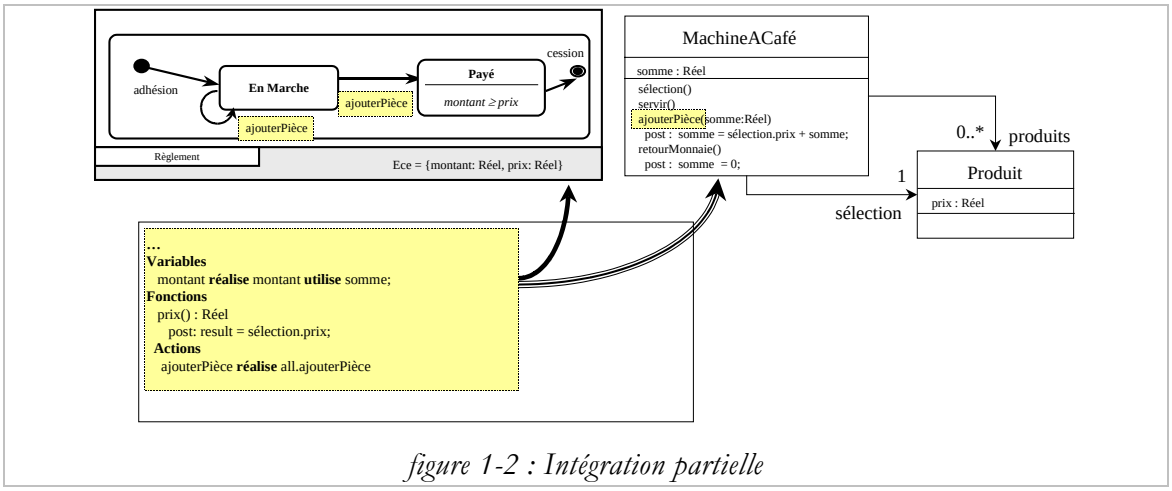


figure 1-2 : Intégration partielle

Nous donnons ci-dessous un exemple de séquencement pour la MachineACafé à partir duquel il est possible de déduire des évolutions non spécifiées dans le rôle et qui doivent être validées par le concepteur lors d'une exécution « faible ». Ce séquencement peut-être obtenu de deux manières :

- Soit le concepteur a implanté dans le langage cible la machine à café dont on veut vérifier le code.
- Soient les spécifications sont rendues exécutables pour en vérifier dynamiquement la cohérence.

	Etat de départ	**Message**	**Etat d'arrivée**	**Commentaire**
1	En Marche	sélection(p)	En Marche	sélection d'un produit p tel que p.prix = 6.0
2	En Marche	ajouterPièce(5.0)	En Marche	insertion d'une pièce de 5F
3	En Marche	ajouterPièce(2.0)	Payé	insertion d'une pièce de 2F, invariant de l'état Payé vrai
4	Payé	retourMonnaie()	En Marche	montant = 0, invariant de l'état Payé faux
Exemple de fonctionnement d'une instance de MachineACafé				

On peut en déduire deux nouvelles propriétés qui doivent être validées par le concepteur afin de compléter la spécification à la fois du comportement Règlement et du rôle MachineACafé :

- La méthode sélection n'a pas changé la valeur du montant. L'invariant de l'état Payé n'est pas vérifié ; la MachineACafé est dans l'état En Marche. La méthode sélection peut donc être intégrée dans une activité réalisée dans l'état En Marche ou dans une action réalisant une transition réflexive dans ce même état.
- La post-condition liée à la méthode retourMonnaie() ne peut être vérifiée en même temps que l'invariant de l'état Payé. retourMonnaie() mène donc obligatoirement dans l'état En Marche. Cette méthode peut être intégrée dans une action qui réalise la transition (Payé, *retourMonnaie*, En Marche).

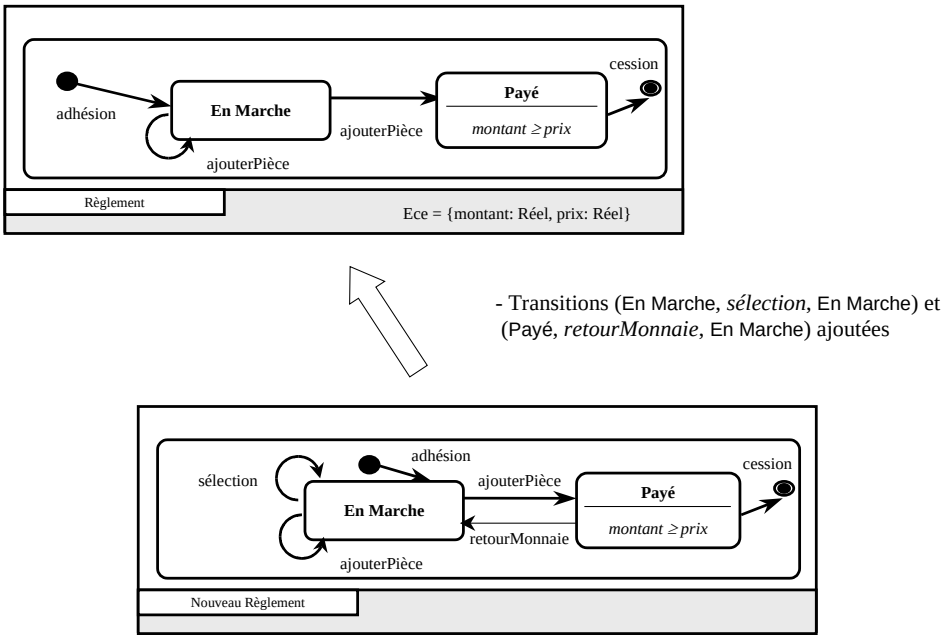


figure 1-3 : Définition interactive d'un nouveau comportement

Nous obtenons ainsi un nouveau comportement (figure 1-3) qui peut-être vu, après validation interactive du concepteur, comme une spécialisation du comportement Règlement (figure 1-2). Si nous considérons qu'il est stable, il peut alors être concrétisé pour une exécution « forte » afin d'aller plus en avant dans les tests.

1.2.2. EXECUTION « FAIBLE » ET ACTIONS, ACTIVITES

Dans une logique « faible », l'utilisation des méthodes ainsi que la réalisation des transitions n'est pas complète. Les règles Rôle [2] et Rôle [3] (chapitre 3 § 2.5.2) ne sont donc plus vraies. L'objectif de l'exécution « faible » est de déterminer dynamiquement l'intégration des méthodes et des transitions dans un rôle donné grâce à l'animation plutôt que de la spécifier statiquement comme cela est réalisé d'habitude.

Dans une sémantique « faible », l'objet doit **au moins** suivre le « contrat dynamique » imposé par le rôle. Cette idée est à comparer avec la différence qui peut exister entre la programmation par contrat et les langages formels. Dans la première, les assertions spécifient ce qui doit être vérifié sur certaines variables alors que dans les seconds, elles spécifient ce qui doit être vérifié pour toutes les variables. Ici, nous nous contentons de vérifier que l'exécution ne va pas à l'encontre de l'évolution partielle spécifiée dans le comportement : lorsqu'une transition peut être franchie, elle est franchie.

1.3.3. EXECUTION « FAIBLE » ET ETATS

Quelle que soit l'exécution, la contrainte élémentaire sur les phénobjets est conservée (cf. chapitre 3 § 2.5.3.2) :

Phénobjet

[1] Un phénobjet est toujours dans un état dont il vérifie la condition (l'invariant).

Cependant, un phénobjet peut être dans plus d'un état à la fois dont il vérifie la condition ; il peut avoir plusieurs états courants. Contrairement à l'exécution « forte », l'exécution « faible » est non obligatoirement déterministe : dans un comportement simple, plusieurs chemins peuvent être empruntés à la fois. La sémantique classique des statecharts est donc fortement modifiée.

L'exécution « faible » est une des raisons pour laquelle il est important de spécifier les invariants d'états. Ces derniers constituent le lien logique entre la structure et le comportement et c'est à partir d'eux que nous levons une partie de l'indéterminisme lié à l'exécution « faible ». Dans certains cas, lorsque les états forment une partition de l'ensemble des valeurs prises par les CE, l'indéterminisme est levé uniquement par les invariants (cf. § 2.5).

1.3. Autorisation vs. interdiction

Au premier abord, nous pouvons considérer l'interdiction utilisée dans le modèle NCR comme une simplification syntaxique très utile pour représenter des comportements complexes. Nous avons dit dans le chapitre 3 que l'interdiction permet d'aller contre le défaut qui veut qu'une transition puisse être franchie dans n'importe quel état co-occurrent.

La possibilité d'interdire est aussi justifiée par l'introduction de l'exécution « faible » dans notre modèle. Dans ce type d'exécution, toute évolution non explicitement interdite qui mène dans un état cohérent est autorisée même si elle n'a pas été spécifiée. Sur la figure 1-4, par exemple, il est possible d'ajouter une pièce dans l'état Payé lors d'une exécution « faible » (puisque rien ne l'interdit) ce qui n'est pas le cas dans l'état En Attente.

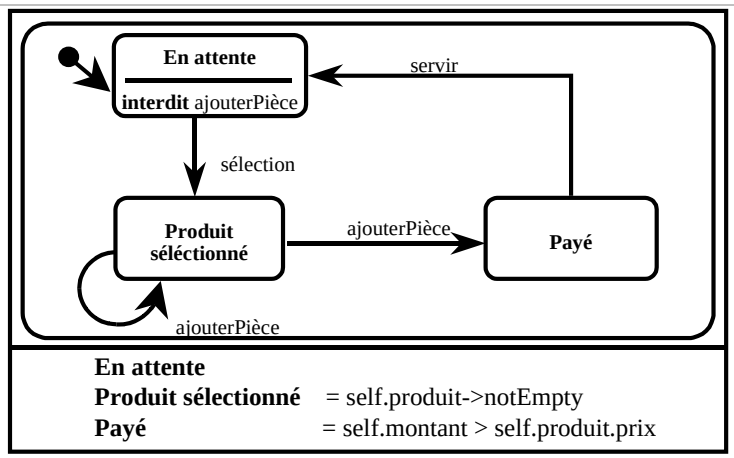


figure 1-4 : Comportement & interdiction

De plus, l'interdiction permet l'expression d'invariants sur les hiérarchies de comportements. Si nous reprenons l'exemple du distributeur (chapitre 3, figure 2-11) pour lequel il n'est pas

possible d'ajouter de pièces dans l'état En Attente, la spécialisation de la *figure 1-5* est interdite à cause de l'ajout de la transition (En Attente, *ajouterPièce*, En Attente).

L'interdiction n'est donc pas qu'un simple sucre syntaxique, elle permet l'expression d'invariants sur une hiérarchie de comportements.

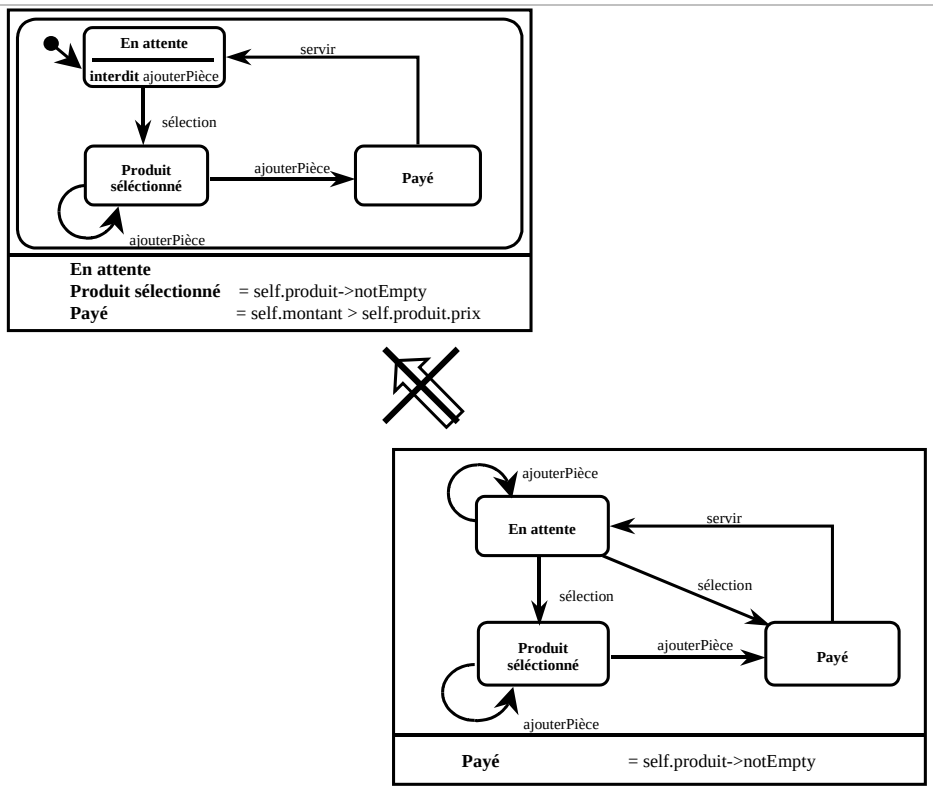


figure 1-5 : Spécialisation interdite de comportement

L'interdiction est comparable à l'« autorisation d'événements » introduite dans les statecharts de la méthode Syntropy [Cook94] : celle-ci associe un ou plusieurs états à un ensemble d'événements qui peuvent survenir lorsque l'objet est dans cet état. Autoriser un événement dans un état revient à lui ajouter une transition réflexive portant l'événement. Il s'agit donc d'une simplification syntaxique. Cependant, ce type de spécification ne convient pas ici car il nécessite de connaître tous les événements qui peuvent survenir et, ce qui est plus dommageable, il oblige à anticiper le raffinement du Statechart en prenant en compte les événements qui seront traités par les statecharts dérivés :

Nous reprenons ci-dessous un exemple d'autorisation lui-même inspiré d'un exemple de la méthode Syntropy [Cook94].

Soit un objet graphique sélectionnable et déplaçable dans une fenêtre. L'objectif est la représentation de ces deux comportements en utilisant l'héritage de statecharts. Nous voyons que la sélection et le déplacement ne sont pas réellement isolés puisque le statechart objetSélectionnable prend en compte les événements de la souris qui conditionnent le

déplacement dans le statechart objetDéplaçable. Ces événements sont ajoutés ici par anticipation sur la redéfinition du statechart objetSélectionable puisqu'ils ne sont pas utilisés pour décrire la sélection de l'objet.

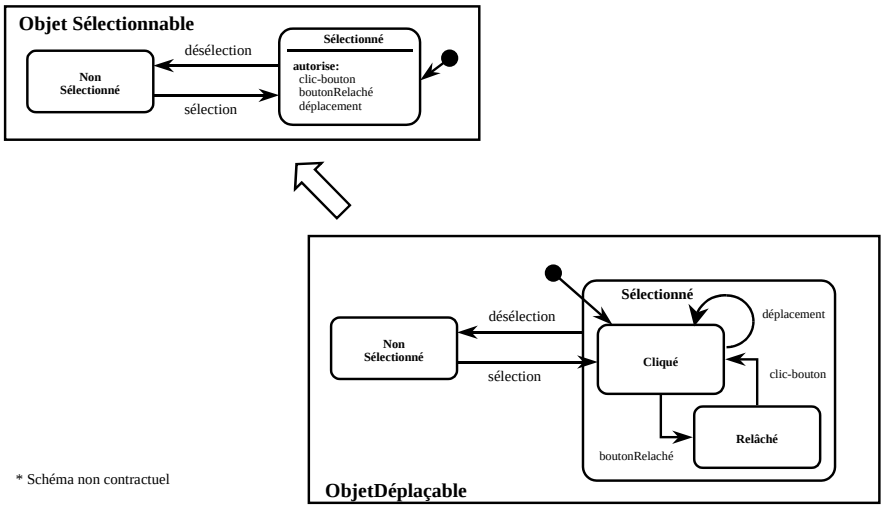


figure 1-6 : Evénements autorisés dans Syntropy

Dans le modèle NCR, nous pouvons définir deux comportements de domaines différents : comportements Sélectionnable et Déplaçable sur la *figure 1-7*. Un comportement équivalent à celui de la figure 1-6 est obtenu par fusion (*figure 1-7*). Cette architecture a l'avantage de spécifier des comportements qui sont réutilisables indépendamment et dans d'autres contextes

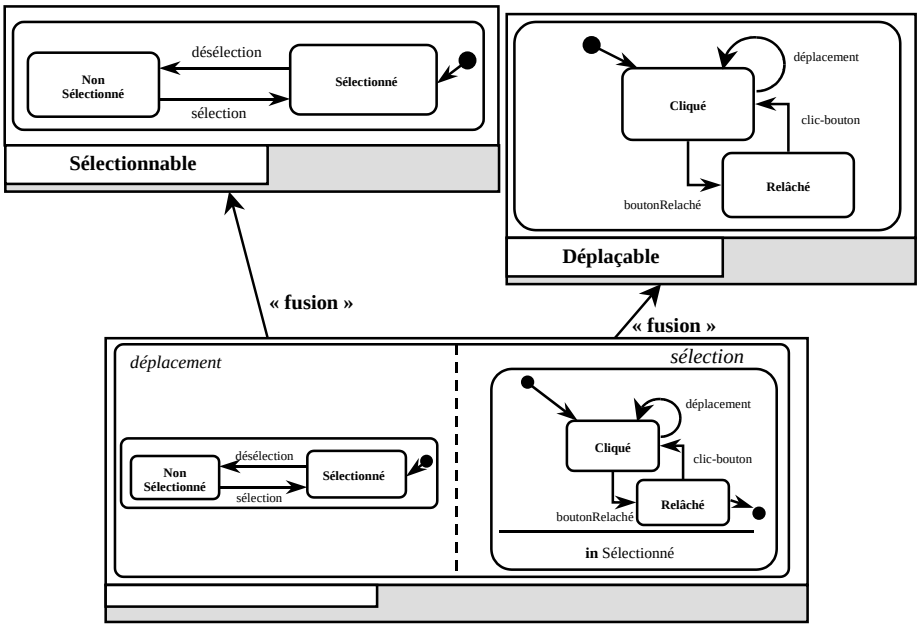


figure 1-7 : Comportements séparés dans NCR

1.4. Comportements & polymorphisme

Nous avons vu que l'architecture du modèle NCR était basée sur des relations d'héritage. Nous pouvons donc abstraire les rôles de deux manières (cf. § 2.3), soit en considérant un rôle comme un sorte de notion, on a là un polymorphisme classique, soit comme une sorte de comportement. Dans ce dernier cas, le type statique de l'objet est de type Rôle et le type dynamique de type Comportement. Au vu du paragraphe précédent, nous devons définir comment il est possible d'animer une instance au travers de son type dynamique dans une logique « forte » et « faible ».

Prenons l'exemple de la figure 1-8. Supposons qu'il existe un rôle R qui concrétise directement le comportement RessourceMultiExemplaires, nous appelons ce dernier comportement statique des instances de R. Nous pouvons voir toute instance de R comme une instance de RessourceMonoExemplaire, ce dernier constitue alors le comportement dynamique de l'instance.

Le comportement de cette instance est restreint à l'exécution :

- Des actions qui réalisent une au moins des transitions définies dans RessourceMonoExemplaire,
- Des activités qui réalisent un au moins des états de RessourceMonoExemplaire.

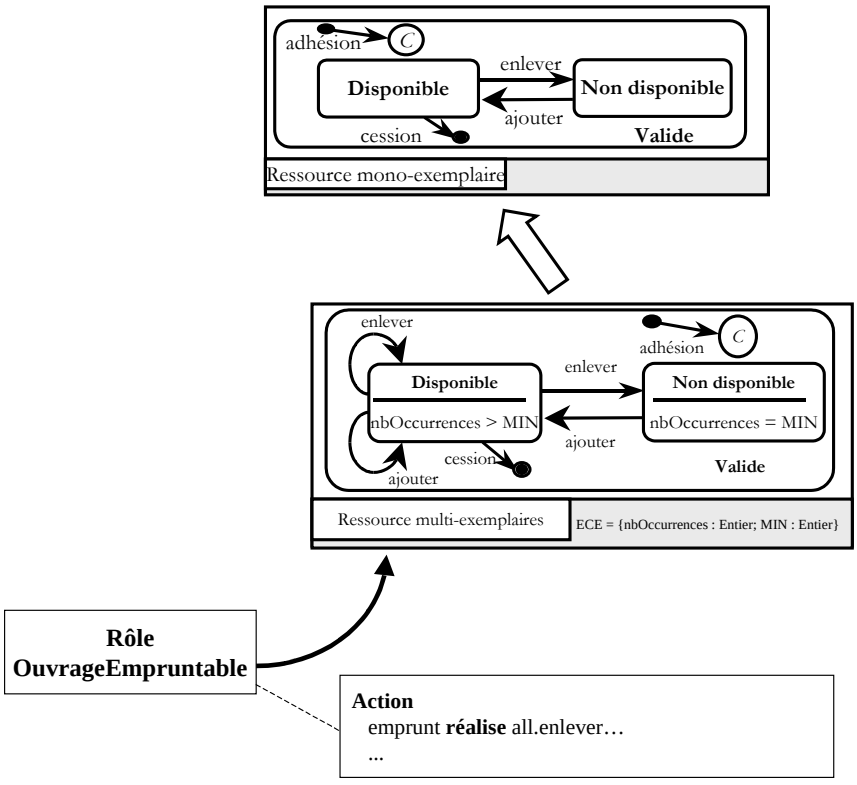


figure 1-8 : Spécialisation de comportements

L'exécution des actions est différente en fonction que nous nous trouvons dans une logique « forte » ou « faible » (cf. § 1.2). Nous simulons aujourd'hui à l'aide du noyau NCR ces deux types d'exécution :

- « Forte »

L'exécution d'une action « forte » (CoersiveAction) est valide s'il existe une transition franchissable dans le comportement dynamique du phénobjet.

La méthode preLink (cf. chapitre 3 § 2.5.3.2) est surchargée pour prendre en compte le polymorphisme. Elle retourne l'ensemble des transitions franchissables par un phénobjet obj lorsqu'il est vu au travers du comportement dynamique behaviour. Sur l'exemple de la figure 1-8, l'exécution de l'action « forte » emprunt retournerait une unique transition (Disponible, *enlever*, NonDisponible).

CoersiveAction : :preLink(Phénobjet obj, Comportement behaviour) :Set(Transition)

Post : result = a.desTransitions->select(t|
behaviour.toutesLesTransitions[29]->exists(t2|t.tousLesParents()->includes(t2)) **and**
self.étatCourant.tousLesParents->includes(t.entrée) **and** t.pré.condition(obj)) ;

// sélectionne l'ensemble des transitions de l'action pour lesquelles il existe une transition parente qui soit une propriété du comportement dynamique behaviour

- « Faible »

L'exécution d'une action est valide s'il existe une transition franchissable dans le comportement statique du phénobjet. Il suffit de vérifier avant exécution la contrainte minimale qui impose qu'une des transitions réalisée par l'action soit une propriété du comportement dynamique behaviour.

WeakAction : :preLink(Phénobjet obj, Comportement behaviour) :Set(Transition)

Pre : a.desTransitions->exists(t|
behaviour.toutesLesTransitions[30]->exists(t2|t.tousLesParents()->includes(t2))) ;

Sur l'exemple de la figure 1-8, l'exécution de l'action emprunt retournerait les deux transitions (Disponible, *enlever*, NonDisponible) et (Disponible, *enlever*, Disponible).

1.5. Conclusion

Dans cette partie, nous avons donné des précisions sur le modèle NCR. Celles-ci ont été choisies pour leur représentativité par rapport au travail que nous menons et aux idées que nous

[29] Comportement : : toutesLesTransitions() : Set(Transition)
post : result = self.toutesLesPropriétés()->select(p|p.isKindOf(Transition)) ;
[30] Comportement : : toutesLesTransitions() : Set(Transition)
post : result = self.toutesLesPropriétés()->select(p|p.isKindOf(Transition)) ;

voulons véhiculer. L'exécution « faible » par exemple est une piste sérieuse que nous explorons en ce moment et qui demande à être formalisée.

Parallèlement à ces travaux sur le modèle lui-même, nous effectuons aujourd'hui des recherches sur la méthode NCR. Nous présentons dans la dernière partie de ce mémoire des fragments du processus NCR. Nous avons « élevé » ces derniers au rang de patrons de conception. En effet, au delà de l'aspect méthodologique, ils abordent des problèmes récurrents liés à la réutilisation comportementale et auxquels nous avons été confrontés dans la littérature concernant ce sujet, ainsi que dans la pratique ou dans l'enseignement de ses principes.

2. FRAGMENTS DU PROCESSUS NCR

NCR est aujourd'hui spécifié en UML et le noyau implanté en java 1.2. Nous savons aujourd'hui simuler tous les comportements qui sont décrits dans ce mémoire. Nous travaillons actuellement sur la définition d'un atelier de conception dédié à l'intégration et à la réutilisation de spécifications comportementales pour les systèmes d'information. Un prototype est en cours d'élaboration et nous espérons qu'il arrivera rapidement à terme. Notre choix s'est pour l'instant porté sur la partie animation ou autrement dit le contrôle dynamique de l'évolution d'objets. Cependant, notre objectif dans cette partie n'est pas de livrer les spécifications d'un possible logiciel NCR mais plutôt de présenter un condensé des idées et des retombées du modèle NCR.

Cette dernière partie se devait d'être fédératrice pour le mémoire et là encore, les patrons nous ont semblé le meilleur vecteur de communication : ils présentent dans un seul formalisme les problèmes rencontrés, la solution habituellement retenue, si elle existe, et finalement notre propre solution conceptuelle et d'implantation. De plus, une modélisation complète des processus liés à la spécification et à l'implantation des composants (notions et comportements) ainsi que ceux destinés plus généralement à l'ingénierie des systèmes d'information par réutilisation de composants n'est actuellement pas possible. En conséquence de quoi, nous nous contentons de présenter dans cette partie quelques principes fondamentaux en utilisant le formalisme des patrons. Nous avons retenu pour cela différents champs :

- **Nom**

  Nom du patron.

- **Intention**

  Problème auquel s'adresse le patron, son point fort.

- **Motivation**

Le problème est illustré par un exemple de modélisation concernant le système d'information de la bibliothèque. Nous référençons dans ce champ les travaux existants susceptibles d'éclairer le lecteur.

- **Description**

Solution conceptuelle décrite à l'aide des concepts NCR.

- **Implantation**

Proposition d'une implantation des points clés du patron dans le langage java (JDK 1.2). Cette solution repose sur le noyau NCR décrit dans le chapitre 3.

- **Patrons liés**

Lien sur des patrons qui soient traitent un problème très proche, soient sont utilisés par ce patron.

- **Dédicace**

Hommage à ceux qui on contribué de près ou de loin à ce patron et de manière générale à cette thèse.

2.1. Plan

Nous avons choisi quatre patrons pour leur intérêt général qui dépasse le simple cadre du modèle NCR. Ceux-ci mettent en évidence des fragments de deux processus NCR complémentaires : le processus d'ingénierie des composants et celui des applications (figure 2-1).

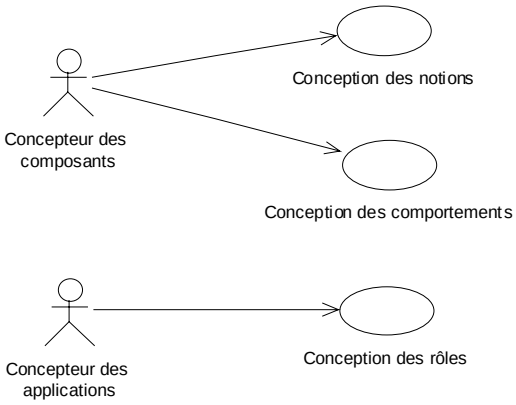


figure 2-1: Deux processus NCR

- Processus d'ingénierie des applications

2.2. Patron « Démarche et réutilisation de comportements »

Ce patron est destiné aux concepteurs des systèmes d'information. Il présente et donne des indications sur la place jouée par chaque dimension structurelle, comportementale et phénoménale, dans le processus de conception NCR.

2.3. Patron « Observer et invoquer, vous n'avez plus à choisir »

Ce patron insiste sur la dualité présente dans les statecharts qui sont utilisés en tant que contrôleurs du comportement en même temps qu'ils autorisent directement son invocation. Cette double vision est intéressante lors du processus de conception car elle offre une réelle interactivité entre l'application développée et les comportements spécifiés pour cette application.

2.4. Patrons « Dynamique des relations » et « Comportement distribué »

Ces deux patrons montrent l'intérêt de découpler les statecharts des classes du système. Le premier identifie plusieurs situations où le comportement significatif est porté par une relation entre classes. Le deuxième patron montre comment il est possible de factoriser un comportement de relation dans une des pattes de la relation.

- Processus d'ingénierie des composants

2.5. Patron « Comportements réutilisables »

Ce patron montre le cheminement suivi pour transformer les statecharts en véritables composants réutilisables. Nous voyons comment il est possible de revenir à un formalisme graphique proche des statecharts à partir de la spécification d'un rôle.

2.2. Démarche NCR

Nom

Démarche et réutilisation de comportements.

Intention

Comment intégrer les spécifications comportementales dans une démarche de conception orientée objet ?

Motivation

De nombreux auteurs proposent aujourd'hui des démarches de développement. Cependant, il semble évident qu'il n'est pas aussi facile d'établir un consensus sur les méthodes de développement que sur des modèles éprouvés depuis longtemps dans le domaine de systèmes d'information. Nous avons montré qu'à partir de seulement deux diagrammes, les diagrammes d'états et les diagrammes de classes, il est possible de définir plusieurs approches, intuitive (cf. chapitre 3 § 1.3.1), opératoire (cf. chapitre 3 § 1.3.2) ou classificatoire (cf. chapitre 3 § 1.3.3), qui souvent ne mènent pas à la même modélisation. Avant de se poser en termes de modèle, notre problématique s'est posée en termes de démarche et d'objectifs :

Comment peut-on intégrer les statecharts dans un processus de conception orienté objet ?

Une première réponse est donnée par la méthode Catalysis et un de ses patrons qui décrit comment nous pouvons intégrer les statecharts dans un système à objets typés. Cette intention peut sembler restrictive par rapport à notre problème mais elle illustre parfaitement la place actuellement tenue par les statecharts, ceux-ci venant se « greffer » sur le processus de développement pour décrire le comportement des objets complexes du système.

- **Nom :**

Utiliser les Statecharts dans un système à objets typés

- **Intention :**

Construire les Statecharts des principaux types du modèle.

- **Description :**

[D'Souza98] définit neuf étapes significatives pour élaborer des statecharts. Pour chacune d'entre elles, il donne quelques recommandations :

1. **Identifier un type avec des états et transitions intéressantes**.
2. **Identifier les états utiles**. Ceux-ci correspondent généralement à un changement radical de comportement.
3. **Dessiner les transitions**. Pour chaque état, nous devons décider de l'applicabilité de chaque opération et la lier à son état résultant. Nous pouvons découvrir de nouveaux états à l'issu de cette étape.
4. **Labeller les transitions avec les opérations qui les déclenchent**. Ajouter les pré-conditions et écrire les post-conditions non indiquées par le changement d'état (par exemple l'incrément d'un compteur).

Un diagramme de transitions d'états est souvent défini de manière informelle dans un premier temps. Les étapes suivantes permettent de le formaliser :

5. **Définir la sémantique des états pour la vue de l'utilisateur**. Faire attention de bien définir ce qu'on entend par l'état aussi bien en terme d'inclusion que d'exclusion.
6. **Définir chaque état comme une fonction booléenne des autres attributs**. Un état doit être défini de manière unique. Les états sont souvent des combinaisons de quelques booléens. Le concepteur doit é**tablir une table de ces combinaisons** en identifiant les combinaisons qui ne correspondent pas à un état. L'ensemble complet des états autorisés forme un invariant de type qui doit être documenté.
7. **Définir les pré et post-conditions** en termes des liens et attributs de l'objet.
8. **Expliciter les ambiguïtés** qui pourraient apparaître. Par exemple, l'absence d'une transition entre deux états peut vouloir dire qu'un objet n'aura jamais la possibilité de passer de l'un à l'autre ou que cette possibilité est réservée aux sous-types.
9. **Intégrer l'information des Statecharts dans la spécification des opérations du système**, notamment lorsque des opérations associées à une transition affectent l'état de plusieurs types d'objets.

Ce patron ne considère pas la réutilisation de statecharts[31]. Il préconise une approche plutôt opératoire des états (sensiblement la même approche que la méthode Syntropy).

[31] Celle-ci ne constitue pas un objectif à atteindre pour les auteurs.

Nous pensons que la réutilisation comportementale ne peut être envisagée sans un processus de développement spécifique pour lequel la part des spécifications structurelles et comportementales est équilibrée. Nous donnons ci-après les grandes lignes d'un tel processus. Ce dernier s'articule autour du modèle NCR.

Description

Nous avons distingué deux processus complémentaires comme dans toute approche par la réutilisation. Le premier est un processus pour la réutilisation. Il consiste à définir les librairies structurelles et comportementales à partir de l'analyse du domaine. Le second est un processus par la réutilisation qui consiste à réutiliser les bibliothèques structurelles et comportementales pour définir des rôles spécifiques à une application donnée. Ces deux processus peuvent être considérés indépendamment :

- Conception pour la réutilisation des composants structurels (figure 2-2).

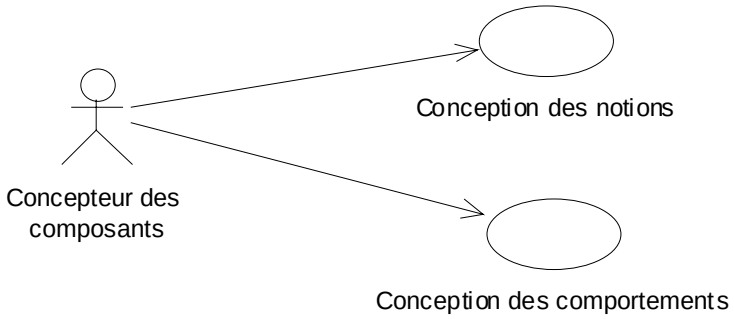


figure 2-2 : Processus pour la réutilisation

Les dimensions structurelle et comportementale sont dédiées à l'identification, la spécification et l'organisation de composants structurels et comportementaux de différents niveaux d'abstraction. Les composants structurels, appelés notions, spécifient les propriétés statiques et fonctionnelles des objets métiers et logiciels des organisations indépendamment des comportements qu'ils pourront adopter au sein des applications. Les composants comportementaux offrent une spécification de comportements abstraits possibles dans cette organisation indépendamment des objets pouvant les adopter au sein des applications. L'idée de base consiste donc, au sein d'une organisation, à favoriser une dissociation complète des propriétés structurelles et comportementales des objets. Un comportement n'exprime plus l'évolution des objets d'une classe mais une évolution potentielle d'un grand nombre d'objets de l'organisation.

Les notions peuvent être déterminées classiquement par une analyse de domaine, le modèle de domaine résultat est représenté en NCR par un diagramme de classes mettant en

évidence les propriétés statiques et fonctionnelles communes et partagées par les objets communs aux organisations de même domaine. Pour des bibliothèques, des exemples classiques de notions sont les documents, les ouvrages, les livres, les abonnés, etc. Dans le cadre d'une organisation, la dimension structurelle est à affiner en fonction de la spécificité des objets de l'entreprise. Pour les « merisiens », il s'agit de prendre en compte les règles de gestion[32] c'est-à-dire les invariants ou contraintes portant sur les objets métiers de l'organisation. Une bibliothèque pourra par exemple imposer un nombre maximal d'emprunts et de réservations par abonné (respectivement 3 et 2 sur la figure 2-3).

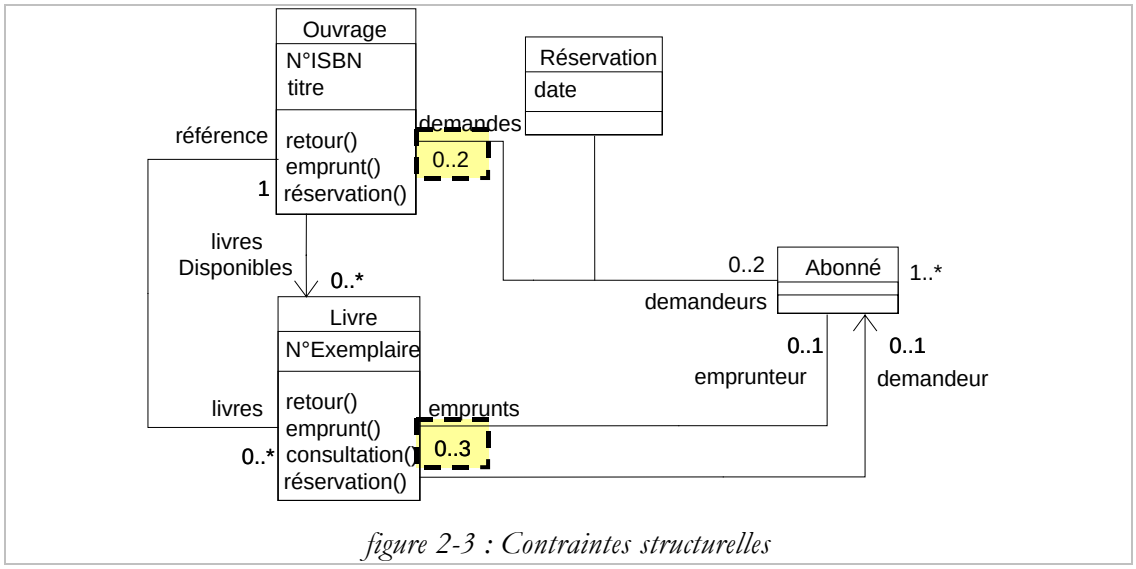


figure 2-3 : Contraintes structurelles

- Conception pour la réutilisation des composants comportementaux (figure 2-2).

Les comportements correspondent à des évolutions abstraites et réutilisables d'objets modélisées par des statecharts. Dans les systèmes d'information, des exemples classiques de comportements réutilisables sont les cycles de conception et de maintenance des produits, d'occupation des ressources, etc. La dimension comportementale capitalise donc des comportements très généraux adaptables à tout domaine par exemple les différents comportements de ressources. L'identification des comportements relève donc plus d'une analyse de situations comportementales types inter-domaines, contrairement aux notions qui correspondent aux propriétés structurelles d'objets de domaine. Ici aussi une organisation utilisant NCR devra étendre ou affiner ces évolutions types en fonction des contraintes de l'organisation, par exemple en fonction de politiques particulières de gestion de ressources. Nous retrouvons comme dans la figure 2-3 une contrainte cette fois comportementale sur la réservation de ressources exprimée par le garde [in réservation.NonRéservé] de la transition ajouter (figure 2-4) : l'ajout d'un élément à une ressource Réservée ne la rend pas Disponible.

[32] Dans la méthode Merise, les règles de gestion sont la « traduction conceptuelle des objectifs et des contraintes acceptées par l'entreprise »… « Leur origine est soit externe à l'entreprise (lois, règlements, etc.), soit interne à l'entreprise (règlements intérieurs, choix de gestion, etc.) » [Collongues86]

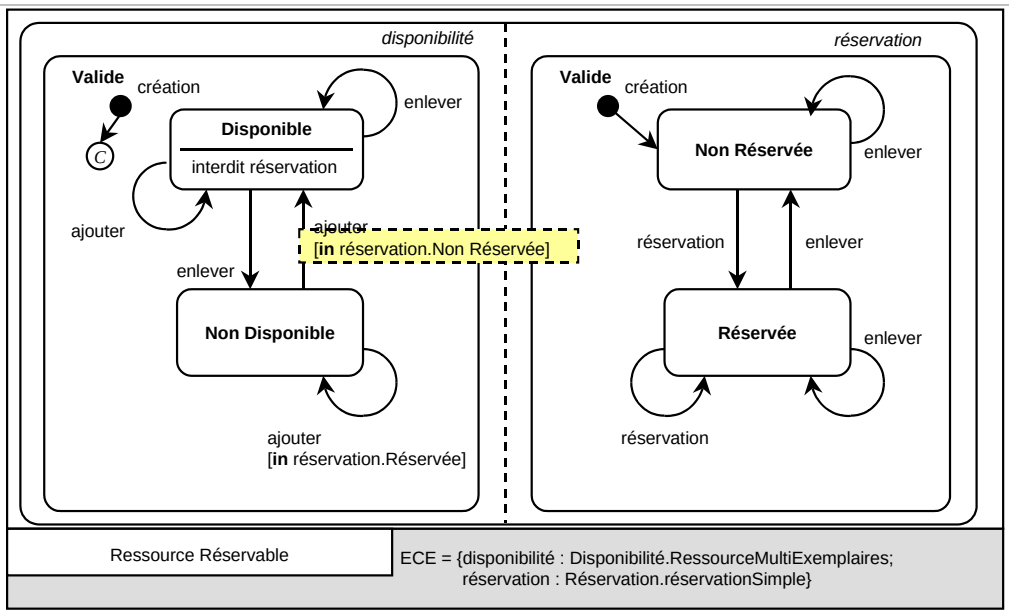


figure 2-4 : Contraintes comportementales

- Conception des rôles pour l'application (figure 2-5).

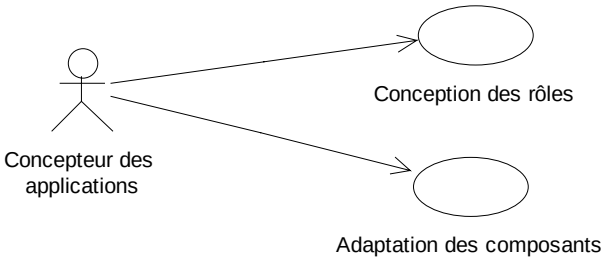


figure 2-5 : Processus par la réutilisation

La troisième dimension est destinée au développement par réutilisation des systèmes d'information. L'objectif est ici de réutiliser, d'adapter et de fusionner les spécifications détenues dans les deux autres dimensions. La spécification d'une application est obtenue en sélectionnant des notions correspondant aux objets métiers (lors de l'analyse) ou logiciel (lors de la conception) intervenant dans l'application et à les doter d'un comportement décrivant leur évolution au sein de cette application.

L'idée de base consiste donc à animer des composants structurels (les notions) par des composants comportementaux (les comportements) pour obtenir des modèles complets d'objets, appelés *rôles*. Le rôle sert de " charnière " conceptuelle entre les dimensions structurelle et comportementale : en général un rôle anime une et une seule notion selon un unique comportement qu'il concrétise.

Dans le processus par réutilisation, il est possible d'exprimer des règles spécifiques à la mise en œuvre de chaque application. Un rôle, par exemple, peut admettre des propriétés

spécifiques qui ne sont issues ni de sa notion ni de son comportement. Il est aussi possible d'adapter des notions et des comportements existants à l'organisation. Dans un système d'information, les propriétés ajoutées dans ce processus correspondent la plupart du temps à l'expression formalisée de règles d'organisation[33].

Les figures 2-4 et 2-6 illustrent les deux niveaux présents dans le modèle :

- Une première contrainte de a été exprimée lors de la conception du comportement RessourceRéservable au sein d'une organisation, par exemple dans une bibliothèque (figure 2-4) : l'ajout d'un élément à une ressource réservée ne la rend pas disponible.
- Lors de l'utilisation de ce composant dans une application cible, par exemple la gestion des emprunts et des réservations des ouvrages, il est possible d'adapter RessourceRéservable pour exprimer une nouvelle contrainte sur les comportements des ouvrages : une ressource (ici un ouvrage) Disponible ne peut être Réservée. Cette contrainte est exprimée sur la figure 2-6 à l'aide d'une interdiction.

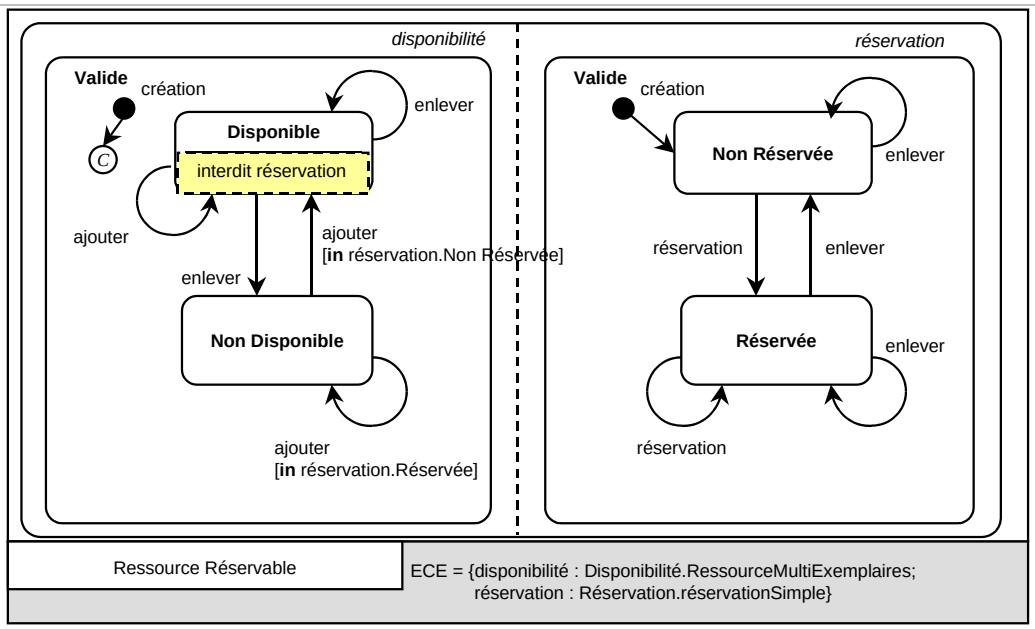


figure 2-6 : Contrainte d'application

Dédicace

Avec des S.I. j'aurai voulu mettre Domi en carafe

Pour connaître encore ce trip au goût de clope.

[33] Dans le méthode Merise, une règle d'organisation traduit l'organisation mise en place pour atteindre les objectifs et les contraintes fixées par les règles de gestion.

De ce burlingue foutoir

Où nous avons jeté le modèle abscons d'une science fatras,

C'est aujourd'hui mon en revoir.

Merci patronne.

2.3. Observation et invocation de statecharts

Nom

Observer et invoquer, vous n'avez plus à choisir[34].

Intention

Les statecharts sont généralement vus comme des contrôleurs (observateurs) d'objets. Une utilisation possible et complémentaire de ce formalisme est l'invocation du comportement d'objet. Nous voyons comment faire cohabiter ces deux visions de l'objet en conservant la possibilité d'étendre et de réutiliser les descriptions obtenues.

Motivation

Prenons l'exemple d'un ouvrage dans une bibliothèque qui a différents comportements dans la gestion d'une bibliothèque, comportements liés à la maintenance, au prêt ou à la réservation. Aujourd'hui, les solutions conceptuelles proposées ne permettent pas à la fois l'observation et l'invocation de ce comportement. Nous avons réellement à choisir. Or l'exemple suivant montre un des intérêts de pouvoir « jongler » d'une représentation à l'autre :

Supposons que l'application soit développée et que nous sommes en phase de test. Nous souhaitons comprendre comment évolue l'ouvrage au travers des différents processus de la bibliothèque. Pour cela nous choisissons (ou décrivons) un comportement qui prend en compte l'évolution de l'ouvrage et intégrons le contrôle de l'ouvrage dans le code de la bibliothèque (cf. patron intégration conceptuelle). Il y a alors deux manières de voir l'évolution de l'objet ouvrage :

- Soit nous utilisons les fonctionnalités de la bibliothèque et nous **observons** les répercussions qu'elles ont sur le comportement de l'ouvrage.
- Soit nous **invoquons** le comportement de l'ouvrage au travers de sa spécification graphique et nous notons les répercussions dans l'application.

[34] Ce titre fait allusion à l'article « Observable or Invocable behaviour, you have to choose » [Ebert94]. Il ne s'agit pas ici d'une polémique mais plutôt d'un « clin d'œil » car le lecteur averti aura remarqué que nous ne parlons pas du problème de la conformité comportementale qui était à l'origine de cet article et de son titre.

De manière générale, ces deux visions complémentaires sont utiles pour comprendre les interactions qui existent entre un objet et son comportement. Elles restent difficiles à mettre en œuvre car font appel à deux types de communication entre objets :

- Lorsque nous observons un objet à l'aide d'un statechart, ce dernier agit comme un contrôleur. L'ajout d'un statechart doit être transparent pour les utilisateurs de l'objet. Nous devons pour cela conserver le protocole de communication : la communication par envoi de messages étant à la base de nombreux modèles à objets, il y a de grandes chances que ce mode de communication ait été choisi et doive être préservé.
- Lorsque nous invoquons le comportement d'un objet au travers d'un statechart, ce dernier agit comme une machine à états prenant en entrée des événements et associant chaque sortie à des services rendus par l'objet. La communication est alors basée sur les événements.

La possibilité de faire cohabiter observation et invocation dans un même modèle passe donc par la coexistence de deux modèles de communication, par messages et par événements. Nous avons identifié deux solutions conceptuelles qui illustrent la première l'observation comportementale et la seconde son invocation :

1. Le patron état (observation) [Gamma95]

Une description complète de ce patron est proposée dans le chapitre 2 § 2.2.1. Nous rappelons que la solution conceptuelle de ce patron consiste à modéliser les états par des classes (figure 2-7 a)).

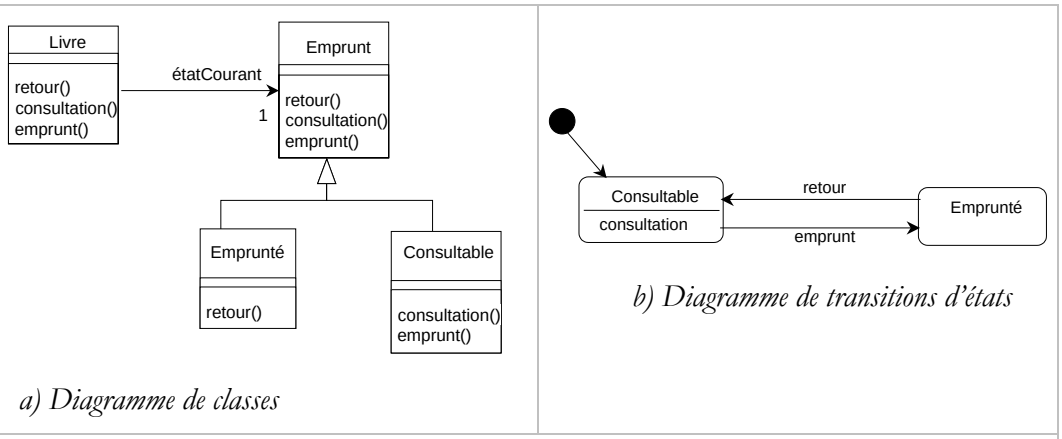


figure 2-7: Patron Etat

L'objectif du patron état est de contrôler les services rendus par un objet. Les états restent transparents pour l'utilisateur, le contrôle étant réalisé par délégation (figure 2-8). Nous restons dans un style de programmation classique (par envoi de messages) :

Une instance de Livre reçoit un **message** (1) dont elle délègue le traitement à l'étatCourant Consultable (2). Ce dernier lui signale le changement d'état (3).

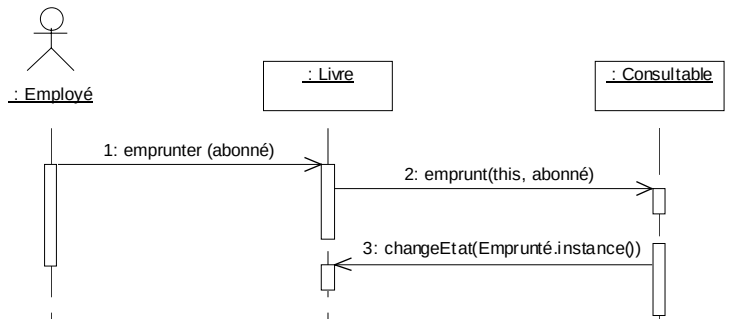


figure 2-8 : Séquencement du patron état (observation)

Cette solution a l'inconvénient de « mélanger » le comportement (le contrôle réalisé par le statechart) et la structure d'un objet (i.e. l'ensemble de ses méthodes) dans les mêmes algorithmes. Il est alors difficile de maintenir et surtout d'étendre le code obtenu. C'est pourquoi, nous trouvons une alternative intéressante au patron état avec le patron « Three Level FSM[35] » bizarrement classé comme un patron de bas niveau par ses auteurs. Ce patron se présente dans notre contexte comme une alternative pour l'invocation du patron état :

2. Le patron état (invocation) [RCM94]

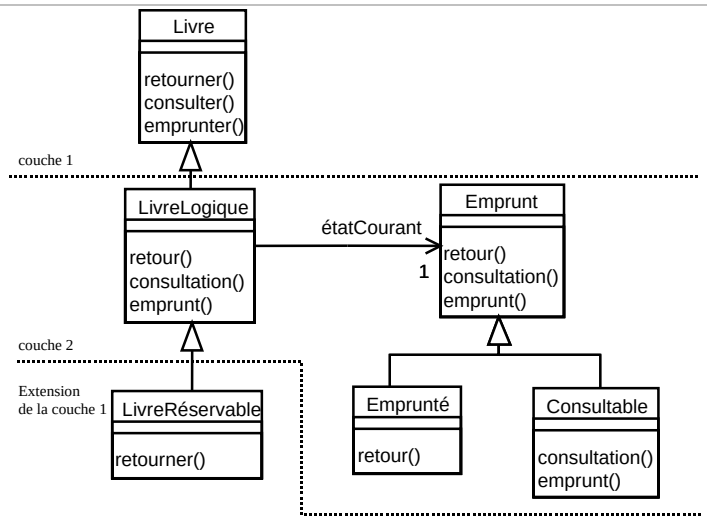


figure 2-9 : Alternative du patron Etat

Les auteurs proposent une variante en trois couches du schéma conceptuel originel du patron état :

1- La première couche décrit les services rendus par l'objet sans tenir compte du contrôle, elle définit en quelque sorte l'interface de l'objet. Sur notre exemple, la classe Livre représente cette couche (figure 2-9).

35 Finite State Machine

2- La deuxième couche implante le contrôle du statechart. Elle définit pour cela les actions qui seront invoquées par les événements du statechart (ils correspondent dans notre exemple aux méthodes de la classe LivreLogique). Les auteurs utilisent un modèle événementiel basique pour implanter cette couche logique. Nous retrouvons à ce niveau la structure du patron état de E. Gamma (classes LivreLogique, Emprunt, Emprunté, Consultable (figure 2-9)).

3- La troisième couche n'est pas présentée ici. Elle est utilisée uniquement lorsque des méthodes de la première couche doivent invoquer des événements de la seconde.

Dans ce patron, la dichotomie événement/action est clairement représentée. Comme le montre le séquencement de la figure 2-10, l'objet est vu au travers de son statechart : envoyer directement un message emprunter(...) à l'objet Livre n'a pas de répercussion sur le comportement associé. En dissociant le comportement (les méthodes de l'objet) de sa logique (le statechart), les auteurs offrent un niveau de réutilisation et d'extension intéressant. En effet, il est facile de réutiliser la classe Livre en dérivant un autre LivreLogique de logique différente. On peut aussi étendre la première couche pour prendre en compte un comportement plus spécifique : le comportement de la classe Livre est redéfini pour prendre en compte la réservation ; seule la méthode retourner est redéfinie dans la classe LivreRéservable. On peut cependant reprocher à ce patron son aspect dissymétrique qui nuit à la clarté des spécifications et donc à sa maintenance lors d'extensions multiples.

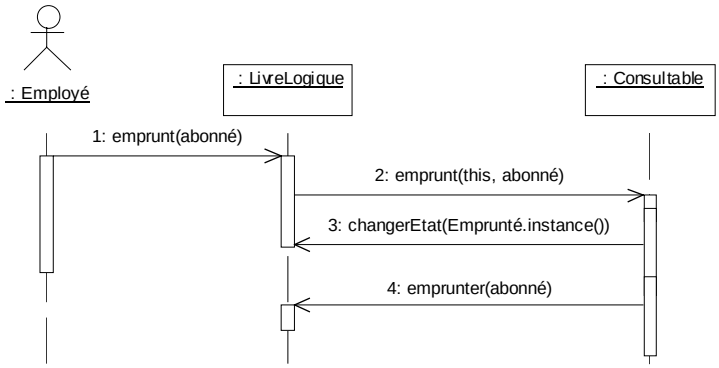


figure 2-10 : Séquencement du patron état (alternative pour l'invocation)

Une instance de LivreLogique reçoit un **événement** emprunt avec un abonné en paramètre (1). Il délègue le traitement à l'étatCourant Consultable (2) qui lui signale en retour le changement d'état (de Consultable à Emprunté) (3). L'état demande alors l'exécution du service correspondant qui a été défini dans la première couche, ici emprunter (4).

Description

Le modèle NCR propose une solution conceptuelle où observation et invocation cohabitent. Celle-ci passe par une véritable intégration conceptuelle de l'objet et de son statechart et par la coexistence de deux modes de communication, par envoi de messages et par production d'événements.

1. Envoi de message et liaison dynamique

Le rôle qui anime une notion hérite de toutes ses méthodes puisque l'animation est réalisée par un lien d'héritage entre notion et rôle. Il doit redéfinir chacune d'entre elle afin d'en contrôler l'exécution mais ne doit en aucun cas modifier les services attendus de la notion (figure 2-11). La visibilité d'un rôle est donc au moins celle de sa notion. Cette clause garantit que les objets qui utilisaient la notion pourront communiquer de la même manière avec le rôle qui l'anime (grâce à la liaison dynamique). Le code généré dans le rôle est ainsi obtenu automatiquement à partir de sa spécification. Seules les méthodes qui autorisaient l'instanciation des notions sont complètement recopiées dans le rôle afin d'instancier des rôles.

La figure 2-10 représente un extrait du noyau NCR écrit en java dans lequel les propriétés exécutables d'un rôle (Action et Activité) implantent l'interface Firable.

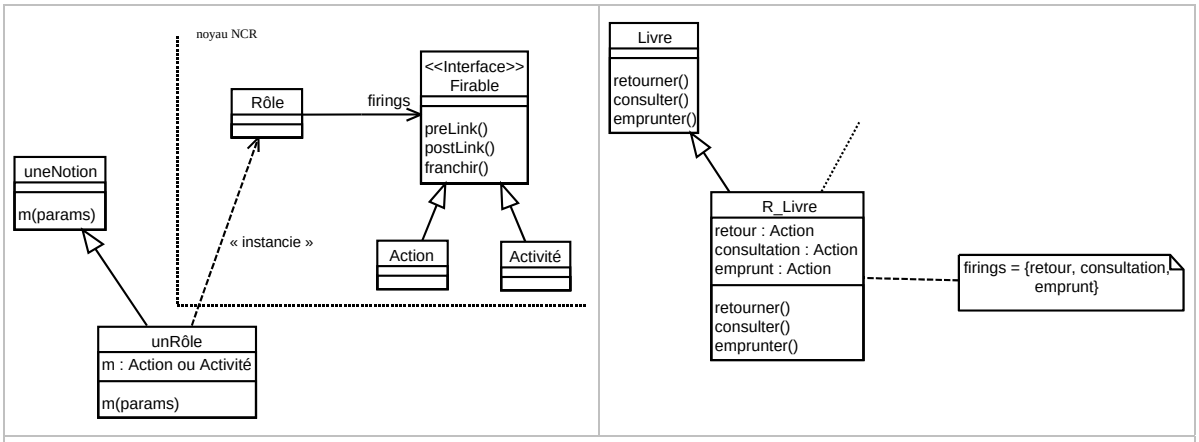


figure 2-11 : Solution NCR pour l'observation

Sur la figure 2-11, les méthodes retourner, consulter et emprunter sont redéfinies dans la classe R_Livre. Nous décrivons ci-dessous (figure 2-12) le séquencement qui permet de contrôler l'exécution des méthodes de notions tout en réutilisant leur code.

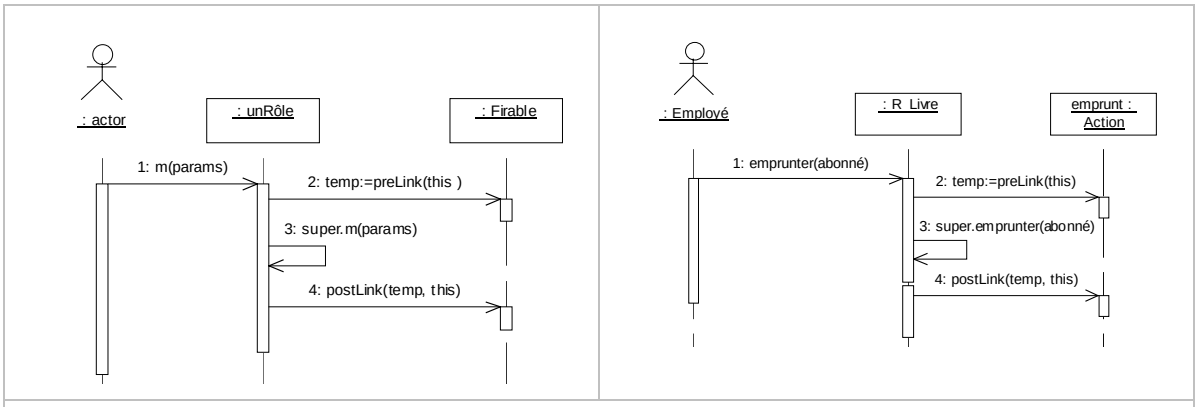


figure 2-12: Contrôle des méthodes de notions

Un phénobjet reçoit un message m (1). Il vérifie qu'il existe des transitions franchissables (vecteur temp) à partir de l'état courant du phénobjet (2)[36]. Il exécute ensuite la méthode correspondante de la notion (3) puis contrôle qu'elle l'a mené dans un état autorisé (4)[37].

2. Emission d'événement et beans

Le modèle événementiel choisi pour l'invocation des comportements d'objet est inspiré des java beans [Englander97].

Le modèle événementiel de java se compose d'objets événement, de récepteurs d'événements et de sources d'événements. Ces objets interagissent de façon standard en invoquant des méthodes pour permettre le déclenchement et la gestion des événements. La figure 2-13 est une illustration de cette interaction. Le récepteur d 'événements se fait lui-même connaître de la source d'événements dont il veut recevoir des notifications d'événements. A un certain moment, la source d'événements déclenche un événement en passant en paramètre l'objet événement ; le récepteur d'événements peut ainsi traiter l'événement.

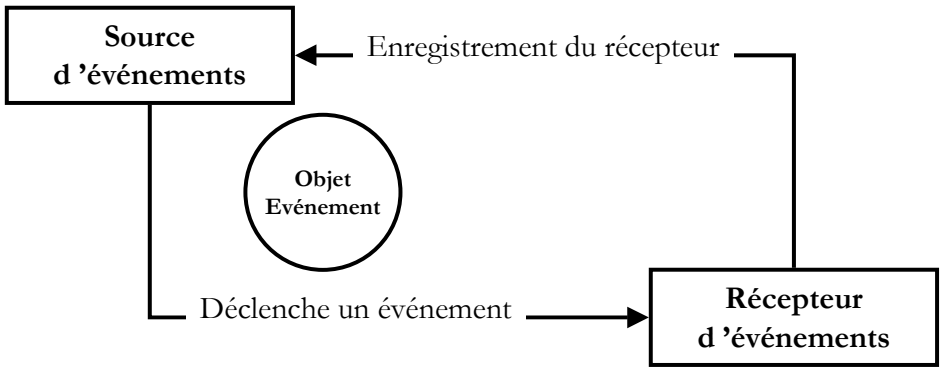


figure 2-13 : Modèle événementiel java

On peut distinguer deux utilisations du modèle événementiel en fonction du contexte dans lequel est utilisé l'objet :

[36] La méthode preLink définie dans le chapitre 3 § 2.5.3.2 renvoie l'ensemble des transitions franchissables.

[37] La méthode postLink définie dans le chapitre 3 § 2.5.3.2 vérifie que le phénobjet est dans un état cohérent.

- Soit il s'agit d'introduire une réelle communication à base d'événements entre objets de l'application, ces derniers interagissant directement entre eux au travers de leur statechart respectif. Dans cette solution, un objet est à la fois source et récepteur.
- Soit le comportement est testé de manière interactive par le concepteur qui manipule les objets à l'aide de sa console. C'est cette solution que nous avons retenue pour des raisons de simplicité et surtout parce qu'elle est facilement automatisable. La figure 2-14 présente le protocole retenu. A la création d'un phénobjet, celui-ci s'enregistre auprès de l'objet collecteur (1). Le collecteur peut alors invoquer le comportement d'un phénobjet en émettant des événements (2). Le phénobjet traite l'événement en franchissant une action ou une activité (3). Tout changement d'état d'un phénobjet est notifié au collecteur (4).

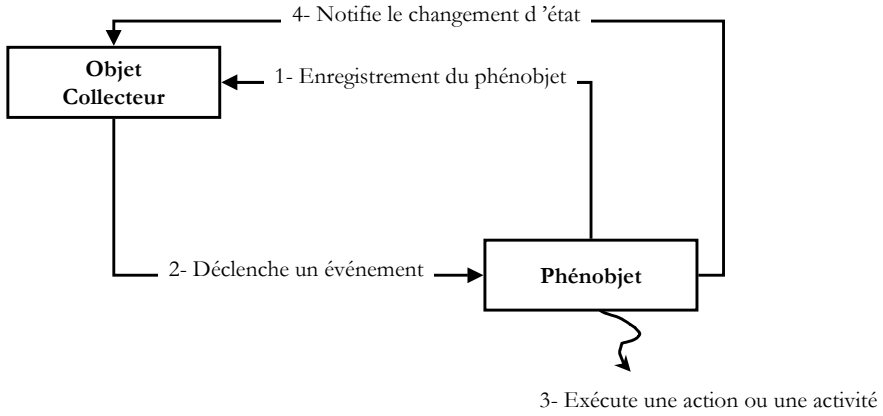


figure 2-14: Solution NCR pour l'invocation

Pour conclure, nous pouvons dire que le modèle NCR constitue aujourd'hui une réponse conceptuelle complète au problème de l'intégration des statecharts dans le modèle à objets. Il offre :

- L'abstraction. Le modèle étant complètement basé sur la relation d'héritage, il est possible d'abstraire un phénobjet en remontant la hiérarchie des notions qu'il anime de même qu'il est possible de remonter la hiérarchie des comportements qu'il concrétise. L'abstraction est donc au cœur de cette solution.
- La réutilisation. Nous l'avons vu, le découplage des notions et des comportements permet d'animer une même notion selon plusieurs comportements et de concrétiser un même comportement pour animer plusieurs notions. La réutilisation est donc plurielle.
- L'extension. Lorsqu'une notion et un comportement sont intégrés dans un rôle, il est toujours possible de redéfinir, soit la notion, soit le comportement. On peut alors respectivement animer ou concrétiser le résultat dans un nouveau rôle qui réutilise les spécifications du premier rôle.

Le tableau ci-après compare les trois solutions introduites dans ce patron :

	Observation	Invocation	Réutilisation	Extension
Patron Etat [Gamma95]	Non	Oui	Non	Difficile
Patron Etat [Englander97]	Oui	Non Pas de contrôle	Structurelle	Limitée et dissymétrique
Solution NCR	Oui	Oui	Structurelle et comportementale	Structurelle et comportementale

Implantation

Nous retrouvons dans la solution NCR, la première couche du patron état pour l'observation. Celle-ci implante les services rendus « hors comportement ». Nous donnons ci-dessous l'implantation partielle du Livre de bibliothèque :

```
public class Livre{

    // attributes

    private String numExemplaire;
    private Ouvrage reference;
    private Abonne emprunteur;
    private Abonne demandeur;
    private Bibliotheque emplacement;


    // constructor
    public Livre(String num, Ouvrage ouv){
      numExemplaire = num;
      reference = ouv;
    }

    // public access methods

public void retour(){
        emprunteur.rendreExemplaire(this);
        emprunteur = null;
        reference.retour(this);
    }

    public void emprunt(Abonne emp){
        if (emp == demandeur){demandeur = null;}
        emprunteur = emp;
        reference.emprunt(this);
        emp.emprunterExemplaire(this);
    }

...
}
```

Le rôle associe cette notion à un comportement de ressource mono-exemplaire :

```
public class R_Livre extends Livre implements Phenomenon_I{

// intégration avec le comportement réalisée par délégation
    public RessourceMonoExemplaire myBehaviour;

// Définition des actions et activités
    public static Action retour;
    public static Action emprunt;
…
    public static CE[] Ece;
```

```
    static{
        try{
          Vector v = new Vector(1);
          v.addElement(RessourceMonoExemplaire.take_1);
          Class[] tab = new Class[1];
          tab[0] = Class.forName("Ncr.Notion.Abonne");
          m = Class.forName("Ncr.Notion.Livre").getDeclaredMethod("emprunt", tab);
          emprunt = new CoersiveAction(m, v, "emprunt");

          v.removeElement(RessourceMonoExemplaire.take_1);
          v.addElement(RessourceMonoExemplaire.let_1);
          tab = null;
          m = Class.forName("Ncr.Notion.Livre").getDeclaredMethod("retour", tab);
          retour = new CoersiveAction(m, v, "retour");
        }
      catch(Exception e){System.out.println("ERROR 4: Action creation");}
    }

// Réalisation des actions et activités tout en redéfinissant les services de la notion

    public void emprunt(Abonne emp){
        Vector temp;
        temp = emprunt.preLink(this);
        super.emprunt(emp);
        emprunt.postLink(temp, this);

    }

    public void retour(){
        Vector temp;
        temp = retour.preLink(this);
        super.retour();
        retour.postLink(temp, this);
    }

}
```

Tous les contrôles nécessaires sont réalisés dans les méthodes de rôles si bien que le franchissement d'une action ou d'une activité en réponse à un événement correspond simplement à l'exécution de la méthode qui réalise cette action ou activité :

```
public class Action extends Propriete implements Firable{

…
public Object franchir(Phenomenon_I obj, Object[] args){
        Object result = null;
        try{
          result = myMethod.invoke(obj, args);
        }
        catch(Exception e){
           System.out.println("ERROR 3: Action invocation mistake");
        }
        return result;
    }
…
}
```

Dédicace

A Philippe pour toutes les fois où on a essayé de lui faire comprendre l'intérêt des patrons.

Car si les patrons meurent, les idées restent et celle-là est de lui.

2.4. Relations et états

2.4.1. PATRON DYNAMIQUE DES RELATIONS

Nom

Dynamique des relations (« Dans quel état tu me vois !»)

Intention

Modéliser l'évolution des occurrences d'une relation à l'aide de comportements.

Motivation

Les états d'un objet sont caractérisés de manière classique par la valeur de ses propriétés, la valeur des propriétés des objets qui lui sont reliés, la valeur des états de ces mêmes objets, ses relations ou absence de relations avec d'autres objets et finalement la valeur des propriétés de ses relations avec d'autres objets (cf. chapitre 1 § 2.1.1.1) [Panet94]. Cette définition des états d'objets permet de modéliser un panel assez large de situations. Elle s'avère parfois insuffisante. C'est le cas par exemple lorsqu'on souhaite modéliser qu'un produit est en rupture de stock dans un dépôt et qu'il ne l'est pas dans l'autre ou qu'un intérimaire est en période d'essai sur un contrat et qu'il a bientôt terminé cet autre contrat. Dans toutes ces situations, l'association classe/statechart n'offre pas de solution convaincante.

Nous prenons l'exemple de la bibliothèque et de ses ouvrages. La bibliothèque est un réseau qui comporte plusieurs antennes dans une même ville. Nous avons parfois à répondre à des questions du style : « Y a t'il des exemplaires disponibles de l'ouvrage de B. Werber dans la bibliothèque principale ? Et dans les autres antennes ? ». Tout se passe comme si un ouvrage avait un état pour chaque antenne : la figure 2-15 montre que les exemplaires de l'Ouvrage sont TousEnRayon dans la bibliothèque de Cordon alors qu'il y a AucunExemplaire disponible au centre de Sallanches. L'état courant des objets est indiqué par un fond gris.

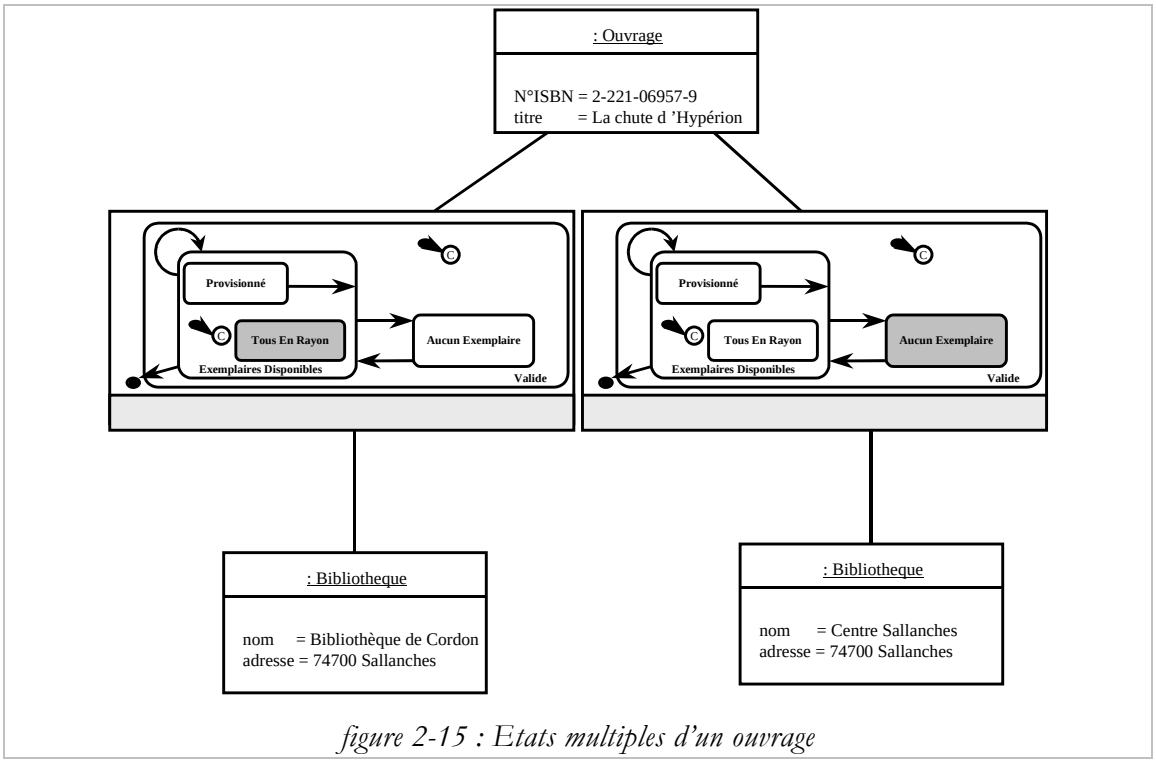


figure 2-15 : Etats multiples d'un ouvrage

Description

Deux questions se posent à la lecture de ce problème : peut-on modéliser cette situation à l'aide de statecharts ? Si oui, peut-elle être modélisée de manière classique ? L'exemple ci-dessus donne une réponse à la première question. La réponse à la seconde n'est pas immédiate. En effet, ce problème relève de la modélisation d'évolutions multiples or les statecharts permettent ce type de représentation. Alors :

Pourquoi la co-occurrence n'apporte t'elle pas de solution dans ce cas là ?

La co-occurrence permet l'expression de comportements multiples dont la multiplicité est fixée. Le nombre de statecharts décrivant le comportement d'un objet par rapport à une relation varie en fonction du nombre d'occurrences de cette relation pour lesquelles l'objet est impliqué. Pour l'exemple de la société d'intérim, la création de nouveaux contrats obligerait à modifier le Statechart associé aux intérimaires en lui ajoutant une dimension d'évolution. Réciproquement si le nombre d'intérimaires variait pour un contrat, nous serions obligés de modifier le statechart de ce dernier. Cette situation ne peut donc pas être modélisée en utilisant la co-occurrence.

Nous ne pouvons représenter ce type de comportement en utilisant les mécanismes classiques car il s'agit ici de représenter non pas une dynamique d'objet mais bel et bien une dynamique de relation entre deux classes : une instance de l'une de ces classes peut être vue au

même moment dans des états différents par les instances de l'autre classe. Ce constat nous amène à proposer une solution associant statechart et relation d'objets. Nous décrivons ci-après cette solution à l'aide des concepts NCR.

Soient deux notions N1 et N2. Soit un comportement C qui décrit potentiellement la relation qui unit N1 à N2. L'intégration de ces trois paradigmes dans la dimension de rôles n'est pas automatique ; nous devons :

1- Animer les notions N1 et N2 avec respectivement les rôles R1 et R2 (figure 2-16).

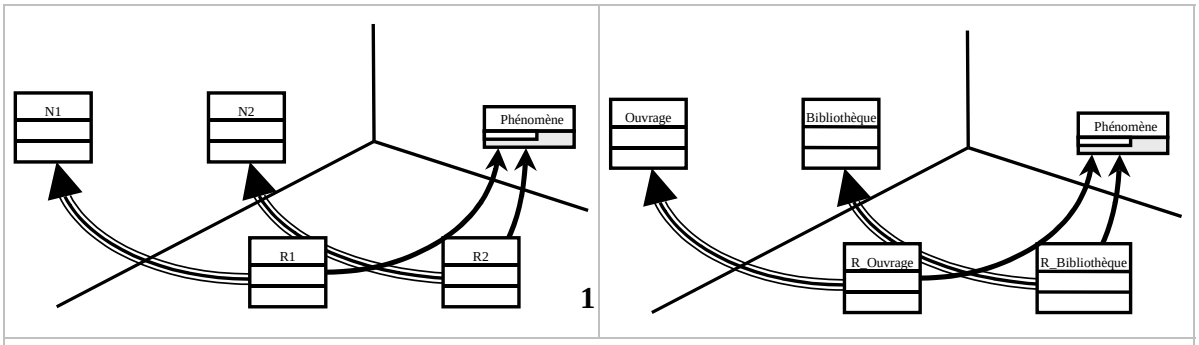


figure 2-16 : Animation des pattes de la relation

Nous n'associons pas ici de comportement spécifique à N1 et à N2. R1 et R2 concrétisent par défaut Phénomène.

2- Créer un rôle R3 qui réifie la relation entre R1 et R2.

Cette réification de la relation est obligatoire lorsqu'elle n'existe pas déjà dans le modèle structurel. On retrouve là un parallèle intéressant avec la définition des classes-associations [Muller97] qui est utilisée dans la dimension des notions pour ajouter des propriétés structurelles aux relations. A ce niveau sont ajoutées les méthodes qui permettent la création de la relation (dans R1 ou R2) et la navigation entre R1, R2 et R3 (figure 2-17). Cette navigation peut être facilitée en utilisant des dictionnaires indexés (cf. implantation).

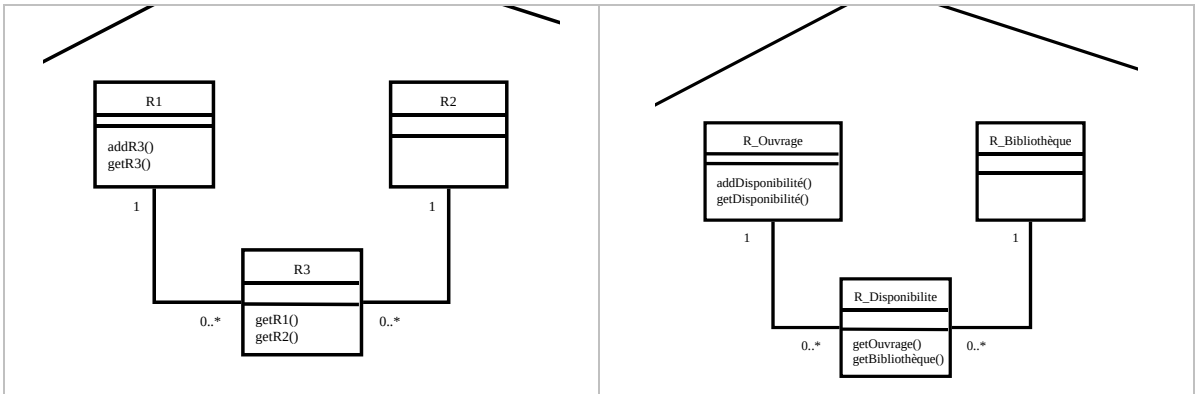


figure 2-17 : Définition de la classe-association

3- Spécifier la concrétisation de C dans R3.

La concrétisation des caractéristiques d'évolution (CE) de R3 est réalisée à l'aide des propres attributs de la relation ou par calcul sur des propriétés de R1 et de R2. La plupart des actions et des activités de R3 sont réalisées par délégation en utilisant les méthodes de N1 et de N2. Dans ce patron, la difficulté réside dans le fait que ce sont les attributs et les méthodes des notions liées qui sont intégrés avec le comportement.

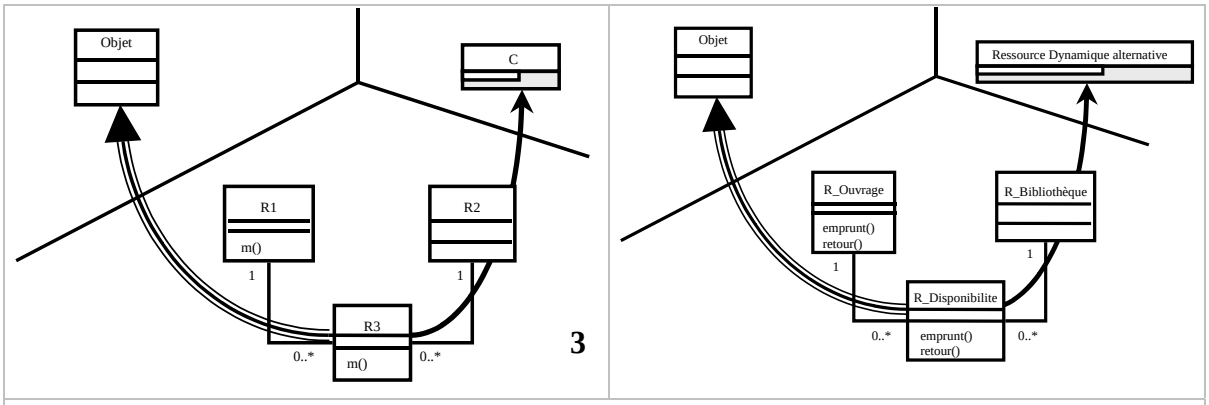


figure 2-18 : Solution complète dans NCR

La relation contrôle l'évolution de ses deux pattes. Les méthodes de R1 et de R2 qui sont utilisées par la relation pour réaliser ses actions doivent être redéfinies, c'est l'exemple des méthodes emprunt et retour, de R_Ouvrage (figure 2-18). Le contrôle peut être vu là aussi (cf. Patron « Observer et invoquer » § 2.3) de deux manières, soit par l'observation des services rendus par les instances de R1 et de R2, soit par l'invocation directe du comportement de la relation R3. La figure ci-dessous représente un séquencement possible pour le contrôle des méthodes de R1 :

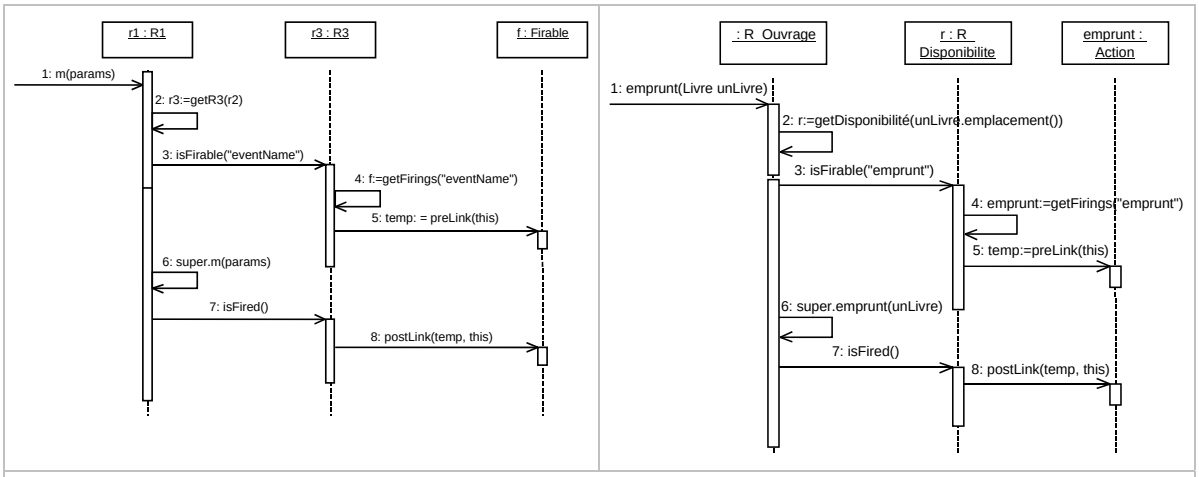


figure 2-19 : Contrôle de l'exécution par la relation

A réception du message m (1), r1 délègue le contrôle à la relation r3 susceptible d'être modifiée après exécution de m. r3 est recherchée dans le dictionnaire (2) à partir de l'indice r2 déduit directement ou indirectement des paramètres de m. On vérifie que l'action correspondante

est franchissable (3..5) avant d'exécuter la méthode m (6) décrite dans la notion N1 (N1 est parente de R1). Le bon franchissement de l'action est ensuite contrôlé (7..8).

La figure 2-20 présente une vue détaillée de la solution présentée dans la figure 2-18.

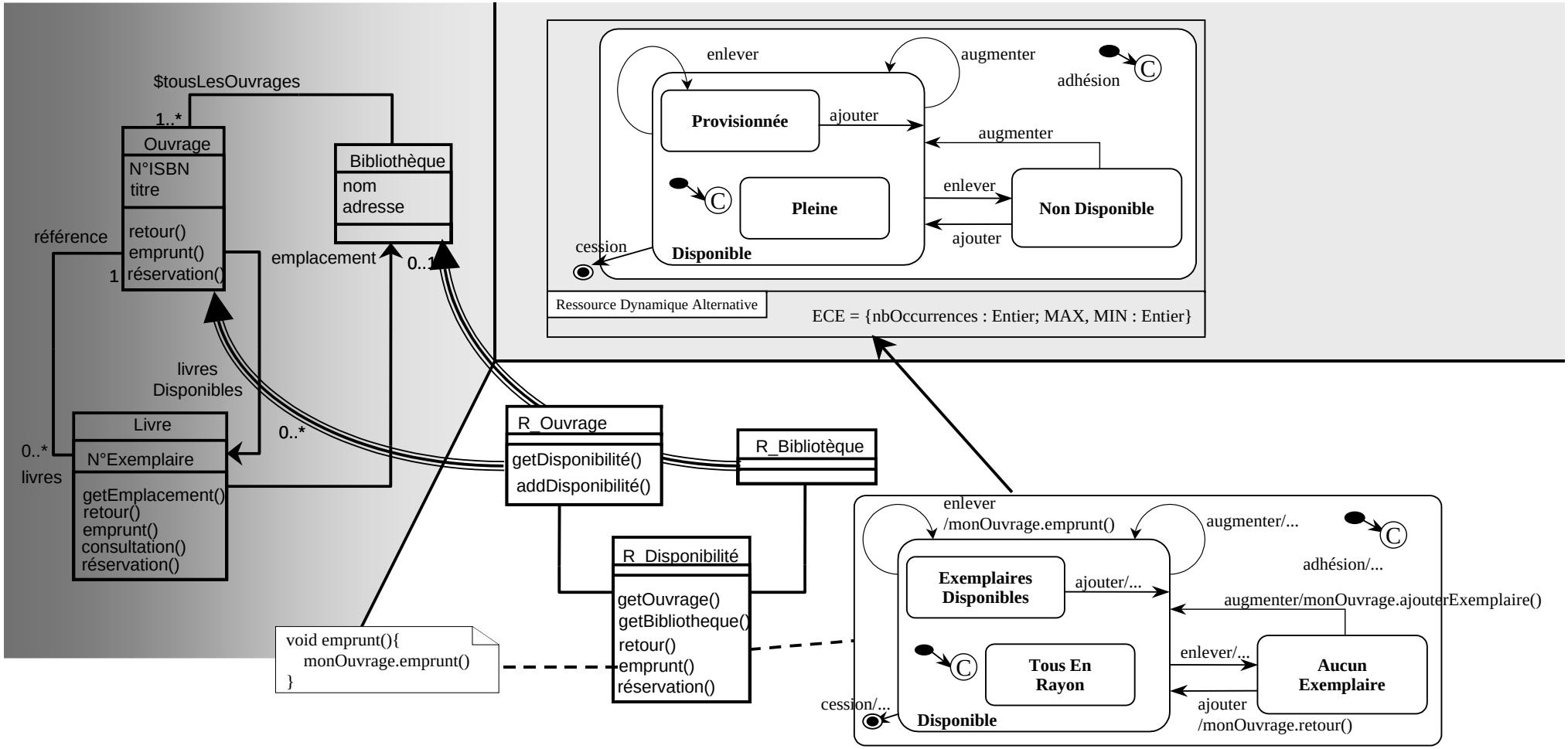


figure 2-20 : Utilisation du patron « Dynamique des relations » pour la bibliothèque

Implantation

Nous implantons dans NCR le statechart contrôlant la dynamique de la relation Ouvrage/Bibliothèque. Nous devons pour cela :

1. Implanter les rôles R_Ouvrage et R_Bibliotheque.

Nous faisons abstraction de l'implantation de R_Ouvrage et de R_Bibliotheque pour nous intéresser uniquement à l'implantation de la relation qui les unit.

2. Créer un rôle R_Disponibilite qui réifie la relation entre R_Ouvrage et R_Bibliotheque.

Nous devons réaliser les relations entre ces trois rôles. Dans cet exemple, seul R_Ouvrage accède R_Disponibilite à l'aide d'un dictionnaire indexé sur les instances de R_Bibliotheque. En effet, nous avons choisi de donner le contrôle de la relation aux instances de R_Ouvrage qui seules peuvent modifier son état.

L'utilisation d'un dictionnaire (Hashtable) permet à une instance de R_Ouvrage d'accéder de manière unique et efficace au lien qui la lie à une bibliothèque du réseau. R_Disponibilite peut accéder à R_Ouvrage et à R_Bibliotheque en les référençant à l'aide de deux attributs (resp. monOuvrage et maBib).

```
public class R_Disponibilite extends PhenObject implements Phenomenon_I{

    // relations entre rôles

    protected R_Ouvrage monOuvrage;
    protected R_Bibliotheque maBib;

    // méthodes publiques permettant de référencer les pattes de la relation

   public R_Ouvrage getOuvrage(){return monOuvrage;}
   public R_Bibliotheque getBibliotheque(){return maBib;}
… }
public class R_Ouvrage extends Ouvrage implements Phenomenon_I{

    // dictionnaire d'instances de R_Disponibilité indexé sur les instances de R_Bibliotheque
        protected Hashtable mesLiens;


    // méthodes pour gérer la relation

    public void ajoutLien(Bibliotheque bib, R_Disponibilite rel){mesLiens.put(bib, rel);}

    public R_Disponibilite getLien(Bibliotheque bib){return (R_Disponibilite)mesLiens.get(bib);}
….}
```

3. Implanter la concrétisation de C dans R3.

Pour cela, nous réalisons dans un premier temps les CE en traduisant les requêtes écrites en OCL :

```
public class R_Disponibilite extends PhenObject implements Phenomenon_I{
    public int MIN(){return 0;}

// return self.monOuvrage.livres->select(l|l.getEmplacement() = maBib)->size()
    public int MAX(){
     int max = 0;
     Enumeration e = monOuvrage.livres.elements();
```

```
      while (e.hasMoreElements())
         if(((Livre)e.nextElement()).getEmplacement() == maBib)
            max++;
      return max;
     }

// return self.monOuvrage.livresDisponibles->select(l|l.getEmplacement() = maBib)->size()
public int nbOccurences(){
   int qty = 0;
      Enumeration e = monOuvrage.livresDisponibles.elements();
      while (e.hasMoreElements())
         if(((Livre)e.nextElement()).getEmplacement() == maBib)
            qty++;
      return qty;
     }
…}
```

Nous implantons ensuite les actions en intégrant les méthodes de R_Ouvrage avec les transitions du comportement RessourceDynamiqueAlternative. La partie réalisation de l'action (la méthode associée) est réalisée par délégation plutôt que par intégration directe pour des raisons techniques. Nous proposons ci-dessous le code développé suivant le séquencement de la figure 2-19 pour l'action emprunt dans le rôle R_Disponibilite et pour la méthode emprunt définie dans le rôle R_Ouvrage.

```
public class R_Disponibilite extends PhenObject implements Phenomenon_I{

// Meta definitions
     public static Action emprunt;
…
     // définition de l'action emprunt par concrétisation de deux transitions take_1 et
     // take_2. L'action anime normalement la méthode « emprunt » de Ouvrage, cette
     // animation est réalisée par délégation (*)
     static{
         try{
           Vector v = new Vector();
           v.addElement(RessourceDynamiqueAlternative.take_1);
           v.addElement(RessourceDynamiqueAlternative.take_2);
           Class[] tab = new Class[1];
           tab[0] = Class.forName("Ncr.Notion.Livre");
           Method m =
            Class.forName("Ncr.Role.Bibliotheque1.R_Disponibilite").getDeclaredMethod("emprunt", tab);
           emprunt = new CoersiveAction(m, v, "emprunt");
         }
       catch(Exception e){System.out.println("ERROR 4: Action creation");}
     }
…
     // (*) réaliser l'action par délégation
     public void emprunt(Livre unLivre){
               monOuvrage.emprunt(unLivre);
     }

…
}


public class R_Ouvrage extends Ouvrage implements Phenomenon_I{

// formal behaviour

     public void emprunt(Livre unLivre){
     // obtenir la relation concernée par l'événement
          R_Disponibilite r = getDisponibilite(unLivre.getEmplacement());
     // tester si la méthode peut être exécutée par rapport à l'état courant de la relation
          r.isFirable("emprunt");

     // réaliser l'action
          super.emprunt(unLivre);

     // contrôler le bon franchissement de transition
          r.isFired();
     }
…
}
```

Patrons liés

Si la relation a un comportement de même domaine qu'une de ses pattes, appliquer le patron « Comportement distribué ».

Dédicace

A vous, relations passées ou bien vivantes, tellement avenantes et quelques fois déconcertantes,

A vous, relations volubiles ne sachant durer, relations pénibles ne voulant cesser,

A vous, relations de boulot et joueurs de squash qui m'ont aidé à terminer cette thèse,

A vous, étudiants qui m'avez supporté,

A vous, lecteur inconnu qui tenez en ce moment cette relation ténue entre vous et moi,

A Frédéric (vous le prendrez avec ou sans dard ?),

A Bernard (quand un thanatonaute rencontre un saint),

A ma famille albygeoise.

Toutes ces relations fragiles qui font de moi un homme.

Nom

Comportement distribué (« Dans quel état vous m'avez mis !»)

Intention

Modéliser un comportement distribué

Motivation

Nous rencontrons fréquemment des situations où les propriétés structurelles d'un objet sont distribuées au travers d'une relation. Le salaire d'un employé par exemple peut-être distribué dans chacune des entreprises qui l'emploie (figure 2-21) ; la performance d'un projet dépend de la performance de chacune des personnes qui participent à ce projet.

Dans tous ces cas, nous retrouvons une idée commune : une propriété peut définir globalement un objet en même temps qu'elle le définit localement au travers de ses relations. La propriété globale est alors la résultante de toutes ces propriétés locales. Ainsi notre employé a une rémunération globale qui est la somme des salaires versés par chaque entreprise ; l'état d'avancement d'un projet est calculé en fonction des tâches accomplies par chaque personne.

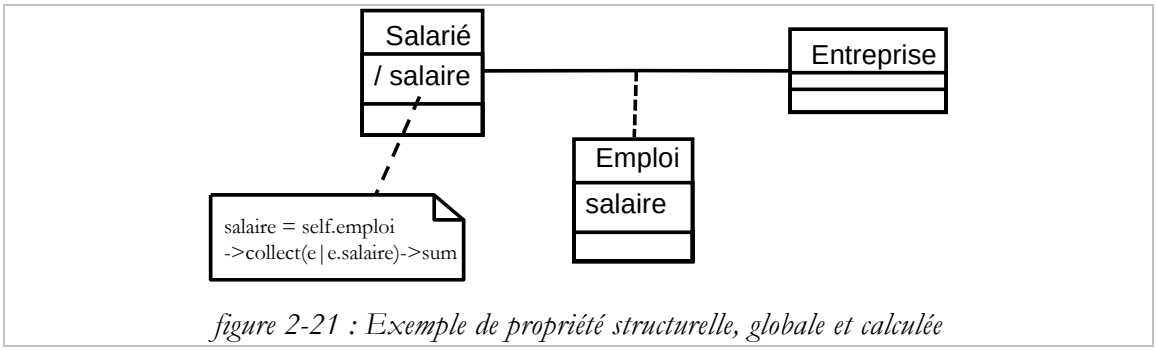


figure 2-21 : Exemple de propriété structurelle, globale et calculée

Nous allons voir que cette situation se rencontre aussi pour les propriétés comportementales d'objets. L'exemple de la bibliothèque est encore là pour l'illustrer. Dans le patron « dynamique des relations », nous nous sommes intéressés à la disponibilité locale des ouvrages pour chaque bibliothèque du réseau. La disponibilité des ouvrages peut aussi être vue globalement pour l'ensemble des bibliothèques, et ce avec un comportement très proche (de même domaine), voir le même (figure 2-22). Nous pouvons faire deux constats :

- Le statechart global est en quelque sorte un statechart calculé, comme pouvait l'être l'attribut du salarié (figure 2-21). Nous déduisons notamment les conditions portées par les états à partir des états locaux.
- Ce type de modélisation est intéressant car le passage d'un comportement centralisé à un comportement distribué se fait avec une forte réutilisation et donc à moindre coût.

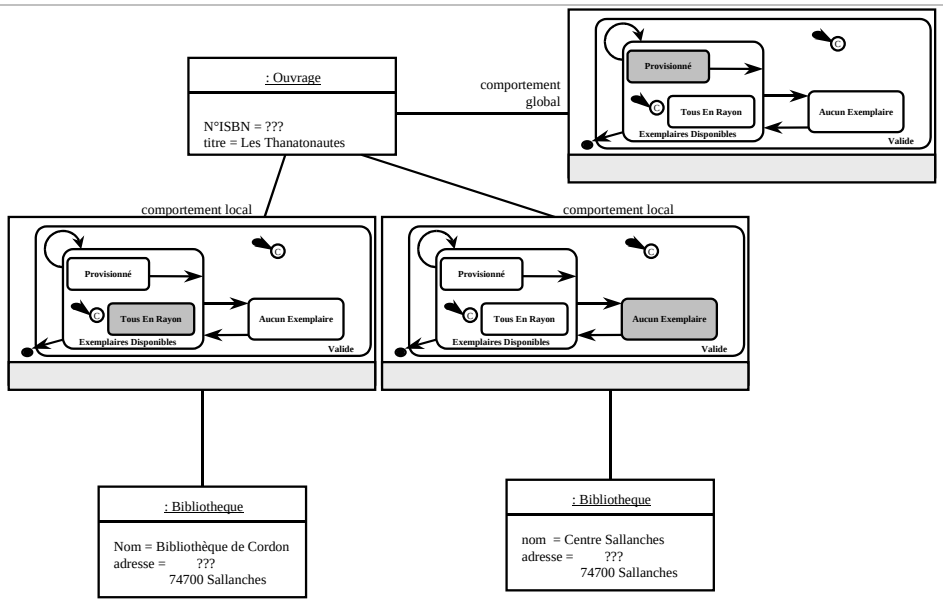


figure 2-22 : Comportement calculé d'un ouvrage

Description

La solution conceptuelle de ce patron utilise celle du patron « Dynamique des relations » (figure 2-18) pour décrire le comportement de chaque relation R_Disponibilité. Il suffit ensuite d'exprimer le comportement global de R_Ouvrage à partir du comportement de ces relations. Les relations qui lient les états globaux aux états distribués sont exprimées dans la dimension comportementale à l'aide de la caractéristique d'évolution occurrences de type Ensemble de RessourceLimitée :

```
//Une ressource est pleine lorsque toutes ses occurrences sont pleines
Pleine          = occurrences->forAll(o|o.in(Pleine));
//Une ressource est provisionnée lorsqu'il existe deux de ses occurrences qui ne sont pas dans le
même état ou bien que toutes ses occurrences sont provisionnées.
Provisionnée    = occurrences->forAll(o1, o2 |
                    o1.getCurrentState().isParent(o2.getCurrentState()) or
                    o2.getCurrentState().isParent(o1.getCurrentState()))
                     implies
                    occurrences->asSequence()->first().getCurrentState().isParent(Provisionnée);
//Une ressource est non disponible lorsque toutes ses occurrences sont non disponibles
NonDisponible   = occurrences->forAll(o|o.in(NonDisponible));
```

L'intégration de la notion Ouvrage et du comportement RessourceCompositeDynamique dans le rôle R_Ouvrage est triviale. La CE occurrences est réalisée par la relation mesLiens qui lie la notion R_Ouvrage à R_Disponibilité, cette dernière concrétisant le comportement RessourceDynamiqueAlternative dont le parent est RessourceLimitée (figure 2-23). Les méthodes de la notion sont intégrées de manière classique. Cependant, l'ordre des contrôles est important. Nous montrons ci-après un séquencement possible pour l'exemple de la figure 2-23.

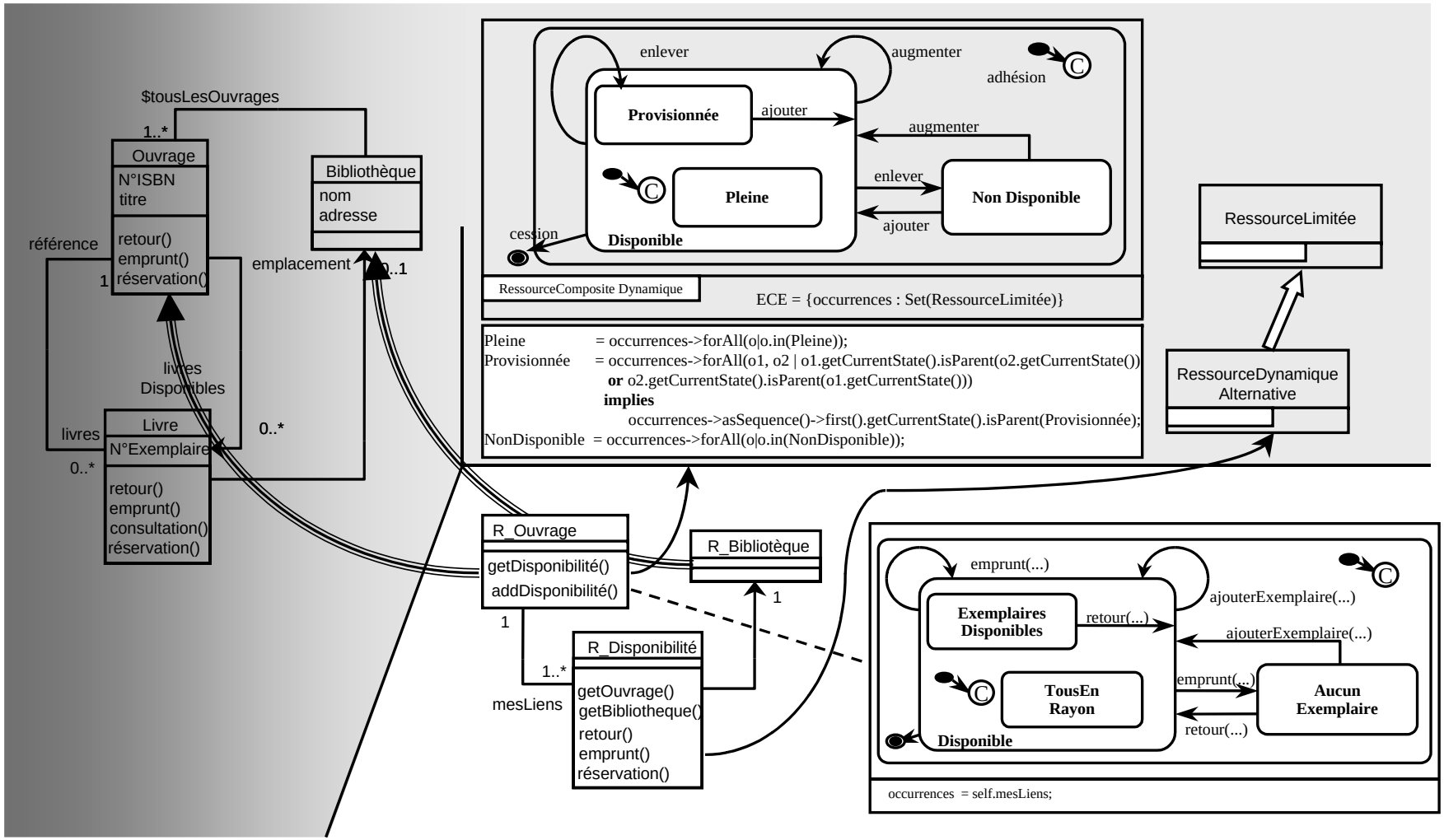


figure 2-23 : Utilisation du patron « Comportement distribué »

A la réception du message d'emprunt (1), une instance de R_ouvrage vérifie qu'il existe des transitions franchissables pour elle (3) et pour la relation concernée par le message (4). Elle exécute ensuite le service correspondant (5). Le contrôle est finalement réalisé pour la relation r (6) et pour l'instance (7), le comportement de cette dernière étant calculé à partir du nouvel état pris par la relation r.

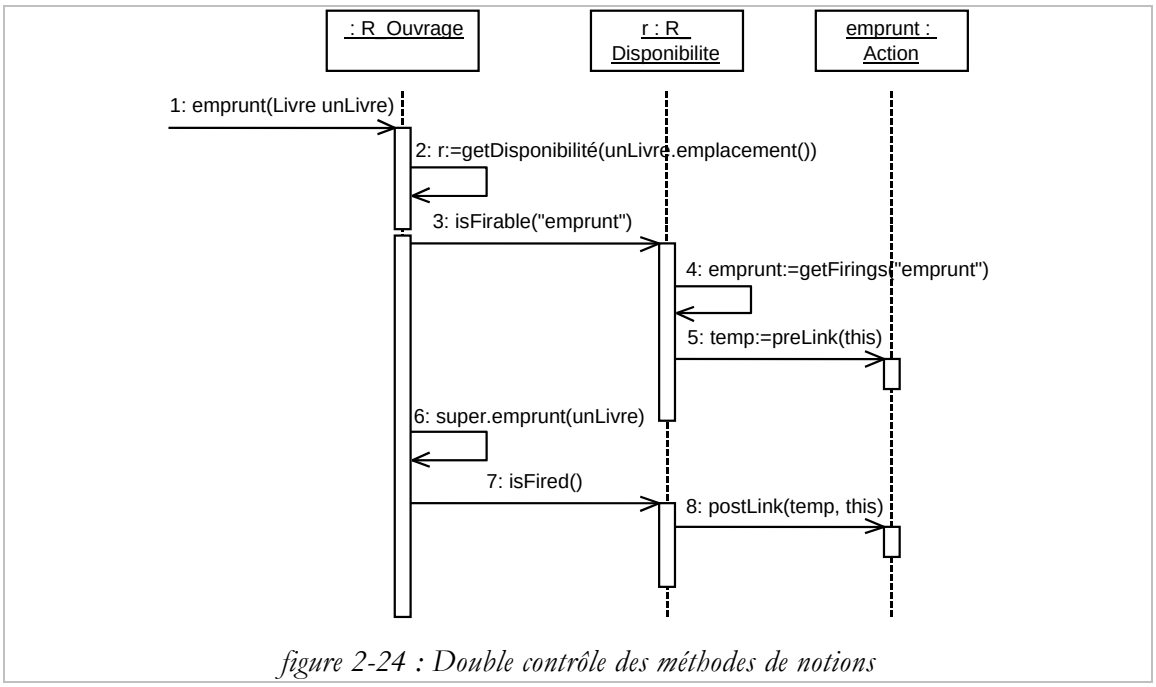


figure 2-24 : Double contrôle des méthodes de notions

Implantation

La définition des états du comportement `RessourceCompositeDynamique` de la figure 2-23 est le résultat des héritages répétés des comportements `RessourceMonoExemplaire, RessourceComposite et RessourceCompositeLimitee`

```
public class RessourceComposite extends RessourceMonoExemplaire{

…

//Déclaration de la caractéristique d'évolution occurrences
    public static CE occurences;

static{
            try{
              // instanciation de la CE occurrences de type Vector
                 occurences = new CE("occurences", "java.Util.Vector");

}
…

// Définition fonctionnelle des états

     public static State isAvailable(Base_I obj){
                Vector v = (Vector)obj.getValue(0);
                Enumeration e = v.elements();
                while (e.hasMoreElements())

     if(((Phenomenon_I)e.nextElement()).getCurrentState().isParent(RessourceMultiExemplaires.i
     sAvailable))
                     return isAvailable;
```

```
                return null;
        }

        public static State unAvailable(Base_I obj){
                Vector v = (Vector)obj.getValue(0);
                Enumeration e = v.elements();
                while (e.hasMoreElements())

    if(!((Phenomenon_I)e.nextElement()).getCurrentState().isParent(RessourceMultiExemplaires.
    unAvailable))
                        return null;
                return unAvailable;
        }
}

public class RessourceCompositeLimitee extends RessourceComposite{

…
// Définition fonctionnelle des états
    public static State isFull(Base_I obj){
            Vector v = (Vector)obj.getValue(0);
            Enumeration e = v.elements();
            while (e.hasMoreElements())

if(!((Phenomenon_I)e.nextElement()).getCurrentState().isParent(RessourceLimitee.isFull))
                  return null;
            return isFull;
    }

    public static State isFree(Base_I obj){
            Vector v = (Vector)obj.getValue(0);
            Phenomenon_I ph1, ph2;
            Enumeration e = v.elements();
            if(e.hasMoreElements()){
                ph1 = (Phenomenon_I)e.nextElement();
                while (e.hasMoreElements()){
                    ph2 = (Phenomenon_I)e.nextElement();
                    if(!(ph2.getCurrentState().isParent(ph1.getCurrentState()) ||
                                ph1.getCurrentState().isParent(ph2.getCurrentState())))
                      return isFree;
                }
                if (ph1.getCurrentState().isParent(RessourceLimitee.isFree))return isFree;
            }
            return null;
    }
}

public class RessourceCompositeDynamique extends RessourceCompositeLimitee {
…
}
```

Dans la classe R_Ouvrage qui concrétise RessourceCompositeDynamique, les méthodes de la notion Ouvrage sont redéfinies pour prendre en compte le contrôle de la figure 2-24.

```
public class R_Ouvrage extends Ouvrage implements Phenomenon_I{
// formal behaviour

    public void emprunt(Livre unLivre){
        Vector temp;
// obtenir la relation concernée par l'événement
        R_Disponibilite r = getDisponibilite(unLivre.getEmplacement());
// obtenir l'ensemble des transitions franchissables
          temp = emprunt.preLink(this);
// tester si la méthode peut être exécutée par rapport à l'état courant de la relation
          r.isFirable("emprunt");

// réaliser l'action
          super.emprunt(unLivre);

// contrôler le bon franchissement de la transition pour la relation
        r.isFired();
// contrôler le bon franchissement de la transition
        emprunt.postLink(temp, this);
    }
```

Dédicace

Aux membres du jury qui ne se sont pas encore perdus dans les méandres de ce mémoire.

2.5. Comportements réutilisables

Nom

Statecharts réutilisables

Intention

Quelles doivent être les caractéristiques des statecharts pour construire des comportements réutilisables ?

Motivation

Nous avons identifié trois points essentiels qui augmentent sensiblement la réutilisation des statecharts. Deux d'entre eux (cf. points 1 et 2) ont déjà été abordé dans le mémoire et nous ne faisons que les rappeler brièvement. Le troisième point est actuellement une piste de recherche dont nous jetons les bases.

1- Un modèle déclaratif

Le concept d'action(cf. chapitre 1 § 1.3.1 {Transition}) n'existe pas dans la dimension comportementale. En effet, pour être réutilisé un comportement doit gagner en déclarativité par rapport à un statechart. Or les actions décrivent comment sont réalisées les transitions de manière procédurale (cf. chapitre 3 § 2.4.2.1 {Transition}). Dans notre modèle, ce n'est que dans le rôle qu'on associe une transition (spécification) à une méthode (réalisation) pour former des actions.

2- Un modèle non événementiel

Un comportement s'intéresse moins aux séquences d'événements qu'aux propriétés opérationnelles de ces séquences. Le concept d'événement qui est d'habitude un concept paramétré qui véhicule l'information nécessaire à la réalisation d'une transition est absent du modèle comportemental. Les événements sont le reflet de choix de communication qui peuvent varier d'un objet à l'autre. Il y a par exemple plusieurs manières de spécifier le service d'ajout pour une ressource : ajouter(o : Objet), ajouter(e : Set(Objet)), ajouter(nbElements : Entier), etc. Or ces services correspondent tous au franchissement d'une même transition. Les événements et leurs paramètres sont déterminés pour chaque rôle lors de l'intégration des transitions du comportement concrétisé et des méthodes de la notion animée (cf. chapitre 3 § 2.5.2.2)).

2- Caractérisation des états

Nous avons distingué deux manières de spécifier et d'utiliser les statecharts (cf. activiste vs. Classificatoire). Chacune d'entre elle peut aboutir à des statecharts qui sont en apparence équivalents. Nous allons montrer que ce n'est pas le cas en ce qui concerne leur potentiel de réutilisation. Ce constat motivera la solution proposée, solution qui n'est autre que le modèle comportemental NCR.

Nous reprenons le comportement d'une ressource multi-exemplaires sous la forme de deux statecharts. Dans le premier (figure 2-25), les états ne sont pas caractérisés et chaque branchement est exprimé à l'aide de pré-conditions.

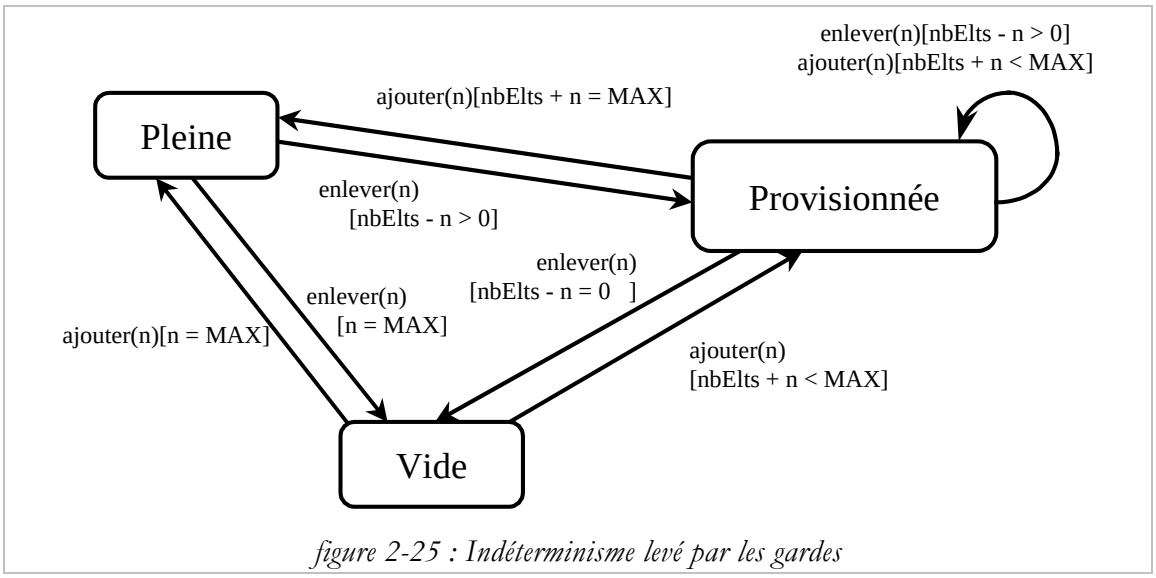


figure 2-25 : Indéterminisme levé par les gardes

Dans le second (figure 2-26), les invariants d'états sont exprimés, l'indéterminisme est donc levé par les états d'arrivée.

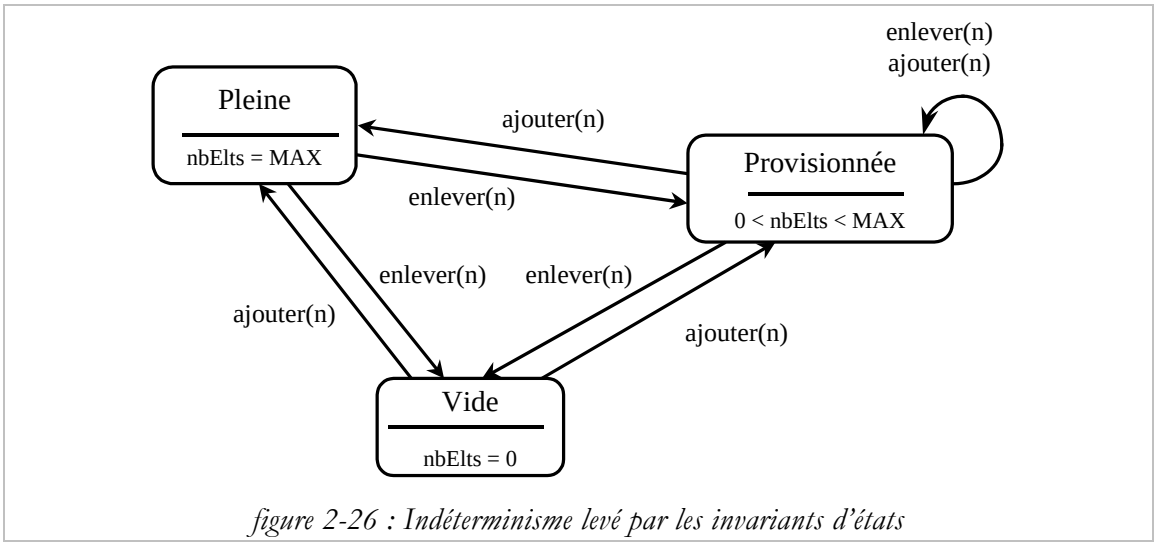


figure 2-26 : Indéterminisme levé par les invariants d'états

Un premier constat s'impose. Le second est visuellement moins chargé et offre donc une meilleure visibilité. Mais comparons ces deux statecharts sur des critères plus « scientifiques » : le second statechart est intéressant car il factorise au niveau des états des propriétés de l'objet qui ne sont pas données explicitement dans le premier (comme le fait d'être vide est équivalent à nbElts = 0). Ces propriétés ne sont pas répétées pour chaque transition. Cependant le premier statechart réalise le contrôle avant le traitement (ici l'ajout où le retrait) alors que le second vérifie les propriétés après franchissement de la transition. Or on sait qu'il vaut mieux prévenir que guérir et l'informatique n'échappe pas à cette règle.

Pour l'animation des statecharts, la première forme semble plus intéressante. Mais qu'en est-il de notre critère de réutilisation ?

Pour spécifier le statechart de la figure 2-25, nous sommes partis d'une spécification minimale des fonctions d'ajout et de retrait.

ajouter(n : Entier)

Post : nbElts = nbElts@pre + n

enlever(n : Entier)

Post : nbElts = nbElts@pre - n

Si maintenant, nous voulons modifier le protocole d'ajout et définir une fonction qui à partir d'un nombre donné de ressources prend ce qu'il faut pour atteindre l'état Pleine (ce qui est un moyen sûr d'être conforme au statechart), alors certaines gardes du statechart deviennent obsolètes et doivent être réécrits. Ce qui n'est évidemment pas le cas du statechart de la figure 2-26.

ajouter(n : Entier)

Post : (n ≥ MAX-nbElts@pre **implies** nbElts = MAX) **or** nbElts = nbElts@pre + n

Nous préconisons de ne pas utiliser de pré-condition lorsque celle-ci est liée au traitement réalisé lors du franchissement de la transition et calculée à partir de l'invariant de l'état d'arrivée. De plus, dans le cas présent les deux statecharts peuvent être déduits assez facilement l'un de l'autre et ce grâce à la double spécification (assertions de méthodes et invariants d'états). La transposition automatique d'une écriture à l'autre est un des objectifs du modèle NCR, elle consisterait à passer d'un comportement tel que celui de la figure 2-26 à un rôle dont le comportement réalisé serait similaire à celui de la figure 2-25.

La figure 2-27 et la figure 2-28 présentent des exemples de passage partiel qui pourraient facilement être obtenus à l'aide d'un moteur d'inférence. A partir des spécifications conjointes de la méthode ajouter de la notion Ouvrage et de la transition ajouter extraite du statechart de la

figure 2-26, nous pouvons déduire deux gardes qui conditionnent l'exécution de l'action ajouter (figure 2-27 et figure 2-28).

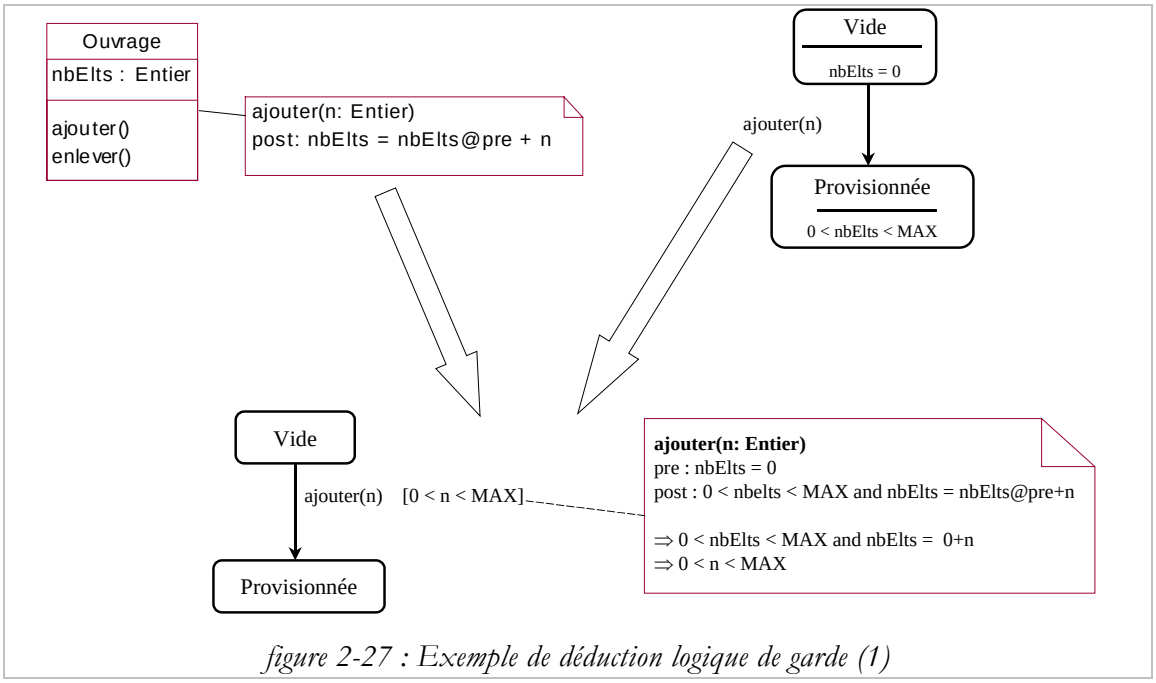


figure 2-27 : Exemple de déduction logique de garde (1)

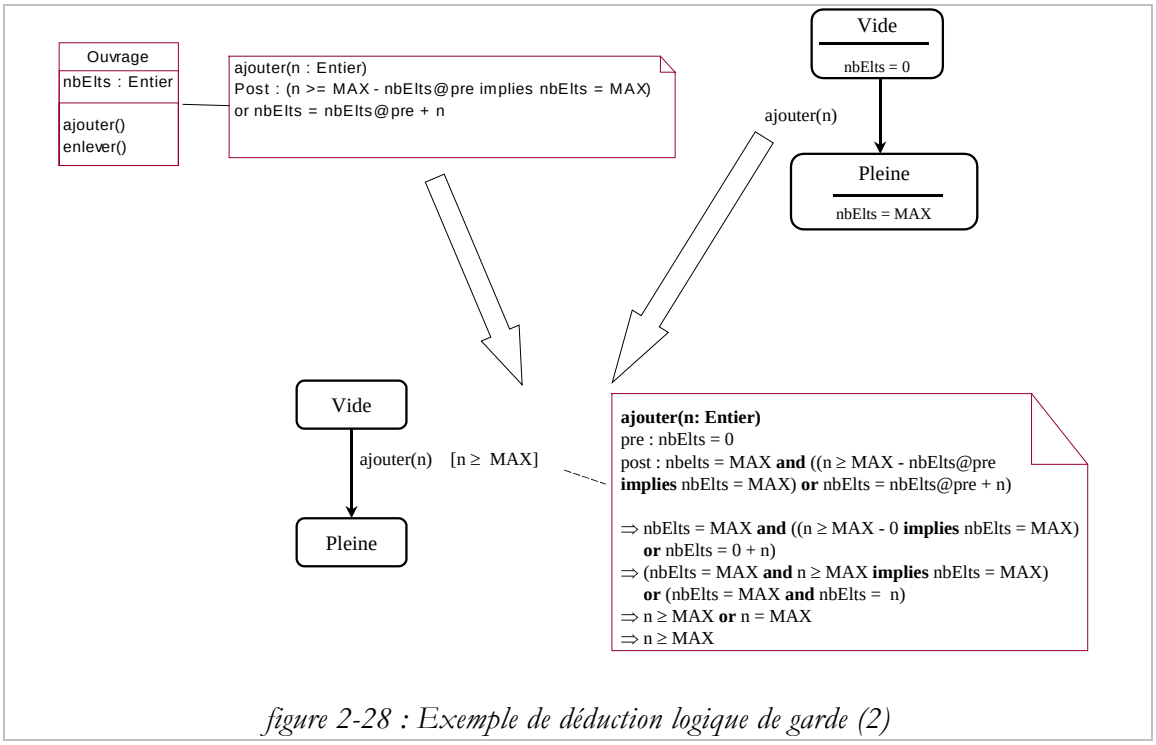


figure 2-28 : Exemple de déduction logique de garde (2)

Cet exemple montre comment on peut vraiment tirer parti des spécifications conjointes du modèle structurel et du modèle dynamique.

Description

Les trois étapes décrites dans la motivation permettent d'épurer le modèle dynamique des statecharts afin de le rendre réutilisable. Elles ont donné lieu aux choix sémantiques réalisés dans les comportements NCR (figure 2-29). L'intégration réalisée ensuite dans les rôles permet de reconstituer un formalisme graphique dont la sémantique est proche de celle des statecharts (l'évolution d'un rôle est représentée à l'aide d'un sous-ensemble du formalisme des statecharts). Sur la figure 2-29, la demi-boucle de droite présente le travail que nous avons réalisé sur le formalisme des statecharts pour obtenir nos comportements ; la demi-boucle de gauche présente le travail qui est réalisé pour obtenir la visualisation d'un rôle à partir de l'intégration d'un comportement et d'une notion.

La dernière étape (figure 2-29 – étape 6) ne peut se faire qu'à un coût de spécification élevé. En effet, si nous ne disposons pas de spécifications structurelles, il est impossible d'inférer les gardes. Elle montre finalement qu'il est nécessaire de bien faire la part des choses **entre la spécification minimale (nous pourrions dire « intrinsèque ») d'une méthode de notion et la spécification induite par le contrôle qui est réalisé dans un rôle**. La première a bien sûr un plus fort potentiel de réutilisation.

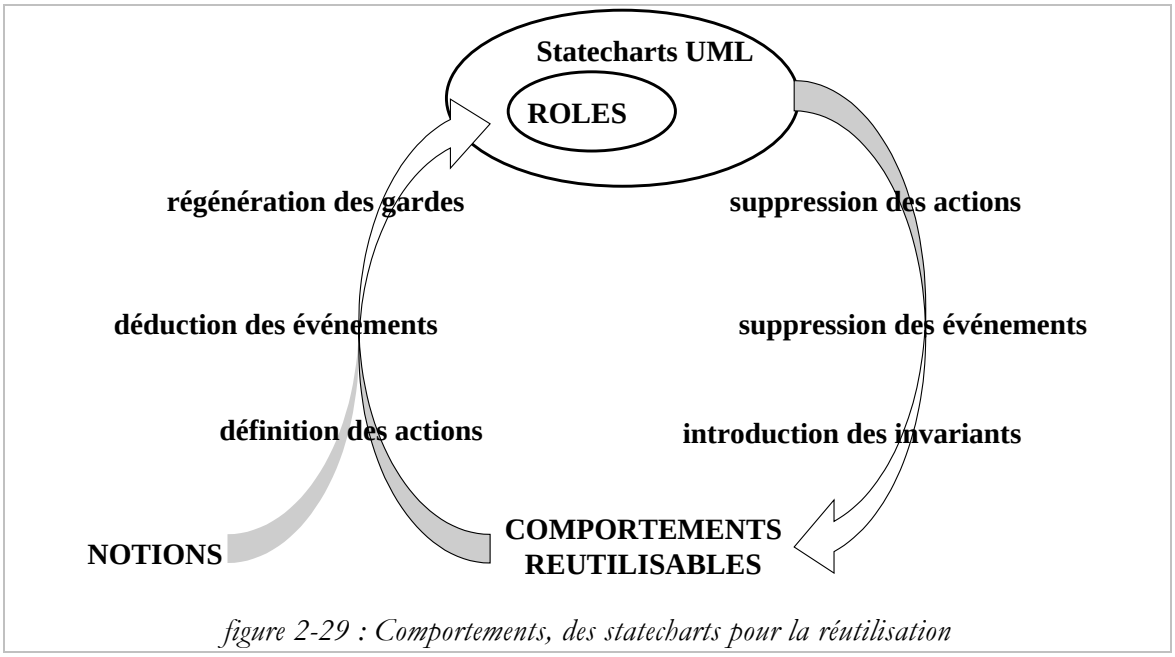


figure 2-29 : Comportements, des statecharts pour la réutilisation

Implantation

Nous donnons dans ce patron l'implantation d'un comportement NCR. Un comportement est une définition abstraite utilisable par tous les rôles. Elle est réalisée par une classe en java. Les

propriétés de cette classe (CE, états et transitions) sont déclarées statiques en java. Cela permet d'avoir une définition unique du comportement dans le système. Le gain en mémoire est important (à comparer de toutes les classes qui sont instanciées dans les solutions utilisant des classes-états). Nous avons vu dans le mémoire que les comportements étaient structurés en domaines hiérarchiques. Nous utilisons l'héritage java classique pour réaliser la décomposition de comportements. Nous proposons ci-après une implantation des comportements de la figure 2-30 :

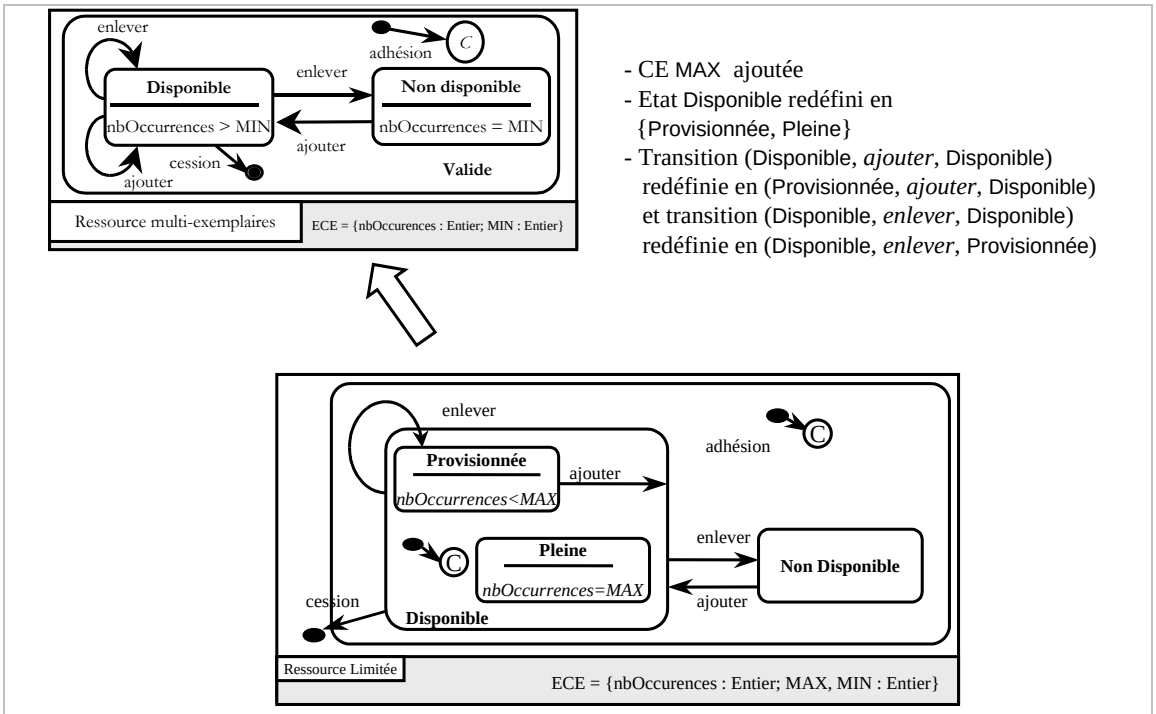


figure 2-30 : Décomposition de comportements

```
package Ncr.Comportment.Resource;

import java.lang.Boolean;
import java.lang.reflect.*;
import Ncr.Kernel.*;

public class RessourceMultiExemplaires extends RessourceMonoExemplaire{

    // Permet de référencer la classe alors qu'il n'existe pas d'instance de cette classe
    public static Class myClass;

    // Déclaration des caractéristiques d'évolution
    public static CE nbOccurences;
    public static CE MIN;

    // Pas de nouvel état par rapport à RessourceMonoExemplaire

    // Déclaration des nouvelles transitions
    public static  Transition take_2;
    public static  Transition take_3;
    public static  Transition let_2;

    // Instanciation des propriétés du comportement
    static{
            try{
                 myClass = Class.forName("Ncr.Comportment.Resource.RessourceMultiExemplaires");

                 nbOccurences = new CE("nbOccurences", "java.lang.Integer");
                 MIN = new CE("MIN", "java.lang.Integer");

                 take_2 = new Transition(null, isAvailable, isAvailable, "take");
```

```
                take_3 = new Transition(null, unAvailable, unAvailable, "take");
                let_2 = new Transition(null, isAvailable, isAvailable, "let");
            }
            catch(Exception e){System.out.println("ERROR 1: comportment "  + myClass.getName()
+ " creation");}
    }


// Définition des invariants d'états sous forme fonctionnelle

    public static State isAvailable(Base_I obj){
           if (((Integer)obj.getValue(0)).intValue() > ((Integer)obj.getValue(1)).intValue())
                return isAvailable;
            else return null;
    }

    public static State unAvailable(Base_I obj){
           if (((Integer)obj.getValue(0)).equals((Integer)obj.getValue(1))){
                return unAvailable;
            }
            else return null;
    }

}
```

et

```
package Ncr.Comportment.Resource;

import java.lang.Boolean;
import java.lang.reflect.*;
import Ncr.Kernel.*;


public class RessourceLimitee extends RessourceMultiExemplaires{

public static Class myClass;

       // CE MAX ajoutée
    public static CE MAX;

       // Déclaration des transitions
    public static  Transition let_2;
    public static  Transition take_2;

       // Déclaration des sous-états {isFree, isFull}
    // public static State isAvailable;
    public static State isFree;
    public static State isFull;


    static{
            try{
                myClass = Class.forName("Ncr.Comportment.Resource.RessourceLimitee");

                MAX = new CE("MAX", "java.lang.Integer");

                isFree = new State(isAvailable, "isFree");
                isFull = new State(isAvailable, "isFull");
// Transition (isAvailable, take_2, isAvailable) redéfinie en (isAvailable, take_2, isFree).
                take_2 = new Transition(RessourceMultiExemplaires.take_2, isAvailable, isFree,
"take");
// Transition (isAvailable, let_2, isAvailable) redéfinie en (isFree, take_2, isAvailable).

                let_2 = new Transition(RessourceMultiExemplaires.let_2, isFree, isAvailable,
"let");
            }
            catch(Exception e){System.out.println("ERROR 1: comportment "  + myClass.getName()
+ " creation");}
    }

// Définition et redéfinition des invariants d'états sous forme fonctionnelle

    public static State isFree(Base_I obj){
           if (((Integer)obj.getValue(0)).intValue() < ((Integer)obj.getValue(2)).intValue()
&& (RessourceMultiExemplaires.isAvailable(obj) != null))
              return isFree;
           else return null;
    }

    public static State isFull(Base_I obj){
```

```
            if (((Integer)obj.getValue(0)).intValue() == ((Integer)obj.getValue(2)).intValue()
&& (RessourceMultiExemplaires.isAvailable(obj) != null))
              return isFull;
            else return null;
    }

    //redefinition de l'état isAvailable : isAvailable => (isFree or isFull)
    public static State isAvailable(Base_I obj){
      try{
           Object args[] = new Object[1];
           args[0] = obj;
           Method isFree = Base.getMethod(obj.getBehaviour(), "isFree");
           State s = (State)isFree.invoke(obj.getBehaviour(), args);
           if (s != null) return s;
           Method isFull = Base.getMethod(obj.getBehaviour(), "isFull");
           return (State)isFull.invoke(obj.getBehaviour(), args);
    // le sens des tests a de l'importance puisqu'on est plus souvent dans isAvailable
    }
    catch(Exception e){
            System.out.println("ERROR 37 : state redefinition corrupted");
            return null;
    }

    }

}
```

Dédicace

A une amie mathématicienne.

CONCLUSION

«Tu m'as donné conscience de tant de sens,
que je me sens con de peu de science.»

Etat des lieux

Nous avons présenté dans ce mémoire « l'épopée » NCR. Celle-ci nous a mené au modèle NCR tel qu'il est spécifié dans le chapitre 3. Ce dernier est le résultat d'une vaste étude du domaine dont nous avons présenté les points clés dans les chapitre 1 et 2.

Le problème s'est tout d'abord posé en termes de modèles à utiliser pour spécifier le comportement individuel des objets du système d'information. Nous nous sommes tout de suite orientés vers le formalisme des statecharts que nous avons décrit de manière exhaustive dans le chapitre 1. Ce formalisme nous offrait potentiellement la puissance d'expression nécessaire pour spécifier graphiquement des comportements complexes tout en conservant un bon rapport complexité de l'objet modélisé/complexité de la spécification.

Malgré cela, notre étude a fait ressortir une sous-utilisation des statecharts dans les méthodes de conception actuelles. Nous avons constaté que ceci était lié plus à la manière de les coupler avec les modèles à objets qu'au formalisme lui-même. Dans le chapitre 1, nous avons identifié deux approches complémentaires qui, et c'est ce que nous avons essayé de montrer, conditionnent grandement leur utilisation dans les méthodes de conception actuelles.

Il était dès lors nécessaire de recadrer notre travail par rapport au domaine général des méthodes de conception à objets. Nous nous sommes tout particulièrement intéressés aux méthodes « dirigées par le comportement » qui connaissent aujourd'hui un franc succès. Nous avons montré dans le deuxième chapitre que le concept de rôle était aujourd'hui fédérateur pour de nombreuses méthodes dont les préoccupations sont la réutilisation et la traçabilité des spécifications.

En utilisant le savoir-faire des patrons, nous avons ensuite comparé la place tenue par les états et les rôles dans les modèles à objets, ceci pour toutes les phases de développement. C'est à partir de ce travail préparatoire que nous avons défini les fondations du modèle NCR.

Dans le chapitre 3, nous avons montré que la réutilisation comportementale n'est pas évidente, même sur un problème simple. Nous avons retenu quatre solutions significatives à ce problème. L'étude de leurs forces et de leurs faiblesses respectives nous a permis de définir le « cahier des charges » du modèle NCR, un modèle pour l'utilisation et la réutilisation de comportements d'objets.

Le modèle NCR répond aujourd'hui à nos attentes en termes de qualité des spécifications fournies. Il autorise :

- L'intégration des diagrammes de classe et des diagrammes d'états. La réification de ces deux modèles à l'aide d'une classification commune a permis l'identification des propriétés génériques aux deux types de diagrammes. Les correspondances entre propriétés de paradigmes différents ont pu être établies grâce aux connecteurs des rôles.

- L'assurance de la traçabilité des spécifications. NCR offre deux vues indépendantes de modélisation qui sont corrélées dans un second temps. L'effort de conceptualisation des correspondances entre nos paradigmes rend l'implantation à la fois plus conforme aux spécifications et plus flexible, il n'y a pas une unique façon de concrétiser nos comportements. De plus, l'implantation des spécifications statiques n'est pas " polluée " par des contrôles dynamiques, d'où une meilleure lisibilité du code.

- La réutilisation possible du comportement. Utilisant pleinement la relation d'héritage, le modèle NCR favorise la réutilisation conjointe des spécifications structurelles et comportementales. Bien que notre modèle ne supprime pas la difficulté de spécifier des comportements, la possibilité de réutiliser ces derniers doit encourager les concepteurs à les intégrer dans le processus de développement.

Perspectives

Trois axes doivent être développés pour améliorer le modèle NCR. Il s'agit d'offrir une démarche convaincante, un modèle à même de prendre en compte des applications plus complexes et un outil qui supporte complètement la démarche.

Démarche

Pour l'instant, notre bibliothèque comportementale est relativement pauvre. Bien qu'elle présente des comportements complexes (jusqu'à une complexité de 3), il nous semble aujourd'hui indispensable de valider le modèle sur des comportements que nous pourrions qualifier de

« métiers », ceci afin d'étudier la portée de notre approche et son gain effectif en termes de réutilisation.

De plus, une utilisation à des fins non « académiques » permettrait de préciser une démarche type supportée par le modèle NCR. En effet, parallèlement à la spécification du modèle, nous avons travaillé sur la formalisation des processus de conception NCR. Les patrons présentés dans le chapitre 4 sont le résultat de ces travaux. Ils donnent quelques pistes intéressantes sur les processus et sur une possible organisation autour des trois dimensions NCR. Le processus que nous envisageons n'est pas sans rappeler celui des approches sur les patrons, la démarche traditionnelle de développement étant remplacée par deux types de processus complémentaires [Moore89] : un processus **pour** la réutilisation dont l'objectif est d'identifier, spécifier et développer des domaines structurels et comportementaux réutilisables et un processus **par** la réutilisation dont l'objectif est de mettre en correspondance ces domaines. La spécification de ces deux processus pour une meilleure réutilisation du comportement est un des objectifs à court terme.

Formalisation

Plusieurs aspects semblent intéressants à formaliser :

1- La synchronisation est exprimée uniquement sur les états. L'idée d'introduire un contrôleur spécifique pour les comportements complexes semble intéressante. Il permettrait d'exprimer des séquencements obligatoires, des exclusions, etc, de facteurs que nous ne conceptualisons pas actuellement. Deux idées de formalisme nous paraissent porteurs, soit rester dans ce que nous savons faire et utiliser un statechart de statecharts, soit utiliser les opérateurs de Allen [Allen95] pour exprimer un contrôle encore plus fin.

2- La régénération des gardes (chapitre 4 § 2.4) n'a pas encore fait l'objet de recherches approfondies de notre part. Il s'agit là d'une piste intéressante mais qui demande un outillage formel plus conséquent. De manière générale, nous ne garantissons pas pour l'instant de contrôles statiques, cette thèse ayant mis en avant l'animation et le contrôle à l'exécution plutôt que la vérification statique des intégrations réalisées.

3- Nous avons privilégié un héritage comportemental multiple sans collisions. L'étude de ces collisions est un problème intéressant qui a été soulevé dans cette thèse et que nous n'avons pas solutionnée pour l'instant. Une des difficultés liées à l'implantation fut d'ailleurs la simulation des héritages multiples à l'aide du langage java.

4- Le méta-modèle pourrait être étendu pour prendre en compte explicitement les relations entre objets. Cela aurait pour avantage d'augmenter le pouvoir d'expression du modèle, notamment en ce qui concerne l'utilisation de méthodes des objets reliés pour réaliser des transitions (idée qui est présente dans la norme). Et plus encore, nous pourrions expliciter les relations que

nous avons identifié dans chaque dimension, relations entre comportements (cf. chapitre 4 § 1.2) et relations entre rôles (cf. chapitre 4 § 2.5).

5- L'idée d'intégrer deux modèles de communication et par la même deux modes de raisonnement fut une problématique des plus intéressante. Cependant, celle-ci devra être étendue pour prendre en compte des architectures plus complexes qui utilisent par exemple les composants java (beans), des processus multiples (threads), etc.

Outillage

Nous travaillons à ce jour sur l'implantation complète de l'atelier NCR en java. En effet, de nombreux points restent à développer :

- Le noyau a été implanté de manière à faciliter la génération automatique de code. Nous en donnons des « morceaux choisis » dans le chapitre 4. Le modèle NCR est exécutable dans sa totalité depuis peu (nous pouvons dire qu'une version stable sortira d'ici la fin de l'année). Il reste de nombreux points à intégrer dont une gestion complète des exceptions qui n'a pas été intégrée au prototype actuel. C'est pourquoi, même si nous savons pertinemment qu'elle est possible, la génération automatique des comportements et des rôles n'est pas mise en œuvre dans le cadre de cette thèse.
- Le module d'animation qui est à notre sens le plus innovant dans le domaine des systèmes d'information (il existe déjà de nombreux outils commerciaux utilisant l'animation à l'aide de statecharts pour décrire les systèmes réactifs) est en cours de développement. Un prototype devrait voir le jour d'ici la fin de l'année. Ce module permettra de simuler le comportement complexe des objets du système, un peu à la manière de la méthode M2PO [Hill93] qui utilise pour cela des réseaux de Petri. Nous pourrons ainsi quantifier précisément le gain réalisé en utilisant les concepts NCR pour la conception et le test d'applications.
- Un des objectifs qui n'a pu être tenu dans cette thèse concerne l'atelier de conception proprement dît. Une véritable discussion doit s'engager sur la spécification graphique de cet atelier. En effet, si la dimension des notions semble facile à mettre en œuvre (elle est présente dans tous les ateliers), il n'en va pas de même de la dimension comportementale. Nous avons bien avancé quelques idées en marge de la thèse pour documenter les comportements sous la forme de patrons mais le problème de la structuration et de la recherche de comportements dans des bibliothèques conséquentes reste entier. De même, la conception de rôles demande de « jongler » en permanence entre des hiérarchies de notions et de comportements. Nous pensons qu'une grande partie des intégrations réalisées dans un rôle peuvent être réalisée graphiquement sans faire appel à des spécifications textuelles.

Bibliographie

[Albano93] A. Albano and Al., AN OBJECT DATA MODEL WITH ROLES, proceedings of 19th VLDB conference, Dublin, 1993

[Alexander77] C. Alexander, and al., A PATTERN LANGUAGE, Oxford University Press, New York, NY, 1977

[Alexander79] C. Alexander, THE TIMELESS WAY OF BUILDING , Oxford University Press, New York, NY, 1979

[Allen95] Arthur Allen, EXTENDING THE STATECHART FORMALISM : EVENT SCHEDULING & DISPOSITION, Austin OOPSLA'95

[Appleton97] B. Appleton, PATTERNS AND SOFTWARE – ESSENTIAL CONCEPT AND TERMINOLOGY, http://www.enteract.com/~bradapp/docs/patterns-intro.html

[Arapis90] C. Arapis , SPECIFYING OBJECT LIFE-CYCLES , Object Management-D. Tsichritzis (Ed) Centre Univ. d'Informatique, Univ. de Geneve, Juillet 90

[Badouel93] D. Badouel, A. Khaled, LA PROGRAMMATION C ET C++, Hermès, 1993

[Baümer97] D. Bäumer and Al., THE ROLE OBJECT PATTERN, PLoP'97 Conference

[Beck87] K. Beck, W. Cunnimghan, USING PATTERN LANGUAGES FOR OBJECT-ORIENTED PROGRAMS, Proceedings of OOPSLA-87, 1987

[Beck89] K. Beck and W. Cunningham, A LABORATORY FOR TEACHING OBJECT-ORIENTED THINKING, Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89) in ACM SIGPLAN Notices 24(10), New Orleans, LA, Oct. 1989, pp. 1-6

[Beck94] K. Beck, R. Johnson, PATTERNS GENERATE ARCHITECTURES, In M. Tokoro and R. Pareschi (Eds.), Object-Oriented Programming, LNCS 821, Berlin :Springer-Verlag, 1994

[Beck95] K. Beck, SMALLTALK BEST PRACTICE PATTERNS VOLUME 1 : CODING (draft review copy), First Class Software Inc., 1995

[Beeck94] M. von der Beeck, A COMPARISON OF STATECHARTS VARIANTS, Proc. of Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT' 94) LNCC 863 -

[Berry88] G. Berry, G. Gonthier, THE SYNCHRONOUS PROGRAMMING LANGUAGE ESTEREL, DESIGN, SEMANTICS, IMPLEMENTATION, Technical Report 327, INRIA, 1988

[Booch91] G. Booch, OBJECT ORIENTED DESIGN WITH APPLICATIONS, The Benjamin / Cummings Publishing Company, Inc, 1991

[Booch98] G. Booch, J. Rumbaugh and I. Jacobson, THE UNIFIED LANGUAGE USER GUIDE, Addison-Wesley Editions, 1998

[Bouzeghoub95] M. Bouzeghoub, G. Gardarin, P. Valduriez, DU C++ À MERISE OBJET, Editions Eyrolles, 1995

[Brunet98] J. Brunet, AN ENHANCED DEFINITION OF COMPOSITION AND ITS USE FOR ABSTRACTION, The 5th international conference on Object Oriented Information Systems, Paris, September 1998

[Cargill92] T. Cargill, C++ PROGRAMMING STYLE, Addison-Wesley, Reading, MA, 1992

[Chen76] P.P.S Chen, THE ENTITY-RELATIONSHIP MODEL, ACM Transactions on Database Systems, 1976

[Coad90] P. Coad, E. Yourdon, OBJECT ORIENTED ANALYSIS, Yourdon Press, Prentice-Hall, 1990

[Coad92] P. Coad, OBJECT-ORIENTED PATTERNS, Communications of the ACM, Vol. 35 - n°9, September 1992

[Coleman92] Coleman, D., F. Hayes and S. Bear, INTRODUCING OBJECTCHARTS, OR HOW TO USE STATECHARTS IN OBJECT ORIENTED DESIGN, IEEE Trans. Soft. Eng. 18 (1992) pp. 9-18

[Collongues86] A. Collongues, J. Hugues, B.Laroche, MERISE 1. MÉTHODE DE CONCEPTION, Dunod Informatique, 1986

[Cook94] Cook, S. And J. Daniels, DESIGNING OBJECT SYSTEMS : OBJECT-ORIENTED MODELLING WITH SYNTROPY, Prentice Hall New York, 1994

[Coplien96] J. Coplien, SOFTWARE DESIGN PATTERNS : COMMON QUESTIONS AND ANSWERS, ftp://st.cs.uiuc.edu/pub/patterns/papers/PatQandA.ps

[CSO99] D. Rieu and all., CONCEPTION DE SYSTÈMES À OBJETS, Support de cours, IUT d'informatique de Grenoble, année 1999

[Dami98] S. Dami, J. Estublier, M. Amiour, APEL: A GRAPHICAL YET EXECUTABLE FORMALISM FOR PROCESS MODELING, Kluwer Academic Publishers, Boston. Process Technology. Edited by E. Di Nitto and A. Fuggeta. January 1998

[Dano97] B. Dano, UNE DÉMARCHE D'INGÉNIERIE DES BESOINS ORIENTÉE OBJET GUIDÉE PAR LES CAS D'UTILISATION, Thèse de l'université de Nantes, novembre 1997

[Desfray94] P. Desfray, OBJECT ENGINEERING, THE FOURTH DIMENSION, Editions Addison-Wesley, 1994

[Dekker92] L. Dekker, B.Carre, MULTIPLE AND DYNAMIC REPRESENTATION OF FRAMES WITH POINTS OF VIEW IN FROME , Représentation Par Objets, La Grande Motte, 1992

[DRG99] Database Research Group, Noyau d'un Système d'Information Générique, document interne CUI, Université de Genève, 1999

[D'Souza92] D. D'Souza, TEACHER! TEACHER!, Jnl. of Object-Oriented Programming, 1992

[D'Souza95] D. D'Souza, BEHAVIOR VS. DATA-DRIVEN, A NON-ISSUE?, http://www.iconcomp.com/papers/data-vs-behavior/index.htm

[D'Souza96] D'Souza, PROJECTION : SUBCLASSES AND STATES, http://www.iconcomp.com/papers/projection/Projection.frm.html

[D'Souza98] D. F. D'Souza, A. C. Wills, OBJECTS, COMPONENTS, AND FRAMEWORKS WITH UML - THE CATALYSIS APPROACH, Editions Addison-Wesley, 1998

[Duffy95] D. Duffy, FROM CHAOS TO CLASSES : OBJECT-ORIENTED SOFTWARE DEVELOPMENT IN C++, London ; New York : McGraw-Hill, pp. 145-161, 1995

[Dupuy98] S. Dupuy, Y. Ledru, M. Chabre-Peccoud, TRANSLATING THE OMT DYNAMIC MODEL INTO OBJECT-Z, 11th International Conference of Z Users (ZUM'98), LNCS 1493, Springer, 1998

[Dyson96] P. Dyson, B. Anderson, STATE PATTERNS, at EUROPLOP'96, Kloster Irsee, Germany, 1996

[Ebert94] J. Ebert, G. Engels, OBSERVABLE OR INVOCABLE BEHAVIOUR - YOU HAVE TO CHOOSE, Technical report TR94-38, University of Leiden, Netherlands, 1994

[Embley92] D.W. Embley and all., OBJECT-ORIENTED SYSTEMS ANALYSIS : A MODEL-DRIVEN APPROACH, Yourdon Press, 1992

[Englander97] R. Englander, JAVA BEANS, GUIDE DU PROGRAMMEUR, Editions O'Reilly, 1997

[Escamilla93] J. Escamilla, SHOOD : UN MODÈLE MÉTA-CIRCULAIRE DE REPRÉSENTATION DES CONNAISSANCES, Doctorat de l'INPG, Grenoble, septembre 1993

[Estublier95] J. Estublier, S. Dami, APEL V3 SPECIFICATIONS, Esprit deliverable, PERFECT project, LSR/IMAG, Grenoble, France, December 1995

[Fowler97a] M. Fowler, ANALYSIS PATTERNS, REUSABLE OBJECT MODELS, Addison-Wesley, 1997

[Fowler97b] M. Fowler, ROLE PATTERNS, PLoP'97 Conference

[Gamma95] E. Gamma and all., DESIGN PATTERNS, Addison-Wesley Professional Computing Series, 1995

[Front-Conte99] A. Front-Conte, SCALP : UN LANGAGE DE PATRONS POUR EXPRIMER LES ASPECTS RÉACTIFS DES SYSTÈMES D'INFORMATION, Numéro spécial de la revue L'OBJET : Patrons Orientés Objet, 1999

[GeO99] Ouvrage collectif, GENIE OBJET, ANALYSE ET CONCEPTION DE L'EVOLUTION, InterEditions, à paraître en mai 1999

[Goldberg84] A. Goldberg, SMALLTALK-80, THE INTERACTIVE PROGRAMMING ENVIRONMENT, Addison-Wesley, 1984

[Gunter94] G. Gunter and all, STRATEGIC DIRECTIONS IN SOFTWARE ENGINEERING AND PROGRAMMING LANGUAGES, ACM Computing Surveys, volume 28, n°4, pp. 727-737, décembre 1996

[Habrias93] H. Habrias, INTRODUCTION À LA SPÉCIFICATION, Editions Masson, 1993

[Harel87] David Harel, STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS, Science of Computer Programming 8 231-274 - 1987

[Harel88] David Harel, ON VISUAL FORMALISMS, Communictions of ACM - 31:5 - 514-530 - 1988

[Harel90] David Harel and all, STATEMATE : A WORKING ENVIRONMENT FOR THE DEVELOPMENT OF COMPLEX REACTIVE SYSTEMS, IEEE Transactions on Software Engineering 16 - 1990

[Harel96a] D. Harel, E. Gery, EXECUTABLE OBJECT MODELING WITH STATECHARTS, Proceedings 18 th Int. Conf. Soft. Eng. IEEE Press 1996 pp.246-257

[Harel96b] David Harel, A. Naamad, THE STATEMATE SEMANTICS OF STATECHARTS, ACM TOSEM 5(4) 293-333 - 1996

[Hartmann93] T. Hartmann, G. Saake, ABSTRACT SPECIFICATION OF OBJECT INTERACTION, Informatik-Bericht 93-08, Technishe Universität Braunschweig, 1993

[Hill93] R.C. Hill, ANALYSE ORIENTÉE OBJETS & MODÉLISATION PAR SIMULATION, éditions Addison-Wesley, 1993

[Hopcroft79] J.E. Hopcroft, J.D. Ullman, INTRODUCTION TO AUTOMATA THEORY, languages and computation. Addison-Wesley Reading 1979

[Huizing89] C. Huizing, W.P. de Roever, EVERYTHING YOU ALWAYS WANTED TO KNOW ABOUT STATECHARTS BUT WERE AFRAID TO ASK, 1989

[Huizing92] C. Huizing, R. Gerth, SEMANTICS OF REACTIVE SYSTEMS IN ABSTRACT TIME, LNCS, vol. 600, Springer, pp. 291-314, 1992

[Icon97a] ICON Computing corporation, WORKING WITH OMT - PART IV: MODEL INTEGRATION, http://www.iconcomp.com/papers/omt4-model-integr/OMT4ModelIntegration.frm.html

[Icon97b] ICON Computing corporation, PROJECTION: SUBCLASSES AND STATES, http://www.iconcomp.com/papers/projection/Projection.frm.html

[Jacobson93] I. Jacobson and all., OBJECT-ORIENTED SOFTWARE ENGINEERING: A USE CASE DRIVEN APPROACH, Wokingham GB ; Reading Mass, Addison-Wesley, 1993

[Jensen85] K. Jensen, COLOURED PETRI NETS, In Petri Nets: Applications and Relationships to Other Models of Concurrency Part I, Lecture Notes in Computer Science Vol. 254, Springer-Verlag, 1987

[Kappel91] G. Kappel, M. Schrefl, OBJECT/BEHAVIOR DIAGRAMS, in proceedings of the 7th international Conference on Data Engineering, pp 530-539, 1991

[Katz93] R. H. Katz, CONTEMPORARY LOGIC DESIGN, Addison Wesley Publishing Company 1993

[Kettani98] N. Kettani and all., DE MERISE À UML, Editions Eyrolles, 1998

[Kristensen95] B. B. Kristensen, OBJECT-ORIENTED MODELLING WITH ROLES, THE SECOND INTERNATIONAL CONFERENCE ON OBJECT-ORIENTED INFORMATION SYSTEMS - OOIS'95. Dublin, 1995

[Kristensen96a] B. B. Kristensen, J. Olsson, ROLES & PATTERNS IN ANALYSIS, DESIGN AND IMPLEMENTATION, 3rd International Conference on Object-Oriented Information Systems, UK, 1996

[Kristensen96b] B. B. Kristensen, K. Osterbye, ROLES : CONCEPTUAL ABSTRACTION THEORY & PRACTICAL LANGUAGE ISSUES, Theory and Practice of Object Systems (TAPOS), 1996

[Kruchten99] P. Kruchten, THE RATIONAL UNIFIED PROCESS, AN INTRODUCTION, Editions Addison&Wesley dans la série ObjectTechnology series

[Legrand97] André Le Grand, SPÉCIALISATION DE CYCLE DE VIE D'OBJET : UNE INTRODUCTION, BDA'97 - Grenoble

[LeGrand98] A. Le Grand, SPECIALIZATION OF OBJECT LIFECYCLE, The 5th international conference on Object Oriented Information Systems, Paris, September 1998

[Léonard91] M. Leonard, T. Estier, G.Falquet, J. Guyot, SIX SPACES FOR GLOBAL INFORMATION SYSTEMS DESIGN, proc. IFIP Working Conf. on the object-oriented approach, 1991

[McGregor93] J.D. McGregor, D.M. Dyer, A NOTE ON INHERITANCE AND STATE MACHINES, ACM SIGSOFT Software Engineering Notes, Vol. 18 n°4, oct. 93

[Marino93] O. Marino, RAISONNEMENT CLASSIFICATOIRE DANS UNE REPRÉSENTATION À OBJETS MULTI-POINTS DE VUE, thèse de doctorat de l'Université Joseph Fourier-Grenoble 1, 1993

[Martin98] R. C. Martin, UML TUTORIAL : FINITE STATE MACHINES, Engineering Notebook Column, C++ Report, June 98

[Maughan94] G. Maughan, B. Durnota, MON : AN OBJECT RELATIONSHIP MODEL INCORPORATING ROLES, CLASSIFICATION, PUBLICITY AND ASSERTIONS, Proceedings of OOIS'94, International Conference on Object Oriented Information Systems, 1994

[Mens97] T. Mens, P. Stayaert, INCREMENTAL DESIGN OF LAYERED STATE DIAGRAMS, Technical Report vub-prog-tr-97-04, Vrije Universiteit Brussel, 1997

[Mens98] T. Mens, C. Lucas, P. Steyaert, SUPPORTING REUSE AND EVOLUTION OF UML MODELS, UML'98, International Workshop, Mulhouse, 1998

[Meyer87] B. Meyer, J-M. Nelson and M. Matsuo, EIFFEL: OBJECT-ORIENTED DESIGN FOR SOFTWARE ENGINEERING, in Proceedings of ESOP'87 European Software Engineering Conference, pp 237-245, AFCET, 1987

[Meyer92] B. Meyer, DESIGN BY CONTRACT, IEEE Computer, 25(10): 40-51, 1992

[Moore89] J. M. Moore, S. C. Bailin, THE KAPTUR ENVIRONMENT: AN OPERATIONS CONCEPT, Report prepared for NASA Goddard Space Flight Center, CTA Incorporated, Rockville, MD, June, 1989

[Morel96] J-M. Morel, EXPÉRIENCES DE RÉUTILISATION AVEC LA MÉTHODE REBOOT, Revue Génie Logiciel, 42:45-50, décembre 1996

[Muller97] P. A. Muller, MODÉLISATION OBJET AVEC UML, Editions Eyrolles, 1997

[OCL97] UML, OBJECT CONSTRAINT LANGUAGE SPECIFICATION , version 1.1 alpha 6

[Nanci96] D. Nanci, B. Espinasse, INGENIERIE DES SYTEMES D'INFORMATION :MERISE, DEUXIEME GENERATION, Editions Sybex, 1996

[Nerson92] J.M. Nerson, APPLYING OBJECT-ORIENTED ANALYSIS AND DESIGN, Communications of theACM, 35(9):63-74, September 1992

[Nguyen98] H. P. Nguyen, DÉRIVATION DE SPÉCIFICATIONS FORMELLES B À PARTIR DE SPÉCIFICATIONS SEMI-FORMELLES, Thèse du Conservatoire National des Arts et Métiers, 1998

[Övergaar98] G. Övergaar, K. Palmkvis, A FORMAL APPROACH TO USE CASES AND THEIR RELATIONSHIPS, UML'98, International Workshop, Mulhouse France, 1998

[Palfinger97] G. Palfinger, STATE ACTION MAPPER, PLoP'97 Conference

[Panet94] G. Panet, R. Letouche, MERISE/2, MODELES ET TECHNIQUES MERISE AVANCES, Les éditions d'organisation, 1994

[Pernici90] B. Pernici, OBJECT WITH ROLES, Proceedings of the conference of Office Information Systems, 1990

[Peterson77] J. Peterson, PETRI NETS, ACM Computing Surveys, 9(3) September 1977

[Ran96] A. Ran, MOODS, MODELS FOR OBJECT-ORIENTED DESIGN OF STATE, Pattern Languages of Program Design 2, Edited by M. Vlissides , J. O. Coplien, and N. L. Kerth, Addison-Wesley, 1996

[Rational97] Rational Software Corporation, RATIONAL OBJECTORY PROCESS 4.1, Technical Report, 1997

[RCM94] R.C.M. Consulting Inc., DISCOVERING PATTERNS IN EXISTING APPLICATIONS (APPENDIX), 1994

[Reenskaugh92] T. Reenskaugh and all., OORASS: SEAMLESS SUPPORT FOR THE CREATION AND MAINTENANCE OF OBJECT ORIENTED SYSTEMS, Journal of Object Oriented Programming, 1992

[Reisig85] W. Reisig, PETRI NETS, Springer-Verlag, Berlin, 1985

[Renouf94] D.W. Renouf, B. Henderson-Sellers, INCORPORATING ROLES INTO MOSES, Proceedings of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS PACIFIC'94), 1994

[Rieu91] D. Rieu Et Al, INSTANCIATION MULTIPLE ET CLASSIFICATION D'OBJETS, 7ieme journées BDA, 1991

[Rieu97] D.Rieu, M. Tollenaere and Al., PATRONS D'OBJETS POUR LES SGDT, 2ème congrès international Franco-Québécois : le génie industriel dans un monde sans frontières, 1997

[Rolland93] C. Rolland, ADAPTER LES MÉTHODES À L'OBJET : CHALLENGES ET EMBÛCHES, Journées Méthodes d'Analyse et de Conception Orientées Objet des Systèmes d'Information, AFCET, Paris, 1993

[Rumbaugh94] J. Rumbaugh, GETTING STARTED, USING USE CASES TO CAPTURE REQUIREMENTS, Journal of Object-Oriented Programming, SIGS ed., septembre 1994

[Rumbaugh95] J. Rumbaugh and al., OMT, MODÉLISATION ET CONCEPTION ORIENTÉES OBJET, Editions Masson, Paris 1995

[St-Marcel96] C. Saint-Marcel, POINT DE VUE, DES GRAPHES D'ÉTATS AUX LANGAGES ORIENTÉS OBJETS, Rapport de D.E.A. Système d'Information, Université J. Fourier, Grenoble, 1996

[St-Marcel98] C. Saint-Marcel, MODÉLISATION DU COMPORTEMENT, UNE APPROCHE PAR LES RÔLES, XVIe Congrès INFORSID, Montpellier, 1998

[St-Marcel2000] C. Saint-Marcel, ANTITHESE, exemplaire unique, aucune édition connue, Grenoble le 1er janvier 2000

[Shlaer92] S. Shlaer, S.J. Mellor, OBJECT LIFECYCLES : MODELING THE WORLD IN STATES, Yourdon Press computing series, 1992

[Schoenfeld96] A. Schoenfeld, DOMAIN SPECIFIC PATTERNS: CONVERSIONS, PERSONS AND ROLES, AND DOCUMENTS AND ROLES, In Proceedings of the 1996 Conference on Pattern Languages of Programming (PLoP'96), Washington University Department of Computer Science, Technical Report WUCS-97-07, 1997

[Sibertin-Blanc85] C. Sibertin-Blanc, HIGH LEVEL PETRI NETS WITH DATA STRUCTURE, Proceedings of the 6th european Workshop on Application and Theory of Petri Nets, Espoo, june 1985

[Sibertin-Blanc98] C. Sibertin-Blanc, COOPERATIVE OBJECTS: PRINCIPLES, USE AND IMPLEMENTATION, In Petri Nets and Object Orientation, Lectures Notes in Computer Science, Springer-Verlag, 1998

[Steyaert96] P. Steyaert and all., REUSE CONTRACTS: MANAGING THE EVOLUTION OF REUSABLE ASSETS, Proceedings of OOPSLA'96, ACM SIGPLAN Notices, 31(10), pp. 268-286, ACM Press, 1996

[UML97] G. Booch, I. Jacobson, J. Rumbaugh, UNIFIED MODELING LANGUAGE, version 1.1, 1997

[Vayda95] T. P. Vayda, LESSONS FROM THE BATTLEFIELD, Austin OOPSLA'95, 1995

[Waldén94] K. Waldén, J.M. Nerson, SEAMLESS OBJECT-ORIENTED SOFTWARE ARCHITECTURE - ANALYSIS AND DESIGN OF RELIABLE SYSTEMS, Prentice Hall 1994, ISBN 0-13-031303-3

[Wirfs-Brock90] Wirfs-Brock R., Wilkerson B., Wiener L., *Designing Object Oriented Software*, Prentice-Hall, 1990