UNIVERSITÉ DE MONTRÉAL

Vers un paradigme transformationnel dans le développement orienté objet

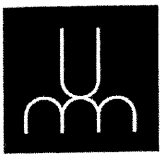
Par

Ismaïl KHRISS

Département d'Informatique et de Recherche Opérationnelle
Faculté des Arts et des Sciences

Thèse présentée à la Faculté des Études Supérieures
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

Mai, 2000



Université de Montréal
Faculté des études supérieures

Cette thèse intitulée:

Vers un paradigme transformationnel dans le développement orienté objet

présentée par:

Ismaïl KHRISS

a été évaluée par un jury composé des personnes suivantes:

Professeur J-Y Nie	Président du jury
Professeur R.K. Keller	Directeur du recherche
Professeur D. Deveaux	Examinateur externe
Professeur F. Lustman	Membre du jury
Professeur E. Merlo	Représentant du Doyen

Thèse acceptée le

Sommaire

Plusieurs modèles de maintenance exigent que les changements soient faits et documentés dans les spécifications des besoins et soient par la suite propagés vers le code source à travers les modèles d'analyse et de conception. Ces modèles supposent donc un haut niveau de traçabilité comme facteur clé de maintenabilité. Cette traçabilité ne peut être obtenue sans une approche transformationnelle au développement des logiciels.

Cette thèse a pour objectif de fournir une approche transformationnelle pour supporter l'ingénierie des besoins à l'aide des scénarios et pour appliquer les patrons de conception au moyen d'algorithmes et de processus. Le langage unifié de modélisation (Unified Modeling Language, UML) a été adopté comme notation de modélisation.

Dans le support de l'ingénierie des besoins, nous proposons, premièrement, un algorithme incrémental pour la synthèse des spécifications dynamiques à partir des scénarios. Cet algorithme prend comme entrée un ensemble de diagrammes de collaboration d'UML et produit en sortie les diagrammes d'états-transitions d'UML de tous les objets collaborant dans les scénarios d'entrée. En outre cet algorithme permet la vérification de la cohérence et de la complétude des scénarios.

Deuxièmement, un autre algorithme est conçu pour la génération d'un prototype de l'interface usager (IU) à partir de la spécification des scénarios. Les scénarios sont acquis sous forme de diagrammes de collaboration enrichis par des informations de l'IU. Ce prototype de l'IU peut être exécuté par un constructeur d'IU ce qui permet non seulement la validation des scénarios avec les utilisateurs mais aussi sa personnalisation et son raffinement éventuel.

L'application automatique des patrons de conception est réalisée par une nouvelle approche pour le raffinement successif des modèles (en format UML) statiques et dynamiques de conception. Ces raffinements successifs sont basés sur des schémas de raffinement. Un schéma de raffinement est composé de deux compartiments. Le premier compartiment décrit le modèle abstrait de conception, et le deuxième compartiment montre le modèle détaillé correspondant après l'application d'un patron de conception. Nous proposons aussi un catalogue de schémas de micro-raffinement

qui permettent non seulement de décrire un schéma de raffinement mais aussi de prouver sa validité.

Mots clés: modélisation orientée objet, transformation, traçabilité, scénario, schéma de raffinement, patron de conception, interface usager, prototypage rapide, UML.

Abstract

Several maintenance models suggest that changes be done and documented in the requirements specification and subsequently propagated through the analysis and design models to the source code. These models require all a high level of traceability. Such traceability can only be obtained by a transformational approach to software development.

The objective of this thesis is to provide a transformational approach for supporting requirements engineering based on scenarios and for applying design patterns by means of algorithms and processes. The Unified Modeling Language (UML) was adopted as the notational framework.

For supporting requirements engineering, we propose first an incremental algorithm for synthesizing behavioral specifications from scenarios. This algorithm generates from a given set of UML collaboration diagrams the UML statechart diagrams of all the objects involved. Moreover, this algorithm allows for the verification of consistency and completeness of the input scenarios.

Secondly, another algorithm is designed to generate a user interface (UI) prototype from scenarios. Scenarios are acquired in the form of UML collaboration diagrams, which are enriched with UI information. This UI prototype can be executed in a UI builder environment for the validation of the scenarios by the users, for customization, and for further refinement.

The automatic application of design patterns is achieved by a new approach to the stepwise refinement of static and dynamic design models represented as UML diagrams. Refinement is based on refinement schemas which are composed of two compartments. The first compartment describes the abstract design model, whereas the second compartment shows the corresponding detailed model after application of one design pattern. We also propose a catalogue of smaller transformations called micro-refinement schemas. These micro-refinement schemas are proven to be correct and can be used to compose correct refinement schemas.

Keywords: object-oriented modeling, transformation, traceability, scenario, refinement schema, design pattern, user interface, rapid prototyping, UML.

Table des matières

Liste des figures

Liste des abréviations

ClassD	diagramme des classes d'UML
CollD	diagramme de collaboration d'UML
IU	interface usager
OCL	langage à objet des contraintes (Object Constraint Language)
OO	orienté objet
StateD	diagramme d'états-transitions d'UML
UIB	bloc de l'interface usager
UML	langage unifié de modélisation (Unified Modeling Language)
UseCaseD	diagramme des cas d'utilisation d'UML

Remerciements

En premier lieu, je tiens à remercier mon directeur de recherche Professeur Rudolf K. Keller de m'avoir accepté au sein de son équipe en plus de m'assurer un support financier tout au long de mon doctorat. Ses remarques pertinentes et éclairées m'ont permis de mieux structurer mes idées et, je l'espère, de mieux les décrire.

Je remercie également M. Bruno Laguë et Bell Canada pour leur collaboration et leur contribution financière dans ce projet.

Mes remerciements vont aussi à l'université de Montréal pour l'aide financière accordée sous forme de bourse d'exemption aux frais de scolarité différentiels.

Je remercie également les membres de mon jury, les professeurs Daniel Deveaux, François Lustman et Jian-Yun Nie pour les commentaires qui ont contribué à l'amélioration de ce document.

Cette thèse doit beaucoup à mes collègues du laboratoire de génie logiciel (Gélo) de l'université de Montréal qui ont permis au «nous», employé tout au long de cette thèse, d'être plus qu'un simple style de langage. Ainsi je remercie Professeur François Lustman pour ses commentaires et remarques très constructives lors de mes présentations au sein du groupe. Je remercie également Mohammed Elkoutbi pour les longs moments que nous avons partagés durant notre collaboration dans certains des travaux de cette thèse. Mes remerciements vont aussi à Siegfried Schönberger qui, avec Rudolf K. Keller, a conçu la version préliminaire de l'algorithme de mon premier travail.

Mes remerciements vont aussi à tous mes ami(e)s pour leur soutien moral. Je remercie particulièrement Hind, Mohammed, Abdesselem, Majid, Kamel et Jawad.

Je remercie tous les membres de ma famille et spécialement mes parents Ahmed et Fatima qui m'ont soutenu tout au long de mes études et qui ont fait en sorte, par leur amour, leur tendresse et leur soutien financier, que je puisse avoir les meilleures conditions possibles pour que je termine mes études supérieures. Qu'ils trouvent ici toute ma gratitude et mon amour pour eux. Mes pensées vont aussi à ma grand-mère décédée en mai 1997 et à mon frère Fouad décédé en mai 1987.

Finalement, je remercie Soumia pour son soutien moral, sa disponibilité et sa compréhension.

À mes parents Ahmed et Fatima

À la mémoire de ma grand-mère et de mon frère Fouad

Introduction

Motivation

La construction d'un système logiciel est en général une tâche très complexe, c'est pour cette raison que le but principal du génie logiciel est de réduire cette complexité. C'est dans cette perspective que plusieurs méthodes ont été proposées. Chaque méthode offre une notation pour supporter les différentes vues d'un système ainsi qu'un processus pour orienter le concepteur dans sa tâche de développement d'un système.

En général, un processus de développement est composé de trois étapes importantes: analyse, conception et implantation. L'étape d'analyse consiste à obtenir une spécification du système. Cette spécification décrit ce que le système doit faire pour satisfaire les besoins des utilisateurs. L'étape de conception a pour objectif de voir comment les éléments composant le système logiciel seront implantés. Ces éléments sont les structures de données, l'architecture du système et le détail de chaque composante. Finalement, l'étape d'implantation ne fait que traduire les éléments de la conception dans un langage compris par l'ordinateur.

En outre, plusieurs modèles de maintenance exigent que les changements soient faits et documentés dans les spécifications des besoins et soient par la suite propagés vers le code source à travers les modèles d'analyse et de conception. Ces modèles supposent donc un haut niveau de traçabilité comme facteur clé de maintenabilité. Cependant, la plupart des méthodes de développement proposent des approches par élaboration. Or, l'élaboration est par essence une tâche manuelle qui ne peut pas être automatisée et qui rend impossible tout support automatique de la traçabilité. C'est pourquoi nous pensons qu'un processus de développement doit être vu comme une approche basée sur des transformations.

Depuis le début des années 1970, plusieurs systèmes transformationnels ont été proposés, dont la plupart n'ont pas survécu pour principalement trois raisons majeures. La première raison réside dans le fait que ces systèmes utilisent des langages ad-hoc et difficiles à utiliser. La deuxième raison est l'incapacité de ces systèmes à supporter des grands systèmes. La troisième raison est que nous avons l'impression qu'ils ont la difficulté à intégrer les progrès récents en génie logiciel. Parmi ces progrès nous pouvons citer ceux qui nous intéressent particulièrement à savoir les

approches orientées objet (OO) pour réduire la complexité grandissante des systèmes logiciels, les techniques d'ingénierie des besoins à l'aide des scénarios [1] comme un excellent moyen d'obtenir des spécifications en conformité avec les attentes des utilisateurs, les notations visuelles pour simplifier l'utilisation et la compréhension des modèles de développement, et enfin les patrons de conception qui offrent un bon véhicule pour la réutilisation et la diffusion des bonnes expériences en conception OO.

Contributions

Le but de cette thèse est de proposer une approche transformationnelle pour (1) supporter le processus d'ingénierie des besoins à l'aide des scénarios et (2) permettre l'application automatique des patrons de conception. Comme notation de modélisation, nous avons adopté le langage unifié de modélisation (Unified Modeling Language, UML). Au delà du fait que ce langage offre une notation visuelle pour la description des différentes vues d'un système, son principal intérêt réside dans le fait qu'il est devenu un standard dans le domaine de la modélisation OO.

Pour l'objectif (1), nous proposons premièrement un algorithme incrémental pour la synthèse des spécifications dynamiques à partir des scénarios. Cet algorithme prend comme entrée un ensemble de diagrammes de collaboration d'UML et produit en sortie les diagrammes d'états-transitions d'UML de tous les objets collaborant dans les scénarios d'entrée. En outre cet algorithme permet la vérification de la cohérence et de la complétude des scénarios.

Deuxièmement, un autre algorithme est conçu pour la génération d'un prototype de l'interface usager (IU) à partir de la spécification des scénarios. Les scénarios sont acquis sous forme de diagrammes de collaboration d'UML enrichis par des informations de l'IU. Ce prototype de l'IU peut être exécuté par un constructeur d'IU ce qui permet non seulement la validation des scénarios avec les utilisateurs mais aussi sa personnalisation et son raffinement éventuel. Les deux algorithmes ont été implantés avec le langage Java et testés sur des exemples de systèmes de petite à moyenne taille. Ces algorithmes, une fois intégrés dans un outil CASE, vont contribuer à rendre le processus d'ingénierie des besoins une tâche plus facile qui va prendre moins du temps et qui va être moins exposée aux erreurs.

Les résultats de l'objectif (1) sont publiés dans [Elkoutbi et al., 1999a; Khriss et al., 1999a; Khriss et al., 1998; Schönberger et al., 2000] et décrits dans les rapports techniques [Elkoutbi et al.,

1. Un scénario est une description partielle ou incomplète des interactions entre un usager et le système pour accomplir une tâche spécifique.

1999b; Khriss et al., 1999b]. Nous avons aussi préparé un site web pour une diffusion plus large du logiciel qui englobe les algorithmes de cet objectif. Nous avons baptisé ce logiciel SUIP pour Scenario-based User Interface Prototyping. Le site est logé à l'adresse <http://www.iro.umontreal.ca/labs.gelo/suip>.

Pour l'objectif (2), nous avons conçu une nouvelle approche pour le raffinement successif des modèles (en format UML) statiques et dynamiques de conception. Ces raffinements successifs sont basés sur des schémas de raffinement. Un schéma de raffinement est composé de deux compartiments. Le premier compartiment décrit le modèle abstrait de conception, et le deuxième compartiment montre le modèle détaillé correspondant après l'application d'un patron de conception.

Nous proposons aussi un ensemble de petits raffinements que nous appelons schémas de micro-raffinement. Le rôle des schémas de micro-raffinement est triple. Primo, ils permettent de décrire un schéma de raffinement puisque ce dernier peut être composé d'une séquence de schémas de micro-raffinement. Secundo, ces derniers permettent de démontrer la validité des schémas de raffinements. Pour cela, il suffit de démontrer la validité de tous les schémas de micro-raffinement qui sont impliqués dans le schéma considéré. Finalement, la généralité des schémas de micro-raffinement les rend réutilisables par plusieurs schémas de raffinement. Notons que ces raffinements n'ont pas encore été implantés mais ils ont été testés (manuellement) sur des exemples de systèmes de petite à moyenne taille.

Les résultats de l'objectif (2) sont publiés dans [Khriss et Keller, 1999a; Khriss et al., 1999c] et décrits dans le rapport technique [Khriss et al., 2000].

Organisation de la thèse

Dans le premier chapitre, nous commencerons par discuter en détail les motivations derrière le besoin d'une approche transformationnelle dans le développement logiciel. Ensuite, nous présenterons quelques approches transformationnelles existantes. Nous conclurons ce chapitre en décrivant les caractéristiques que doit avoir une bonne approche transformationnelle.

Dans la première section du deuxième chapitre, nous présenterons le langage UML que nous avons choisi comme notation dans nos travaux. En particulier, nous nous intéresserons à quatre types de diagrammes: les diagrammes de classes, les diagrammes de cas d'utilisation, les diagrammes de collaboration, et enfin les diagrammes d'états-transitions. Pour une bonne compréhension des différents algorithmes, nous donnerons dans les annexes A, B et C, respectivement les grammaires des diagrammes de classes, de collaboration et d'états-transitions. Ensuite, nous

introduirons la notion des patrons de conception tout en mettant l'accent sur l'intérêt de ces derniers dans les transformations.

Dans le troisième chapitre, nous présenterons une vue générale de notre approche transformationnelle pour le développement logiciel. En premier lieu, nous décrirons les différentes tâches d'un processus d'ingénierie des besoins à l'aide des scénarios. Ensuite nous montrerons comment nous supportons ces tâches par un processus transformationnel. Dans la deuxième partie du chapitre, nous parlerons du deuxième type de transformations à savoir l'application automatique des patrons de conception.

Dans le quatrième chapitre, nous décrirons notre premier algorithme du processus transformationnel de l'ingénierie des besoins. À partir d'un scénario décrit dans un diagramme de collaboration d'UML, cet algorithme permet de dériver le comportement dynamique partiel des objets, en forme de diagrammes d'états-transitions d'UML, collaborant dans le scénario. Le pseudo-code de l'algorithme se trouve en annexe D. Nous terminerons ce chapitre par une discussion du présent algorithme.

Dans le cinquième chapitre, nous détaillerons deux autres algorithmes du processus d'ingénierie des besoins. Le premier permet de faire une analyse d'un diagramme d'états-transitions. Le deuxième a pour objectif d'intégrer deux diagrammes d'états-transitions. Le pseudo-code des deux algorithmes se trouve respectivement en annexe E et F. Nous terminerons ce chapitre par une discussion des deux algorithmes.

Dans le sixième chapitre, nous présenterons notre travail qui s'attaque à l'intégration de la conception générale avec la conception détaillée à l'aide d'une approche de raffinements successifs basée sur l'application des patrons de conception. Nous utiliserons le patron Observer pour illustrer notre approche. Un autre exemple, celui du patron Mediator, sera donné dans la première section de l'annexe H. Nous décrirons aussi, en annexe G, les schémas de micro-raffinement, leur pseudo-code et les preuves de leur validité. Dans la deuxième section de l'annexe H, nous présenterons le pseudo-code du schéma de raffinement proposé pour le patron Observer.

Dans le septième chapitre, nous commencerons par décrire les différents étapes de l'algorithme sous-jacent à la dernière phase de notre processus d'ingénierie des besoins. Cet algorithme consiste à générer un prototype de l'IU à partir de la spécification des scénarios. Ensuite nous esquisserons notre vision d'un environnement logiciel supportant une approche transformationnelle pour le développement des systèmes OO.

Nous terminerons cette thèse par une conclusion générale où nous ferons une synthèse de nos

recherches et de ce qui pourrait être fait comme suite à nos travaux.

Chapitre 1 Transformations en développement logiciel

Plusieurs modèles de maintenance exigent que les changements soient faits et documentés dans les spécifications des besoins et soient par la suite propagés vers le code source à travers les modèles d'analyse et de conception [Basili, 1990]. Ces modèles supposent donc un haut niveau de traçabilité comme facteur clé de la maintenabilité. Cette traçabilité ne peut pas être assurée par les processus de développement conventionnels tel que le modèle en cascade proposé par Royce [Royce, 1970]; plutôt, elle exige une approche transformationnelle dans le développement logiciel.

Depuis le début des années 1970, plusieurs systèmes transformationnels ont été proposés, la plupart ont échoué pour plusieurs raisons tels que l'étroitesse de leur champ d'application ou leur inadaptation pour les systèmes OO. Cela n'empêche pas qu'une approche transformationnelle peut réussir si elle arrive à bien tenir compte des expériences passées.

Dans ce chapitre, nous allons commencer par présenter les motivations derrière une approche transformationnelle dans le développement logiciel. Ensuite, nous allons expliquer pourquoi nous considérons les notions de transformation et de traçabilité comme équivalentes. Puis, nous allons faire une discussion sur un échantillon d'approches transformationnelles existantes. Cette discussion va nous permettre de conclure par ce que nous voyons comme caractéristiques que doit avoir une bonne approche transformationnelle.

1.1 Problématique

Dans cette section, nous allons en premier lieu discuter les avantages et inconvénients des cycles de vie traditionnels suivi par une présentation des différentes vues d'un système OO auxquelles nous nous intéressons. Comme nous allons voir dans le prochain chapitre, UML offre un ensemble de diagrammes pour supporter ces vues. Puis nous allons donner des définitions de la notation de traçabilité que nous trouvons dans la littérature. Nous allons clore cette section par les motiva-

tions derrière le besoin d'une approche transformationnelle dans le développement logiciel.

1.1.1 Cycle de vie des systèmes

La construction d'un système logiciel est en général une tâche très complexe, c'est pour cette raison que le but principal du génie logiciel est de réduire cette complexité. C'est dans cette perspective qu'a été proposé le modèle en cascade [Royce, 1970]. Il est considéré comme le premier vrai cycle de vie d'un système logiciel qui offre un processus clair et systématique. Il est composé de quatre étapes (voir figure 1.1):

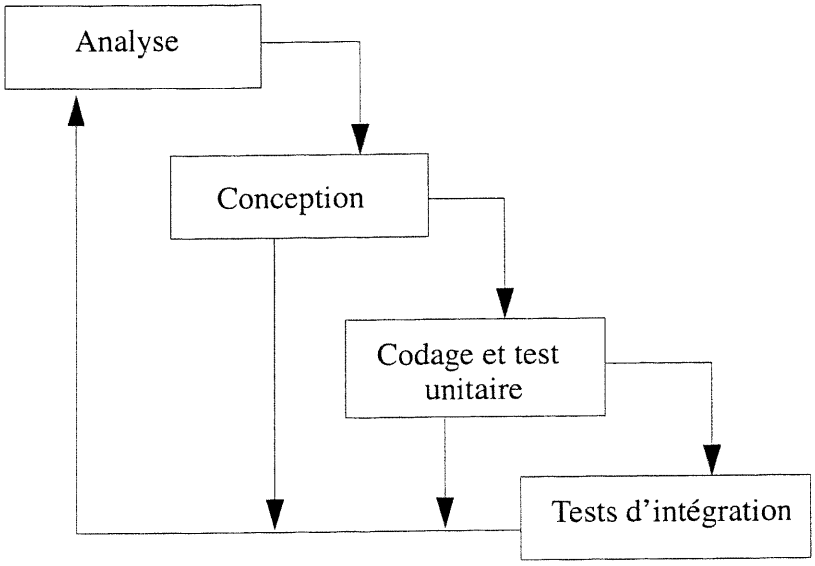


Figure 1.1: Modèle en cascade

1. Analyse: elle consiste à procurer une description détaillée des besoins du système. La description doit être complète, cohérente, lisible et révisable par les diverses parties intéressées. Le résultat de cette étape est un document de spécification qui définit ce que le système doit faire pour satisfaire aux besoins des utilisateurs.
2. Conception: dans cette étape, on s'intéresse à la manière avec laquelle les éléments composant le système logiciel seront implantés. Tous les éléments d'un système seront pris en compte durant cette étape, tels que les structures de données, l'architecture du système et les détails de chaque composante.
3. Codage et test unitaire: elle consiste à traduire la conception dans un langage compris par

l'ordinateur. En outre chaque composante du système est testée individuellement pour détecter d'éventuelles erreurs.

4. Tests d'intégration: cette étape a pour rôle de tester l'ensemble du système.

L'avantage principal du modèle en cascade est de fournir des points de mesure concrets, sous forme de documents; il augmente ainsi la visibilité sur l'état d'avancement du système en cours de développement. Cependant, il suit un processus séquentiel où chaque étape doit être terminée avant que la suivante ne commence. Cette rigidité cause un ensemble de problèmes comme le fait remarquer Boehm [Boehm, 1988]:

- Le modèle n'adresse pas de façon adéquate les problèmes liés aux changements dans les besoins de l'utilisateur.
- Le modèle suppose une progression relativement uniforme des étapes d'élaboration. En effet, la démarche en cascade ne donne des résultats satisfaisants que lorsqu'il est effectivement possible d'enchaîner les phases sans trop de problèmes. Il faut que l'ensemble des besoins soit parfaitement connu et le problème complètement compris par les analystes; il faut aussi que la solution soit facile à déterminer par les concepteurs. Or, les projets ne se présentent pas tous sous ces conditions idéales et ceci pour plusieurs raisons telles que la méconnaissance des besoins par l'utilisateur ou des problèmes engendrés par des choix technologiques.
- Le modèle ne prend pas en compte le genre de développement évolutif rendu possible par le prototypage rapide et les langages de quatrième génération.
- Le modèle n'adresse pas les modes récents de développement de programmes en liaison avec les possibilités de programmation automatique, les possibilités de transformation de programmes et les capacités d'outils basés sur la connaissance.

Le cycle en cascade est souvent représenté sous la forme d'une lettre V pour faire apparaître que le développement des tests est effectué de manière synchrone avec le développement logiciel. La figure 1.2 montre un exemple de modèle en V, dans lequel les tests fonctionnels sont spécifiés lors de l'analyse, les tests d'intégration lors de la conception et les tests unitaires pendant la phase de codage [Pressman, 1997].

Il existe un certain nombre de cycles alternatifs pour remédier à ces problèmes. Le plus connu est le modèle en spirale de Boehm [Boehm, 1988] où le cycle de développement est vu comme un processus incrémental et itératif. Le cycle de vie itératif et incrémental est basé sur l'évolution de prototypes exécutables, et donc sur l'évaluation d'éléments concrets. Il s'oppose ainsi au cycle de vie en cascade qui repose sur l'élaboration de documents.

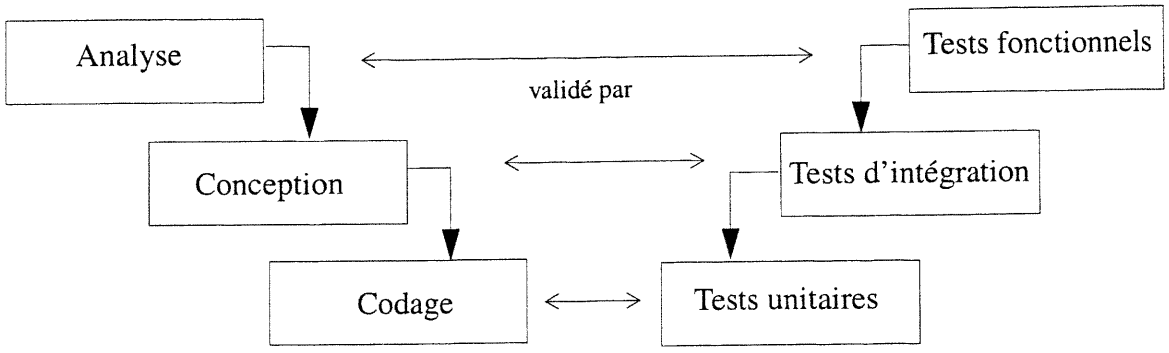


Figure 1.2: Modèle de développement en V

1.1.2 Vues d'un système orienté objet

L'architecture d'un système OO est considérée à partir de plusieurs points de vues complémentaires, à l'image de celles décrites par Kruchten [Kruchten, 1995] dans son modèle dit des 4 + 1 vues (voir figure 1.3).

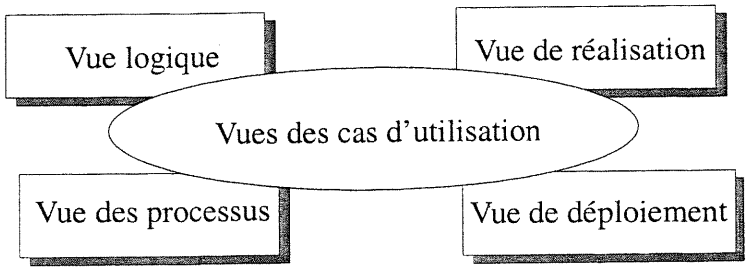


Figure 1.3: Vues d'un système

La vue logique décrit les aspects statiques et dynamiques d'un système en termes de classes et d'objets et se concentre sur l'abstraction, l'encapsulation et l'uniformité. Le système est décomposé dans un jeu d'abstraction-clés issues du domaine du problème. La vue de réalisation s'intéresse à l'organisation des modules en montrant l'allocation des classes dans les modules et l'allocation des modules dans les sous-systèmes. La vue des processus représente la décomposition du système en flots d'exécution [1] (processus, threads, tâches), la synchronisation entre flots et l'allocation des objets et des classes au sein des différents flots. La vue des processus se préoc-

1. À partir du chapitre 2, le terme flot d'exécution va plutôt désigner un thread.

cupe également de la disponibilité du système, de la fiabilité des applications et des performances. La vue de déploiement décrit les différentes ressources matérielles et l'implantation du logiciel dans ces ressources.

Les cas d'utilisation décrivent le comportement du système du point de vue de l'utilisateur. Ils permettent de définir les limites du système et ses relations avec l'environnement. En fait, un cas d'utilisation est une manière spécifique d'utiliser le système. C'est l'image d'une fonctionnalité du système, déclenchée en réponse à la stimulation par un acteur externe.

Les cas d'utilisation forment la colle qui unifie les quatre vues précédentes du système. Ils motivent et justifient les choix d'architecture. Ils permettent d'identifier les interfaces critiques, ils forcent les concepteurs à se concentrer sur les problèmes concrets, ils démontrent et valident les autres vues d'architecture.

1.1.3 Notion de traçabilité

La traçabilité fait déjà partie des recommandations des standards de la qualité en génie logiciel [IEEE/Std.1219, 1992; IEEE/Std.982.1, 1989; IEEE/Std.839, 1984; ISO/9000-3]. Or la définition donnée est très générale et s'intéresse plutôt à la traçabilité des besoins. Par exemple, un document du département de défense américain définit la traçabilité comme un moyen de s'assurer que les besoins des clients sont satisfaits [DOD-STD-2167A, 1988].

Lindvall et Snadhal donnent une définition plus précise de la traçabilité [Lindvall et Snadhal, 1996]. Ils distinguent deux niveaux de traçabilité: la traçabilité verticale et la traçabilité horizontale (voir figure 1.4). La traçabilité verticale est la possibilité de retracer les éléments dépendants à l'intérieur d'un modèle, alors que la traçabilité horizontale permet de retracer les éléments correspondants entre les modèles des différentes étapes du cycle de développement.

L'objectif de la traçabilité est de supporter la cohérence à l'intérieur d'un modèle ou entre les différents modèles d'un système. L'idée sous-jacente est qu'un système n'est pas seulement du code mais aussi un ensemble de documents qui sont les «livrables» des étapes du processus de développement. Ainsi, les éventuelles modifications du système seront faites dans la spécification des besoins et, grâce à la traçabilité, elles pourront être propagées au code source à travers les modèles d'analyse et de conception.

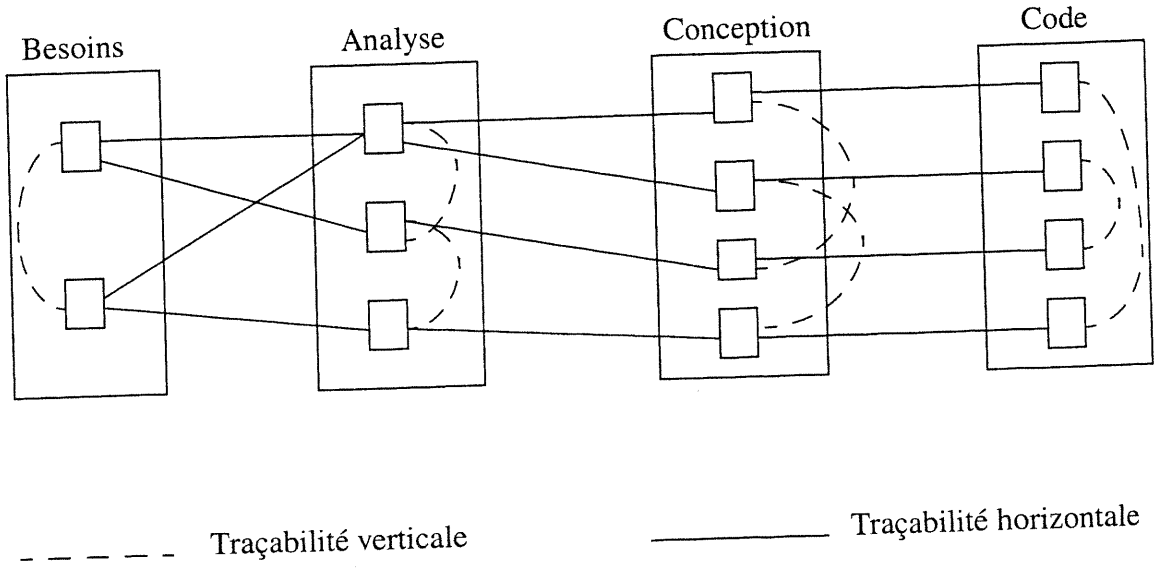


Figure 1.4: Modèle de traçabilité

Actuellement, peu d'outils CASE supportent la traçabilité et ceci seulement d'une façon manuelle. Par exemple, Objectory SE [Objective Systems, 1993b], qui supporte la méthode Objectory de Jacobson [Objective Systems, 1993a] [2], permet l'introduction de liens physiques entre les éléments d'un modèle comme moyen de traçabilité. Cependant, les décisions d'utilisation de ces liens sont laissées à l'expertise du concepteur. En d'autres termes, Objectory n'utilise en aucune façon ces liens.

1.1.4 Besoin d'une approche de transformation

La plupart des méthodes d'analyse et de conception OO proposent des approches de développement par élaboration [3]. Par exemple, Rumbaugh et al. résument dans le processus d'OMT de la façon suivante [Rumbaugh et al., 1991]: *"the analysis models are elaborated, refined, and then optimized to produce a practical design"*. Or, l'élaboration est par essence une tâche manuelle qui ne peut pas être automatisée. Ceci rend impossible tout support automatique de la traçabilité. Pour cette raison, nous pensons qu'un processus de développement doit être vu comme une approche basée sur des transformations.

2. Objectory n'est autre que l'ancêtre de la méthode OOSE [Jacobson et al., 1993].
3. Cette remarque a aussi été faite par Shlaer et Mellor [Shlaer et Mellor, 1993].

1.2 Traçabilité versus transformation

Un processus de développement d'un système n'est autre qu'un un processus de raffinement des modèles abstraits (les besoins sont le plus haut niveau d'abstraction) vers des modèles de niveau plus concret (le code source est le niveau le plus concret). Ceci nous permet de donner les définitions suivantes:

Définition 1.1. Une transformation [4] est l'opération de raffinement d'un modèle d'un niveau abstrait vers un modèle de niveau plus détaillé.

Nous considérons la traçabilité et les transformations comme deux notions équivalentes, ce que nous explicitons dans la définition suivante.

Définition 1.2. Il y a une traçabilité entre deux types de modèles A et B si et seulement si il existe une relation de transformation T tel que: $\forall X \in A \exists Y \in B / X T Y$. En fait, grâce à la relation T, nous pouvons savoir qu'il existe une traçabilité entre les éléments X et Y.

Notons que dans tout le reste de ce travail, nous nous appuierons sur ces définitions.

1.3 Approches de transformation

1.3.1 Cadre pour la classification

Partsch et Steinbrüggen ont défini un cadre pour la classification des approches de transformation [Partsch et Steinbrüggen, 1986]. Ce cadre est constitué de cinq dimensions: objectif, niveau du support, forme des règles de transformation, langages utilisés et éventail d'application.

Objectif

Une approche de transformation peut avoir au moins un des trois objectifs suivants. Le premier objectif est de permettre la synthèse des programmes à partir d'une description formelle du problème. Le deuxième objectif est le support de modifications de programmes. Une modification

4. En fait, une transformation est plus générale qu'un raffinement comme nous allons voir dans la section 1.3.1. Cependant, dans cette thèse, nous ne nous intéressons pas aux autres types de transformation.

peut consister, par exemple, à optimiser les structures de données ou à adapter des programmes à un style donné de programmation. Finalement, le troisième objectif consiste à vérifier la validité des programmes.

Niveau du support

En général, les approches de transformation contiennent une collection prédéfinie de transformations disponibles, appelée catalogue, et un mécanisme pour leur application. Une approche peut permettre l'extension de cette collection par la définition de nouvelles règles définies par l'utilisateur lui-même ou créées à partir de la composition des règles existantes. Une approche peut aussi fournir une aide pour la sélection des règles du catalogue.

Les systèmes de transformation peuvent être totalement automatiques ou semi-automatiques. Ils peuvent avoir aussi des facilités pour la documentation du processus de développement. Et enfin, ils peuvent aussi supporter la possibilité pour l'évaluation des programmes. Cette évaluation peut être de deux manières: le système peut contenir des outils pour les tests ou pour l'analyse des programmes.

Forme des règles de transformation

Il existe deux formes possibles pour la représentation des règles de transformation: algorithmique ou schématique. Une règle algorithmique prend un programme et en ressort un nouveau alors qu'une règle schématique est représentée par une paire de programmes ou un est remplacé par l'autre.

Langages utilisés

En fonction de l'objectif des approches, plusieurs langages peuvent être utilisés et rentrent dans deux catégories: les langages de spécification et les langages d'implantation. Un langage de spécification peut aller d'un langage naturel formalisé à un langage purement descriptif tel que les notations mathématiques ou la logique des prédicats. Les langages d'implantation peuvent être un langage fonctionnel tel que LISP, procédural tel que C, ou OO tel que Java.

Éventail d'applications

Cette dimension est la plus importante. En effet, la limitation de l'éventail d'applications est la

principale source de critiques des approches transformationnelles.

1.3.2 Exemples d'approches de transformation

Dans la littérature, il existe une multitude d'approches de transformation. Ci-dessous, nous allons nous limiter à quelque exemples représentatifs. Notons que dans les chapitres 5, 6 et 7 nous allons présenter d'autres travaux connexes aux domaines spécifiques de nos transformations.

Le projet SAFE/TI

Le projet SAFE/TI est considéré comme étant un des premiers projets dans le domaine des systèmes tranformationnels. Il a été initié par R. Balzer en 1973 [Balzer 1973]. L'intérêt de ce projet réside dans le fait qu'il illustre bien non seulement les grandes ambitions qu'ont eues les pionniers dans ce domaine mais aussi les raisons des critiques des systèmes transformationnels.

Le projet est constitué de deux sous-projets: SAFE (pour Specification Acquisition from Experts) [Wile et al., 1977; Balzert et al. 1980] et TI (Transformational Implementation) [Balzer et al., 1976]. L'objectif de SAFE est de synthétiser une spécification formelle du système à partir d'une description informelle écrite en langage naturel. Le langage de spécification s'appelle GIST. Une description d'un système en GIST est composée d'un ensemble d'états (les types d'objets et les relations entre eux), des transitions entre les états, et d'un ensemble de contraintes sur les états et les transitions.

Le sous-projet TI a comme rôle d'obtenir une implantation du système à partir de sa spécification par l'application d'une suite de transformations. Les transformations sont conservées dans un catalogue. L'éventail d'applications de cette approche se limite à des petits systèmes, et nous y trouvons un justificateur de lignes [Balzer 1977], un éditeur de texte [Balzer et al., 1976], un compresseur de texte [Wile 1983], et le problème de huit reines [Balzer 1981].

Le projet SAFE/TI a connu un grand échec. La raison principale de son échec réside dans le fait que son objectif était extrêmement ambitieux: obtenir une implantation d'un système à partir d'une description informelle du problème et de sa solution écrite en langage naturel!

Shlaer et Mellor

Shlaer et Mellor [Shlaer et Mellor, 1992] proposent une méthode de conception OO par des trans-

formations. En premier lieu, le système à construire est décomposé en partitions appelées domaines où chacune est analysée séparément. Un domaine est un monde abstrait habité par un ensemble d'objets se comportant suivant des règles et politiques qui caractérisent le domaine. Nous pouvons citer comme exemples de domaines, le domaine d'application, c'est-à-dire le domaine pour lequel est conçu le système, le domaine de l'IU ou le domaine de l'architecture logicielle du système.

Les domaines sont hiérarchiquement classifiés d'un niveau abstrait à un niveau plus détaillé. Ensuite, un moteur de transformation est construit pour traduire les éléments d'un domaine en éléments correspondant du domaine plus bas suivant des règles définies par le concepteur du système. Notons que Shlaer et Mellor donnent simplement un cadre pour les transformations pour les méthodes OO mais ne fournissent pas un catalogue de transformations qui peuvent être réutilisées dans la conception des systèmes.

Moriconi et al.

Moriconi et al. [Moriconi et al., 1995] proposent un ensemble de transformations schématiques qui conservent la validité des architectures. Plusieurs styles d'architectures ont été étudiés tels que la mémoire partagée ou "pipe-filter". Ils utilisent la théorie du premier ordre dans leurs preuves de validité et définissent des règles syntaxiques pour une composition correcte des raffinements individuels. Le travail de Moriconi et al. est peut être appliqué à grande échelle dans la construction des systèmes non OO mais reste difficile à adapter pour le monde OO.

Lano et Bicarregui

Lano et Bicarregui [Lano et Bicarregui, 1999] présentent une représentation sémantique pour UML (Unified Modeling Language, cf. chapitre 2) appelée Real-time Action Logic (RAL) que nous avons utilisée dans nos travaux (cf. chapitre 6). Ils fournissent aussi un ensemble de transformations sur les diagrammes de classes et les diagrammes d'états-transitions. Nous y trouvons, par exemple, des transformations pour éliminer les association optionnelles dans les diagrammes de classes et pour renforcer les conditions de garde sur les transitions des diagrammes d'états-transitions.

Ces transformations sont très utiles puisqu'elles sont applicables pour n'importe quelle conception décrite avec UML; cependant elles doivent être composées pour produire des transformations à un niveau d'abstraction plus intéressantes pour la réutilisation.

1.4 Caractéristiques d'une bonne approche de transformation

Dans la section précédente, nous avons discuté quatre travaux tout en mettant l'accent sur les raisons qui limitent leur application à grande échelle. Cette discussion nous permet de dégager un plan pour avoir une nouvelle approche afin d'éviter les erreurs du passé. Nous allons introduire ce plan sous forme de quatre caractéristiques que doit avoir une bonne approche de transformation. Notons que dans le cadre d'un congrès sur les systèmes transformationnels [Sant'Anna et al., 1999], auquel nous avons assisté [Khriss et Keller, 1999a], deux séances ont été réservées pour discuter des expériences du passé et des pistes prometteuses pour l'avenir des systèmes transformationnels. Plusieurs intervenants ont mis l'accent sur une de ces quatre caractéristiques que nous présentons ci-dessous.

Une approche de transformation doit supporter un langage de spécification expressif et facile à utiliser. L'expressivité et la facilité d'utilisation sont deux conditions contradictoires. En effet, un langage qui devient de plus en plus expressif tend à devenir difficile à utiliser. C'est pourquoi il doit y avoir un compromis entre les deux conditions. Un langage doit pouvoir exprimer la complexité de plus en plus grandissante des logiciels avec toutes ces multiples vues tout en restant le plus possible facile à comprendre et à utiliser. D'ailleurs, un consensus est en train de se réaliser autour du fait qu'un langage de spécification doit être visuel. Ceci explique l'intérêt grandissant autour des méthodes qui offrent ce genre de langages tels que OMT [Rumbaugh et al., 1991] ou UML [Rational et al., 1997], qui est devenu d'ailleurs un standard pour la modélisation OO.

Une approche de transformation doit supporter les nouvelles techniques du génie logiciel. Le génie logiciel est une discipline qui s'enrichit régulièrement de nouvelles techniques qui deviennent plus tard soient des standards, soient des pratiques appuyées par un certain consensus. Il est primordial pour une approche transformationnelle d'intégrer ces techniques pour qu'elle soit en phase avec ce qui se fait ailleurs. Par exemple, une approche transformationnelle ne supportant pas le monde OO serait actuellement inutilisable à grande échelle.

Les règles de transformation doivent être à un niveau adéquat d'abstraction. L'idéal serait à un niveau proche du langage naturel. Malheureusement, vu les expériences passées, cela semble irréaliste. En contre-partie le niveau ne doit pas être à un niveau très détaillé ou très fin ce qui ne permet pas un grand gain de productivité. Selon Batory [Batory, 1999], il faut dépasser le niveau d'une instruction dans un module comme cela se fait actuellement dans plusieurs approches pour arriver à un niveau de composant qui peut être une classe ou un ensemble de classes comme c'est

le cas avec les patrons de conception (cf. chapitre 2).

Les règles de transformation doivent être réutilisables. Une approche de transformation ne doit pas être simplement un cadre pour les transformations ou des transformations dépendantes d'un système mais doit plutôt fournir des règles de transformation réutilisables par des systèmes appartenant à des domaines différents.

Dans ce chapitre, nous avons présenté les motivations derrière une approche transformationnelle dans le développement logiciel. Ensuite, nous avons expliqué pourquoi nous considérons les notions de transformation et de traçabilité comme équivalentes. Puis nous avons conclu, à la suite d'une discussion sur un échantillon d'approches transformationnelles existantes, par quatre caractéristiques que doit avoir une bonne approche transformationnelle. Ces caractéristiques vont constituer la base de notre approche transformationnelle que nous allons présenter dans le chapitre 3. Mais avant, nous allons introduire dans le prochain chapitre deux progrès récents dans le monde du développement OO que nous supportons dans notre approche, à savoir, le langage UML qui est un langage visuel pour la modélisation des systèmes OO, ainsi que les patrons de conception qui permettent de documenter et de communiquer l'expertise en matière de conception logicielle OO.

Chapitre 2 Progrès récents en conception orientée objet

Deux événements ont attiré dans les dernières années l'attention de la communauté qui s'intéresse à la conception OO: l'émergence du langage unifié de modélisation (UML) comme standard dans la modélisation OO et la prise de conscience de la notion des patrons de conception comme moyen de documentation et de diffusion de bonnes conceptions.

Dans ce chapitre, nous allons en premier lieu présenter le langage UML que nous avons choisi comme notation dans nos travaux. Ensuite, nous allons introduire la notion des patrons de conception tout en mettant l'accent sur l'intérêt de ces derniers dans les transformations.

2.1 Le langage unifié de modélisation (UML)

2.1.1 Historique

En 1993, on a recensé jusqu'à cinquante méthodes orientées objet [Booch et al., 1997]. Cependant, aucune d'elles n'a satisfait complètement ses utilisateurs. Partant de ce constat, certains auteurs ont commencé à emprunter des autres des concepts manquants dans leurs méthodes. Ainsi, par exemple, de nouvelles versions des méthodes Booch et OMT sont apparues sous les noms Booch'93 [Booch, 1994] et OMT-2 [Rumbaugh, 1995a; Rumbaugh, 1995b; Rumbaugh, 1995c] qui se sont d'ailleurs beaucoup rapprochées.

Au début de l'année 1995, Booch et Rumbaugh ont décidé d'unifier leurs méthodes. Le résultat de cette unification a été la première version d'UML (version 0.8) [Booch et Rumbaugh, 1995]. Et au cours de l'année 1996, Jacobson (concepteur de la méthode OOSE [Jacobson et al., 1993]) les a rejoints. Dans le même temps, un consortium de partenaires a été créé pour travailler à la définition de la version 1.0 d'UML; il regroupe notamment: DEC, HP, i-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI et Unisys. De cette col-

laboration naît la description d'UML version 1.1 [Rational et al., 1997] remise à l'OMG (Object Management Group [UML, 1998]) en Septembre 1997 et accepté comme standard en Novembre 1997. Remarquons que la plus récente version d'UML est la version 1.3 [Rumbaugh et al., 1999]. Cette version contient des changements mineurs par rapport à la version 1.1 et n'a aucun impact sur nos travaux à part une petite différence dans la notation en ce qui concerne les barres d'ouverture et de fermeture des flots d'exécution (voir plus loin) qui deviennent avec la nouvelle version des états de jonction (représentés par un petit cercle).

En fait, UML peut être considéré comme une unification de la méthode OOSE, une approche qui offre un excellent support pour l'ingénierie des besoins, de la méthode OMT-2, adéquate pour l'étape d'analyse et les systèmes d'information et enfin de Booch'93, qui offre une notation riche pour la phase de conception.

2.1.2 Les éléments de la notation

UML décrit un système sous forme de multiples vues liées mais distinctes, dont chacune saisit un aspect particulier du système. Chaque vue donne lieu à un modèle et est représentée graphiquement par un diagramme. UML définit neuf types de diagrammes différents: les diagrammes de classes, les diagrammes de cas d'utilisation, les diagrammes de suivi d'événements, les diagrammes de collaboration, les diagrammes d'objets, les diagrammes d'états-transitions, les diagrammes d'activités, les diagrammes de composants et les diagrammes de déploiement.

Dans ce qui va suivre nous n'allons présenter que les types de diagrammes utilisés dans nos travaux et qui sont: les diagrammes de classes, les diagrammes de cas d'utilisation, les diagrammes de collaboration, et les diagrammes d'états-transitions. Notons que nous donnons respectivement en annexes A, B et C les grammaires des diagrammes de classes, des diagrammes de collaboration et des diagrammes d'états-transitions. Ces grammaires sont utiles pour la compréhension des différents algorithmes de transformation qui vont être présentés par la suite.

Diagrammes de classes

Un diagramme de classes (ClassD [1]) est utilisé dans la conception logique d'un système pour montrer l'existence de classes et leurs relations. Durant l'analyse, un ClassD indique les rôles et les responsabilités communes des entités qui participent au comportement du système. Durant la

1. Par la suite, ClassD va désigner un diagramme de classes selon la notation d'UML.

conception, il permet de représenter la structure des classes qui forment l'architecture du système.

Un diagramme de classes est constitué principalement de classes et de relations: association, généralisation, et différentes sortes de dépendances, telles que réalisation et usage. Une classe décrit un concept du domaine d'application ou de solution. Elle est représentée graphiquement par un rectangle à trois compartiments. Le premier compartiment donne le nom de la classe, le deuxième décrit ses attributs et le troisième présente la liste de ses opérations. Les relations entre les classes sont exprimées graphiquement par des chemins entre les rectangles. Chaque type de relation est déterminé par la texture de la ligne et des décorations sur le chemin et ses extrémités. La figure 2.1 montre un ClassD d'une version simplifiée du système de gestion d'un guichet automatique bancaire (GAB) [Rumbaugh et al., 1991]. `Un client possède un ou plusieurs comptes` est un exemple d'association entre classes. Il y a aussi une relation d'héritage entre la classe `Compte` et les deux classes `Compte_Epargne` et `Compte_Chèque`.

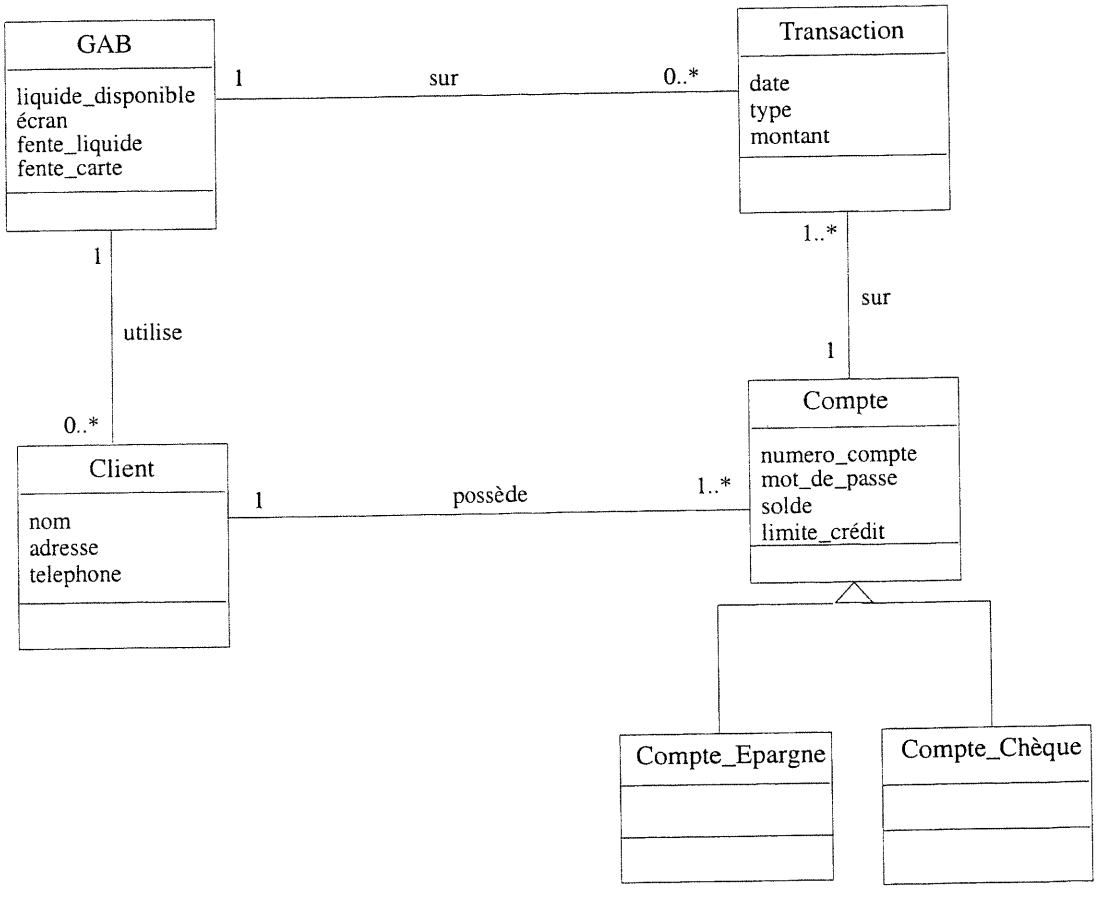


Figure 2.1: Le ClassD du système GAB

Diagrammes de cas d'utilisation

Un diagramme de cas d'utilisation (UseCaseD [2]) modélise les fonctionnalités du système telles que perçues par les utilisateurs externes, appelés acteurs. Il montre les acteurs et les cas d'utilisation dans lesquels ils participent. Un acteur peut être une personne physique, un processus ou toute autre chose qui intéragit avec le système. Il est décrit graphiquement par un petit dessin représentant une personne. Un cas d'utilisation est une unité fonctionnelle cohérente exprimée comme une transaction entre les acteurs et le système. Il est représenté par une ellipse et peut être décrit à différents niveaux de détail. Un cas d'utilisation peut être décomposé en cas d'utilisation plus simples. La description de cette décomposition est exprimée dans un UseCaseD par deux types de relations entre les cas d'utilisation: `«uses»` (relation d'utilisation) et `«extends»` (relation d'extension).

La relation d'utilisation entre cas d'utilisation signifie qu'une instance d'un cas d'utilisation source comprend également le comportement décrit par le cas d'utilisation destination. La relation d'extension entre cas d'utilisation signifie que le cas d'utilisation source étend le comportement du cas d'utilisation destination. La figure 2.2 montre un UseCaseD du système `GAB`. Le système est composé de quatre cas d'utilisation: `Dépôt`, `Identification`, `Retrait`, et `Balance`. Le UseCaseD comprend trois relations d'utilisation telles que celle qui existe entre le cas d'utilisation `Dépôt` et celui `Identification`.

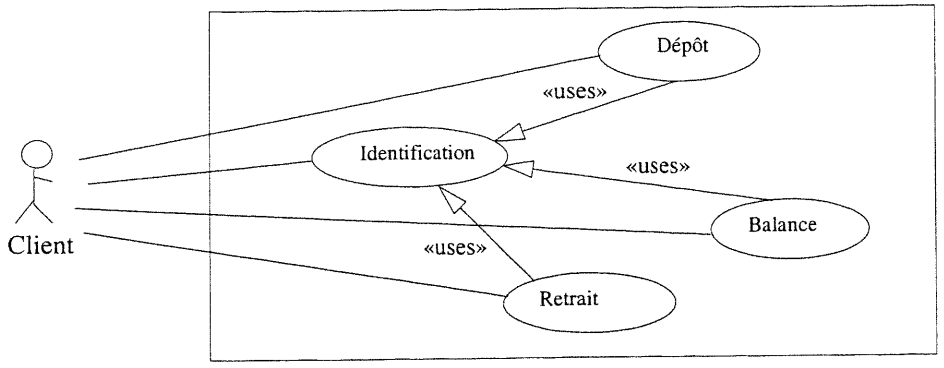


Figure 2.2: Le UseCaseD du système GAB

Diagrammes de collaboration

Un diagramme de collaboration (CollD [3]) est utilisé pour montrer les interactions entre les objets.

2. Par la suite, UseCaseD va désigner un diagramme de cas d'utilisation selon la notation d'UML.

À la différence d'un diagramme de suivi d'événements, un CollD montre les relations entre les objets dans le même contexte qu'un diagramme d'objets. En outre, le temps n'est pas capturé comme une dimension séparée, plutôt les séquences de messages et les tâches parallèles sont déterminées grâce à la numérotation. Par exemple, la suite `1.1 msg1`, `1.2 msg2`, `1.3a msg3`, `1.3b msg4` veut dire que le message `msg2` suit le message `msg1` et que par la suite les deux messages `msg3` et `msg4` sont envoyés en parallèle. Les CollDs peuvent aussi exprimer l'envoi d'une répétition de messages ou un envoi conditionnel.

À l'analyse, un CollD permet de spécifier un scénario d'un cas d'utilisation qu'on retrouve dans un UseCaseD. Et à la conception, il spécifie une opération d'une classe. La figure 2.3 montre un exemple de CollD exprimant un scénario régulier `retraitRégulier` du cas d'utilisation `Retrait` appartenant au système `GAB`.

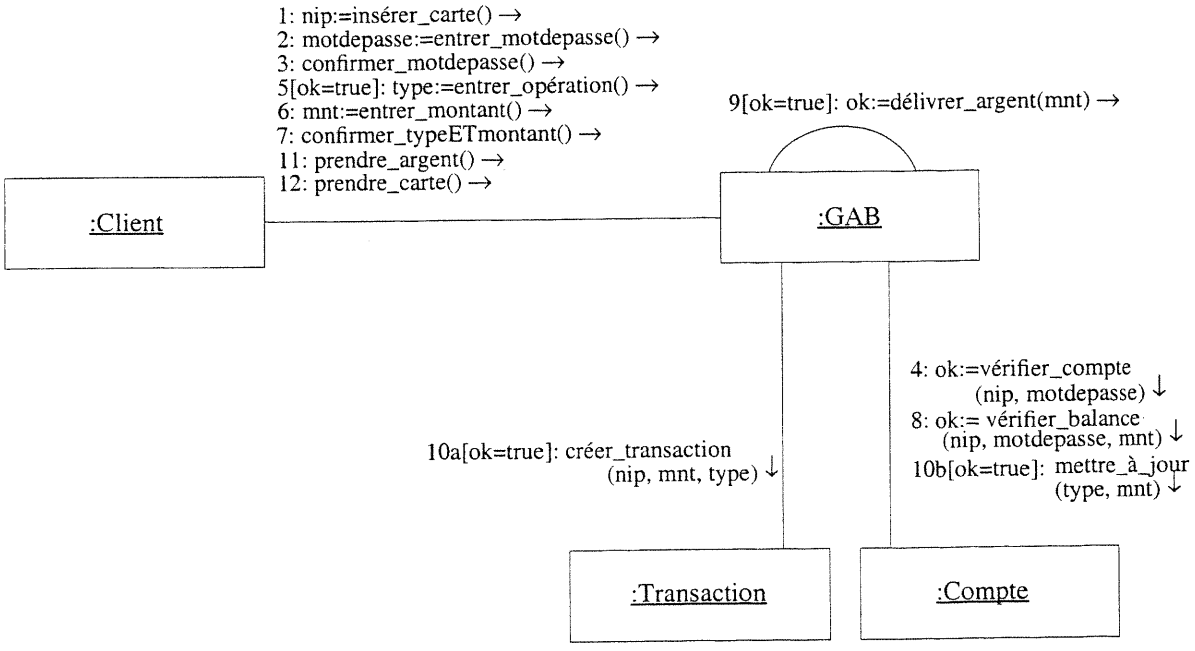


Figure 2.3: Le CollD du scénario `retraitRégulier` du système GAB

Diagrammes d'états-transitions

Un diagramme d'états (StateD [4]) permet de montrer l'espace des états d'une classe, les événements qui causent une transition d'un état à un autre, et les actions qui résultent d'un tel change-

3. Par la suite, un CollD va désigner un diagramme de collaboration d'UML.
4. Par la suite, un StateD va désigner un diagramme d'états d'UML.

ment. UML a adopté la notation de Harel [Harel, 1988]. Cette notation offre une approche simple, mais très expressive, qui est nettement supérieure aux automates à états finis conventionnels grâce aux notions d'états imbriqués et d'états orthogonaux.

L'imbrication des états donne une profondeur aux StateDs et permet de contrôler l'explosion combinatoire des états et des transitions dans les systèmes complexes. L'imbrication des états donne des états de type OU. Par exemple si l'état A est de type OU et contient les états B et C alors si un objet entre dans l'état A, il peut être soit dans l'état B ou dans l'état C. Le concept d'états orthogonaux permet une décomposition de type ET des états. Cela veut dire que si un système se trouve dans l'état A, lequel contient les sous-états B et C, cela signifie que le système se trouve dans l'état A ainsi que dans les deux sous-états B et C. En fait, les états orthogonaux permettent d'exprimer la concurrence dans un système.

En plus des deux types d'états introduits ci-haut, UML définit quatre autres types d'états: les états initiaux, les états finaux, les états réguliers, et les états de synchronisation. Les états initiaux sont représentés par un point noir. Les états finaux sont représentés par un point noir encerclé. Les états réguliers sont représentés par des rectangles arrondis. Les états de synchronisation sont représentés par une barre verticale. Il y a deux types d'états de synchronisation: les barres d'ouverture des flots d'exécution (thread) `splitBar` [5] (split bar) et les barres de fermeture des flots d'exécution `mergeBar` [6](merge bar). La figure 2.4 montre un exemple de StateD qui décrit le comportement dynamique d'une transmission automatique [Rumbaugh et al., 1991]. L'état `Marche avant` est un exemple d'état de type OU.

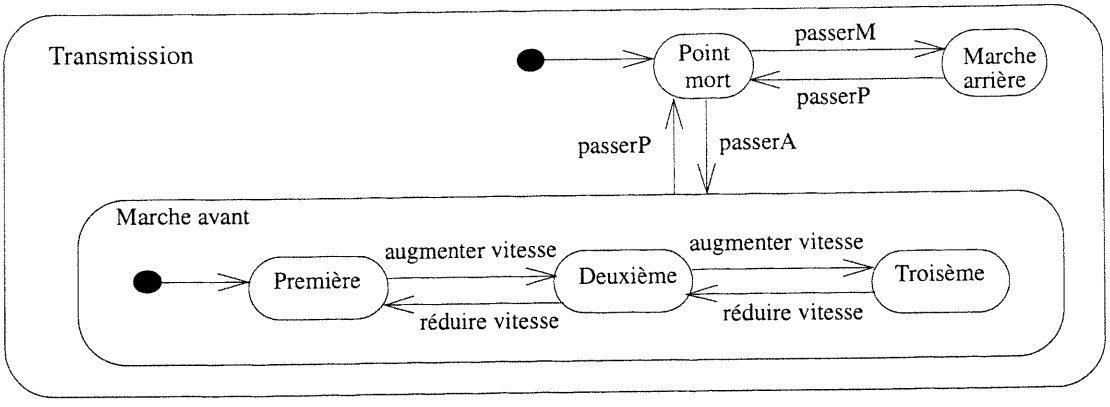


Figure 2.4: Le StateD d'une transmission automatique

5. Par la suite, splitBar va désigner une barre d'ouverture des flots d'exécution selon la notation d'UML.
6. Par la suite, mergeBar va désigner une barre de fermeture des flots d'exécution selon la notation d'UML.

La figure 2.5 montre un autre exemple de StateD qui capture partiellement le comportement dynamique de la classe GAB vue précédemment. L'état `Emission` est un exemple d'état de type ET. En outre, l'exemple montre l'utilisation des barres `splitBar` et `mergeBar` dans la synchronisation des flots d'exécution.

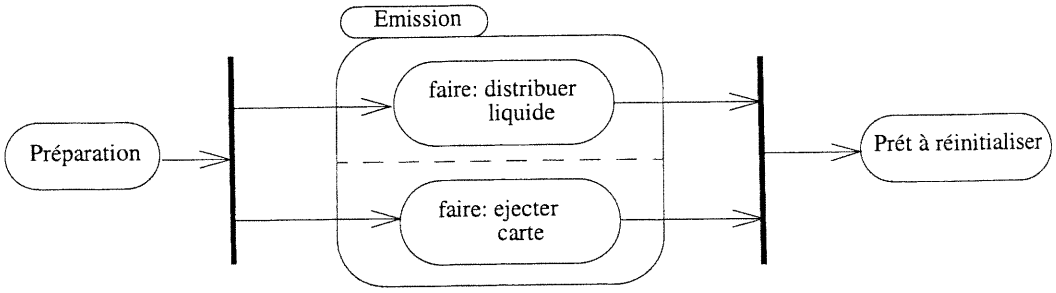


Figure 2.5: Le StateD partiel de la classe GAB

2.2 Les patrons de conception

Les concepteurs expérimentés vous diront qu'une conception réutilisable et facilement adaptable est difficile sinon impossible à réussir convenablement du premier coup. Avant de finir leur conception, ils tentent généralement à plusieurs reprises de la réutiliser et la modifient chaque fois.

Les concepteurs d'expérience savent qu'il ne faut pas chercher à résoudre un problème en partant de mécanismes de base, mais plutôt, qu'il vaut mieux réutiliser des solutions qui ont déjà fait leurs preuves. Quand ils détiennent une solution, ils l'utilisent systématiquement. C'est cette expérience qui contribue à faire d'eux des experts.

Le but des patrons de conception est de recueillir cette expertise en matière de conception logicielle OO. Chaque patron de conception, systématiquement, nomme, explique et évalue un concept important qui figure fréquemment dans les systèmes surtout OO.

En général, un patron de conception est décrit à l'aide de quatre éléments essentiels:

1. Le *nom* du patron est un moyen de décrire en un ou deux mots un problème de conception, ses solutions et leurs conséquences. Ceci permet de travailler a un degré d'abstraction plus élevé et de disposer d'un vocabulaire pour la documentation et la communication des conceptions.
2. Le *problème* décrit les situations où le patron s'applique. Il expose le sujet à traiter et son con-

texte. Le problème comporte parfois une liste de conditions à satisfaire pour que le patron s'applique adéquatement.

3. La *solution* décrit les éléments qui constituent la conception, les relations entre eux, leur part dans la solution et leur coopération.
4. Les *conséquences* sont les effets résultants de la mise en oeuvre du patron et les variantes de compromis que celle-ci entraîne. Ces conséquences sont déterminantes pour l'évaluation des alternatives de conception et pour l'appréciation des avantages et des inconvénients de l'application du patron de conception.

Dans ce qui suit, nous allons décrire deux patrons de conception qui vont être utilisés dans le chapitre 6 et dans l'annexe H: Observer et Mediator. Mais au préalable nous allons montrer l'intérêt des patrons de conception dans les transformations.

2.2.1 Intérêt des patrons de conception dans les transformations

Nous avons déjà vu dans le premier chapitre les différentes phases du cycle de vie des systèmes. En particulier, nous avons vu qu'au cours de la phase d'analyse l'emphase est mise sur ce qui devrait être fait, le quoi, indépendamment de la manière de le faire, le comment. Par contre, au cours de la conception, des décisions doivent être prises concernant la façon de résoudre le problème, d'abord à un niveau général, puis à des niveaux de détail de plus en plus fins.

À un niveau général, le concepteur du système doit décomposer le système en sous-systèmes, allouer les sous-systèmes aux processeurs et aux tâches, choisir une approche de gestion des données, traiter le partage des accès aux ressources globales, etc. À un niveau plus détaillé, le concepteur doit donner une description détaillée de toutes les classes qui composent son système. Plus précisément, il doit concevoir des algorithmes pour les opérations, concevoir les associations, ajuster la structure des classes pour accroître l'héritage, etc.

Il y a un consensus autour du fait que les patrons de conception offrent de bonnes solutions à un grand ensemble de problèmes qui peuvent survenir dans l'étape de conception comme ceux mentionnés ci-haut [Buschmann et al., 1996; Gamma et al., 1995; Rumbaugh et al., 1999]. En particulier, et comme nous allons voir par la suite, nous voyons la place des patrons de conception dans la conception détaillée. Ainsi, cette étape revient à réaliser des transformations lors des raffinements du modèle résultant de l'étape de conception générale. Pour cela, nous avons besoin de décrire de façon rigoureuse les transformations qui résultent dans les occurrences des patrons de conception. À titre d'exemple, supposons que dans le modèle de conception générale, on trouve

que deux classes sont liées par une contrainte. Le concepteur doit décider de la façon avec laquelle il va réaliser cette contrainte. Le concepteur n'aura alors qu'à appliquer le patron Observer qui offre une solution élégante à ce problème.

Notons qu'il existe deux autres types de patrons utiles pour d'autres étapes du cycle de vie, à savoir les patrons architecturaux [Buschmann et al., 1996] lors de la conception générale et les patrons spécifiques à un domaine (aussi appelés d'analyse) [Coad et al., 1995; Fowler, 1997] lors de l'étape d'analyse.

2.2.2 Observer

Le patron Observer définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour [Gamma et al., 1995].

Le patron Observer peut être utilisé dans les situations suivantes:

- Quand un concept a deux représentations dont une dépend de l'autre. Encapsuler ces deux représentations dans des objets distincts permet de les réutiliser et de les modifier indépendamment.
- Quand la modification d'un objet nécessite de modifier d'autres objets dont on ne connaît pas le nombre.
- Quand un objet doit être capable de faire une notification à d'autres objets, sans faire d'hypothèses sur la nature de ces autres objets.

La figure 2.6 montre la structure du patron Observer. Il est constitué de quatre classes: `Subject`, `Observer`, `ConcreteSubject`, et `ConcreteObserver`. Un objet de la classe `Subject` (sujet) a un nombre quelconque d'observateurs (`Observer`) et fournit une interface pour attacher et détacher les objets observateurs. La classe `Observer` définit une interface de mise à jour pour les objets qui doivent être notifiés de changements dans un sujet. Un objet de la classe `ConcreteSubject` mémorise les états qui intéressent les objets `ConcreteObserver` et envoie une notification à ses observateurs lorsqu'il change d'états. Un objet de la classe `ConcreteObserver` gère une référence sur un objet `ConcreteSubject`, mémorise l'état qui doit rester pertinent pour le sujet, et fait l'implémentation de l'interface de mise à jour de l'observateur pour conserver la cohérence de son état avec le sujet.

La figure 2.7 décrit un exemple de collaboration typique entre les objets que comprend le patron Observer. L'objet `aConcreteSubjet` notifie ses observateurs de tout changement se produisant qui pourrait rendre l'état de ses observateurs incompatible avec le sien. Après avoir été informé d'un changement sur son sujet, un objet de la classe `ConcreteObserver` fait une demande d'information au sujet pour mettre son état en conformité avec celui-ci.

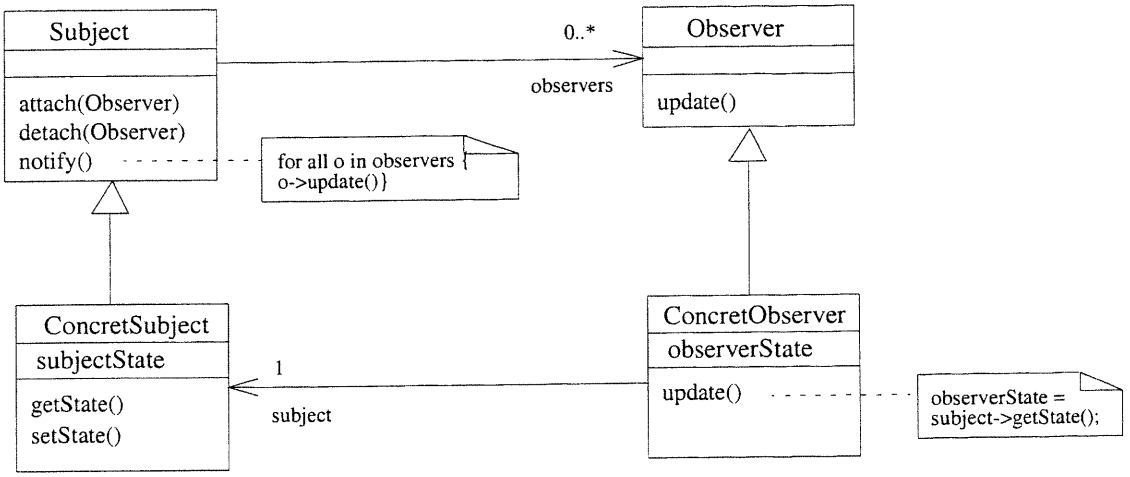


Figure 2.6: La structure du patron Observer

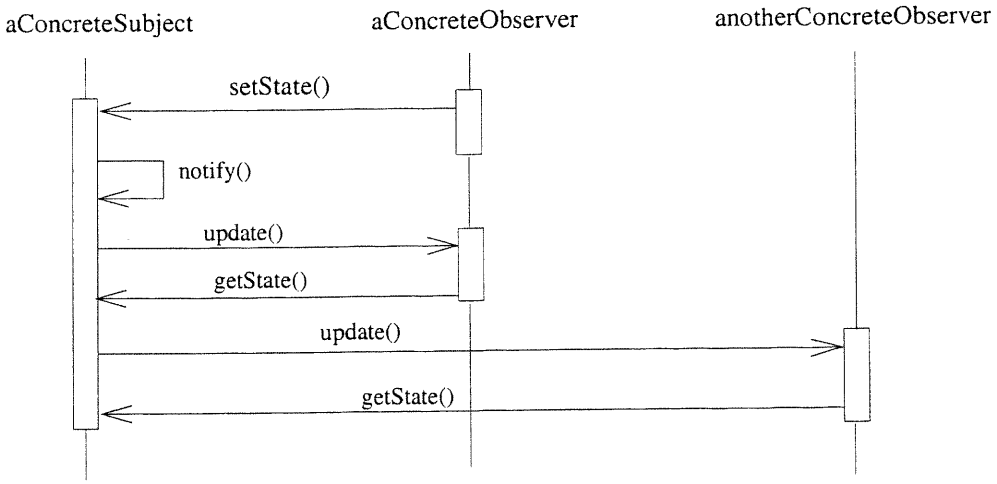


Figure 2.7: Collaboration entre les objets du patron Observer

Les avantages et les inconvénients du patron Observer sont les suivants:

- Le couplage entre sujets et observateurs est abstrait et minimal. En effet, tout ce que sait un

sujet est qu'il possède une liste d'observateurs, chacun de ceux-ci se conformant à l'interface simple de la classe abstraite `Observer`.

- La notification émise par un sujet n'a pas à spécifier ses destinataires. La notification est automatiquement diffusée à tous les objets intéressés qui ont souscrit. Le sujet ne s'occupe pas du nombre des objets intéressés; sa seule responsabilité est de notifier ses observateurs. Ceci donne la liberté d'ajouter ou de retrancher, à tout instant, des observateurs.
- Du fait que les observateurs ignorent la présence les uns des autres, ils peuvent être complètement aveugles au coût final d'une modification d'un sujet. Une opération sur le sujet peut engendrer une cascade de mises à jour sur les observateurs et les objets qui en dépendent.

2.2.3 Mediator

Le patron Mediator définit un objet qui encapsule les modalités d'interaction d'un certain ensemble d'objets [Gamma et al., 1995]. Il favorise le couplage faible en dispensant les objets de se faire explicitement référence, et il permet donc de faire varier indépendamment les relations d'interaction.

Le patron Mediator peut être utilisé lorsque:

- Les objets d'un ensemble communiquent d'une façon bien définie mais très complexe.
- La réutilisation d'un objet est difficile, du fait qu'il fait référence à beaucoup d'autres objets et communique avec eux.
- Un comportement distribué entre plusieurs classes doit pouvoir être spécialisé sans une pléthore de dérivations.

La figure 2.8 montre la structure du patron Mediator. La classe `Mediator` définit une interface pour communiquer avec les objets collègues. La classe `ConcreteMediator` réalise le comportement coopératif en coordonnant les objets de la classe `Colleague`. Chaque classe `Colleague` connaît son objet `Mediator` et s'adresse à celui-ci à chaque fois qu'il veut communiquer avec un autre collègue.

Les avantages et les inconvénients du patron Mediator sont les suivants:

- Il limite la création de sous-classes en regroupant les éléments d'un comportement, qui autrement auraient été distribués parmi plusieurs objets. Le changement de ce comportement ne requiert donc que de dériver une sous-classe de la classe Mediator.

- Il réduit le couplage entre collègues.
- Il simplifie les protocoles objet en remplaçant des interactions plusieurs à plusieurs par des interactions un à plusieurs entre un Mediator et ses collègues.
- Il centralise le contrôle ce qui peut conduire à un objet Mediator complexe.

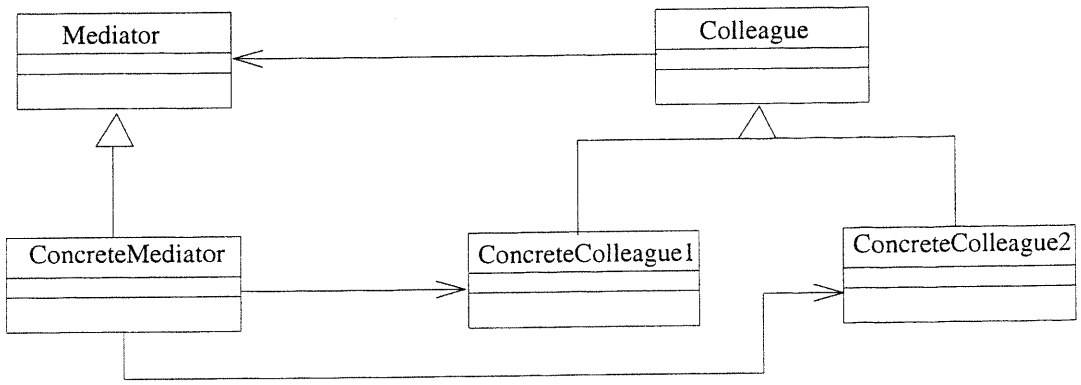


Figure 2.8: La structure du patron Mediator

Chapitre 3 Aperçu de l'approche de transformation

Dans cette thèse nous nous sommes attaqués à deux objectifs pour la réalisation d'une approche transformationnelle dans le développement des systèmes logiciels [1]. Le premier objectif consiste à offrir un support transformationnel au processus d'ingénierie des besoins basé sur les scénarios. Le deuxième objectif cherche à appliquer automatiquement les patrons de conception dans le but de rendre systématique leur implantation dans les systèmes OO et d'éliminer la nature élaborative de la conception OO. Comme langage de modélisation, nous avons utilisé le langage de modélisation unifié UML qui offre une notation standardisée pour la description des systèmes OO en développement.

Un scénario est une description partielle ou complète des interactions entre un usager et le système pour accomplir une tâche spécifique [Booch, 1994]. Les scénarios sont considérés comme une bonne technique pour capturer les besoins des usagers parce que ces derniers les utilisent de façon naturelle pour décrire le comportement souhaité d'un système. Dans la première section de ce chapitre, nous allons en premier lieu décrire les différentes tâches du processus d'ingénierie des besoins à l'aide des scénarios. Ensuite nous allons montrer comment nous supportons ces tâches par une approche transformationnelle.

Dans la deuxième section, nous allons décrire le deuxième type de transformation que nous supportons à savoir l'intégration de la conception à haut niveau avec celle détaillée par l'application automatique des patrons de conception.

1. Notons que notre travail ne couvre pas la totalité d'un processus de développement logiciel. Voir [Jacobson et al., 1999] pour une vue globale d'un processus de développement par élaboration avec le langage UML.

3.1 Support de l'ingénierie des besoins basé sur les scénarios

Un processus typique pour l'ingénierie des besoins basé sur les scénarios est composé de cinq étapes (voir figure 3.1): acquisition des scénarios, génération de la spécification, vérification de la spécification, génération d'un prototype, et enfin validation du prototype avec les utilisateurs. En premier lieu, l'analyste commence par acquérir les scénarios des utilisateurs. Puis, une spécification qui décrit le comportement dynamique du système est synthétisée à partir des scénarios. Ensuite, l'analyste vérifie la spécification dans le but de détecter et corriger des scénarios incohérents et/ou incomplets. Suit la génération d'un prototype du système à partir de la spécification. Finalement, la cinquième étape a pour rôle de valider le prototype avec les utilisateurs. En cas de scénarios invalides détectés lors de la troisième étape ou de la dernière étape, l'analyste retourne à la première étape et recommence les différentes étapes jusqu'à ce que tous les scénarios soient valides.

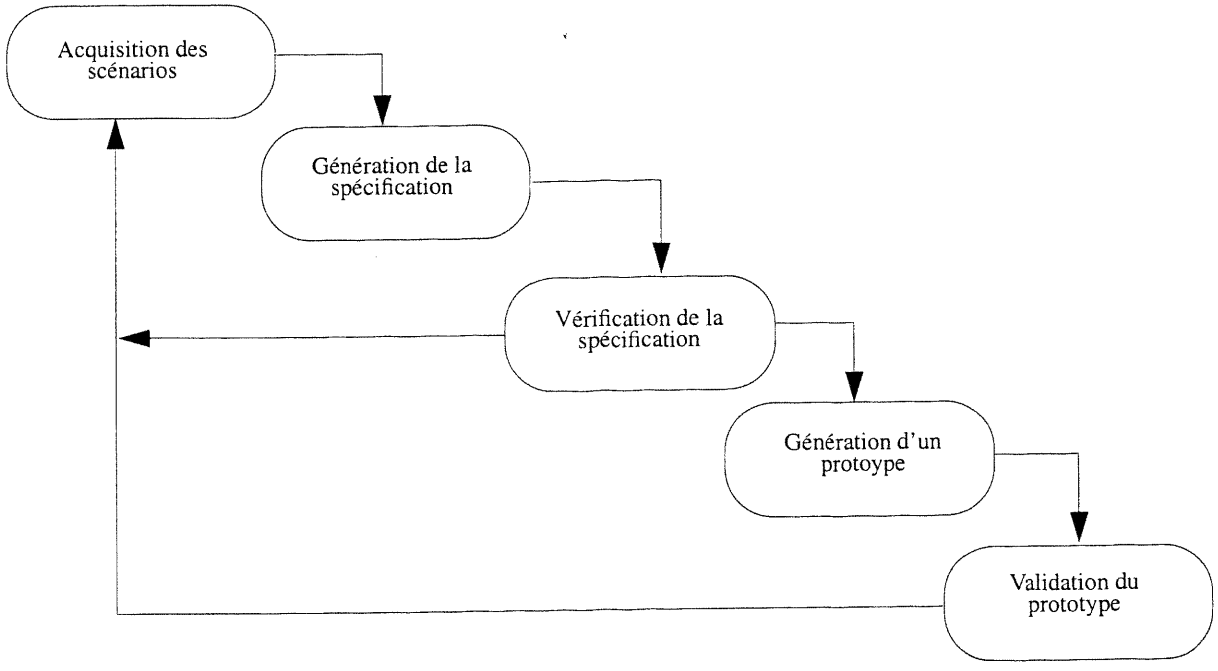


Figure 3.1: Processus d'ingénierie des besoins basé sur les scénarios (adapté de [Hsia et al., 1994])

Tant que ce processus reste non supporté par des outils automatiques, il constitue une tâche difficile qui prend beaucoup du temps et qui est exposée aux erreurs. C'est pourquoi notre premier

objectif consiste à supporter ce processus par des outils automatiques. En particulier, nous voulons automatiser les trois étapes au milieu du processus à savoir les étapes 2, 3 et 4. Pour cela, nous avons développé une approche transformationnelle afin de spécifier le comportement dynamique du système à partir des scénarios et d'obtenir un prototype de l'interface usager (IU). Ce prototype peut être simulé et permet ainsi la validation des scénarios avec les usagers. Notons que le comportement dynamique d'un système est décrit par le comportement des objets qu'il englobe. Notre approche transformationnelle est composée de cinq activités (voir figure 3. 2), qui vont être décrites ci-dessous:

1. Acquisition des scénarios,
2. Génération du comportement partiel des objets,
3. Analyse du comportement partiel des objets,
4. Intégration des comportements partiels des objets,
5. Génération et validation du prototype de l'IU.

3.1.1 Acquisition des besoins

UML propose un bon cadre pour l'acquisition des scénarios par l'utilisation des UseCaseDs pour capturer les fonctionnalités d'un système et les diagrammes de suivi d'événements ou les CollDs pour la description des scénarios.

Dans cette activité, l'analyste commence par l'élaboration du UseCaseD et du ClassD du système. La figure 2.1 montre le ClassD du système GAB introduit dans la section 2.1.2 alors que la figure 2.2 montre son UseCaseD. Puis une analyse détaillée est effectuée pour chaque classe du ClassD dans le but d'identifier les attributs et les opérations avec leurs pre- et post-conditions respectives. À titre d'exemple, la classe GAB est représenté dans la figure 3.3. Finalement, l'analyste acquiert, pour chaque cas d'utilisation du UseCaseD, les différents scénarios sous forme de CollDs. Les figures 3.4 et 3.5 décrivent deux scénarios pour le cas d'utilisation `Retrait` du système GAB. Remarquons que le scénario présenté dans la figure 3.4 est augmenté par rapport à celui de la figure 2.3 par des contraintes que nous expliquerons par la suite.

Les scénarios d'un cas d'utilisation donné sont classés par leur type et leur fréquence d'utilisation. Nous considérons deux types de scénarios: les *scénarios réguliers* qui sont exécutés dans des situations régulières et les *scénarios d'exception* qui sont exécutés dans des cas d'exception comme dans des cas d'erreurs ou dans des situations anormales. La fréquence d'utilisation d'un scénario est un chiffre variant entre *un* et un seuil supérieur fixé à *dix* dans nos travaux et sera

affecté par l'analyste. Dans notre cas, le cas d'utilisation `Retrait` possède un scénario régulier `RetraitRégulier` avec une fréquence égale à dix et un scénario d'exception `RetraitAvecErreurNIP` ayant une fréquence égale à quatre. Cette classification est utilisée dans l'algorithme de génération d'un prototype de l'IU (cf. section 7.1).

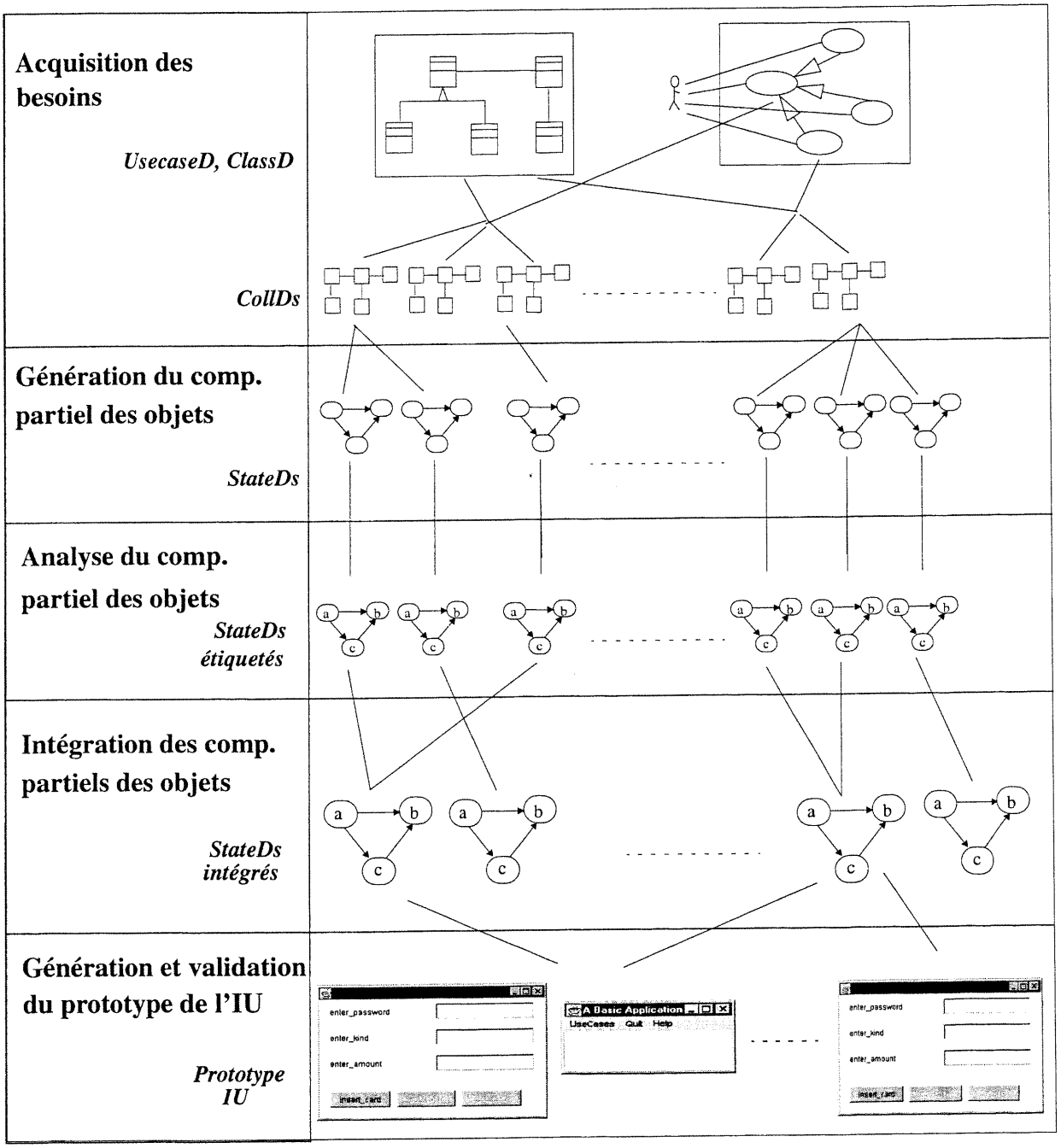


Figure 3.2: Vue générale de l'approche

GAB

liquide_disponible: boolean = true

écran: string = "principale"

fente_liquide: string = "fermée"

fente_carte: string = "vide"

insérer_carte(): string
pre: liquide_disponible=true and écran="principale" and fente_argent="fermée" and fente_carte="vide"
post: liquide_disponible=true and écran="entrer mot de passe" and fente_argent="fermée" and fente_carte="pleine"

entrer_motdepasse(): string
pre: liquide_disponible=true and écran="entrer mot de passe" and fente_argent="fermée" and fente_carte="pleine"
post: liquide_disponible=true and écran="confirmer mot de passe" and fente_argent="fermée" and fente_carte="pleine"

entrer_opération(): string
pre: liquide_disponible=true and écran="entrer type opération" and fente_argent="fermée" and fente_carte="pleine"
post: liquide_disponible=true and (écran="Dépôt" or écran="Retrait") and fente_argent="fermée" and fente_carte="pleine"

entrer_montant(): float
pre: liquide_disponible=true and (écran="Dépôt" or écran="Retrait") and fente_argent="fermée" and fente_carte="pleine"
post: liquide_disponible=true and écran="confirmer type et montant" and fente_argent="fermée" and fente_carte="pleine"

délivrer_argent(m: float)
pre: liquide_disponible=true and écran="Retrait en cours" and fente_argent="fermée" and fente_carte="pleine"
post: liquide_disponible=true and (écran="prendre_argent" or écran="Fond insuffisant") and (fente_argent="ouverte or fente_argent="fermée") and fente_carte="pleine"

prendre_argent()
pre: liquide_disponible=true and écran="prendre_argent" and fente_argent="ouverte" and fente_carte="pleine"
post: liquide_disponible=true and écran="prendre_carte" and fente_argent="fermée" and fente_carte="retirée"

prendre_carte()
pre: liquide_disponible=true and écran="prendre_carte" and fente_argent="fermée" and fente_carte="retirée"
post: liquide_disponible=true and écran="principale" and fente_argent="fermée" and fente_carte="vide"

afficher_erreur()
pre: liquide_disponible=true and écran="Fonds insuffisants" and fente_argent="fermée" and fente_carte="pleine"
post: liquide_disponible=true and écran="prendre_carte" and fente_argent="fermée" and fente_carte="retirée"

confirmer_motdepasse()
pre: liquide_disponible=true and écran="confirmer mot de passe" and fente_argent="fermée" and fente_carte="pleine"
post: liquide_disponible=true and (écran="Mot de passe incorrecte" or écran="entrer type opération") and fente_argent="fermée" and fente_carte="pleine"

confirmer_typeETmontant()
pre: liquide_disponible=true and écran="confirmer type et montant" and fente_argent="fermée" and fente_carte="pleine"
post: liquide_disponible=true and (écran="Fond insuffisants" or écran="Retrait en cours" écran="Dépôt en cours") and fente_argent="fermée" and fente_carte="pleine"

Figure 3.3: La Classe GAB

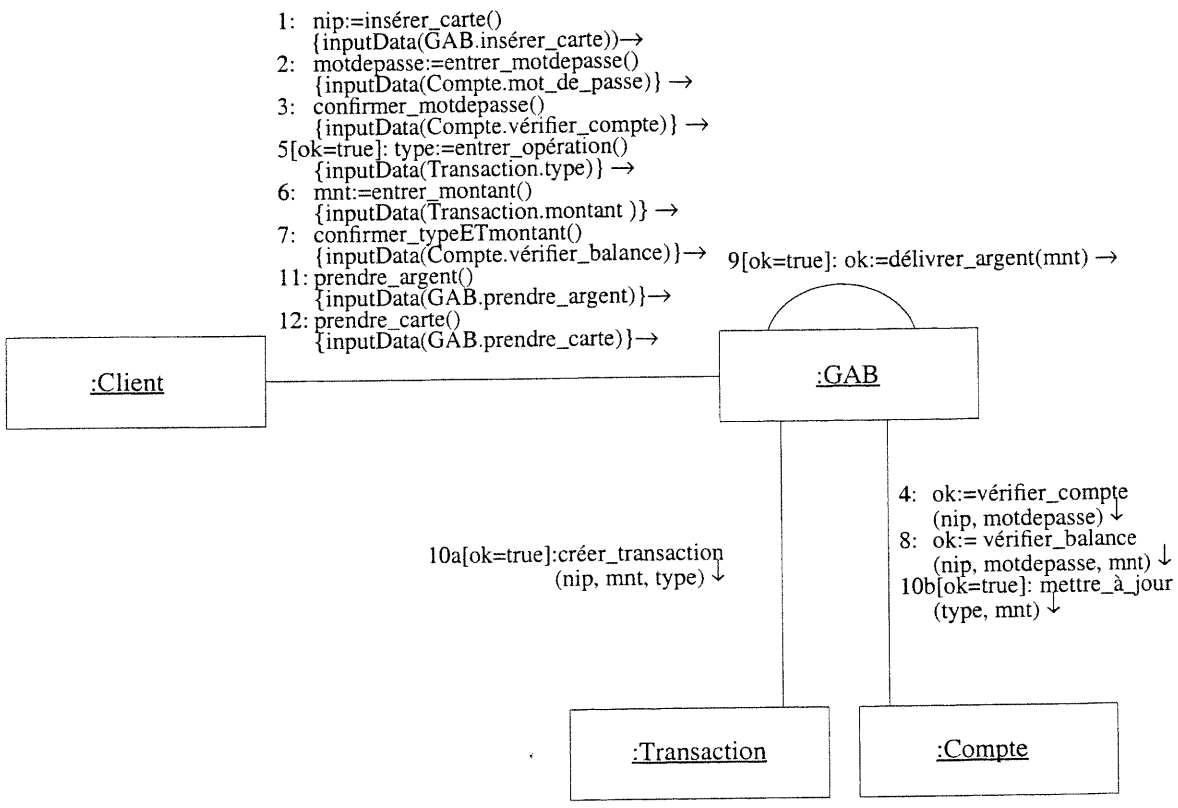


Figure 3.4: Le CollD du scénario `retraitRégulier` du système `GAB`

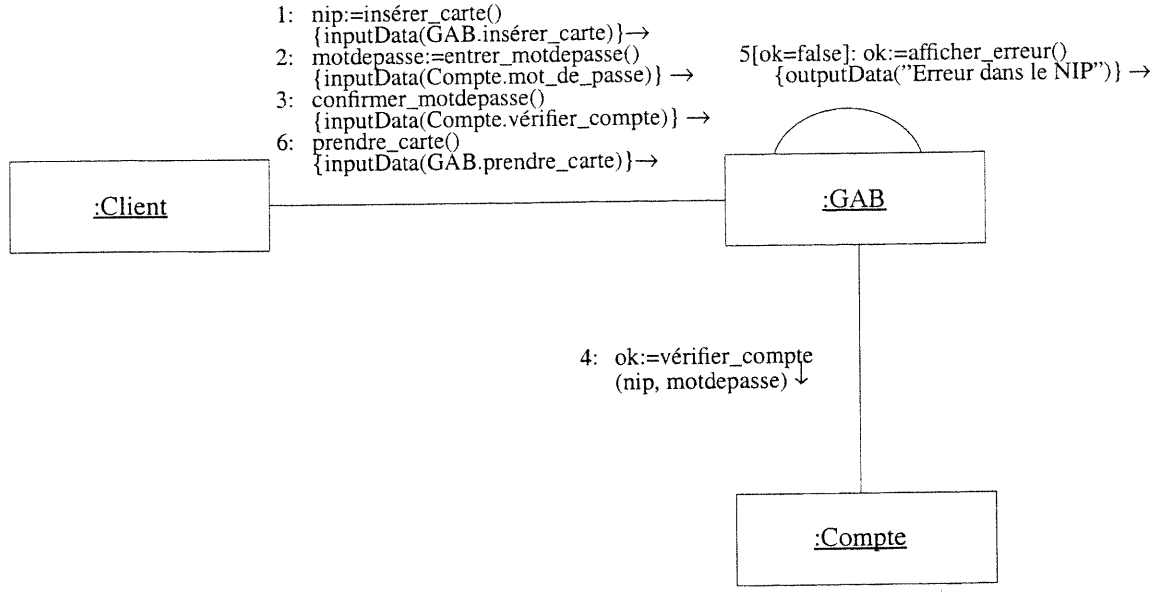


Figure 3.5: Le CollD du scénario `retraitAvecErreurNIP` du système `GAB`

Dans le système GAB, l'objet GAB est un objet spécial appelé *objet d'interface*. Un objet d'interface est un objet à travers lequel l'utilisateur interagit avec le système pour rentrer des données et recevoir des résultats. Un *message interactif* dans un CollD est défini comme un message qui est envoyé à un objet d'interface. Pour les besoins de génération du prototype de l'IU, nous définissons deux contraintes qui peuvent être associées aux messages interactifs: inputData et outputData. La contrainte inputData indique que le message correspondant est le résultat d'une entrée de l'utilisateur alors que la contrainte outputData indique que le message correspondant concerne une sortie à afficher dans l'IU.

Une contrainte inputData ou outputData attachée à un message définit aussi une relation de dépendance entre ce message et une opération ou un attribut d'une classe. Cette opération ou cet attribut représentent la partie *modèle* de l'architecture *MVC* (Model View Controller) [Gamma et al., 1995] qui sépare la partie présentation d'un système, c'est-à-dire son IU, de la partie domaine d'application. Dans le CollD de la figure 3.4 par exemple, le message `2:motdepasse:=entrer_motdepasse()` a une contrainte de type inputData avec l'attribut `mot_de_passe` de la classe `Compte`.

En fait, les messages des CollDs annotés avec des contraintes les reliant avec le ClassD du système vont permettre de faire une sélection automatique des types d'objets d'interaction (*widgets*) que nous allons retrouver dans les différentes fenêtres de l'IU. Cette sélection se base sur les différentes règles que nous avons pu tirer de la littérature [Bodart et al., 1994; IBM 1991; Maher, 1994]. Ci-dessous nous donnons une liste de règles (règles 3.1 à 3.7) que nous supportons dans notre implantation actuelle:

Règle 3.1. Si la contrainte d'un message interactif est de type inputData et le lien est une méthode d'une classe alors l'objet d'interaction est un bouton.

Règle 3.2. Si la contrainte d'un message interactif est de type inputData, le lien est un attribut d'une classe, et le type de l'attribut est une énumération de taille inférieure à six alors l'objet d'interaction est un bouton radio.

Règle 3.3. Si la contrainte d'un message interactif est de type inputData, le lien est un attribut d'une classe, et le type de l'attribut est une énumération de taille supérieure à six alors l'objet d'interaction est une liste active.

Règle 3.4. Si la contrainte d'un message interactif est de type inputData, le lien est un attribut d'une classe, et le type de l'attribut n'est pas une énumération alors l'objet d'interaction est un

champ de saisie (*Input Field*).

Règle 3.5. Si la contrainte d'un message interactif est de type outputData et le lien est une constante (chaîne de caractère) alors l'objet d'interaction est une étiquette (*Label*).

Règle 3.6. Si la contrainte d'un message interactif est de type outputData, le lien est un attribut d'une classe, et le type de l'attribut est soit une énumération de taille inférieure à six soit d'un autre type alors l'objet d'interaction est un champ de texte (*Text Field*).

Règle 3.7. Si la contrainte d'un message interactif est de type outputData, le lien est un attribut d'une classe, et le type de l'attribut est une énumération de taille supérieure à six alors l'objet d'interaction est une liste passive.

En appliquant ces règles sur les CollDs des figures 3.4 et 3.5 nous obtenons les CollDs décrits dans les figures 3.6 et 3.7 respectivement.

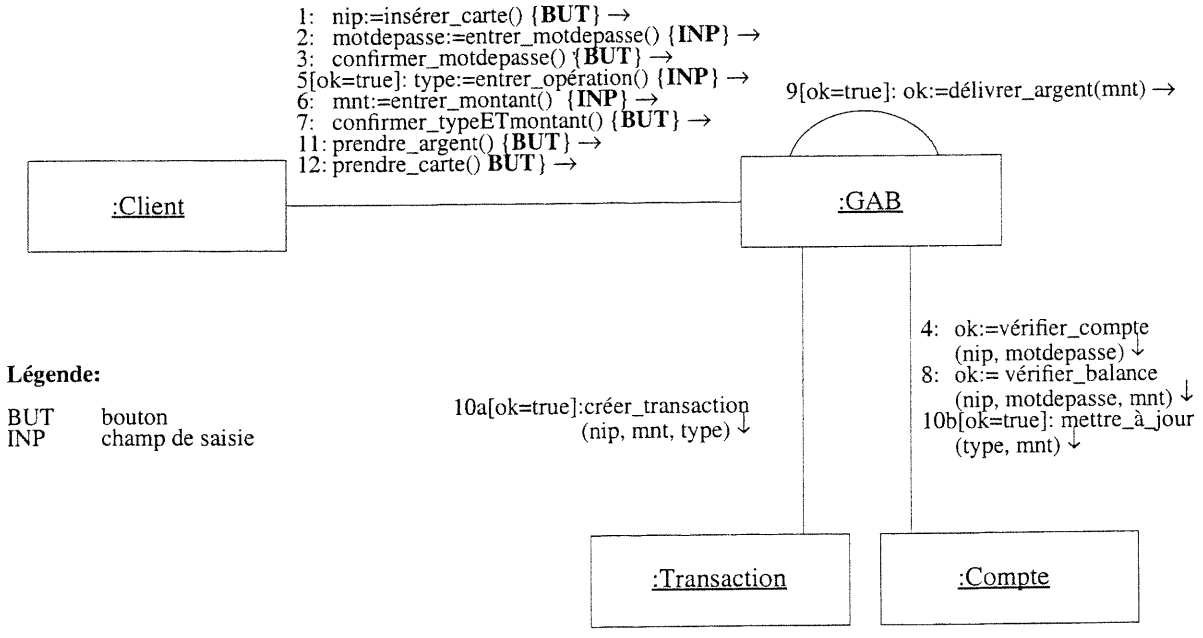


Figure 3.6: Le CollD du scénario `retraitRégulier` après sélection automatique des objets d'interaction

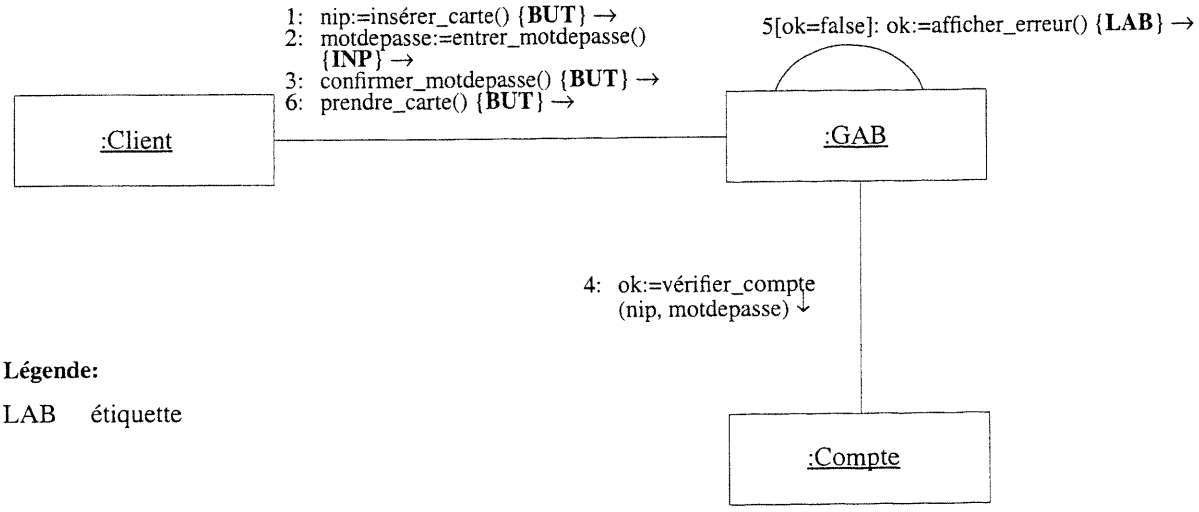


Figure 3.7: Le CollD du scénario `retraitAvecErreurNIP` après sélection automatique des objets d'interaction

3.1.2 Génération du comportement partiel des objets

Dans cette activité, nous appliquons pour tous les CollDs l'algorithme `CTS` (CollD To StateD) dans le but de générer des spécifications partielles des objets participant dans les différents scénarios.

La transformation d'un CollD en StateDs est un processus composé de cinq étapes. La première étape créé un StateD pour chaque classe dont les objets sont impliqués dans le CollD. La deuxième étape introduit comme variables d'états toutes les variables du CollD qui ne sont pas des attributs d'objets. La troisième étape crée des transitions pour les objets émetteurs des messages alors que la quatrième étape les crée pour les objets récepteurs de messages. Le rôle de la dernière étape est de relier dans le bon ordre les différentes transitions nouvellement créées par les deux étapes précédentes. Le séquencement suit le type d'un message dans un CollD: itération, conditionalité, concurrence et messages à prédécesseurs multiples. Dans le prochain chapitre nous allons décrire en détail l'algorithme `CTS`.

Après l'application de l'algorithme `CTS` aux scénarios `RetraitRégulier` et `RetraitAvecErreurNIP`, nous obtenons pour l'objet `GAB` les StateDs partiels décrits dans les figures 3.8 et 3.9 respectivement.

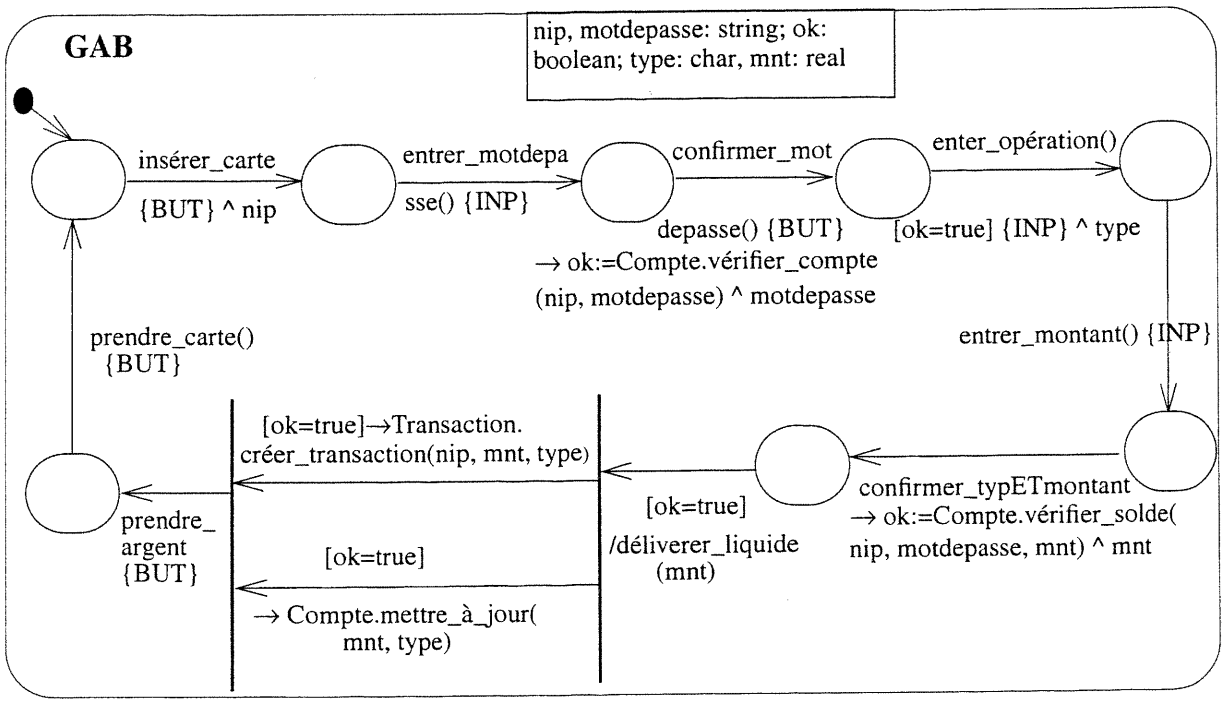


Figure 3.8: StateD de la classe `GAB` généré du scénario `RetraitRégulier` de la figure 3.6

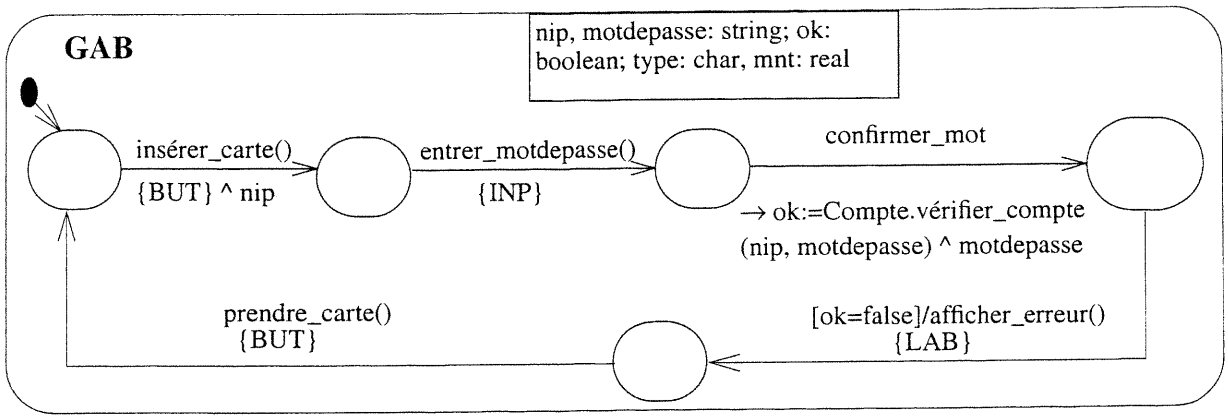


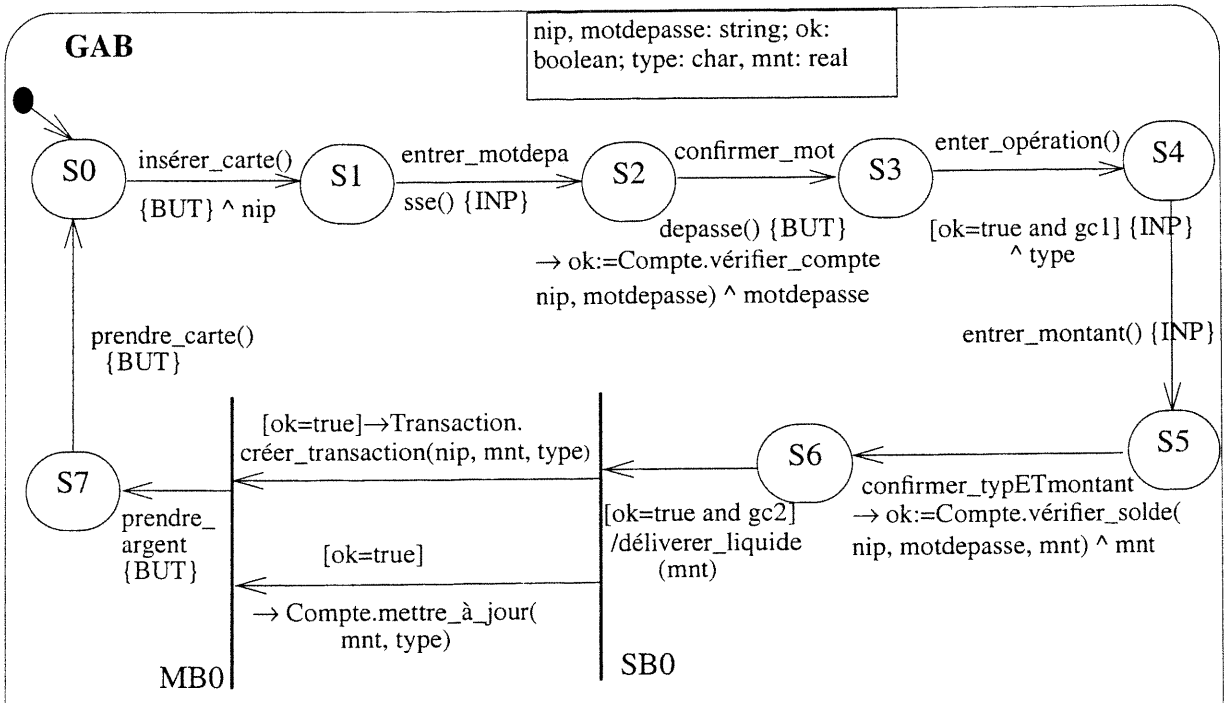
Figure 3.9: StateD de la classe `GAB` généré du scénario `RetraitAvecErreurNIP` de la figure 3.7

3.1.3 Analyse des spécifications partielles

Les StateDs partiels obtenus dans l'étape précédente ne sont pas étiquetés, c'est-à-dire, les états ne possèdent pas de noms. Or, l'algorithme d'intégration (cf. section 3.1.4), basé sur les états, exige des StateDs étiquetés. En outre, les scénarios acquis peuvent être incohérents ou incomplets. C'est pourquoi nous avons conçu un algorithme d'analyse qui a pour rôle d'étiqueter un

StateD tout en vérifiant sa cohérence et sa complétude. En fait, cette vérification va aussi assurer la cohérence et la complétude du StateD synthétisé à partir des StateDs étiquetés.

Cet algorithme est basé principalement sur les pre- et post-conditions des opérations des classes du système en question. Dans la section 5.1 nous allons présenter les détails de cet algorithme. En appliquant l'algorithme d'analyse au StateD de la figure 3.8 nous obtenons le StateD présenté dans la figure 3.10 et annoté avec des étiquettes expliquées dans la légende. Pour le StateD de la figure 3.9 nous obtenons le StateD présenté dans la figure 3.11.



Légende:

S0 = liquide_disponible=true and écran="entrer type opération" and fente_argent="fermée" and fente_carte="pleine"
S1 = liquide_disponible=true and écran="entrer mot de passe" and fente_argent="fermée" and fente_carte="pleine"
S2 = liquide_disponible=true and écran="confirmer mot de passe" and fente_argent= "fermée" and fente_carte="pleine"
S3 = liquide_disponible=true and (écran="entrer type opération" or écran="Mot de passe incorrecte") and fente_argent="fermée" and fente_carte="pleine"
S4 = liquide_disponible=true and (écran="Dépôt" or écran="Retrait") and fente_argent= "fermée" and fente_carte="pleine"
S5 = liquide_disponible=true and écran="confirmer type et montant" and fente_argent= "fermée" and fente_carte="pleine"
S6 = liquide_disponible=true and (écran="Dépôt en cours" or écran="Retrait en cours" or écran="Fonds insuffisants") and fente_argent="fermée" and fente_carte="pleine"
S7 = liquide_disponible=true and écran="prendre_carte" and fente_argent="fermée" and fente_carte="retirée"
gc1 = liquide_disponible=true and écran="entrer type opération" and fente_argent="fermée" and fente_carte="pleine"
gc2 = liquide_disponible=true and écran="Retrait en cours" and fente_argent="fermée" and fente_carte="pleine"

Figure 3.10: StateD étiqueté obtenu à partir du StateD de la figure 3.8

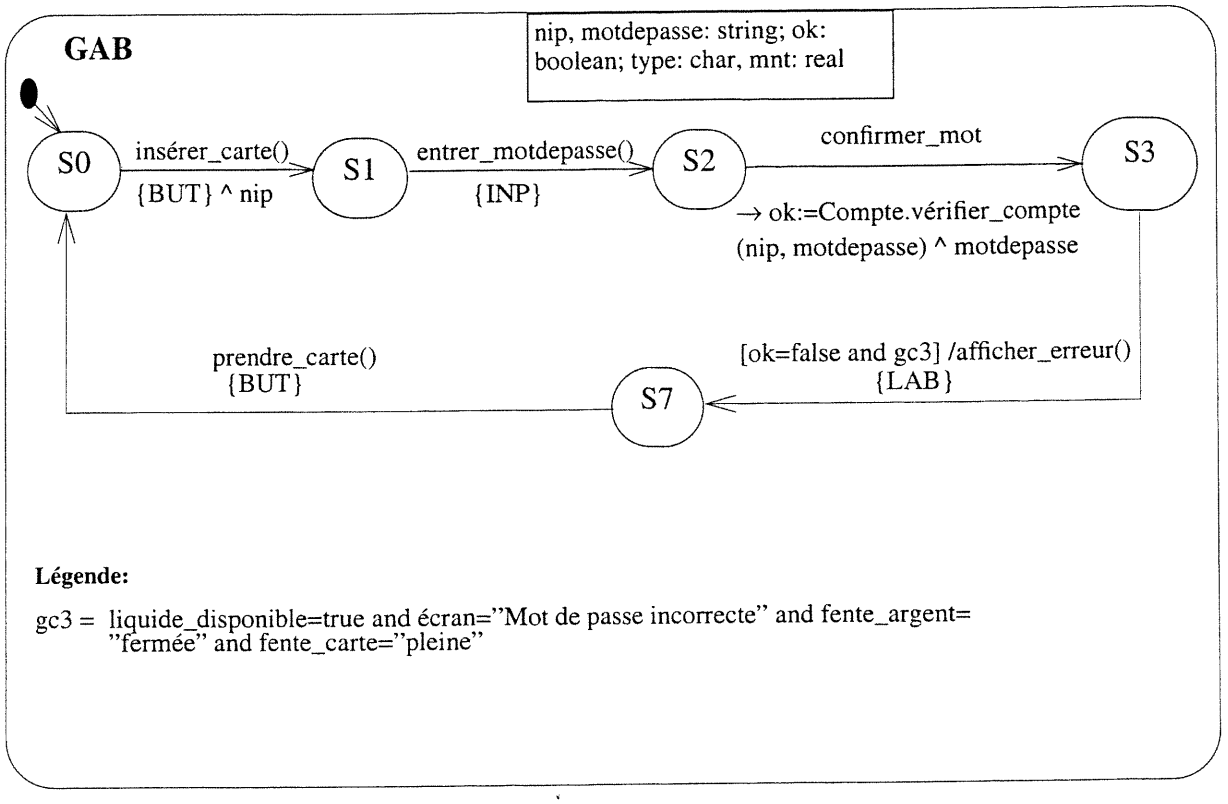


Figure 3.11: StateD étiquetée obtenu à partir du StateD de la figure 3.9

3.1.4 Intégration des spécifications partielles des objets

L'objectif de cette activité est d'intégrer pour chaque objet du système tous ses StateDs partiels en un seul StateD. Initialement, nous procédons à une intégration par cas d'utilisation puis nous fusionnons les StateDs des différents cas d'utilisation. Cette intégration en deux étapes est requise pour l'activité de la génération d'un prototype de l'IU que nous allons voir dans la prochaine section. En plus de l'intégration, cette activité permet la vérification de la cohérence des StateDs résultants. Cette vérification a pour but de trouver d'éventuelles incohérences entre les différents scénarios à intégrer. L'algorithme sous-jacent à cette activité sera décrit en détail dans la section 5.2. La figure 3.12 montre le résultat de l'intégration des deux StateDs partiels de la classe GAB, qui sont présentés dans les figures 3.10 et 3.11 respectivement et qui correspondent aux deux scénarios du cas d'utilisation `Retrait`.

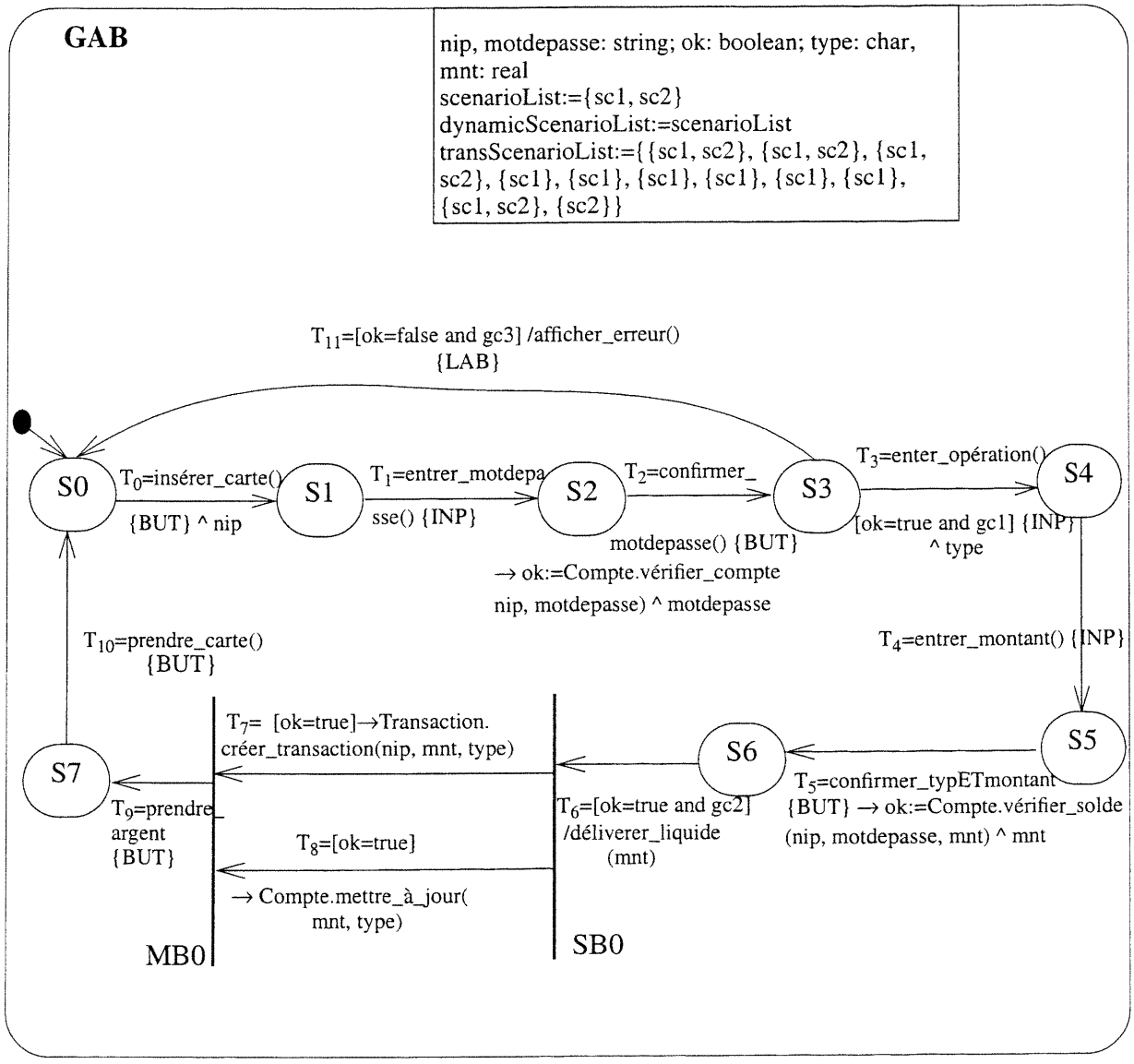


Figure 3.12: Le StateD résultant après intégration des StateDs des figures 3.10 et 3.11

3.1.5 Génération et validation d'un prototype de l'interface usager

Dans cette activité, un prototype de l'IU est dérivé pour tous les objets d'interface du système à partir de leurs StateDs respectifs. Non seulement l'aspect statique du prototype mais aussi son aspect dynamique sont générés; ce qui permet la simulation des différents scénarios pour pouvoir les valider avec les utilisateurs.

Pour chaque objet d'interface, le prototype généré comprend un menu pour que l'utilisateur puisse passer d'un cas d'utilisation à un autre ainsi que les différentes fenêtres et boîtes de dialogues nécessaires pour accéder aux fonctionnalités du système. Le contrôle du dialogue des prototypes correspond aux StateDs des objets d'interface. Dans l'implantation actuelle de notre algorithme, les prototypes sont des applications en Java acceptés par le constructeur d'IU Visual Café [Symantec, 1997]. Ceci facilite une éventuelle personnalisation de l'IU obtenu.

Appliqué au StateD de la figure 3.12, l'algorithme de génération produit un prototype qui comprend les fenêtres présentées dans la figure 3.13. La figure 3.14 montre le menu d'accès aux différents cas d'utilisation. En plus du cas d'utilisation `Retrait`, il y a une option pour l'accès au cas d'utilisation `Dépôt` que nous n'avons pas décrit dans ce document. L'algorithme de génération sera présenté en détail dans la section 7.1.

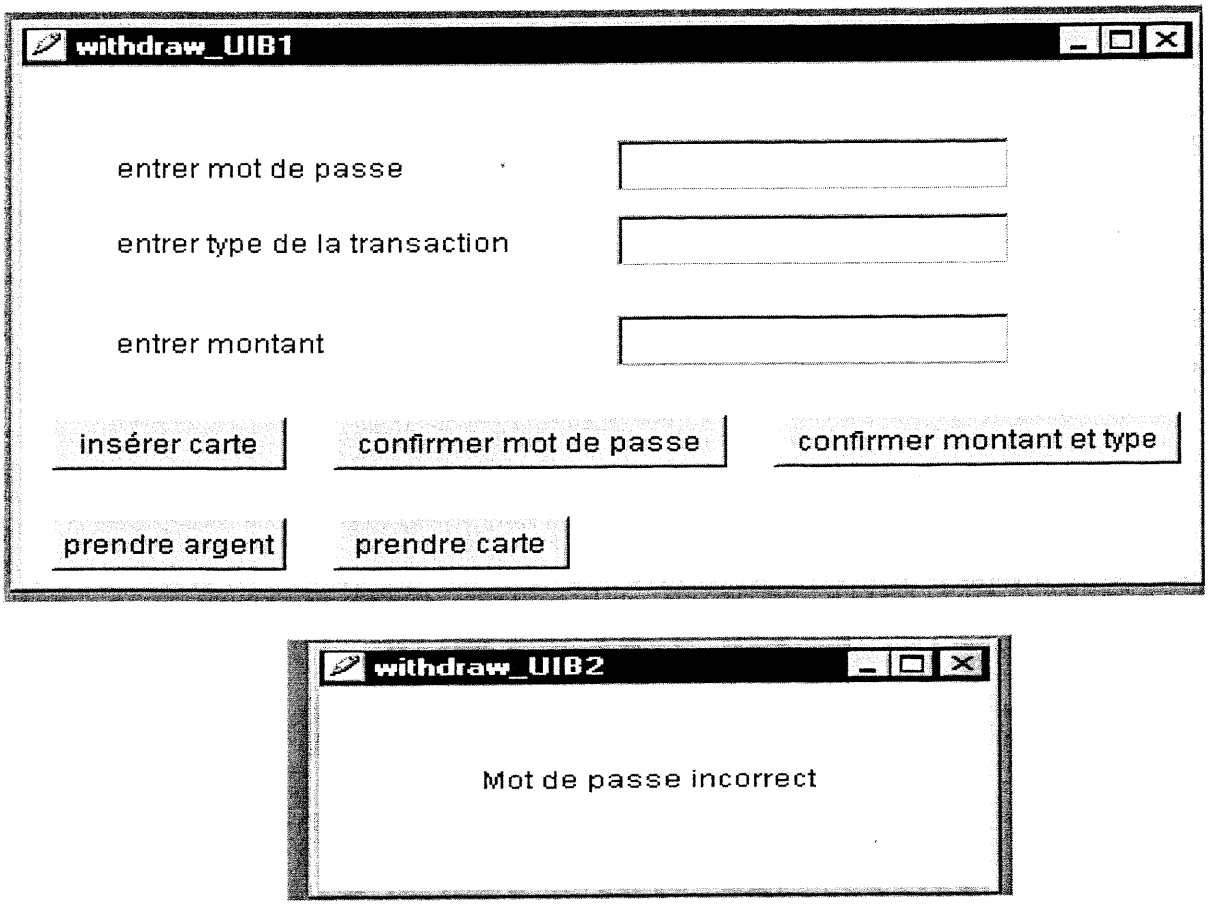


Figure 3.13: Les deux fenêtres générées pour le cas d'utilisation `Retrait`

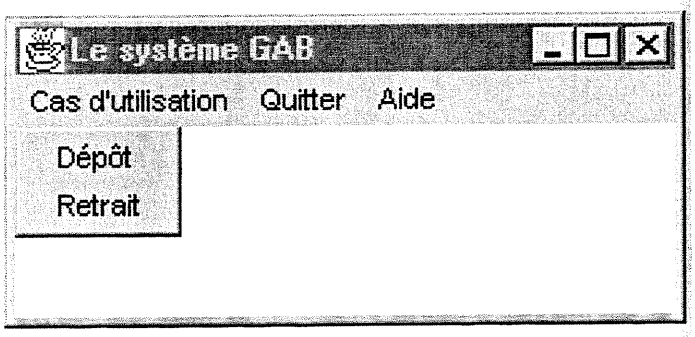


Figure 3.14: Le menu d'accès aux cas d'utilisation

3.2 Application automatique des patrons de conception

3.2.1 Description de l'approche

L'objectif de notre approche est de fournir un support automatique pour la transition d'une conception générale vers une conception détaillée. Cette transformation est réalisée par des raffinements successifs où chaque étape est prouvée correcte. Chaque raffinement est basé sur l'application d'un patron de conception. Dans chaque étape de transformation (voir figure 3.15), le concepteur commence par analyser le modèle de conception dans le but de choisir un schéma de raffinement approprié. Le modèle de conception se présente dans un ClassD, des StateDs et des CollDs. Ensuite, le concepteur spécifie les éléments du modèle du ClassD à raffiner. Finalement, le schéma de raffinement est automatiquement appliqué sur les différents diagrammes représentant le système. Ainsi, par exemple, nous pouvons appliquer dans un premier raffinement une occurrence du patron Observer, puis dans le deuxième raffinement une occurrence du patron Mediator et dans le dernier raffinement une autre occurrence du patron Observer.

Un raffinement est décrit graphiquement par un schéma appelé *schéma de raffinement*. Un schéma de raffinement est paramétré par les éléments du modèle de conception et est composé de deux compartiments. Le premier compartiment décrit le modèle abstrait de la conception et le second compartiment montre le modèle détaillé correspondant qui résulte de l'application du patron de conception en question. Dans le chapitre 6, nous allons présenter, comme illustration de notre approche, le schéma de raffinement que nous avons développé pour le patron Observer décrit dans le chapitre 2. Un autre schéma de raffinement sera donné en annexe H. Ce schéma correspond au

patron Mediator décrit aussi dans le chapitre 2.

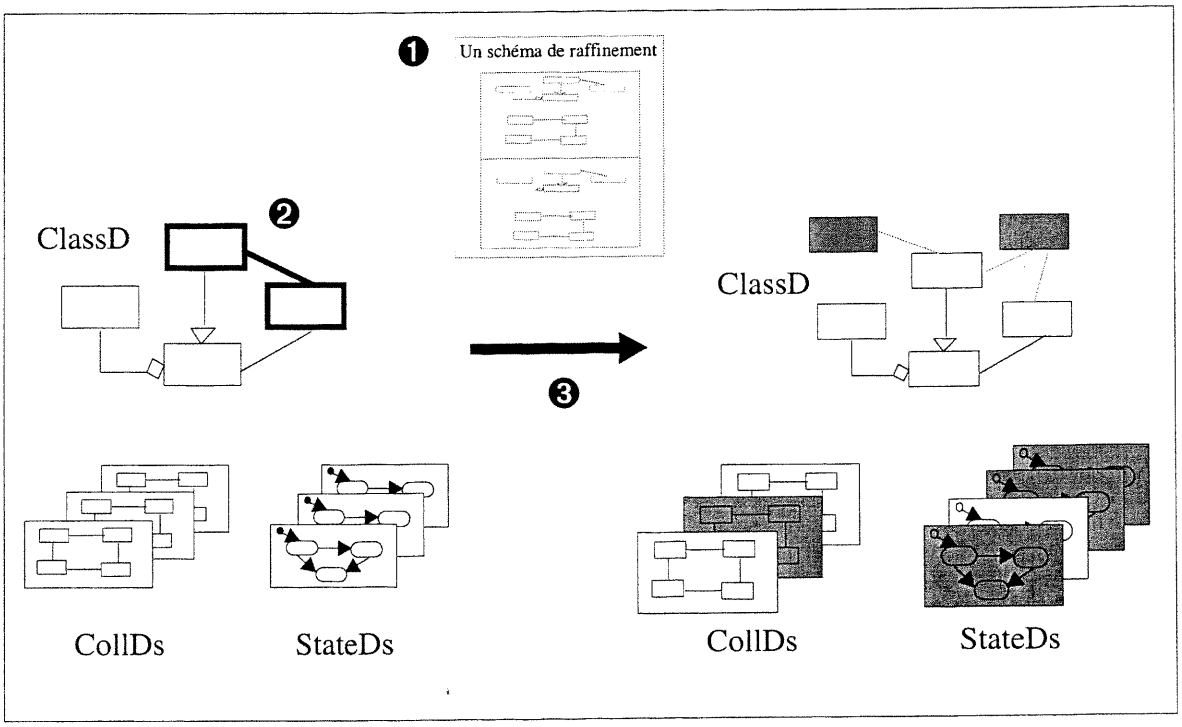


Figure 3.15: Les étapes d'un raffinement:

1- analyser le modèle et choisir un schéma de raffinement approprié,
2- spécifier les éléments du modèle du ClassD à raffiner,
3- générer le modèle détaillé correspondant.

3.2.2 Notion de schéma de micro-raffinement

Il nous est impossible de supporter tous les patrons de conception existant actuellement dans la littérature vu leur grand nombre qui ne cesse d'ailleurs de croître. Cependant, ces patrons de conception se basent principalement sur un petit nombre de techniques utilisées dans le monde OO telles que l'abstraction, l'héritage ou la délégation. Nous nous sommes restreints à étudier un ensemble de cinq patrons de conception qui sont: Observer, Mediator, Proxy, Façade [Gamma et al., 1996] et Forwarder-Receiver [Buschmann et al., 1996]. Ces patrons de conception sont intéressants pour trois raisons principales. La première raison réside dans le fait qu'ils permettent de bien illustrer l'utilisation des patrons de conception comme moyen d'intégrer la conception générale avec celle détaillée. La deuxième raison est que leur application engendre la mise à jour non seulement de l'aspect statique des modèles de conception mais aussi de leur aspect dynamique.

Finalement, la troisième raison vient du fait que ces patrons réutilisent les techniques que nous avons mentionnées ci-dessus.

Après avoir déterminé les schémas de raffinement pour les cinq patrons de conception, nous avons décomposé ces schémas en petits schémas que nous appelons *schémas de micro-raffinement*. En fait, un schéma de micro-raffinement sera aussi décrit par un schéma mais la différence est qu'il n'est pas spécifique à un seul patron de conception. En outre, puisque nous nous intéressons à avoir des raffinements valides, c'est-à-dire, des raffinements qui préservent le comportement du modèle de conception du système, il suffit de démontrer la validité des schémas de micro-raffinement pour que nous puissions nous assurer de la validité des schémas de raffinements les composant.

Donc le but des schémas de micro-raffinement est triple. Le premier but consiste à décrire un schéma de raffinement à partir d'une composition de schémas de micro-raffinement. Le deuxième but est la constitution d'un catalogue de schémas de micro-raffinement pour leur réutilisation dans la définition de schémas de raffinement pour de nouveaux patrons de conception. Le dernier but est la démonstration de la validité d'un schéma de raffinement par les schémas de ses micro-raffinements. Nous décrivons en annexe G les cinq schémas de micro-raffinement les plus réutilisés par les cinq schémas de raffinement. Nous donnons aussi leur pseudo-code avec leur preuve de validité.

3.2.3 Définition de nouveaux schémas de raffinement

La figure 3.16 montre, en forme de diagramme d'activité, le processus pour définir de nouveaux schémas de raffinement. En premier lieu, le concepteur choisit un patron de conception à ajouter dans le catalogue des schémas de raffinement. Ensuite, il définit son schéma de raffinement. Ceci consiste à définir le modèle abstrait du schéma puis de voir comment sera le modèle détaillé après l'application du patron de conception.

Puis, le concepteur essaiera de décomposer le schéma de raffinement en petits raffinements tout en essayant de réutiliser le plus possible les schémas de micro-raffinement déjà existant dans le catalogue. Si tous les schémas de micro-raffinement obtenus sont dans le catalogue, le concepteur n'a qu'à développer le schéma de raffinement comme une composition de schémas de micro-raffinement. Dans le cas contraire, le concepteur doit développer les schémas de micro-raffinement manquant tout en s'assurant de leur validité.

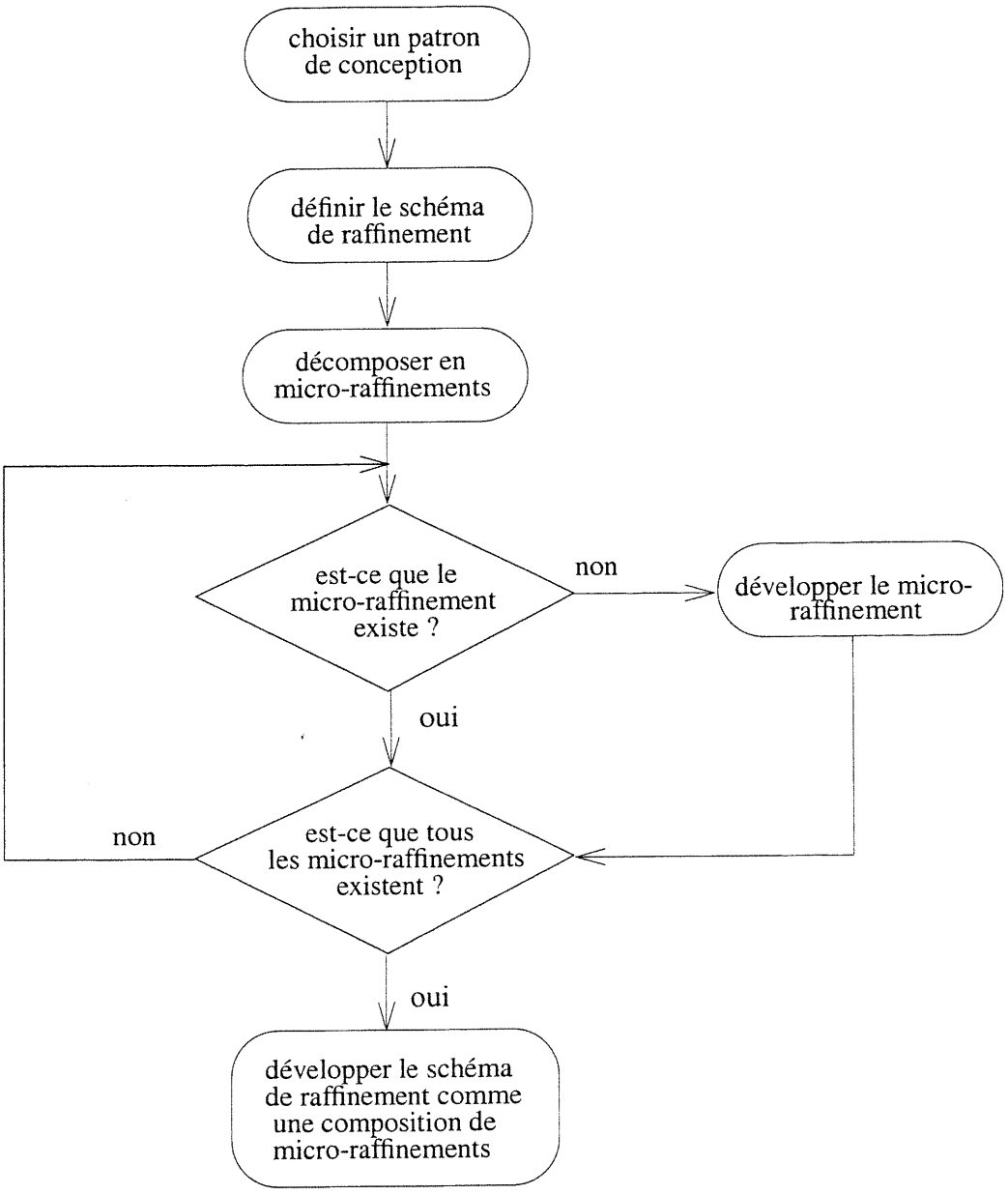


Figure 3.16: Méthodologie pour la définition de nouveaux schémas de raffinement

Après avoir donné une vue générale de notre approche pour atteindre les deux objectifs de notre thèse, nous allons présenter dans les chapitres 4, 5 et 7 les détails concernant le premier objectif (voir figure 3.2). Dans le chapitre 6, nous allons décrire en détail notre approche par rapport au deuxième objectif.

Chapitre 4 Génération du comportement dynamique partiel des objets à partir des scénarios

Dans ce chapitre [1], nous allons présenter l'algorithme [2] qui permet de générer le comportement dynamique partiel des objets, capturé dans des StateDs, à partir d'un scénario décrit dans un CollD. Le pseudo-code de l'algorithme se trouve en annexe D. Remarquons que pour une bonne compréhension de l'algorithme, il est nécessaire de lire les grammaires des CollDs et des StateDs se trouvant, respectivement, en annexes B et C. Nous allons clore ce chapitre par une discussion sur plusieurs points qui concernent le présent algorithme à savoir StateDs non pertinents, décisions de conception, extension de l'algorithme, complexité de l'algorithme, implantation de l'algorithme et expériences.

4.1 Description de l'algorithme

La génération des StateDs des objets à partir d'un CollD est composée de cinq étapes:

1. création des StateDs vides.
2. création des variables d'état pour les StateDs.
3. création des transitions pour les objets émetteurs.
4. création des transitions pour les objets récepteurs.
5. séquencement des transitions des StateDs.

La première étape crée un StateD pour chaque classe d'objets impliquée dans le CollD. La

1. Les résultats de ce chapitre sont publiés dans [Schönberger et al., 2000].
2. Les deux algorithmes du chapitre suivant étendent cet algorithme pour la synthèse de plusieurs scénarios. C'est pourquoi nous ne discutons des travaux reliés au présent algorithme que dans le prochain chapitre.

deuxième étape introduit, comme variables d'état, toutes les variables qui ne sont pas des attributs dans aucun des objets du CollD. Les variables considérées sont les variables de retour des messages, les paramètres de messages ou les variables qu'on retrouve dans les expressions d'itération ou de condition.

Appliquée à l'exemple de la figure 4.1, la première étape crée cinq StateDs vides (`Contrôleur`, `Fil`, `Ligne`, `Bead` et `Fenêtre`). La deuxième étape introduit `i`, `n`, `ligne`, `r0`, et `r1` comme des variables d'état pour `Fil`, et `fenêtre` pour `Ligne` (voir figure 4.2).

La troisième étape crée des transitions pour les objects émetteurs de messages. Ceci consiste à déterminer la partie événement d'une transition, créer des transitions auxiliaires pour les transitions qui attendent plus d'un événement externe, déterminer la partie envoi d'événement (`sendClause`), ainsi que préparer le contenu des données temporaires qui vont être utilisées dans la cinquième étape. La troisième étape est organisée autour de douze sous-étapes numérotées de 3.1 à 3.12 (cf. annexe D).

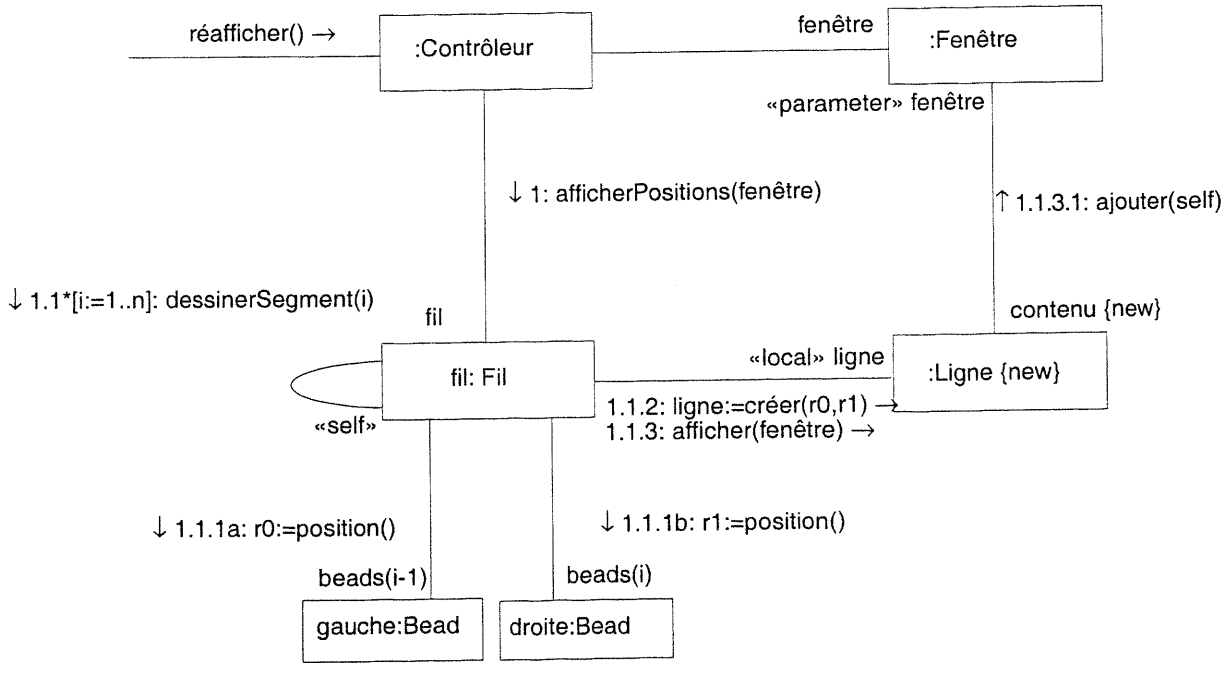


Figure 4.1: Le CollD de l'opération `réafficher()` (adaptée de [Rational et al., 1997], figure 37)

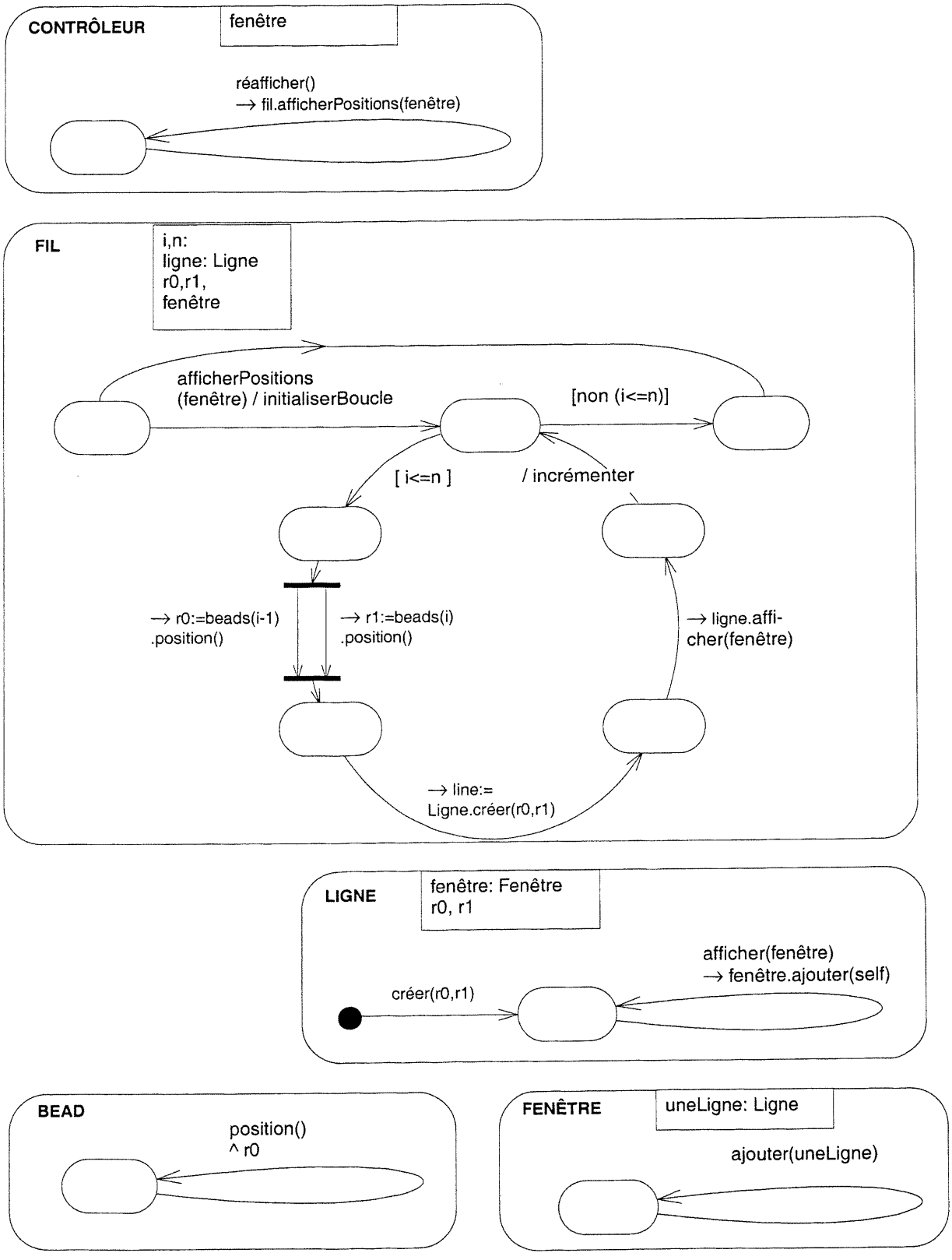


Figure 4.2: Les StateDs résultant de la transformation du CollD de la figure 4.1

Appliquée à l'exemple de la figure 4.1, la troisième étape crée les transitions identifiées par les envois d'événements suivants: `afficherPositions` dans `Contrôleur`; `position`, `créer`, et `afficher` dans `Fil`; et enfin `ajouter` dans `Ligne`. La sous-étape 3.2 ne fait que renvoyer l'algorithme à la sous-étape 3.10 à chaque fois qu'il se retrouve devant un message d'un objet à lui - même («`self`»). Les sous-étapes 3.3 et 3.4, qui traitent les messages avec un ou plusieurs prédécesseurs, ne sont pas exécutés puisque l'exemple ne contient pas de tels messages. Les sous-étapes 3.5 et 3.6 déterminent l'indicateur de synchronisation (`syncIndicator`) et la valeur de retour des transitions: → `afficherPosition` dans `Contrôleur`; → `r0:= position`, → `r1:= position`, → `créer`, et → `afficher` dans `Fil`; ainsi que → `ajouter` dans `Ligne`.

La sous-étape 3.7 détermine l'objet récepteur de la partie envoi d'événement des transitions: `fil` pour `afficherPositions` dans `Contrôleur`; `Ligne` pour `créer`, `beads(i-1)` et `beads(i)` pour `position`, et `ligne` pour `afficher` dans `Fil`; et finalement `fenêtre` pour ajouter dans `Ligne`. La sous-étape 3.8 traite les messages à prédécesseurs multiples et est par conséquent non applicable dans cet exemple. La sous-étape 3.9 met à jour les paramètres de la partie envoi d'événement des transitons: `afficherPositions(fenêtre)` dans `Contrôleur`; `position()`, `créer(r0, r1)`, et `afficher(fenêtre)` dans `Fil`; et `ajouter(self)` dans `Ligne`. Les sous-étapes 3.10, 3.11, et 3.12 préparent le contenu des données temporaires: numéro de séquencement, les informations de récurrence, et le drapeau "`new`" dans un lien. Ces données vont être utilisées dans la cinquième étape: l'information de récurrence va permettre la création de l'action `/initialiserBoucle`, et le drapeau "`new`" la création de la transition `Ligne.create` dans `Wire` (voir figure 4.2).

La quatrième étape crée des transitions pour les objets récepteurs des messages. Ceci consiste à déterminer l'événement et la valeur du retour d'une transition. Appliquée à l'exemple de la figure 4.1, la quatrième étape crée les transitions `réafficher()` dans `Contrôleur`, `afficherPositions(fenêtre)` dans `Fil`, `créer(r0,r1)` et `afficher(fenêtre)` dans `Ligne`, `position()` dans `Bead`, et `ajouter(uneLigne)` dans `Fenêtre` (voir figure 4.2).

La cinquième étape consiste à déterminer le bon séquencement des transitions des différents StateDs. Ceci permet la création des états et des barres (`splitBar` et `mergeBar`) nécessaires à un séquencement correct des transitions. Appliquée à l'exemple de la figure 4.1, cette étape permet la création d'un ensemble d'états et leur lien avec les transitions appropriés dans les StateDs des différents objets (voir figure 4.2). Cette étape sera décrite en détail dans la prochaine section.

4.2 Séquencement des transitions

Le but de la cinquième étape est de faire le bon séquencement des transitions générées par les étapes 3 et 4 de l'algorithme. Le séquencement est réalisé par la connexion des états et des barres `splitBar` et `mergeBar` avec les transitions appropriées. Le séquencement des messages se base sur les quatre types de messages: messages avec itération, messages conditionnels, messages concurrents et messages à prédécesseurs multiples. Notons que pour le cas de messages simples, le séquencement consiste à relier les transitions correspondant aux messages successifs par des états intermédiaires. Par exemple dans la figure 4.3, les messages `msg3` et `msg4` sont deux messages successifs transformés en deux transitions reliées par un état intermédiaire.

4.2.1 Messages avec itération

La transformation d'un message avec un indicateur d'itération ("*") consiste à placer tous les messages qui appartiennent à ce message dans un ensemble d'états et de transitions formant une boucle. La figure 4.3 montre comment un ensemble de messages contenant un message avec itération est transformé en un StateD.

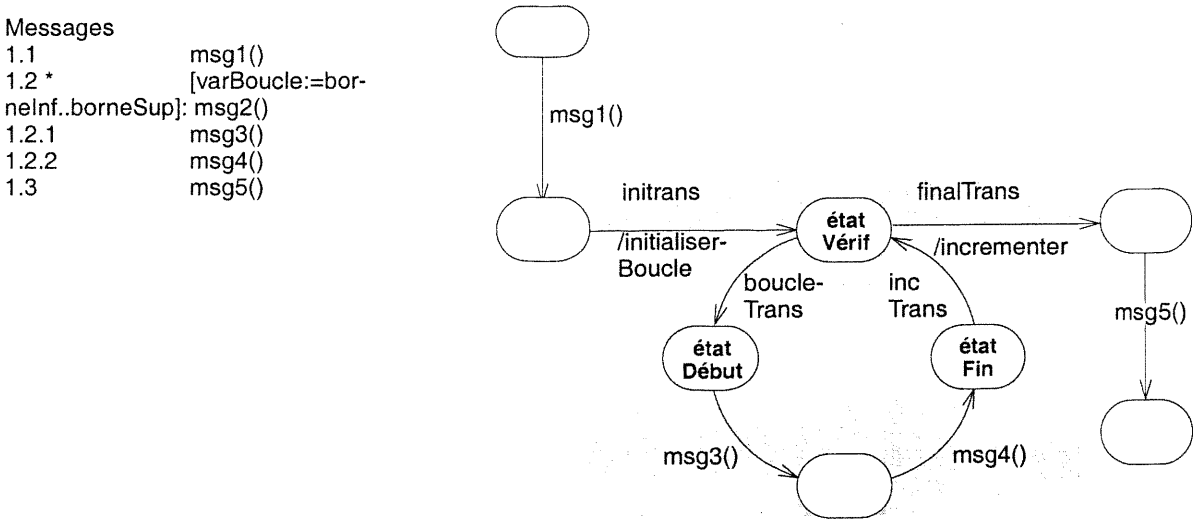


Figure 4.3: Transformation d'une liste de messages contenant un message avec itération

Soit `(1.2)` la transition dérivée du message avec le numéro de séquencement `1.2`. L'indicateur d'itération du message `msg2()` est temporairement sauvegardé dans la transition `(1.2)` (précisément dans `tempRecurrence`; voir la sous-étape 3.11 de l'algorithme). Dans la cinquième étape, la zone grise de la figure 4.3 est générée en remplacement de la transition `(1.2)` du message

`msg2()`, puis liée aux deux transitions correspondant, respectivement, à `msg1()` et `msg5()`. Le message `msg2()` est transformée en transitions `initTrans`, `boucleTrans`, `incTrans`, et `finalTrans`, ainsi qu'en états `étatVérif`, `étatDébut`, et `étatFin`. Les messages appartenant à l'itération (`msg3()` et `msg4()`) sont transformés en transitions et états à l'intérieur de la boucle.

La cinquième étape utilise l'information contenue dans `tempRecurrence` de la transition `(1.2)` dans le but de vérifier si elle indique une itération ("`*`") et d'ajouter les conditions `[varBoucle <= borneInf]` et `[varBoucle > borneSup]`, respectivement, dans les transitions `boucleTrans` et `finalTrans`.

Le CollD de la figure 4.1 (→ `1.1*[i:=1..n]: dessinerSegment(i)`) montre un exemple d'un message avec itération. La transformation de ce message se reflète dans la figure 4.2 (StateD de la classe `Fil`).

Un message avec itération qui est envoyé à un autre objet est traité essentiellement de la même manière que le message envoyé par un objet à lui même («`self`»). Les transitions `initTrans`, `boucleTrans`, `incTrans`, et `finalTrans` sont générées pour l'objet qui envoie le message avec itération. Tous les sous-messages de ce dernier sont transformés dans la même façon que les messages réguliers.

4.2.2 Messages conditionnels

Un groupe de messages conditionnels [3] est un groupe qui contient des messages avec une condition de garde ainsi que leurs sous-messages respectifs. Deux cas se présentent dans la transformation d'un groupe de messages conditionnels suivant le fait que les conditions de garde soient exclusives ou non. La figure 4.4 montre la transformation dans le cas où les conditions sont exclusives, alors que la figure 4.5 la montre dans le cas contraire.

Dans la figure 4.4, les conditions de garde sont exclusives `(x>0 et x<-2)`. Les transitions correspondant à `msg2()` et `msg4()` se répartissent à partir d'un état commun `étatCond` et se rencontrent dans un autre état commun `étatFusion`. `nullTrans` est la transition qui est exécutée dans le cas où aucune des conditions de garde ne soit satisfaite. La partie `condition` de `nullTrans` est construite à partir du `et logique` de la négation de toutes les conditions de garde des messages condi-

3. Notons que la structure des numéros de séquencement d'un groupe de message conditionnels est la même que pour les messages concurrents. En fait, les différentes lettres ne font que définir les différents groupes de messages conditionnels. La distinction entre les messages concurrents et les messages conditionnels se fait sur l'existence ou l'absence de conditions dans la structure des messages.

tionnels.

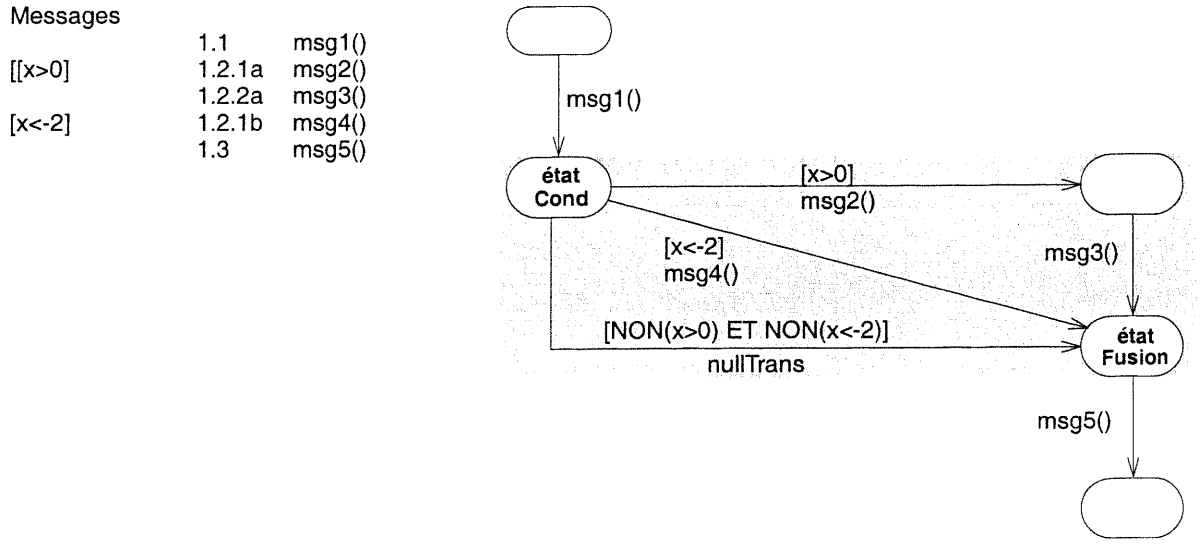


Figure 4.4: Transformation d'un exemple de messages conditionnels dans le cas où les conditions de garde sont exclusives

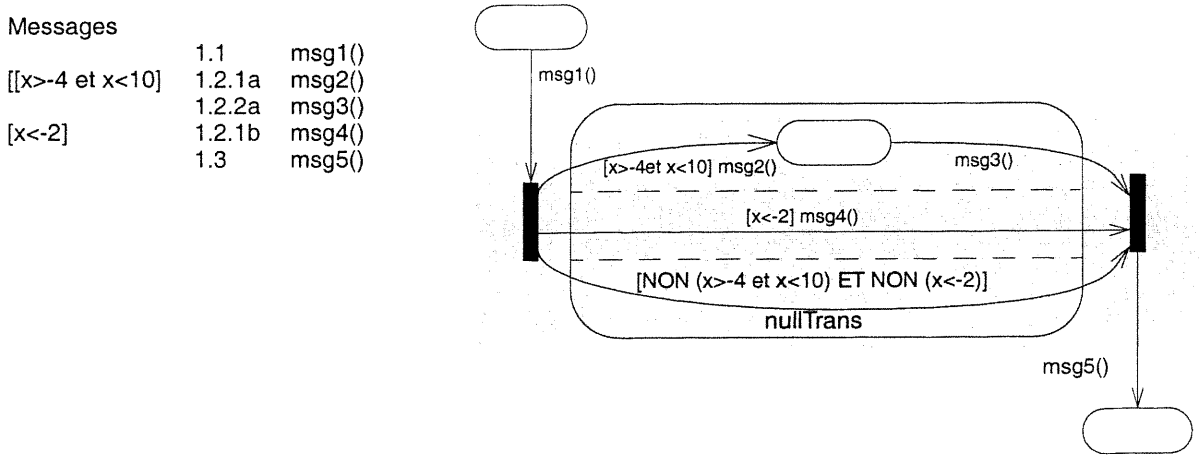


Figure 4.5: Transformation d'un exemple de messages conditionnels dans le cas où les conditions de garde ne sont pas exclusives

Dans la figure 4.5, les conditions de garde ne sont pas exclusives `((x>-4 et x< 10) et x<-2)`. La transformation ressemble beaucoup à celle d'un message concurrent (voir ci-dessous). Les transitions correspondant à `msg2()` et `msg4()` forment un ensemble de flots d'exécution concur-

rents. Les flots d'exécution s'ouvrent à partir d'une barre `splitBar` commune et se ferment dans une barre `mergeBar` commune. Chaque flot est placé à l'intérieur d'un sous-état d'un état concurrent.

La figure 4.6 montre un exemple de CollD contenant des messages conditionnels (→ `1.1a[fenêtre.affichage=#on]: dessinerSegment`, → `1.1b[fenêtre.affichage=#off]: flashBorder()`). La figure 4.7 montre la transformation correspondante.

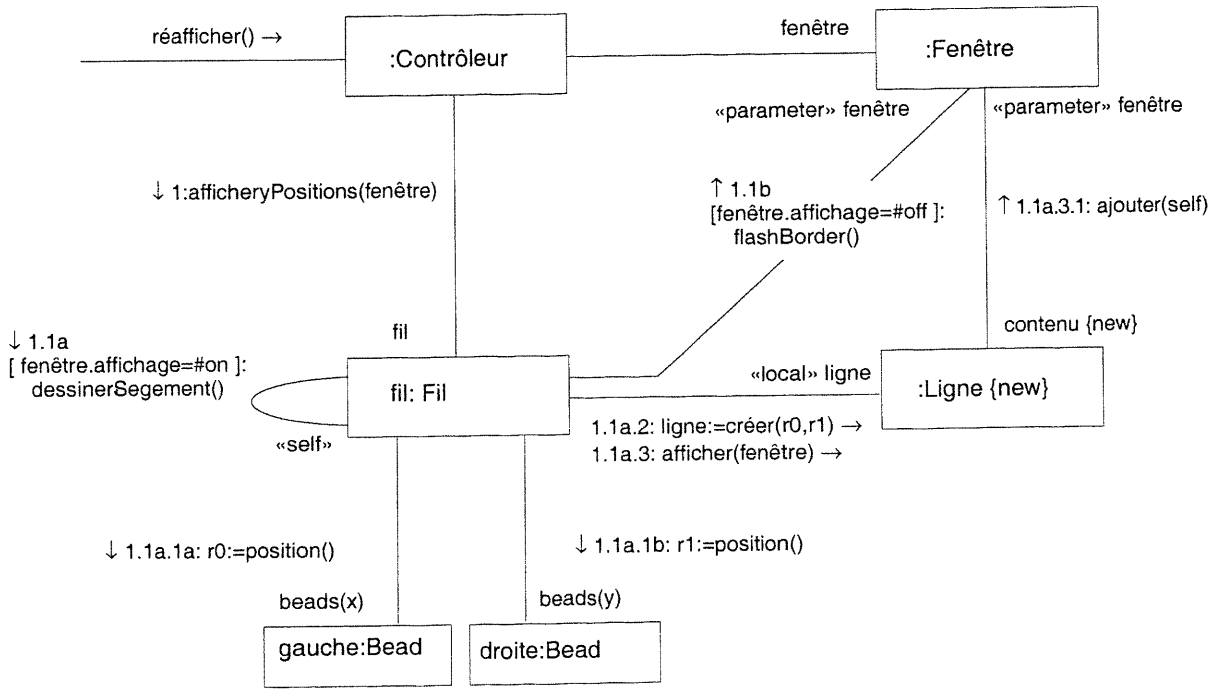


Figure 4.6: Exemple d'un CollD avec messages conditionnels (extension de la figure 4.1)

4.2.3 Messages concurrents

Dans le cas de messages concurrents (contenant un indicateur de concurrence, c'est-à-dire une lettre), un flot d'exécution est créé pour chaque lettre différente. Les flots d'exécution s'ouvrent à partir d'une barre `splitBar` commune et se ferment dans une barre `mergeBar` commune. Chaque flot d'exécution est placé à l'intérieur d'un sous-état d'un état concurrent. La figure 4.8 montre comment des messages appartenant à deux flots d'exécution sont transformés en un StateD.

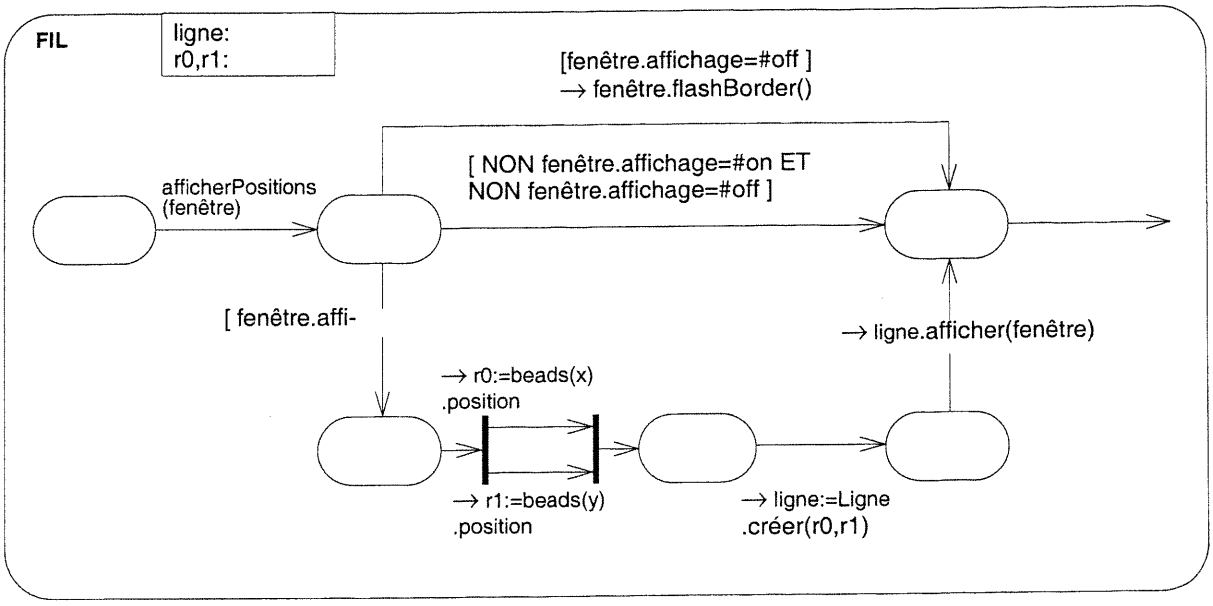


Figure 4.7: Le StateD de la classe `Fil` résultant de la transformation du diagramme de collaboration de la figure 4.6

La figure 4.1 montre un CollD contenant des messages concurrents (→ `1.1.1a: r0:=position()`, → `1.1.1b: r1:=position()`). Le StateD (de la classe `Fil`) correspondant est décrit dans la figure 4.2.

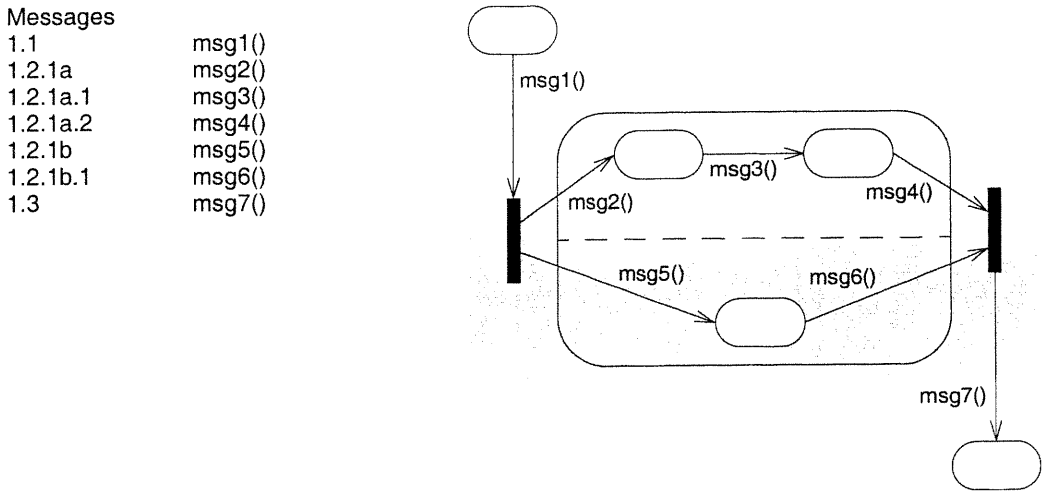


Figure 4.8: Transformation de messages concurrents

4.2.4 Messages à prédécesseurs multiples

Les prédécesseurs d'un message sont spécifiées dans le champ `predecessor` d'un message (voir annexe B). Ils indiquent les numéros de séquencement des messages qui doivent être envoyés avant que le message en question puisse être envoyé à son tour. Pour chaque élément que contient `predecessor`, une nouvelle transition est créée. Ces transitions vont se synchroniser par la liaison avec une barre `mergeBar` nouvellement créée. Pour distinguer les messages du même nom, le numéro de séquencement de chaque message est concatené avec le nom de l'événement de la transition correspondante. La figure 4.9 montre comment le message `msg2()` qui contient deux prédécesseurs (les messages avec les numéros de séquencement `1.7a` et `1.6b`) est transformé en éléments d'un StateD.

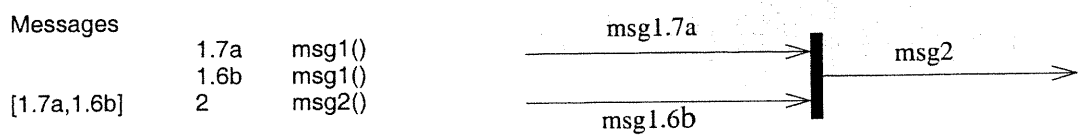


Figure 4.9: Transformation d'un message avec multiple prédécesseurs

4.3 Compression des diagrammes d'états-transitions

L'algorithme de transformation peut implanter différentes techniques pour réduire le nombre d'états des StateDs obtenus. Dans sa forme actuelle, notre algorithme implante les cinq techniques suivantes:

- fusion de deux transitions ne contenant que leur partie action.
- fusion de deux transitions ne contenant que leur partie envoi d'événement.
- fusion d'une transition ne contenant que sa partie action avec une transition ne contenant que sa partie envoi d'événement.
- fusion d'une transition sans les parties action et envoi d'événement avec une transition ne contenant que sa partie événement. Par exemple, dans le StateD de la classe `Fil` décrit dans la figure 4.2, la transition `afficherPositions(fenêtre) /initialiserBoucle` est le résultat de l'application de cette technique sur les deux transitions d'origine `afficherPositions(fenêtre)` et `/initialiserBoucle`.

- élimination des transitions dupliquées. Si deux transitions sont semblables (même `fromNode`, `toNode`, `guardCondition`, `{action}`, et `{sendClause}` (voir annexe C)) alors l'une des deux transitions est supprimée. Par exemple, dans le StateD de la classe `Bead` décrit dans la figure 4.2, cette technique est utilisée pour éliminer la transition dupliquée `position() ^r1`.

4.4 Discussion

Dans ce qui suit, nous allons discuter plusieurs points concernant l'algorithme de transformation présenté ci-haut: StateDs non pertinents, décisions de conception, extension de l'algorithme, complexité de l'algorithme, implantation de l'algorithme et expériences.

StateDs non pertinents

Durant la conception, les StateDs de certaines classes peuvent être ignorés. En effet, ces classes fournissent des services qui n'exigent pas un ordre particulier entre ses fonctions. Dans ce cas, les StateDs ne sont pas nécessaires. Cependant, il faut aussi tenir compte du fait qu'un StateD non pertinent pour un domaine d'application peut l'être dans un autre domaine.

Décisions de conception

Durant la construction de l'algorithme, nous avons pris certaines décisions de conception et ceci à différents endroits. Nos choix de conception ont été guidés par notre souci de simplicité et de lisibilité des StateDs générés.

À titre d'exemple, pour un message attendant un résultat, plusieurs possibilités s'offrent pour transformer un CollD en un StateD. Considérons les quatre messages de la figure 4.10, appartenant à deux objets `objetA` et `objetB`. `msg1` est envoyé par `objetA`, alors que les messages `msg2`, `msg3` et `msg4` sont envoyés par `objetB`. La figure 4.10 montre deux possibilités de construction de StateD pour `objetB`.

Le StateD de gauche suppose l'envoi de tous les sous-messages du message qui attend un résultat avant que ce dernier ne peut être retourné à l'émetteur. En effet, `objetB` attend la réception de l'événements `msg1`, puis envoie les messages `msg2`, `msg3` et `msg4`, avant de retourner le résultat à `objetA`. Le StateD de droite montre une solution alternative pour la transformation de ses messages. `objetB` attend la réception de l'événement `msg1` pour envoyer le message `msg2` et retourner `résultat`. C'est seulement après, qu'il peut envoyer les messages `msg3` et `msg4`.

En se basant sur les informations contenues dans un CollD, il est difficile de décider à quel niveau le résultat doit être retourné. Notre algorithme implante la dernière approche, c'est-à-dire, celle décrite par le StateD de droite. Notons aussi que la façon avec laquelle est traité un message à multiple prédécesseurs (voir section 4.2.4) ou les techniques de compression décrites dans la section précédente font aussi partie de ces choix de décision que nous avons pris.

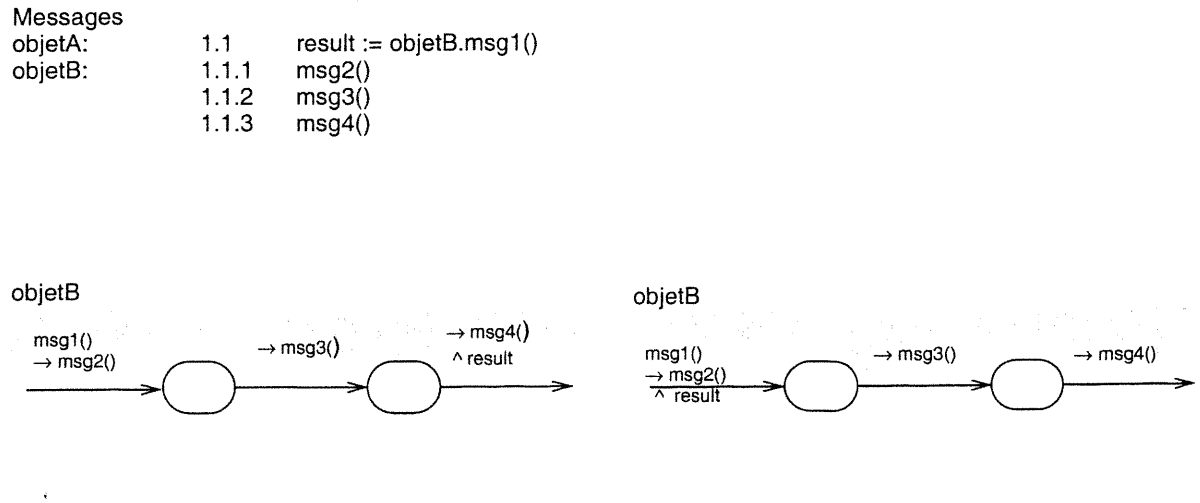


Figure 4.10: Deux possibilités de transformer un message

Extension de l'algorithme

Remarquons que notre algorithme a été conçu de façon incrémentale pour inclure les différents sous-ensembles des CollDs et des StateDs. Cette incrémentalité nous permet d'affirmer que l'algorithme peut être facilement étendu pour couvrir les notions de CollD non encore couvertes (voir annexe B), mais de plus peut être modifié pour suivre les différents changements apportés à UML. Il est à noter que nous avons apporté à plusieurs occasions des modifications à la version initiale de l'algorithme (qui était conforme à la version 0.8 d'UML). Maintenant, l'algorithme reste conforme à la version standard d'UML (version 1.1).

Complexité de l'algorithme

Pour la discussion de la complexité C de l'algorithme, soient N_o le nombre des objets dans un CollD, N_l le nombre de liens et N_m le nombre de messages. Puisque l'algorithme est composé de cinq étapes, C est la somme de C_1, C_2, C_3, C_4 et C_5 (avec C_i représentant la complexité de l'étape i). La première étape crée un StateD pour chaque classe présente dans un CollD. Par conséquent, C_1 est de l'ordre de N_o. La deuxième étape crée des variables d'état en se basant sur des informations se trouvant dans les liens ou dans les messages. C_2 appartient donc à $O(\max(N_o, N_l, N_m))$.

La troisième étape crée des transitions pour les objets émetteurs des messages, donc il y a au plus N_m messages à traiter. Le traitement d'un message est de l'ordre de N_m du fait que cette étape contient une méthode qui fait la recherche, pour chaque message, d'un message avec un numéro de séquencement donné (voir étape 3.4 en annexe D). C_3 appartient par conséquent à $O(N_m^2)$. Le même raisonnement s'applique pour la quatrième étape qui crée des transitions pour les objets récepteurs. Donc C_4 est de $O(N_m^2)$. La cinquième étape consiste à faire un séquencement des transitions générées. Une liste de transitions qui a une taille n'excédant pas N_m est utilisée. Le séquencement commence par trier la liste de transitions puis traite chaque élément de cette liste. L'algorithme a une complexité de $O(N_m * \log_2 N_m)$. Le traitement d'une transition consiste entre autres à faire une recherche sur les autres transitions et par conséquent il est égal à $O(N_m)$. Le traitement de toutes les transitions est par conséquent de l'ordre de N_m^2, et C_5 appartient à $O(N_m^2)$. Finalement, la complexité globale de l'algorithme est de $O(N_m^2)$.

Implantation de l'algorithme et expériences

Notre algorithme a été implanté avec un système de 15 classes et quelques 2100 lignes de code en langage Java (commentaires non inclus). Pour représenter les CollDs et les StateDs, nous avons défini des formats textuels. Pour le test de l'algorithme, nous avons généré des cas de test pour couvrir tous les types de messages: messages séquentiels, messages avec itération, messages conditionnels, messages concurrents et messages à prédécesseurs multiples. En outre, nous avons conçu des cas de test pour vérifier les cinq techniques de compression d'un StateD. Le nombre total des cas de test tourne autour de vingt.

En plus, nous avons testé l'algorithme sur des exemples de systèmes de petite à moyenne taille, allant jusqu'à des CollDs qui comprennent 10 objets et 50 messages. Ces exemples ont été pris de plusieurs sources: des exemples décrits dans la documentation d'UML [Rational et al., 1997], un système de gestion de guichet automatique [Rumbaugh et al., 1991], un système de gestion des feux d'un carrefour [Rumbaugh et al., 1991], une extension du système de gestion de bibliothèque [Eriksson et Penker, 1998], et finalement un système de gestion d'une station de service [Coleman et al., 1994]. Pour tous ces cas de test, le temps d'exécution était excellent (moins d'une seconde sur un Sun Sparc 10/514 à 4 processeurs SuperSparc de 50 MHz).

Notons que tous les exemples de CollDs présentés dans ce chapitre ont été traités par notre algorithme. Les diagrammes correspondant aux StateDs résultats ont été dessinés manuellement sur la base des descriptions textuelles générées par l'algorithme.

Chapitre 5 Synthèse du comportement dynamique des objets

Dans ce chapitre [1], nous allons en premier lieu présenter deux algorithmes. Le premier permet de faire une analyse d'un StateD. Cette analyse revient à étiqueter chaque StateD partiel généré par l'algorithme du chapitre précédent tout en vérifiant sa cohérence et sa complétude. Le deuxième algorithme a pour objectif d'intégrer deux StateDs partiels étiquetés d'un même objet. Le pseudo-code des deux algorithmes se trouve dans les annexes E et F respectivement. Puis, nous allons présenter les algorithmes que nous avons conçus pour vérifier la cohérence et la complétude des scénarios. Par la suite, nous allons faire une comparaison de notre approche avec des travaux reliés à la problématique de la synthèse de scénarios. Nous allons terminer ce chapitre par une discussion sur plusieurs points qui concernent les deux algorithmes soient: restriction sur les conditions, problème de chevauchement des scénarios, complexité des algorithmes ainsi que implantation des algorithmes et expériences.

5.1 Analyse du comportement dynamique partiel des objets

Dans cette section, nous allons commencer par donner un ensemble de définitions utilisées pour la compréhension de l'algorithme qui fait l'analyse d'un StateD. Ensuite, nous allons décrire l'algorithme d'analyse. Notons que cet algorithme est basé principalement sur les pre- et post-conditions des opérations des classes du système en question. La syntaxe de ces conditions (voir annexe A) suit un sous-ensemble du langage OCL [Rational et al., 1997] défini par UML pour la description des contraintes entre les éléments du modèle d'un système. Les définitions ci-dessous sous-entendent de telles conditions.

1. Les résultats de ce chapitre sont publiés dans [Khriss et al., 1999a; Khriss et al., 1998] et décrits dans le rapport technique [Khriss et al., 1999b].

5.1.1 Définitions

Définition 5.1. Soit c une condition de type orExpression sur un tuple de variables [2] v_i. *Eval(c)* est la fonction qui retourne l'ensemble des tuples de valeurs des variables v_i qui vérifient la condition c.

Définition 5.2. Un *état* s d'une classe C est défini par une condition c(s) de type orExpression sur les attributs de C.

Définition 5.3. Deux conditions c_1 et c_2 de type orExpression sont *égales* si et seulement si Eval(c_1) est égale à Eval(c_2).

Définition 5.4. Deux états s_1 et s_2 sont *égaux* si et seulement si leurs conditions c(s_1) et c(s_2) respectifs sont égales.

Définition 5.5. Une condition c_1 de type orExpression *raffine* une condition c_2 de type orExpression (ou c_1 est plus restrictive que c_2) si et seulement si Eval(c_1) $\subseteq$ Eval(c_2).

Définition 5.6. Un état ss est un *sous-état* d'un état s si et seulement si ss est dessiné à l'intérieur du rectangle au coins arrondis qui représente l'état s. s est aussi appelé *super-état* de ss.

Par exemple, si on prend le cas d'une classe `A` ayant comme attributs deux entiers a_1 et a_2 alors un état d'un objet de la classe `A` est défini par une condition sur a_1 et a_2. La figure 5.1 montre un super-état `S0=`a_1`>0 et `a_2`>0` de la classe `A`. Cet état contient deux sous-états `S1=`a_1`>5 et `a_2`>3` et `S2=`a_1`>0 et `a_1`<2 et `a_2`>0 et `a_2`<3`. Si `S3=`a_1`>7` était dessiné à l'intérieur du rectangle aux coins arrondis de l'état `S0` alors `S3` serait aussi un sous-état de `S0`.

Définition 5.7. Un état s est *un état de type* `OU` si et seulement si les sous-états ss_i de s vérifient les conditions (1) $\forall i, 1 \leq i \leq n$. c($ss_i$) raffine c(s) et (2) $\forall i, j, 1 \leq i \leq n, 1 \leq j \leq n$ et $i \neq j$. Eval(c(ss_i)) $\cap$ Eval(c(ss_j)) = $\varnothing$.

Par exemple, l'état `S0=`a_1`>0 et `a_2`>0` mentionné ci-dessus est un état de type `OU` de la classe `A`. En effet, les sous-états `S1=`a_1`>5 et `a_2`>3` et `S2=`a_1`>0 et `a_1`<2 et `a_2`>0 et `a_2`<3` vérifient bien les conditions (1) et (2) de la définition 5.7. Si `S3=`a_1`>7` et `S4=`a_1`>2 et `a_2`>1` devenaient aussi des sous-états de `S0` alors `S0` ne serait plus un état de type `OU` car `S3` ne vérifie pas la condition (1)

2. Une variable peut être un attribut d'une classe, un paramètre d'une opération d'une classe ou une variable dans un StateD.

alors que S4 ne vérifie pas, par rapport à S1, la condition (2).

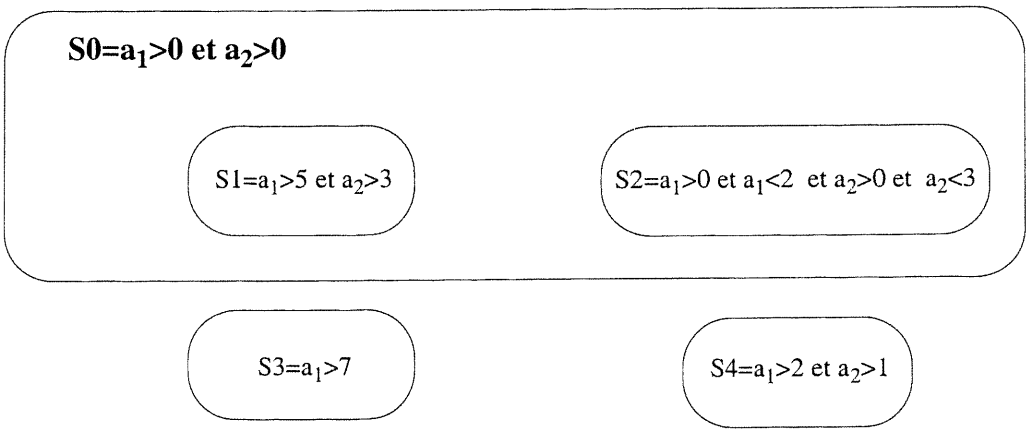


Figure 5.1: Illustration d'un état de type OU

Définition 5.8. Un état s est *un état de type ET* si et seulement si s est un super-état contenant n sous-états concurrents ss_k ($n \geq 2$) tel que $\forall i, j, 1 \leq i \leq n, 1 \leq j \leq n$ et $i \neq j$: $c(ss_i)$ ne raffine pas $c(ss_j)$, $c(ss_j)$ ne raffine pas $c(ss_i)$ et $Eval(c(ss_i)) \cap Eval(c(ss_j)) \neq \varnothing$.

Notons qu'un état concurrent peut être un état simple (c'est-à-dire ne contenant pas de sous-états), un état de type OU où bien un état de type ET. La définition 5.8 assure en premier lieu qu'il n'y ait pas de relation d'inclusion entre les conditions correspondantes aux états concurrents. Deuxièmement, elle assure qu'un objet d'une classe donnée ne se trouve pas dans des états concurrents où il existe une incohérence entre leurs conditions respectives. En effet prenons le cas d'une classe A ayant comme attributs deux entiers a_1 et a_2. L'état $S0=a_1>0$ et $a_2>0$ tel que décrit dans la figure 5.2 est bien un état de type ET de cette classe. Un objet de la classe A peut être à la fois dans l'état $S1=a_1>0$ (dans un de ses sous-états $S2=a_1>5$ et $S3=a_1>0$ et $a_1<5$) et dans l'état $S4=a_2>3$ sans qu'il n'y ait d'incohérence entre leurs conditions respectives. Supposons maintenant que l'état $S5=a_1>5$ et $a_2>0$ et $a_2<3$ devient aussi un sous-état concurrent de S0. L'état S5 ne raffine ni S1 ni S4 et vice versa, par contre $Eval(c(S4)) \cap Eval(c(S5)) = \varnothing$, donc il ne peut pas être un sous-état concurrent dans S0.

Notons que les définitions ci-dessus sont conformes à la sémantique informelle donnée par UML [RATL, 1997] à condition que nous considérions qu'un état d'une classe donnée est défini par une condition de type orExpression sur les attributs de la classe. À cet effet, notre définition de l'état

(définition 5.2) est la seule restriction que nous faisons par rapport à UML.

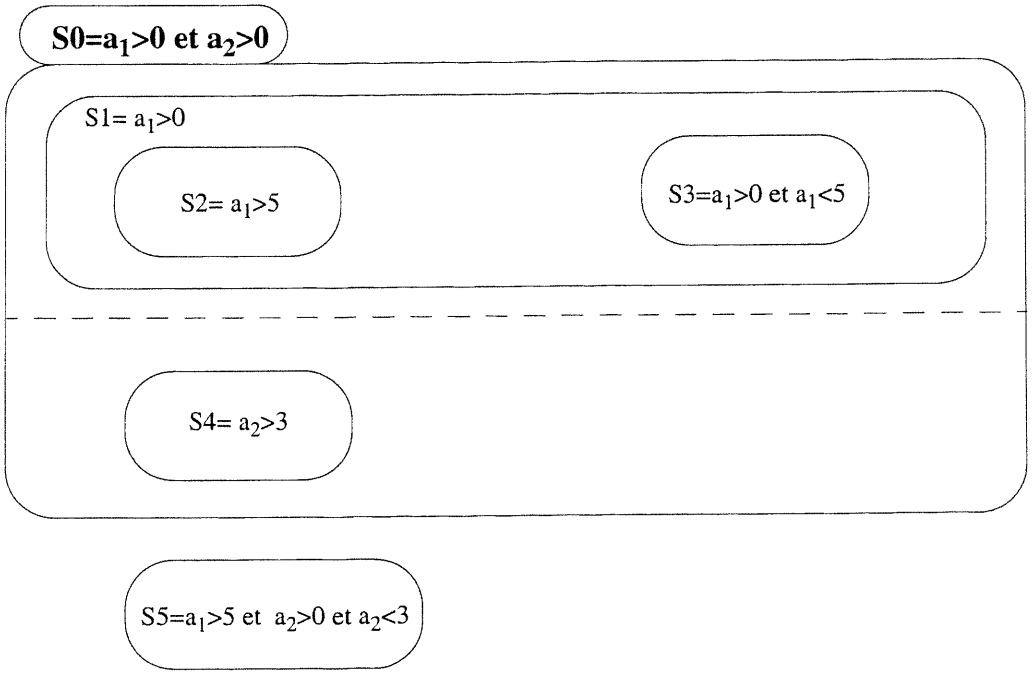


Figure 5.2: Illustration d'un état de type ET

Définition 5.9. Soit trans une transition d'un StateD d'une classe C donnée qui est caractérisée par [event] [guardCondition] {/ action} {sendClause} [^ returnValue]. Si event existe alors *preTrans(trans)* est égale à la pre-condition de l'opération de la classe C correspondante à event, sinon *preTrans(trans)* est égale à la pre-condition de l'opération de la classe C correspondante au premier élément de {/action} [3]. Si la partie {/action} existe alors *postTrans(trans)* est égale à la post-condition de l'opération de la classe C correspondante au dernier élément de {/action}, sinon *postTrans(trans)* est égale à la post-condition de l'opération de la classe C correspondante à event.

Définition 5.10. Soit c une condition de type preCondition ou postCondition. Si c est une orExpression alors *preCond(c)* est égale à TRUE et *postCond(c)* est égale à c. Dans le cas contraire, c'est-à-dire que c est égale à IF orExpression_{11} THEN orExpression_{12} ENDIF {OR IF orExpression_{i1} THEN orExpression_{i2} ENDIF}$_{i=2..n}$, alors *preCond(c)* est égale à la disjonction de toutes les orExpression_{i1}, i=1..n, et *postCond(c)* est égale à la disjonction de toutes les

3. Notons que l'algorithme du chapitre précédent (cf. section 4.3) assure, sauf dans un seul cas, de la présence d'au moins une action si event n'existe pas. Le cas d'exception survient lorsque le fromNode ou le toNode de la transition est un splitBar ou un mergeBar. Ces derniers sont des pseudo-états que nous n'étiquetons pas et par conséquent nous n'aurons pas à chercher ni leur preTrans ni leur postTrans.

orExpression$_{i2}$, i=1..n.

Définition 5.11. Soient c_1 une condition égale à IF orExpression$_{11}$ THEN orExpression$_{12}$ ENDIF {OR IF orExpression$_{i1}$ THEN orExpression$_{i2}$ ENDIF}$_{i=2..n}$ et c_2 une orExpression. *preCond(c_1, c_2)* est la disjonction de toutes les orExpression$_{i1}$, i=1..n, qui raffinent c_2, et *postCond(c_1, c_2)* est la disjonction de toutes les orExpression$_{i2}$, i=1..n, dont la orExpression$_{i1}$ correspondante raffine c_2. Dans le cas où aucune orExpression$_{i1}$ ne raffine c_2 alors *preCond(c_1, c_2)*, et *postCond(c_1, c_2)* retournent la valeur FALSE.

Définition 5.12. Soient c une condition de type orExpression, c'est-à-dire que c est égale à basicExpression$_{11}$ {AND basicExpression$_{i1}$}$_{i=2..n_0}$ {OR basicExpression$_{1k}$ {AND basicExpression$_{jk}$}$_{j=2..n_k}$}$_{k=2..n_1}$, et C une classe donnée. *attributePart(c, C)* est la condition dérivée de c telle que chaque basicExpression dans c qui exprime une contrainte sur une variable qui n'est pas un attribut de C est remplacée par la valeur TRUE.

Définition 5.13. Soient c une condition de type orExpression, c'est-à-dire que c est égale à basicExpression$_{11}$ {AND basicExpression$_{i1}$}$_{i=2..n_0}$ {OR basicExpression$_{1k}$ {AND basicExpression$_{jk}$}$_{j=2..n_k}$}$_{k=2..n_1}$, et C une classe donnée. *parameterPart(c, C)* est la condition dérivée de c où chaque basicExpression dans c qui exprime une contrainte sur un attribut de C est remplacée par la valeur TRUE.

Définition 5.14. Soient *c* une condition de type orExpression qui est égale à andExpression$_1$ {OR andExpression$_i$}$_{i=2..n}$ et C une classe donnée. *attributePartList(c, C)* est la liste formée par attributePart(andExpression$_i$)$_{i=1..n}$.

Notons que andExpression est un cas spécial de orExpression, par conséquent la définition 5.12 s'applique (ainsi que la définition 5.13, voir ci-dessous).

Définition 5.15. Soient *c* une condition de type orExpression qui est égale à andExpression$_1$ {OR andExpression$_i$}$_{i=2..n}$ et C une classe donnée. *parameterPartList(c, C)* est la liste formée par parameterPart(orExpression$_i$)$_{i=1..n}$.

Définition 5.16. Soit *c* une condition de type orExpression qui est égale à andExpression$_1$ {OR andExpression$_i$}$_{i=2..n}$ (respectivement, *c* est égale à ifExpression$_1$ {OR ifExpression$_i$}$_{i=2..n}$). *expressionListOfOR(c)* est la liste formée par (andExpression$_i$) $_{i=1..n}$ (respectivement, formée par (ifExpression$_i$)$_{i=1..n}$).

5.1.2 Algorithme d'analyse

L'algorithme d'analyse d'un StateD est composée de quatre étapes:

1. vérification de la cohérence de la description de la classe en entrée,
2. calcul des fonctions *preTrans* et *postTrans* des transitions du StateD,
3. étiquetage des états du StateD,
4. vérification de la cohérence du StateD.

La première et la quatrième étapes concernent l'aspect de vérification et seront par conséquent décrites dans la section 5.3. Dans la deuxième étape, l'algorithme calcule la valeur de *preTrans* et *postTrans* pour toutes les transitions du StateD. Par exemple, la figure 5.3 montre la description de la classe C ainsi que son StateD non étiqueté qui comprend deux transitions T_1 et T_2. D'après la définition 5.9, nous obtenons preTrans(T_1) = pre(e_1) = a_1<=0 et a_2<=1, postTrans(e_1) = if a_1=0 et a_2=1 et p>0 then a_1=1 et a_2=2 endif ou if a_1<0 et a_2<1 et p>0 then a_1>2 et a_2>3 endif, preTrans(T_2) = pre(e_2) = a_1=3 et a_2=4 et p>0 ou a_1>3 et a_2>4 et p<=0 et postTrans(T_2) = post(e_2) = a_1=2 et a_2=3.

Dans la troisième étape, l'algorithme étiquette les états du StateD non étiqueté en parcourant sa liste des transitions. Notons que cette liste est construite, par l'algorithme que nous avons décrit dans le chapitre précédent, de façon à ce qu'elle soit triée en suivant l'ordre de séquencement des messages dans le scénario correspondant [4, 5]. Pour chaque transition trans, l'algorithme commence par calculer la valeur de deux conditions c_1 et c_2 ainsi que celle d'une liste de conditions Sc. Ces conditions constituent la base de l'opération d'étiquetage. Deux cas sont à considérer. Le premier cas survient quand *postTrans(trans)* est une *orExpression* alors c_1 sera égale à *preTrans(trans)* si le fromNode de trans n'est pas encore étiqueté (puisqu'une transition précédente aurait déjà pu l'étiqueter) sinon c_1 sera égale à *la condition de fromNode*, c_2 sera égale à *postTrans(trans)* si le toNode de trans n'est pas encore étiqueté sinon c_2 sera égale à *la condition de toNode,* et Sc sera égale à *parameterPartList(preTrans(trans))* (voir définition 5.15). Le deuxième cas survient quand *postTrans(trans)* est une *ifExpression {OR ifExpression}* alors si le fromNode de trans est déjà étiqueté par une transition précédente alors c_1 sera égale à *la condition de fromNode* sinon l'algorithme fera une tentative avec le premier élément de la liste retournée par

4. Les messages concurrents sont triés en ordre alphabétique par rapport aux lettres survenant dans le numéros de séquencement des messages. Par exemple, un CollD contenant les messages 1, 3, 2a.1, 2a.2 et 2b seront triés de la façon suivante: 1, 2a.1, 2a.2, 2b, 3.
5. Une autre approche serait de parcourir le StateD à partir de son état initial. Mais puisque la liste des transitions est déjà triée, nous avons opté pour cette solution qui est plus simple algorithmiquement.

expressionListOfOR(postTrans(trans)) (voir définition 5.16) (voir plus loin la condition d'échec de la tentative et l'action à faire dans ce cas) et c_1 prendra la valeur de *preCond(premier élément de expressionListOfOR(postTrans(trans)))*, c_2 sera égale à *postCond(postTrans(trans), c_1)* (voir définition 5.11) si le toNode de `trans` n'est pas encore étiqueté sinon c_2 sera égale à *la condition de toNode,* et `sc` sera égale à *parameterPartList(premier élément de expressionListOfOR(post-Trans(trans)).*

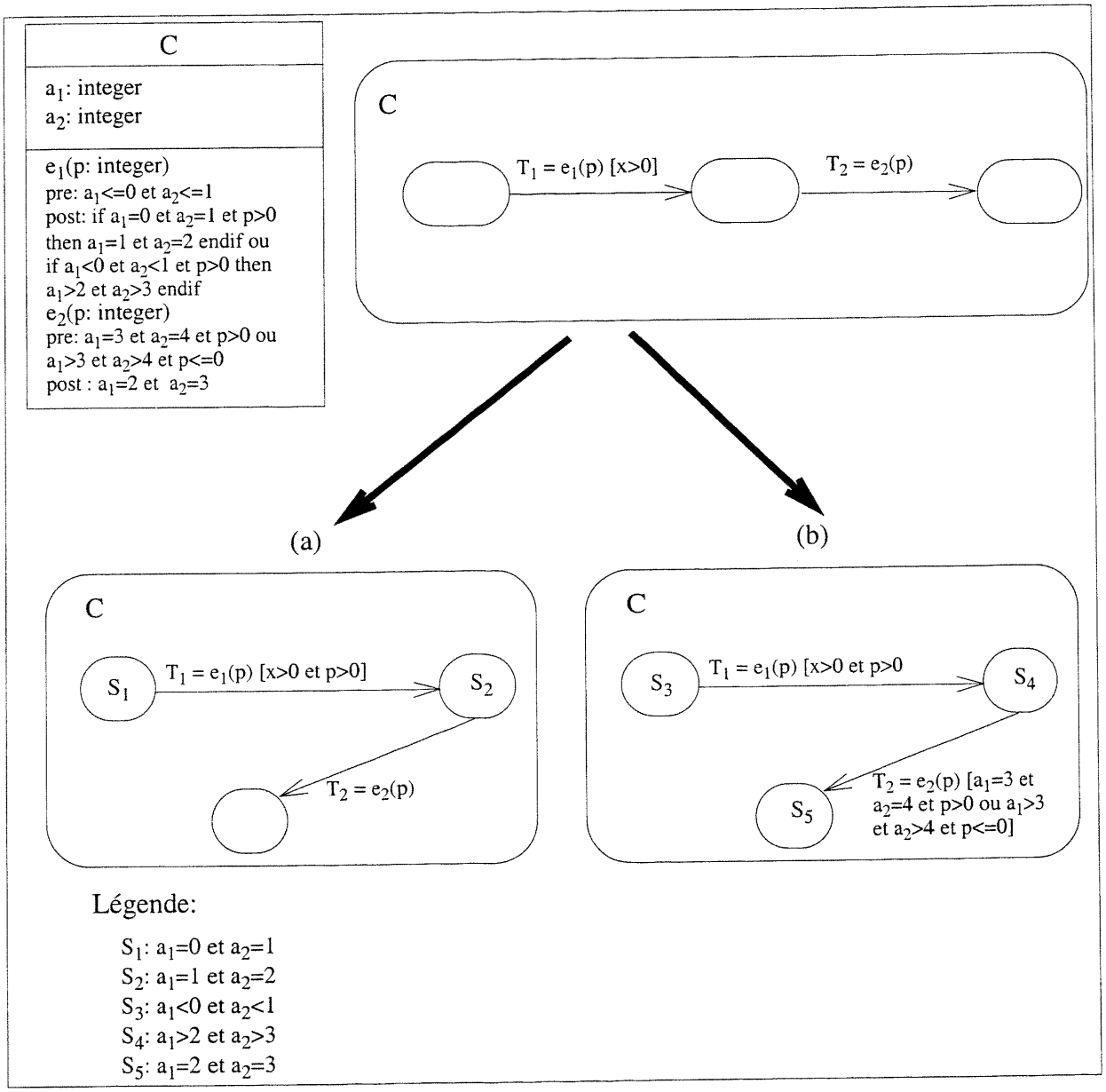


Figure 5.3: Étiquetage d'un StateD: (a) étiquetage impossible (T_2 est incohérent avec son from-Node)
(b) étiquetage réussi (T_2 est cohérent avec son fromNode)

Dans l'exemple de la figure 5.3, `postTrans(`T_1`)` est une ifExpression {OR ifExpression} et le fromNode et le toNode de T_1 ne sont pas encore étiquetés alors c_1 = `preCond(if` a_1`=0 et` a_2`=1 et p>0 then` a_1`=1 et` a_2`=2 endif)` = a_1`=0 et` a_2`=1 et p>0, c2 = postCond(if` a_1`=0 et` a_2`=1 et p>0 then` a_1`=1 et` a_2`=2 endif ou if` a_1`<0 et` a_2`<1 et p>0 then` a_1`>2 et` a_2`>3 endif,` a_1`=0 et` a_2`=1 et p>0)` = a_1`=1 et` a_2`=2` et `Sc = {p>0}`.

L'état `fromNode` de `trans` est étiqueté par *attributePart(c_1)* s'il n'est pas encore étiqueté, et l'état `toNode` par *attributePart(c_2)* (voir définition 5.12) s'il n'est pas encore étiqueté. La condition de garde de `trans` dépend de `Sc`. Si `Sc` contient un seul élément alors la condition de garde sera remplacée par la conjonction disjonctive de sa propre valeur avec celle de l'élément de `Sc`. Sinon, c'est-à-dire que `Sc` contient plus qu'un seul élément alors la condition de garde sera remplacée par la conjonction disjonctive de sa propre valeur avec celle de *preCond(c_1)* (voir définition 5.10). Ainsi pour la transition T_1 de la figure 5.3, nous avons S_1`=attributePart(`c_1`)` = a_1`=0 et` a_2`=1,` S_2 = `attributePart(`c_2`)` = a_1`=1 et` a_2`=2` et T_1`.guardCondition =` T_1`.guardCondition et Sc[1] = x>0 et p>0`.

Ensuite, l'algorithme vérifie que la transition suivante `suivTrans` reste cohérente avec son fromNode et son toNode. Cela veut dire que *preTrans(suivTrans) raffine la condition de fromNode* et que *la condition de toNode* raffine *postTrans(suivTrans)* si fromNode et toNode sont étiquetés (voir définition 5.20). Si `suivTrans` est cohérente avec son fromNode et son toNode alors l'algorithme continue son chemin en traitant `suivTrans`; dans le cas contraire, il revient en arrière (il fait du "backtracking") pour chercher la dernière transition qui lui reste encore des éléments à essayer dans sa liste retournée par la fonction expressionListOfOR de sa post-condition. Dans l'exemple de la figure 5.3, T_2 n'est pas cohérente avec S_1, donc il faut revenir à la dernière transition qui lui reste des éléments à essayer. Ceci est le cas de la transition T_1, nous aurons donc c_1 = `preCond(if` a_1`<0 et` a_2`<1 et p>0 then` a_1`>2 et` a_2`>3 endif)` = a_1`<0 et` a_2`<1 et p>0, c2 = postCond(if` a_1`=0 et` a_2`=1 et p>0 then` a_1`=1 et` a_2`=2 endif ou if` a_1`<0 et` a_2`<1 et p>0 then` a_1`>2 et` a_2`>3 endif,` a_1`<0 et` a_2`<1 et p>0)` = a_1`>2 et` a_2`>3` et `Sc = {p>0}`. S_3`=attributePart(`c_1`)` = a_1`<0 et` a_2`<1,` S_4 = `attributePart(`c_2`)` = a_1`>2 et` a_2`>3` et T_1`.guardCondition =` T_1`.guardCondition et Sc[1] = x>0 et p>0`. Maintenant, T_2 est cohérente avec son fromNode et son toNode n'est pas encore étiqueté alors l'algorithme peut continuer pour traiter la transition T_2. Nous aurons ainsi, `postTrans(`T_2`)` est une orExpression et le fromNode de T_2 est déjà étiqueté alors c_1 = `condition de` T_2`.fromNode =` a_1`>2 et` a_2`>3`, le toNode de T_2 n'est pas étiqueté alors c_2 = `postTrans(`T_2`)` = a_1`=2 et` a_2`=3, Sc = {p>0, p<=0}>`, S_4 est déjà déterminée par T_1, S_5 = `attributePart(`c_2`)` = a_1`=2 et` a_2`=3` et T_2`.guardCondition=`T_2`.guardCondition et preCond(`c_1`)` = a_1`=3 et` a_2`=4 et p>0 ou` a_1`>3 et`

a_2>4 et p<=0.

Notons que dans le cas où l'algorithme a épuisé tous les éléments de expressionListOfOR des post-conditions de toutes les transitions sans qu'il ne termine l'opération d'étiquetage, il signale un message d'erreur à l'analyste indiquant une incohérence entre les transitions du StateD.

5.2 Intégration des comportements dynamiques partiels des objets

L'algorithme d'intégration est composée de cinq étapes:

1. Validation des états
2. Intégration des variables de composition
3. Fusion des états
4. Fusion des transitions
5. Vérification de la cohérence du StateD résultant.

La cinquième étape est une étape de vérification et sera par conséquent discutée dans la prochaine section. Notez que la première étape peut être aussi considérée comme une étape de vérification, mais comme nous allons voir elle n'obéit pas aux mêmes règles. C'est pourquoi nous avons préféré de ne pas l'inclure dans la section 5.3.

5.2.1 Validation des états

L'algorithme d'intégration est conçu pour intégrer n'importe quel StateD (au delà des StateDs générés par l'algorithme précédent). C'est pourquoi l'étape de validation des états est introduite avant la fusion des états pour vérifier si deux états de même nom apparaissent dans les deux StateDs dans des niveaux de hiérarchie différents. Par exemple, supposons que l'algorithme doit fusionner pour l'objet `Obj` le StateD $\texttt{StateD}_1$ de la figure 5.4 avec le StateD $\texttt{StateD}_2$ de la figure 5.4. L'algorithme va détecter les erreurs suivants:

L'état b existe dans plusieurs endroits dans le deuxième StateD,
L'état e se trouve à des niveaux différents de hiérarchie dans les deux StateDs.

Si l'algorithme détecte ce genre d'erreurs, l'algorithme d'intégration ne peut pas intégrer les deux

StateDs. Dans ce cas, le concepteur doit corriger ces erreurs. Supposons que ce dernier a corrigé ces erreurs en remplaçant `e` avec `l` dans $\texttt{StateD}_1$, et `b` avec `k` dans $\texttt{StateD}_2$ tel que montré dans la figure 5.5.

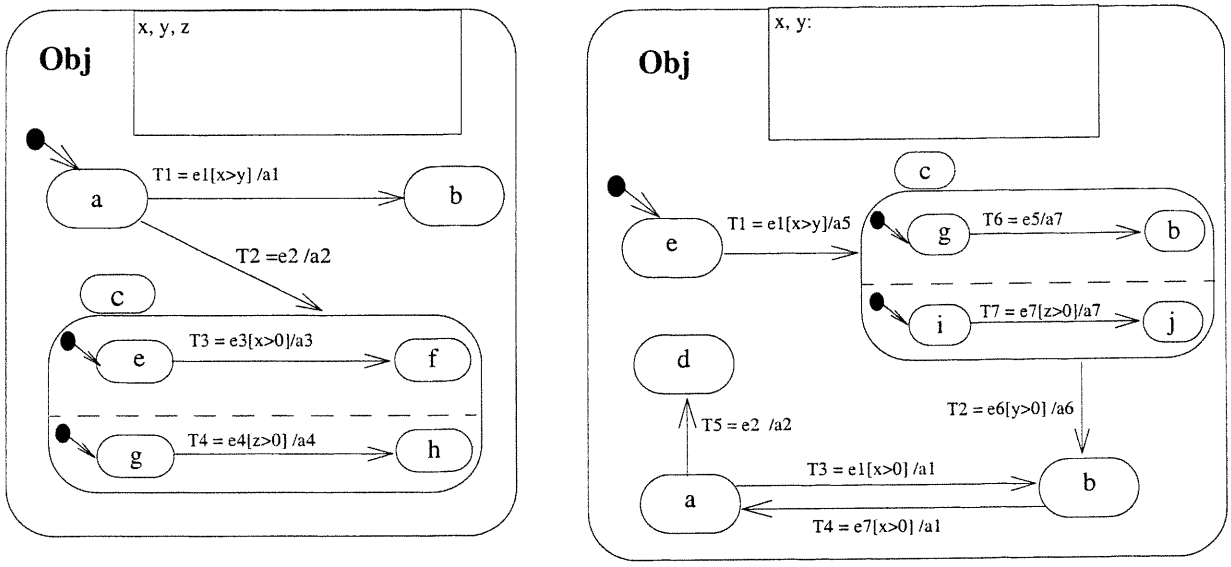


Figure 5.4: StateD_1 (à gauche) et StateD_2 (à droite) avant correction

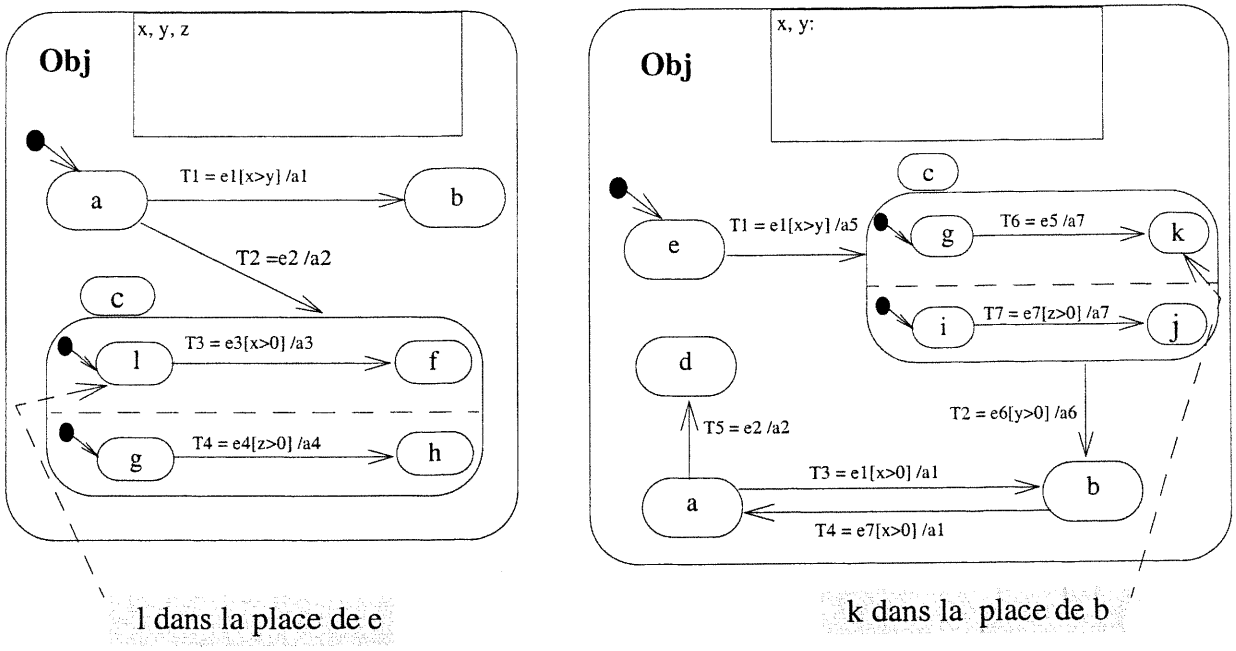


Figure 5.5: StateD_1 (à gauche) et StateD_2 (à droite) après correction

5.2.2 Intégration des variables de composition

En général, après l'intégration de plusieurs scénarios, la spécification résultante capture plus que ce qui est spécifiée dans les scénarios d'entrée. La figure 5.6 fournit un exemple illustrant ce problème (les scénarios sont représentés par des StateDs). En effet, si on fusionne les deux scénarios `Sc1` and `Sc2`, la spécification résultante `Sc` va capturer non seulement `Sc1` et `Sc2`, mais aussi de nouveaux scénarios correspondants respectivement aux séquences `(T1, T2, T7, T8)` et `(T5, T6, T3, T4)`.

Pour remédier à ce problème, nous avons défini trois variables de composition: *scenarioList*, *dynamicScenarioList* et *transScenarioList* (voir figure 5.7). *scenarioList* est l'ensemble de noms de scénarios; il mémorise donc les scénarios que le StateD capture. *dynamicScenarioList* est aussi un ensemble de noms de scénarios. Il est initialisé à *scenarioList* et peut changer durant l'exécution du StateD. À chaque exécution, il garde les noms de scénarios qui restent possibles dans la prochaine exécution. *transScenarioList* est un tableau d'ensembles de noms de scénarios. Il conserve l'information sur les scénarios qui sont concernés par chaque transition du StateD.

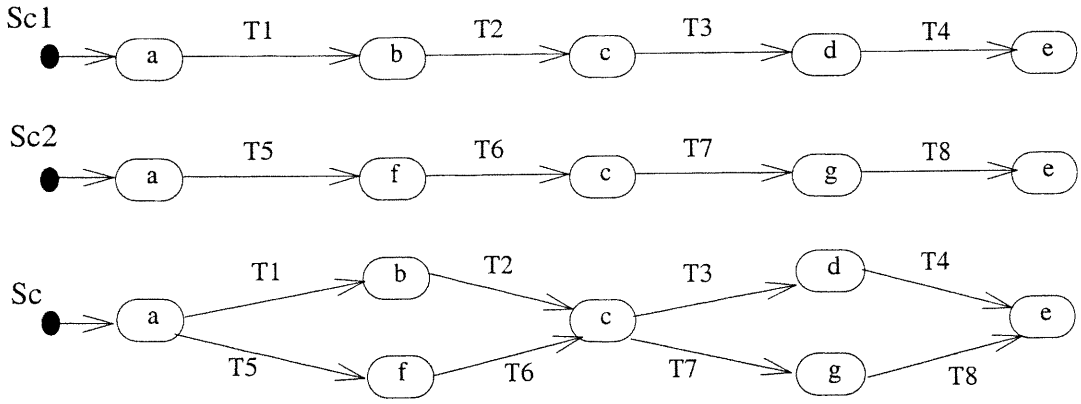


Figure 5.6: Le problème de chevauchement entre `Sc1` et `Sc2`

Pour toutes les transitions d'un StateD qui terminent un scénario, nous ajoutons une action de réinitialisation *ra* égale à *dynamicScenarioList:= scenarioList*. Pour les autres transitions, nous introduisons une condition spéciale *sc* égale à *[(transScenarioList[tr]* $\cap$ *dynamicScenarioList* $\neq \varnothing$*)]* (tr est l'indice d'une transition), et une action spéciale *sa* égale à *dynamicScenarioList:= dynamicScenarioList* $\cap$ *transScenarioList[tr]*.

Par exemple, pour l'exemple de la figure 5.5, nous obtenons les StateDs tel que décrit dans la figure 5.7. $\texttt{StateD}_1$ spécifie deux scénarios `sc1` et `sc2` déjà intégrés alors que $\texttt{StateD}_2$ spécifie un scénario `sc3`.

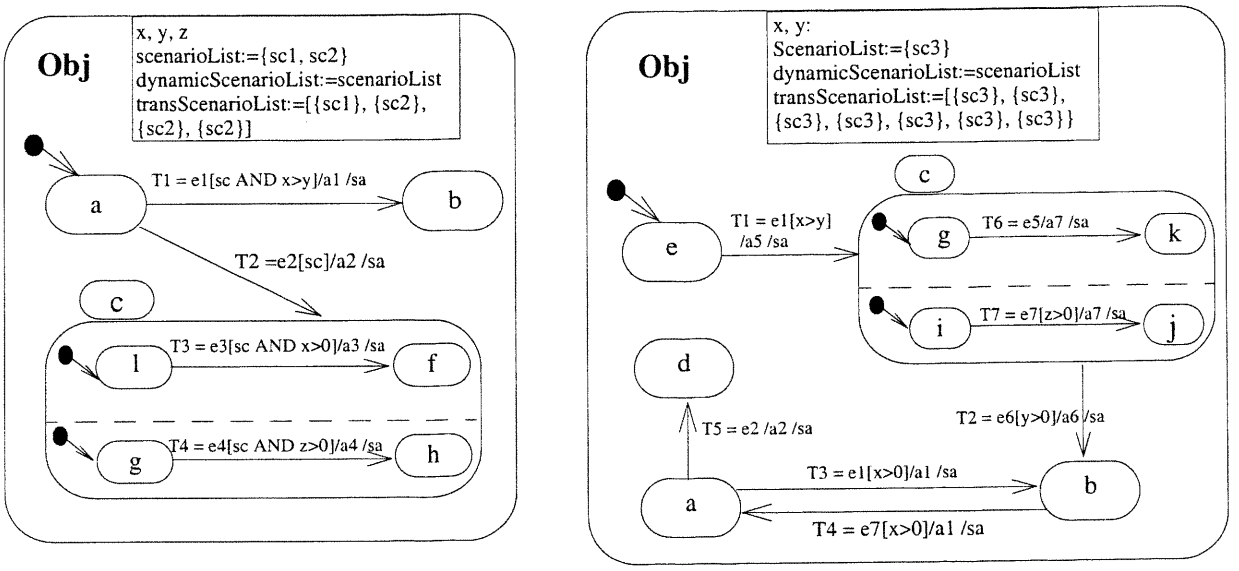


Figure 5.7: StateD_1 (à gauche) et StateD_2 (à droite) avec variables de composition

5.2.3 Fusion des états

Lorsqu'aucune erreur n'est détectée dans l'étape de vérification des états, l'algorithme procède à la fusion des états des deux StateDs ($\texttt{StateD}_1$ et $\texttt{StateD}_2$) du plus haut jusqu'au plus bas niveau de la hiérarchie des StateDs. Pour chaque niveau, l'algorithme vérifie les états initiaux des deux StateDs. Par exemple dans la figure 5.7, l'état `a` est l'état initial du plus haut niveau de $\texttt{StateD}_1$ et l'état `e` est celui de $\texttt{StateD}_2$.

Si les états initiaux d'un même niveau sont les mêmes alors une *fusion de type OU* est effectuée, ce qui correspond à la réunion de tous les états du même niveau des deux StateDs dans un état composite de type `OU`. Si les états initiaux sont différents alors une *fusion de type ET* est effectuée. Ceci revient aussi à réunir tous les états de même niveau dans un état composite de type OU, sauf qu'ici un nouvel état de type `ET` est créé pour contenir les états initiaux comme sous-états concurrents. La figure 5.8 montre les deux cas pour l'intégration de StateD_1 et StateD_2. Au plus haut niveau, une fusion de type `ET` est réalisée alors qu'au niveau de l'état c, une fusion de type `OU` est réalisée.

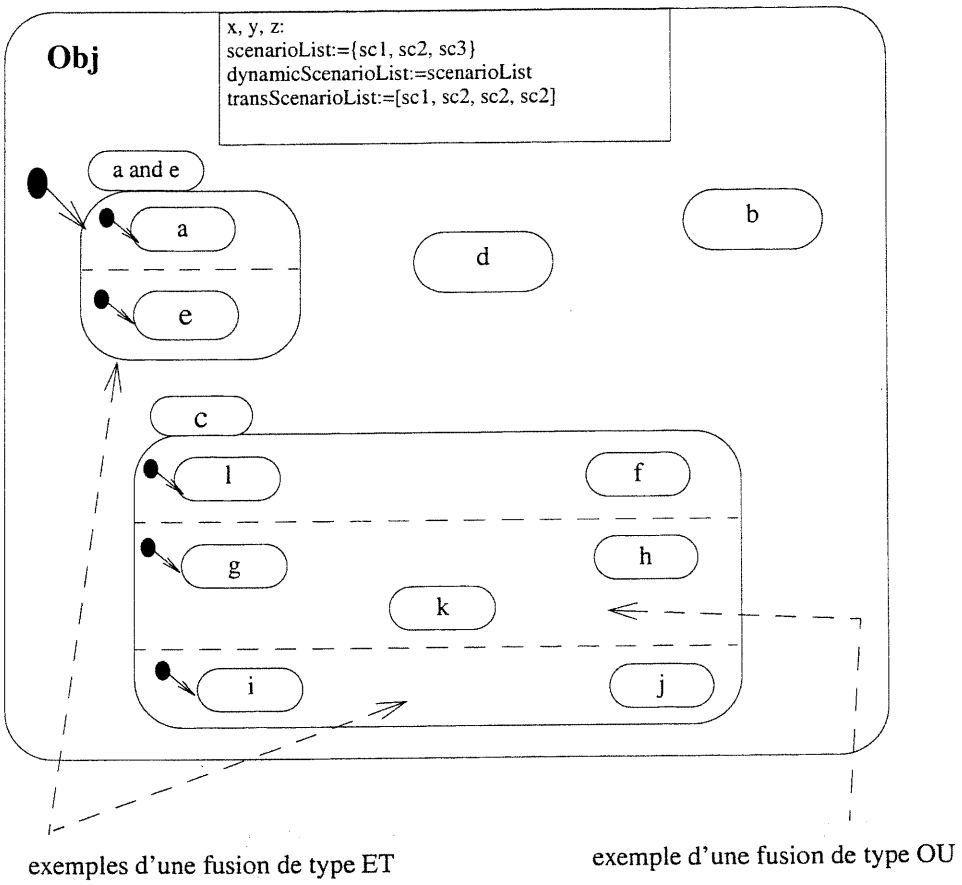


Figure 5.8: Le StateD résultant après la troisième étape de l'intégration de $StateD_1$ et $StateD_2$

5.2.4 Fusion des transitions

Dans cette étape, l'algorithme cherche dans les deux StateDs des paires de transitions ayant le même quintuplet, soient l'état `fromNode`, l'état `toNode`, le champ `event`, le champ `{/action}`, le champ `{sendClause}` (voir annexe C), pour qu'elles soient fusionnées. La condition de garde de la transition fusionnée devient la disjonction des conditions de garde des deux transitions. La figure 5.9 donne le résultat d'intégration des deux StateDs $StateD_1$ et $StateD_2$ de la figure 5.8 après cette étape.

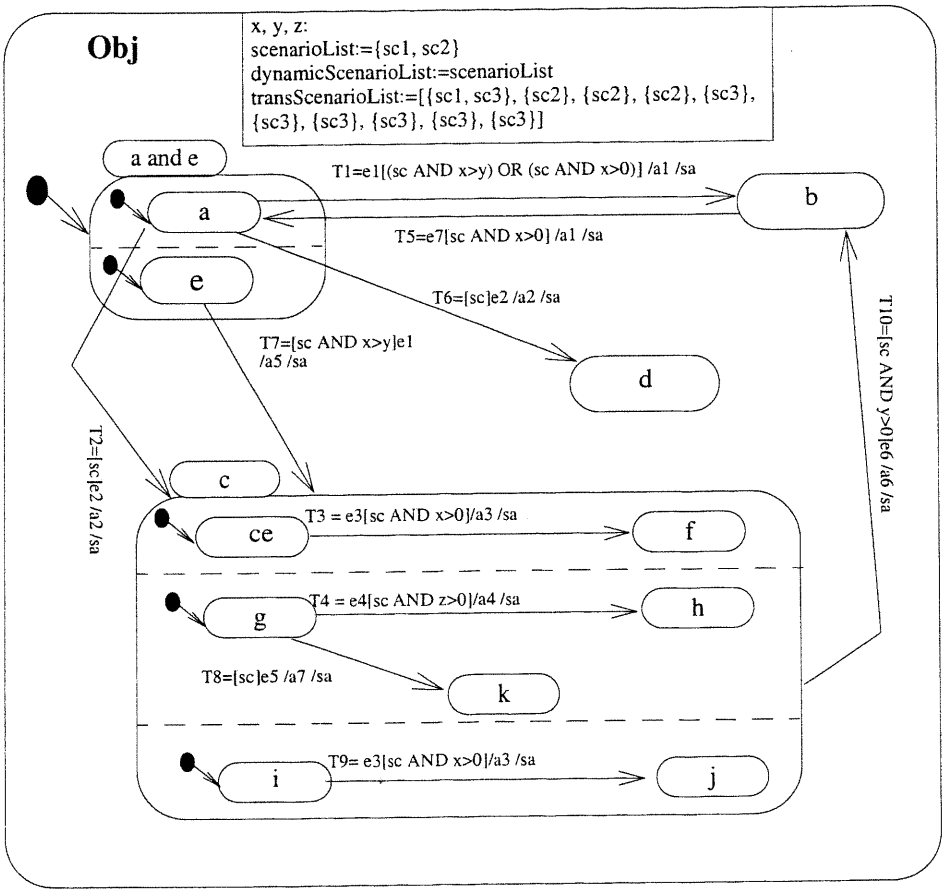


Figure 5.9: Le StateD résultat après l'étape 4 de l'intégration de $StateD_1$ et $StateD_2$

5.3 Vérification des comportements dynamiques des objets

Dans cette section, nous allons commencer par donner quelques définitions qui vont être utilisées dans les algorithmes de vérification. Notons que tous ces algorithmes concernent la vérification de la cohérence des scénarios. Remarquons aussi que l'objectif de notre approche est de dériver une spécification qui capture exactement les besoins tels que décrits par les utilisateurs. Cependant, quelques aspects d'incohérence, tel que le non déterminisme, cachent souvent de dangereux problèmes d'incomplétude. C'est pourquoi quelques problèmes d'incomplétude vont être résolus dans la vérification de la cohérence des scénarios. En outre, nous allons donner dans la section 5.3.3 quelques règles qui permettent de compléter une spécification donnée.

5.3.1 Définitions

Définition 5.17. Soit c une orExpression. *c* est *cohérente* si et seulement si Eval(c) ≠ Ø.

Définition 5.18. Soit c une ifExpression {OR ifExpression}. *c* est *cohérente* si et seulement si ∀i, j, $1 \leq i \leq n_i$, $1 \leq j \leq n_j$. Eval(expressionListOfOR(preCond(c))[i]) ≠ Ø et Eval(expressionListOfOR(postCond(c))[j]) ≠ Ø.

Définition 5.19. Soit m une méthode d'une classe C donnée, et soient pre(m) et post(m) la pre- et la post-condition de la méthode m respectivement. *post(m) est cohérente avec pre(m)* si et seulement si preCond(post(m)) raffine preTrans(m).

Définition 5.20. Dans un StateD, une transition trans est cohérente avec son fromNode et son toNode si et seulement si fromNode et toNode sont étiquetés, preTrans(trans) raffine la condition de son fromNode et la condition de son toNode raffine postTrans(trans).

Définition 5.21. Un *StateD exhibe un comportement non déterministe* si et seulement si il existe une transition $trans_1$ qui est caractérisée par $fromNode_1$ [$event_1$] [$guardCondition_1$] / {/$action_1$} {$sendClause_1$} [^$returnValue_1$] $toNode_1$ et une transition $trans_2$ par $fromNode_2$ [$event_2$] [$guardCondition_2$] {/$action_2$} {$sendClause_2$} [^$returnValue_2$] $toNode_2$ telles que $fromNode_1$ raffine $fromNode_2$ ou $fromNode_2$ raffine $fromNode_1$, $event_1$=$event_2$, $guardCondition_1$ et $guardCondition_2$ ne sont pas exclusives (c'est-à-dire que $eval(guardCondition_1) \cap eval(guardCondition_2) \neq \emptyset$), et:

(1) {$action_1$}≠{$action_2$} ou {$sendClause_1$}≠{$sendClause_2$}, et $toNode_1$= $toNode_2$, ou

(2) {$action_1$}={$action_2$}, {$sendClause_1$}={$sendClause_2$}, et $toNode_1$≠$toNode_2$.

Définition 5.22. Un *StateD* d'une classe C donnée est *cohérent* si et seulement s'il satisfait les conditions ci-dessous:

(1) les conditions de garde de toutes les transitions sont cohérentes,

(2) il n'a pas un comportement non déterministe,

(3) tous les états considérés comme étant de type OU satisfont la définition 5.7,

(4) tous les états considérés comme étant de type ET satisfont la définition 5.8,

Proposition 5.1. Si un StateD exhibe un comportement non déterministe causé par l'existence d'un ensemble de transitions qui satisfont la condition (2) de la définition 5.21 et les états *toNode* de ces transitions appartiennent au même niveau hiérarchique et satisfont les deux conditions de la

définition 5.8 alors l*e non déterminisme peut être résolu.*

Preuve. Pour résoudre ce non déterminisme, il suffit de créer un état de type ET qui contient tous les états toNode des transitions qui satisfont la condition (2) de la définition 5.21. Puis il faut ne garder qu'une seule de ces transitions. L'état de type ET nouvellement créé, qui satisfait bien la définition 5.8, devient le toNode de cette transition.

5.3.2 Cohérence

Nous avons défini trois sortes de vérification pour la cohérence des scénarios: la cohérence de la description des classes, la cohérences des transitions avec ses états de départ et d'arrivée et la cohérence des StateDs. Les trois sortes de vérification sont utilisées par l'algorithme d'analyse des spécifications partielles. L'algorithme d'intégration de deux StateDs n'utilise que la troisième sorte de vérification.

Cohérence de la description des classes

L'objectif ici est de vérifier que la description d'une classe donnée est cohérente. Cette opération s'assure que les pre- et post-conditions de toutes les méthodes de la classe sont cohérentes (voir définitions 5.17 et 5.18). En outre, la post-condition de chaque méthode doit être cohérente avec sa pre-condition (voir définition 5.19). En cas d'erreur, le développeur est invité à faire des corrections sur la description de cette classe.

Cohérence des transitions avec ses états de départ et d'arrivée

Cette opération est utilisée lors de l'opération d'étiquetage pour vérifier qu'une transition reste cohérente avec son fromNode et son toNode qui ont été étiquetés par une transition précédente (voir définition 5.20). Cette incohérence peut être causée par une description incohérente ou incomplète d'un scénario donné. Dans les deux cas, l'algorithme invite l'analyste à faire des corrections sur la description du scénario en question.

Par exemple, si nous supposons que dans la description du scénario `retraitRégulier` (voir figure 3.4), le message `2: motdepasse:=entrer_motdepasse()` $\rightarrow$ a été omis, le StateD générée de la classe GAB sera comme celui décrit dans la figure 5.10. Durant l'activité d'étiquetage, l'algorithme pourra détecter que la transition T_2 n'est plus cohérente avec son fromNode. En effet, le traitement de la transition T_1 va étiqueter son toNode avec la condition

`liquide_disponible=true and écran="entrer mot de passe" and fente_argent="fermée" and fente_carte="pleine"`, alors que `preTrans(`T_2`)` = `cliquide_disponible=true and écran="confirmer mot de passe" and fente_argent="fermée" and fente_carte="pleine"` (voir figure 3.3) et `preTrans(`T_2`)` ne raffine pas la condition exprimée par le fromNode de la transition T_2.

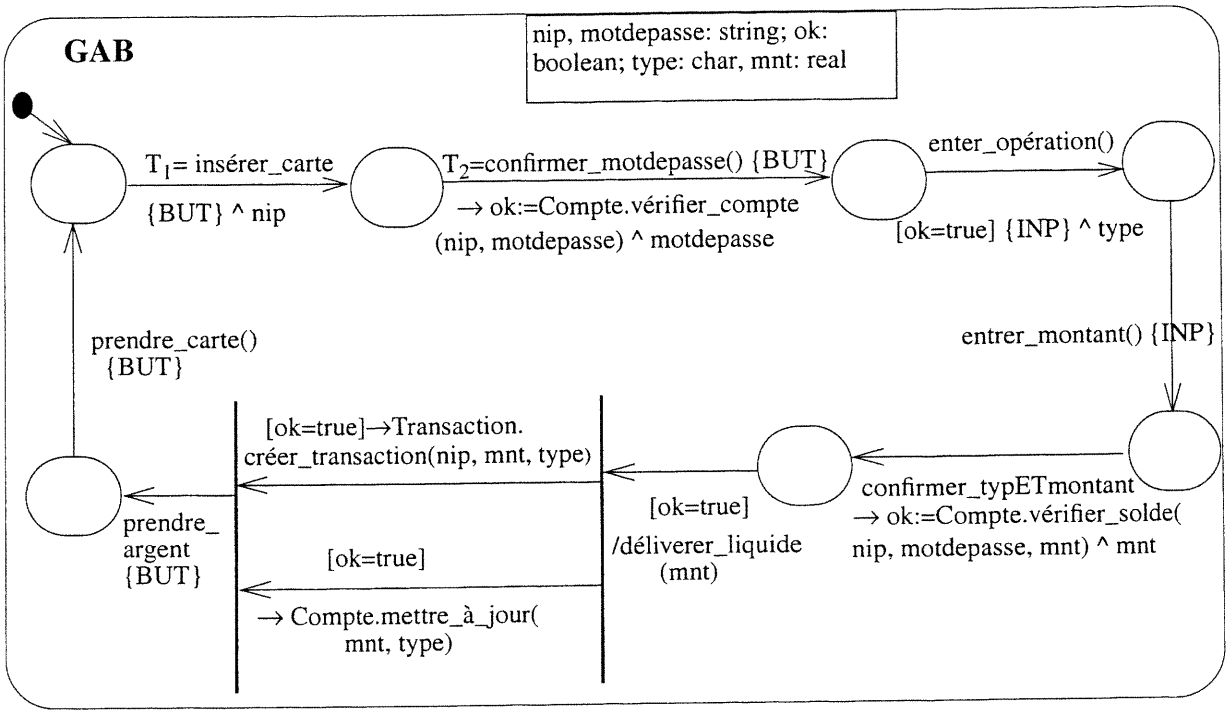


Figure 5.10: StateD de la classe `GAB` générée d'une version erronée du scénario donnée dans la figure 3.4

Cohérence des StateDs

Cette opération vérifie la cohérence d'un StateD donné (voir proposition 5.1). Deux cas sont à considérer. Si l'incohérence du StateD est causée par les conditions (1), (3) et (4) alors le développeur doit corriger les scénarios qui causent cette incohérence. Sinon, c'est-à-dire que l'incohérene est causée par un StateD non déterministe (condition (2)), alors l'algorithme propose, si c'est possible (voir proposition 5.1), un nouveau StateD où le non déterminisme est éliminé puis invite le développeur à confirmer le résultat (puisque comme nous avons déjà mentionné que le non déterminisme peut cacher des problèmes d'incomplétude). À titre d'exemple, la figure 5.9 montre un StateD avec un comportement non déterministe. En effet, quand le StateD est dans l'état `a and e`, il peut aller à l'état `c` ou à l'état `d` à la réception de l'événement `e2`. La résolution du non déterminisme est réalisée par la création d'un état de type `ET` qui contient ces deux états (c'est-à-dire

les états c et d) comme sous-états concurrents (voir figure 5.11).

5.3.3 Complétude

Il existe plusieurs façon pour définir une spécification complète. En effet, une spécification complète peut être définie comme celle qui contient tous les comportements requis par les utilisateurs. Une autre définition suggère qu'une spécification complète comme celle qui contient tous les aspects requis pour le système à construire même ceux qui ne sont pas exigés par les utilisateurs. La première définition est assurée par notre approche puisque les spécifications sont automatiquement générées à partir des scénarios. En outre, la vérification de la cohérence des scénarios permet la détection des scénarios incomplets causés par des erreurs dans leur description.

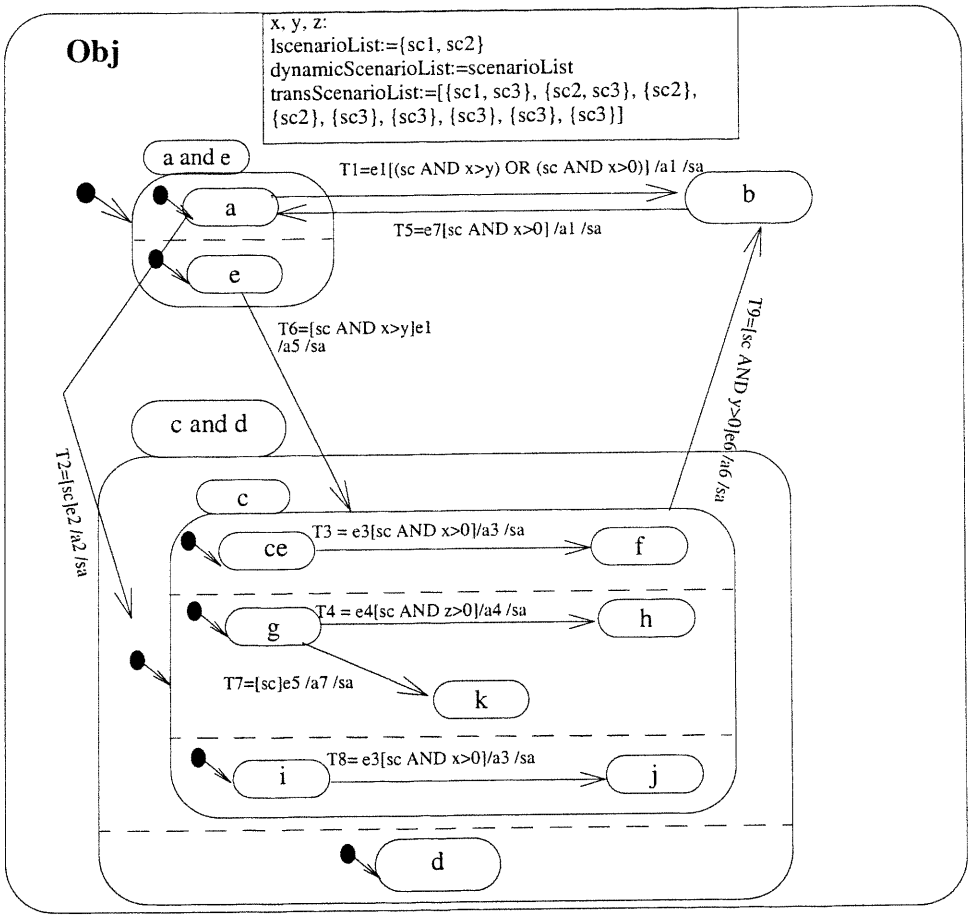


Figure 5.11: Le StateD résultant après l'élimination du non déterminisme du StateD de la figure 5.9

Pour le deuxième genre de complétude, nous pouvons utiliser des règles fournies par Heimdahl et Leveson [Heimdahl et Leveson, 1996]. En effet, ces règles vérifient la complétude d'une spécification donnée en ce qui concerne un ensemble de critères reliés à la robustesse, c'est-à-dire à la capacité d'avoir une réponse pour n'importe quelle entrée possible. Pour un StateD, la robustesse implique les règles suivantes:

1. Chaque état doit avoir une transition pour toutes les entrées possibles.
2. La disjonction des conditions des transitions qui se déclenchent à partir d'un état doit former une tautologie,
3. Chaque état doit avoir une transition définie dans le cas où il n'y a pas d'entrée après une période donnée (time-out).

5.4 Travaux reliés

Plusieurs travaux se sont intéressés à la synthèse des scénarios et/ou à la vérification des comportements dynamiques. Dans ce qui suit nous allons présenter les travaux que nous jugeons comme les plus pertinents.

5.4.1 Koskimies et Mäkinen

Dans le domaine du développement OO, le travail de Koskimies et Mäkinen [Koskimies et Mäkinen, 1994] est probablement le plus proche du notre. Ils présentent un algorithme SMS (state machine synthesis) pour la synthèse des diagrammes des StateDs à partir d'un ensemble de scénarios. Leur travail est basé sur l'application des idées de l'algorithme proposé par Biermann et Krishnaswamy [Biermann et Krishnaswamy, 1976] qui synthétise un programme à partir des exemples de traces d'exécution.

À l'opposé de l'algorithme SMS, notre approche utilise les possibilités de concurrence offerte par les StateDs pour supporter la concurrence qu'on peut trouver dans un CollD. L'incrémentalité de SMS est différent de la nôtre. En effet pour ce dernier, la trace d'un nouveau scénario est concaténée avec la trace des scénarios déjà intégrés dans le but de produire un nouveau StateD qui capture tous les scénarios, tandis qu'avec notre approche l'intégration d'un nouveau scénario se fait en intégrant le StateD des scénarios précédemment traités avec le StateD nouvellement généré à

partir du CollD du nouveau scénario. En d'autres termes, notre approche propose des algorithmes qui non seulement transforment un CollD en StateDs mais aussi qui permettent d'intégrer des StateDs. Finalement, notre approche diffère de l'algorithme SMS par les décisions de conception prises. À titre d'exemple, SMS définit un état à partir d'une action d'un objet, alors que notre approche le définit comme une condition sur les attributs d'un objet.

Koskimies et son groupe ont développé un outil appelé SCED [Koskimies et al., 1998] qui supporte *la conception par exemple* (synthèse des scénarios) et *la conception par animation.* Cette dernière consiste à faire une exécution symbolique des StateDs dans le but de générer des traces d'exécution dans une forme d'un diagramme de séquence. SCED permet aux développeurs d'augmenter des scénarios par la spécification des événements à envoyer par des objets existants et/ou la définition des objets manquants. La conception par l'animation permet à SCED d'être utilisé dans certaines tâches de la retro-ingénierie (reverse engineering).

5.4.2 Deharnais et al.

Desharnais et al. [Deharnais et al., 1998] définissent un scénario comme une union de deux relations *Re* et *Rs* où *Re* représente la relation de l'environnement qui capture toutes les actions possibles de l'environnement, et *Rs* la relation correspondant à la réaction du système. L'intégration des scénarios est donnée par la composition des relations de scénarios.

Cette approche ne supporte que les systèmes séquentiels et par conséquent ne permet pas de capturer des comportements concurrents.

5.4.3 Glinz

Glinz propose les StateDs comme formalisme de représentation des scénarios [Glinz, 1995]. En outre, il propose des extensions à ce formalisme pour la composition des scénarios. Cette composition est basée sur l'ordre d'exécution des scénarios et est de quatre types: séquence (A après B), alternance (A ou B), itération (n fois A), concurrence (A concurrent avec B).

Glinz ne traite pas le problème de chevauchement des scénarios et laisse au développeur le soin de régler le problème avant la composition, c'est-à-dire que le développeur doit intégrer à la main des scénarios qui se chevauchent.

5.4.4 Dano et al.

Dano et al. proposent une approche basée sur les cas d'utilisation pour produire une spécification du comportement dynamique d'un système [Dano et al., 1997]. Au préalable, les cas d'utilisation sont capturés dans des tables appelées PUCT (pour Partial Use Case Table). Chaque table contient une suite séquentielle de fonctions qui permettent d'atteindre l'objectif du cas d'utilisation. Elle contient aussi, pour chaque fonction, les états des objets concernés par le cas d'utilisation et des conditions supplémentaires autres que les états.

Après, ils génèrent un réseau de Petri pour chaque PUCT. Ces réseaux de Petri sont par la suite composés pour obtenir un seul réseau de Petri qui modélise le comportement dynamique de tout le système. La composition est basée sur des liens temporels entre les différents cas d'utilisation. Les liens sont: avant, rencontre, commence, termine, égale, durant et chevauche.

La principale limite de ce travail vient du fait que les états doivent être définis explicitement par le développeur. En outre ce travail ne supporte que des cas d'utilisation exprimés sous forme d'une suite séquentielle de fonctions.

5.4.5 Kawashita

Kawashita propose une approche d'intégration des scénarios pour obtenir une spécification d'un système [Kawashita, 1996]. Cette approche est divisée en deux phases: une analyse des scénarios et leur intégration.

L'analyse des scénarios a pour but d'extraire des scénarios toutes les données utiles pour la phase d'intégration. Ces données sont les objets avec leurs attributs, leurs relations dans les scénarios et les liens possibles entre les scénarios.

L'intégration des scénarios consiste à obtenir une spécification complète du système. Elle est obtenue en combinant les états des objets dans un seul automate à états finis étendu. Cette intégration ne tient pas compte du fait que les scénarios peuvent dépendre ou non les uns des autres et ne traite pas la concurrence.

5.4.6 Lee et al.

Lee et al. [Lee et al., 1998] proposent une formalisation des scénarios avec une extension des réseaux de Petri, appelée les *Constraint-based Modular Petri Nets* (CMPNs). Un CMPN possède une structure interne et une interface externe. La structure interne d'un CMPN est semblable à celle d'un réseau de Petri. L'interface externe est spécifiée par son nom et un ensemble de places et transitions partagées. Ces places et transitions partagées sont utilisées pour la synchronisation d'un ensemble de CMPNs. Pour chaque cas d'utilisation, un CMPN est dérivé à partir de sa description informelle (une description tabulaire).

La spécification résultante consiste en un ensemble de CMPNs concurrents où chaque CMPN spécifie un cas d'utilisation, alors que notre approche génère, pour chaque objet, un StateD qui intègre tous les scénarios. Les places des CMPNs sont extraites des pre- et post-conditions des actions. Cependant, Lee et al. supposent que les états d'un système sont connus et les pre- et post-conditions des actions ne font que référencer à ces états. Or, dans notre approche, les états des objets sont dérivés des conditions sur les attributs d'un objet. Les auteurs donnent aussi un ensemble de règles pour trouver des incohérences et des incomplétudes dans les CMPNs.

5.4.7 Somé et al.

Somé et al. proposent un algorithme d'intégration de scénarios dans une spécification en utilisant un formalisme basé sur les automates temporisés où l'exécution des transitions est conditionnée par des variables et des contraintes d'horloges [Somé et al., 1995].

Pour chaque scénario, l'algorithme d'intégration consiste à chercher dans la spécification une trace partielle qui accepte ce scénario. Dans le cas où cette trace n'existe pas, la spécification est augmentée par des états et des transitions pour qu'elle puisse exécuter le scénario à intégrer. Même s'il peut générer un ensemble d'automates temporisés qui s'exécutent en parallèle, l'algorithme ne traite pas le problème de concurrence au sein d'un scénario.

5.4.8 Heimdahl et Leveson

Heimdahl and Leveson [Heimdahl et Leveson, 1996] proposent un outil pour la vérification de la complétude et de la cohérence dans les StateDs. Par conséquent, cet outil peut analyser les StateDs qu'on synthétise. À l'instar de notre approche, l'explosion des états est éliminée par

l'application de l'analyse à un niveau élevé d'abstraction. Ceci veut dire qu'au lieu de générer un graphe d'accessibilité, l'analyse est effectuée directement sur le modèle.

Un ensemble de règles de composition définies par Heimdahl et Leveson sont aussi identifiées dans notre approche. Par exemple, la règle *composition avec union,* qui interdit que deux transitions soient exécutées en même temps à partir d'un même état, est aussi identifiée par notre travail (voir définition 5.22). Enfin, il faut prendre note que Heimdahl et Leveson ne supportent que des conditions de garde plus restrictives que les nôtres. Ces conditions ne contiennent que des prédicats indépendants. En effet, si une condition de garde `gc` est égale à `p et q` alors il faut que le prédicat `p` $\Rightarrow$ `q ou q` $\Rightarrow$ `p` soit égal à faux pour que `gc` soit supportée par leur travail. Or, notre approche supporte ce genre de conditions.

5.4.9 STATEMATE

STATEMATE [Harel et al., 1990] est un outil commercial qui supporte des langages visuels pour la description d'un système en développement dans trois vues différentes: structurelle, fonctionnelle, et dynamique. La vue dynamique est capturée par des StateDs. L'outil permet la génération automatique du code ainsi que la simulation des systèmes pour les besoins de vérification.

Nous considérons STATEMATE comme un outil complémentaire à notre approche. Les StateDs synthétisés par notre approche pourraient être passés à STATEMATE pour la simulation et l'analyse. En outre, les StateDs décrits dans STATEMATE pour les objets d'interface pourraient être utilisés comme entrées de notre approche pour la génération d'un prototype de l'IU (voir section 7.1). Par conséquent, les deux outils combinés permettent la simulation des aspects fonctionnels et IU d'un système. En effet, STATEMATE peut simuler les StateDs des objets non IU d'un système, alors que notre approche peut simuler son interface usager.

5.5 Discussion

Ci-dessous, nous allons discuter quatre points importants concernant notre approche: restriction sur les conditions, problème de chevauchement des scénarios, complexité et implantation des algorithmes et expériences.

Restriction sur les conditions

Nous avons vu que la syntaxe d'une condition (c'est-à-dire, pre- et post-condition d'une méthode, condition de garde d'une transition) n'est définie que sur un sous-ensemble d'OCL (voir annexes A et C). Cette restriction n'a pas un grand impact sur la généralité de notre approche. En effet, un grand nombre de systèmes réels peuvent être supportés par notre approche puisqu'il suffit de voir que notre restriction est moins forte que les hypothèses faites par d'autres travaux (par exemple le travail de Heimdahl et Leveson vu dans la section précédente).

En outre, cette restriction nous permet d'effectuer des opérations sur les conditions (l'opération *raffine* (voir définition 5.5), de savoir si deux conditions sont égales (voir définition 5.4), et de savoir si deux conditions sont exclusives ou non (voir définition 5.7) dans un temps polynômial; or Heimdahl et Leveson ont indiqué que ce genre d'opérations prennent un temps exponentiel [6] [Heimdahl et Leveson, 1996].

Problème de chevauchement des scénarios

Dans ce travail, nous avons résolu le problème de chevauchement entre les scénarios par la définition de trois variables de composition tout en respectant la syntaxe et la sémantique des StateDs. Notons que dans certains cas le chevauchement peut être désiré dans le but d'explorer de nouveaux scénarios qui n'ont pas été encore considérés. L'exécution ou la non-exécution de l'étape 2 de l'algorithme d'intégration rend possible ou exclut le chevauchement entre scénarios.

Complexité des algorithmes

Nous avons vu dans ce chapitre deux algorithmes, le premier qui fait l'analyse d'un StateD, et le second qui fait l'intégration de deux StateDs.

Pour la discussion de la complexité c_A de l'algorithme d'analyse d'un StateD, soient N_{OP} le nombre des opérations dans la description d'une classe, N_T le nombre des transitions dans un StateD,

6. En effet, dans notre approche, ces opérations sont réalisées à partir de l'utilisation de la fonction d'évaluation Eval (voir définition 5.1). Or cette fonction est polynômiale. En ce qui concerne le travail de Heimdahl et Leveson, ces opérations passent par le calcul des propositions. Rappelons que ce dernier est un formalisme qui étudie la vérité des propositions (ou prédicats) obtenues par la composition de propositions élémentaires. Par exemple, la formule `p et q` est une expression du calcul des propositions et n'a de valeur que si l'on affecte des valeurs de vérité aux propositions qui y figurent. Une fonction qui affecte des valeurs de vérité aux propositions est appelée une fonction d'interprétation. Or, pour savoir si une expression est vraie, il faut vérifier s'il existe au moins une fonction d'interprétation qui la rend vraie. Cette recherche prend un temps exponentiel.

N_S le nombre des états dans un StateD, et N_{or} le nombre maximum de ifExpression que peut avoir les post-conditions des transitions du StateD. Puisque notre algorithme est composée de quatre étapes, C_A est la somme de C_{A1}, C_{A2}, C_{A3} et C_{A4} où C_{Ai} représente la complexité de l'étape i. L'étape 1 vérifie la cohérence de toutes les opérations dans une classe. C_{A1} est par conséquent de $O(N_{OP})$. L'étape 2 calcule la pre- et la post-condition de toutes les transitions du StateD. Par conséquent, C_{A2} appartient à $O(N_T)$. L'étape 3 étiquette tous les états du StateD. L'étiquetage consiste à un parcours, avec backtracking, de toutes les transitions du StateD pour trouver la bonne suite des ifExpressions des post-conditions des transitions avec laquelle elle peut réussir. Donc au pire cas, C_{A3} appartient à $O(N_{or}{}^{N_T})$. L'étape 4 vérifie la cohérence du StateD résultant, ce qui revient à vérifier la cohérence des conditions de garde de toutes les transitions, à vérifier l'existence d'un comportement non déterministe et à vérifier la cohérence entre les sous-états de tous les états composites du StateD. C_{A3} est par conséquent de $O(N_T+N_T{}^2+N_S{}^2)$. Finalement, C_A appartient à $O(N_{or}{}^{N_T})$.

Notons que la complexité exponentielle de l'algorithme d'analyse n'est pas problématique pour deux raisons principales. La première raison est que cette complexité est obtenue dans le pire cas, c'est-à-dire que l'algorithme ne réussit qu'après avoir effectué toutes les tentatives possibles. La deuxième raison est que l'algorithme en question traite des StateDs partiels où le nombre des transitions n'est pas très grand puisque chacun de ces StateDs ne spécifie qu'un seul scénario.

Pour la discussion de la complexité C_I de l'algorithme d'intégration de deux StateDs, soient N_{T1} (respectivement N_{T2}) le nombre des transitions dans le premier StateD (respectivement dans le second StateD), N_{S1} (respectivement N_{S2}) le nombre des états dans le premier StateD (respectivement dans le second StateD). Puisque notre algorithme est composé de 5 étapes, C_I est la somme de C_{I1}, C_{I2}, C_{I3}, C_{I4} et C_{I5} où C_{Ii} représente la complexité de l'étape i. L'étape 1 vérifie l'existence des conflits entre les états des deux StateDs. C_{I1} est par conséquent de $O(N_{S1} * N_{S2})$. L'étape 2 introduit les trois variables de composition dans les deux StateDs. Donc C_{I2} appartient à $O(\max(N_{T1}, N_{T2}))$. L'étape 3 fusionne les états des deux StateDs niveau par niveau. L'étape 3 est donc de $O(\max(N_{S1}{}^2, N_{S2}{}^2))$. L'étape 4 fusionne les transitions des deux StateDs. Par conséquent, l'étape 4 appartient à $O(N_{T1} * N_{T2})$. L'étape 5 vérifie la cohérence du StateD résultant de l'intégration des deux StateDs. Donc C_{I5} est égale à C_{A3} et par conséquent elle appartient à $O(\max(N_{T1}, N_{T1}) + \max(N_{T1}, N_{T1})^2+\max(N_{S1}, N_{S1})^2)$. Par conséquent, C_I est de $O(\max(N_{T1}, N_{T1})^2+\max(N_{S1}, N_{S1})^2)$.

Implantation des algorithmes et expériences

Les deux algorithmes ont été implantés avec un système de 22 classes et quelques 4500 lignes de code en langage Java (commentaires non inclus). Les deux algorithmes ont été testés sur des exemples de systèmes de taille petite allant jusqu'à six scénarios pour un seul système.

Pour le test de l'algorithme d'analyse, nous avons en premier lieu testé individuellement toutes ses fonctions importantes. Une emphase a été mise sur les opérations qui traitent les conditions: évaluation d'une condition, égalité de deux conditions, union de deux conditions et intersection de deux conditions. Ensuite, nous avons pris les mêmes cas de test de l'algorithme précédent pour tester l'algorithme d'analyse en entier (voir section 4.4). Puis, nous avons dérivé des cas de test pour valider toutes les opérations de vérification: cohérence de la description des classes, cohérence des transitions avec ses états de départ et d'arrivée et cohérence des StateDs.

Pour le test de l'algorithme d'intégration, nous avons pris comme cas de test les exemples de systèmes qui sont décrits dans la section 4.4. Le nombre total des cas de test des deux algorithmes tourne autour de 35. Pour tous ces cas de test, le temps d'exécution était excellent (quelques secondes sur un Sun Sparc 10/514 à 4 processeurs SuperSparc de 50 MHz).

Chapitre 6 Application automatique des patrons de conception

Dans la section 3.2, nous avons présenté une vue générale de notre deuxième objectif, à savoir fournir un support automatique pour la transition d'une conception générale vers une conception détaillée. Cette transition est réalisée par des raffinements successifs où chaque étape est basée sur l'application d'un patron de conception (voir figure 3.15). Comme exemple d'illustration, nous allons décrire dans la première section de ce chapitre [1] le schéma du raffinement que nous avons défini pour le patron Observer ainsi que les schémas de ses micro-raffinements. Notons que nous décrivons en annexe H le schéma de raffinement que nous avons défini pour le patron Médiator.

Dans la deuxième section, nous allons présenter le cadre sémantique pour les preuves de validité des différents raffinements. Ce cadre contient, en plus d'une définition formelle d'un raffinement valide, une description sémantique du sous-ensemble d'UML qui est pertinent pour notre travail. La section sera close par une preuve de la validité du schéma de raffinement Observer. Dans la troisième section, nous allons présenter les travaux les plus reliés à l'application automatique des patrons de conception. Nous allons terminer ce chapitre par une discussion sur plusieurs points qui concernent notre approche de raffinement avec les patrons de conception: décomposition et composition des raffinements, intérêts de l'approche pour les méthodes OO de développement et pour les outils CASE.

6.1 Schéma de raffinement pour Observer

6.1.1 Description

Comme exemple d'illustration, nous allons utiliser une partie de la conception du système de gestion d'une station de service. Le but du système est de contrôler le débit d'essence, gérer le règle-

1. Les résultats de ce chapitre sont publiés dans [Khriss et Keller, 1999a; Khriss et al., 1999c] et décrits dans le rapport technique [Khriss et Keller, 1999b].

ment des clients, et contrôler le niveau des réservoirs d'essence. La figure 6.1 montre deux classes, `Pompe` et `Écran`, du ClassD du système. La classe `Pompe` est conçue pour contrôler le débit d'essence, et la classe `Écran` pour montrer le volume d'essence délivré. La figure 6.2 décrit les modèles dynamiques (StateDs) de ces deux classes.

Le développement de ce système exige, entre autres, une façon pour implanter la contrainte existante entre les classes `Pompe` et `Écran`. Le patron Observer (cf. chapitre 2), offre une bonne solution à ce problème. En effet, il définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour [Gamma et al., 1995].

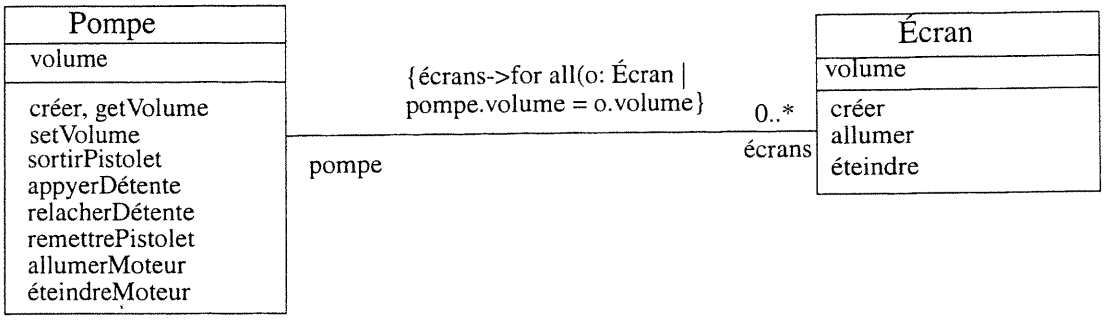


Figure 6.1: Une partie du ClassD du système de gestion d'une station de service

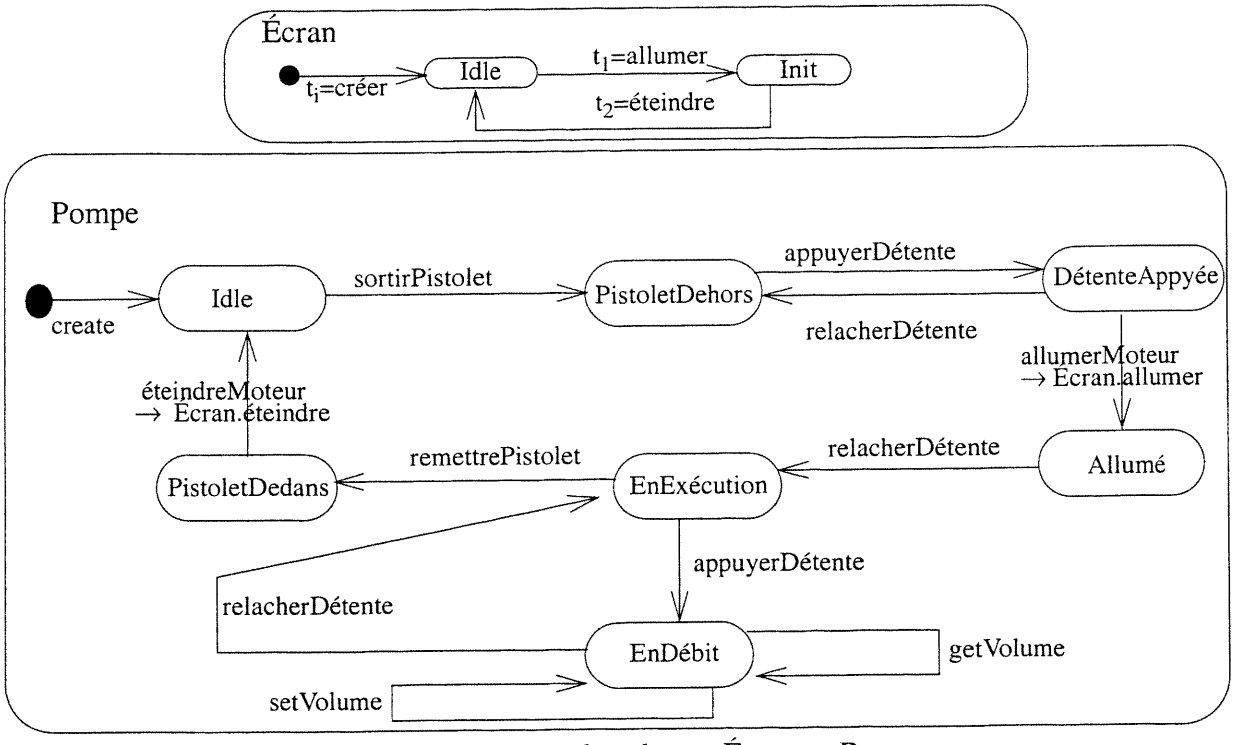


Figure 6.2: StateDs des classes Écran et Pompe

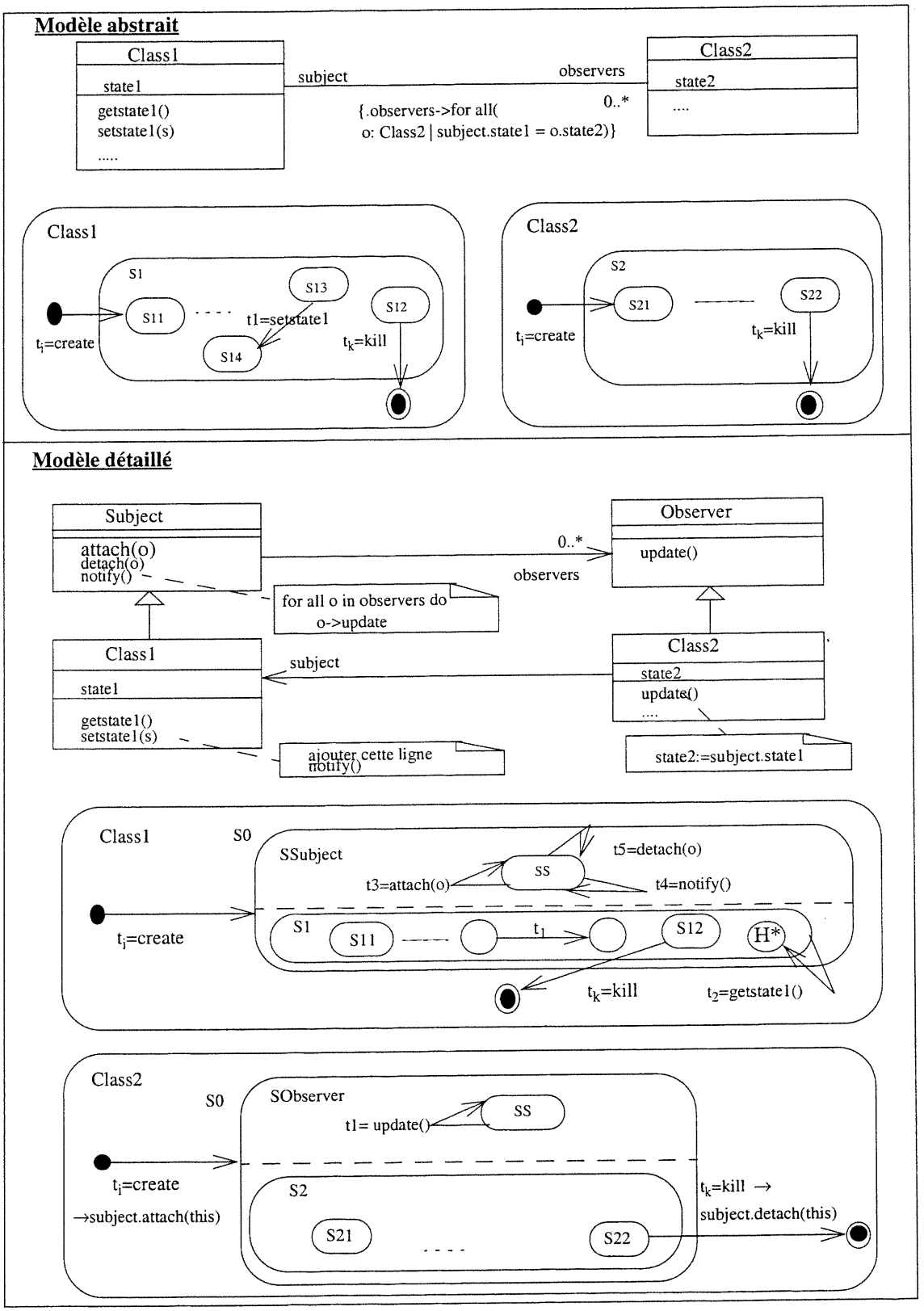


Figure 6.3: Schéma du raffinement du patron Observer

La figure 6.3 montre le schéma de raffinement que nous avons défini pour le patron Observer. Ce schéma de raffinement a huit paramètres (voir le pseudo-code dans la section H.2): une association entre deux classes avec une contrainte, l'attribut de la première classe correspondante à l'extrémité gauche de l'association, l'attribut de la deuxième classe correspondant à l'extrémité droite de l'association, l'opération qui retourne la valeur de l'attribut de la première classe, l'opération qui modifie la valeur de l'attribut de la première classe, la classe Subject, la classe Observer, le classD du système. Le schéma de raffinement d'Observer met à jour le ClassD du système et les StateDs des deux classes en question. En fait, ces changements sont le résultat des différents schémas de micro-raffinement utilisés par le schéma de raffinement d'Observer. Nous allons décrire en détail ces raffinements dans la prochaine section.

Les figures 6.4 et 6.5 montrent comment la conception à haut niveau du système de gestion d'une station de service est transformée en une conception détaillée lors de l'application du schéma de raffinement du patron `Observer`. Cette transformation est appliquée pour implanter la contrainte entre les classes `Pompe` et `Écran`. La classe `Pompe` devient une sous-classe de `Subject`. Son comportement est mis à jour pour pouvoir notifier à tous ses observateurs, et en particulier à la classe `Écran`, tout changement de son attribut `volume`. La classe `Écran` devient une sous-classe de `Observer`. Son comportement est mis à jour pour que à chaque création d'un objet, ce dernier devient un observateur d'un objet de `Subject` et par conséquent d'un objet de `Pompe`.

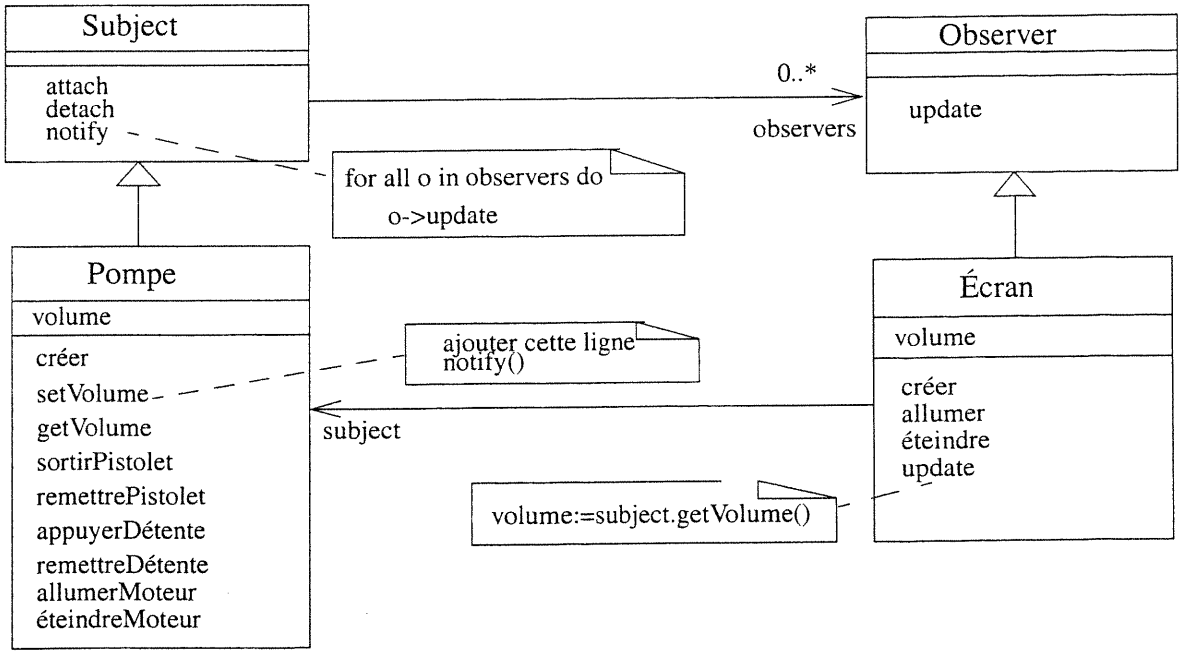


Figure 6.4: Le ClassD après application du schéma de raffinement de Observer

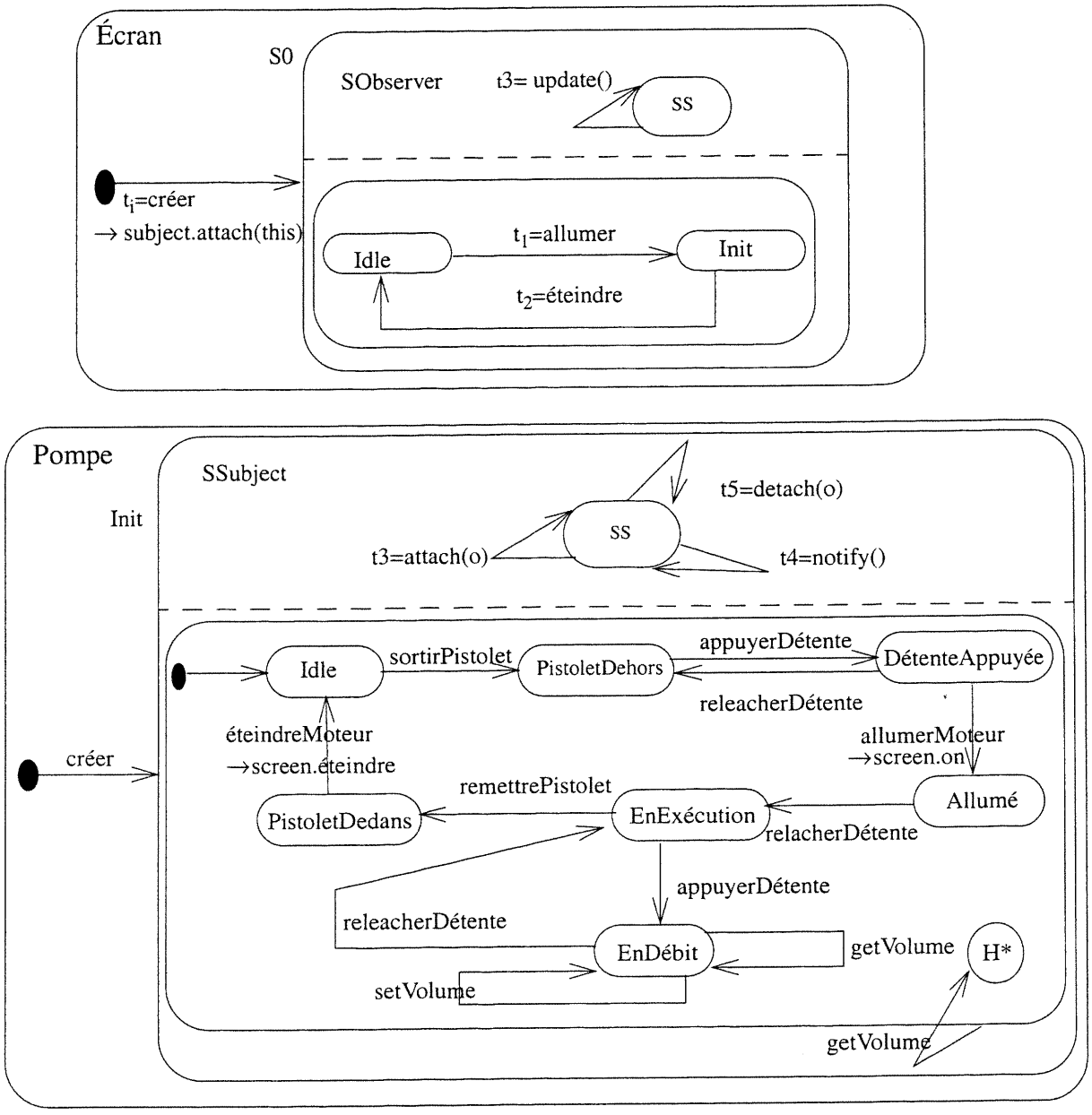


Figure 6.5: StateDs des classes Écran et Pompe après l'application du schéma de raffinement de Observer

6.1.2 Les schémas de micro-raffinement utilisés par Observer

Le schéma de raffinement du patron Observer est composé d'une séquence de quatre schémas de micro-raffinement. Premièrement, le modèle abstrait est transformé en appliquant deux fois le

schéma de micro-raffinement *héritage* (voir figure 6.6). Ce raffinement ajoute une nouvelle classe `Subject` (respectivement `Observer`) qui devient une super-classe pour la classe `Class1` (respectivement `Class2`). Le StateD de `Class1` (respectivement de `Class2`) est mis à jour pour ajouter le comportement de `Subject` (respectivement de `Observer`).

Deuxièmement, le schéma de micro-raffinement *ajout d'une action à une transition* est appliquée deux fois sur le StateD de la classe `Class2` (voir figure 6.7). Ce StateD est mis à jour pour ajouter une action à la transition t_i et une autre à la transition t_k. Troisièmement, l'association bi-directionnelle `<Class1, Class2>` est remplacée par deux associations uni-directionnelles `<Subject, Observer>` et `<Class2, Class1>` (voir figure 6.8).

Finalement, la contrainte sur les deux associations est remplacée par une notification automatique à `Class2` pour mettre à jour son attribut dès que l'attribut de `Class1` correspondant est modifié (voir figure 6.9). En effet, l'instruction `notify()` est ajoutée à la fin de l'opération `setstate1` de `Class1` dans le but de notifier tous les observateurs (et par conséquent aux objets de `Class2`) de `Class1` pour mettre à jour leurs attributs. Ensuite, le StateD de `Class1` est modifié dans le but d'accepter en tout temps l'événement `getstate1`. La transition `t2` du StateD de `Class1` veut dire que quand un objet est dans n'importe quel sous-état de `S1` et reçoit un événement `getstate1`, `t2` est déclenchée sans le faire changer d'état.

6.2 Preuve de validité des raffinements

6.2.1 Un cadre sémantique pour les preuves

Un formalisme pour UML

Pour une représentation formelle des modèle d'UML, nous utilisons un formalisme proposé par Lano et Bicarregui [Lano et Bicarregui, 1999] appelé Real-time Action Logic (RAL). Cette logique est une synthèse de la logique à temps réel [Jahanian et Mok, 1986] avec la logique temporelle et linéaire [Ostroff, 1989] et le formalisme d'*object calculus* [Fiadeiro et Maibaum, 1991].

Une classe `C` d'UML est représentée par une théorie Γ_C. Une théorie est décrite par son nom, les types de ses symboles, un ensemble d'attributs représentant les variables de la classe, un ensemble d'actions qui peuvent affecter une donnée (telles que une opération, une transition d'un StateD ou une méthode) ainsi qu'un ensemble d'axiomes qui sont des propriétés logiques décrivant des con-

traintes sur les symboles.

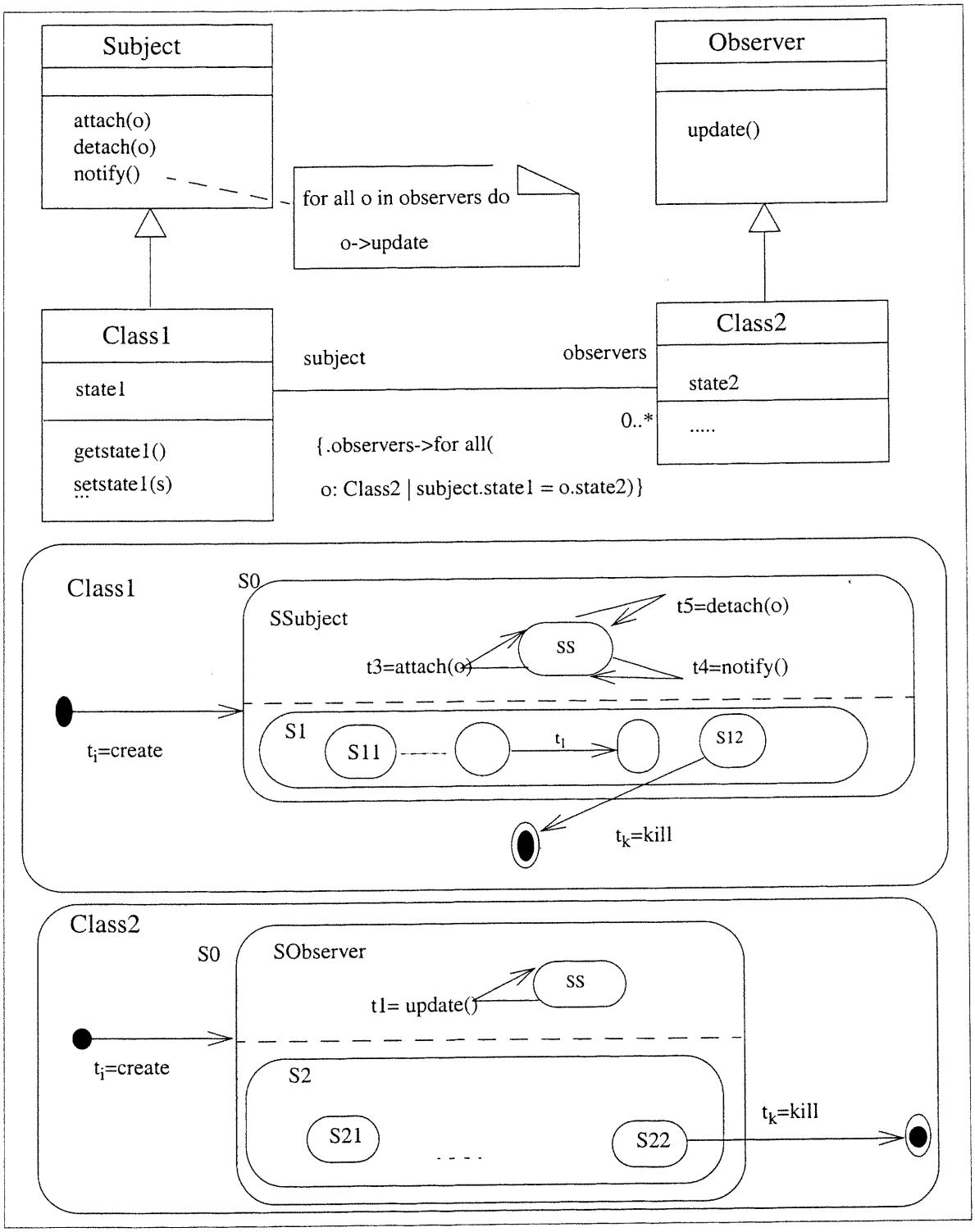


Figure 6.6: Le résultat du premier schéma de micro-raffinement *héritage* utilisé par Observer

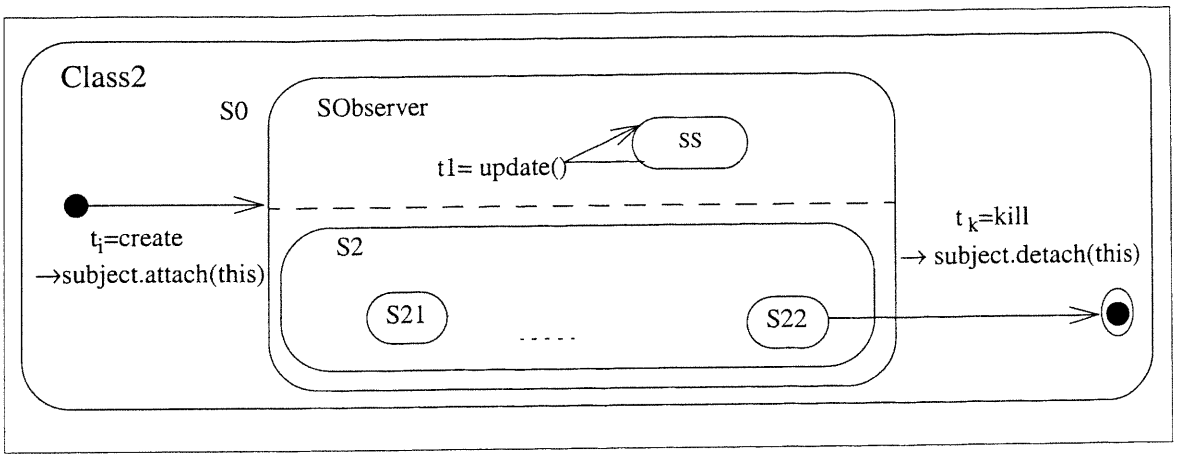


Figure 6.7: Le résultat du deuxième schéma de micro-raffinement *ajout d'une action à une transition* utilisé par Observer

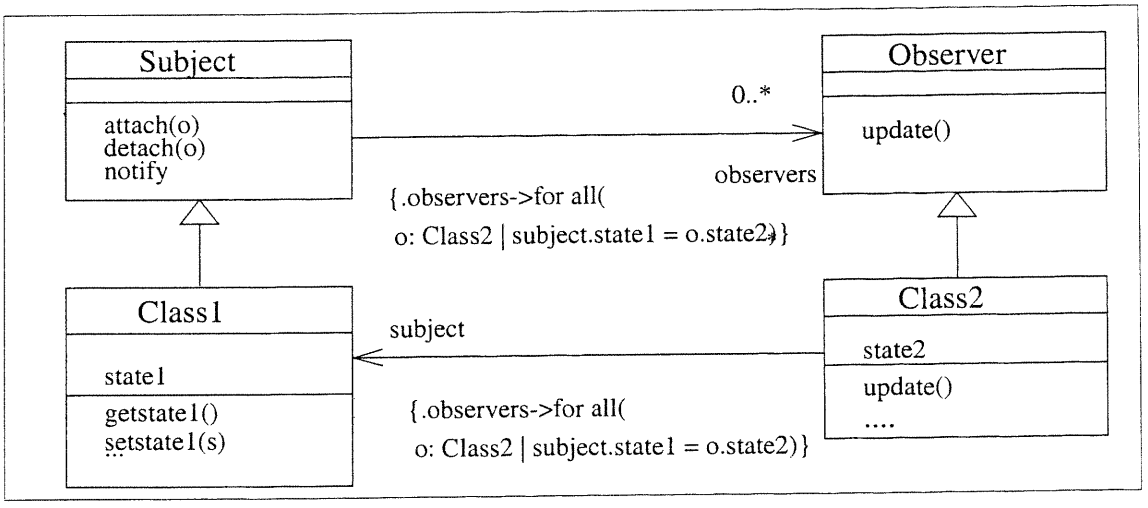


Figure 6.8: Le résultat du troisième schéma de micro-raffinement *changement d'association* utilisé par Observer

Les théories peuvent être connectées par des morphismes de théories, ce qui permet de décrire un modèle d'UML (un ClassD par exemple) à partir d'un assemblage de théories des éléments tels que une classe ou une association. En outre, les théories utilisent, en plus des notations mathématiques standard, les notations empruntées de la logique à temps réel, de la logique temporelle et linéaire et du formalisme d'object calculus. Nous trouvons par exemple:

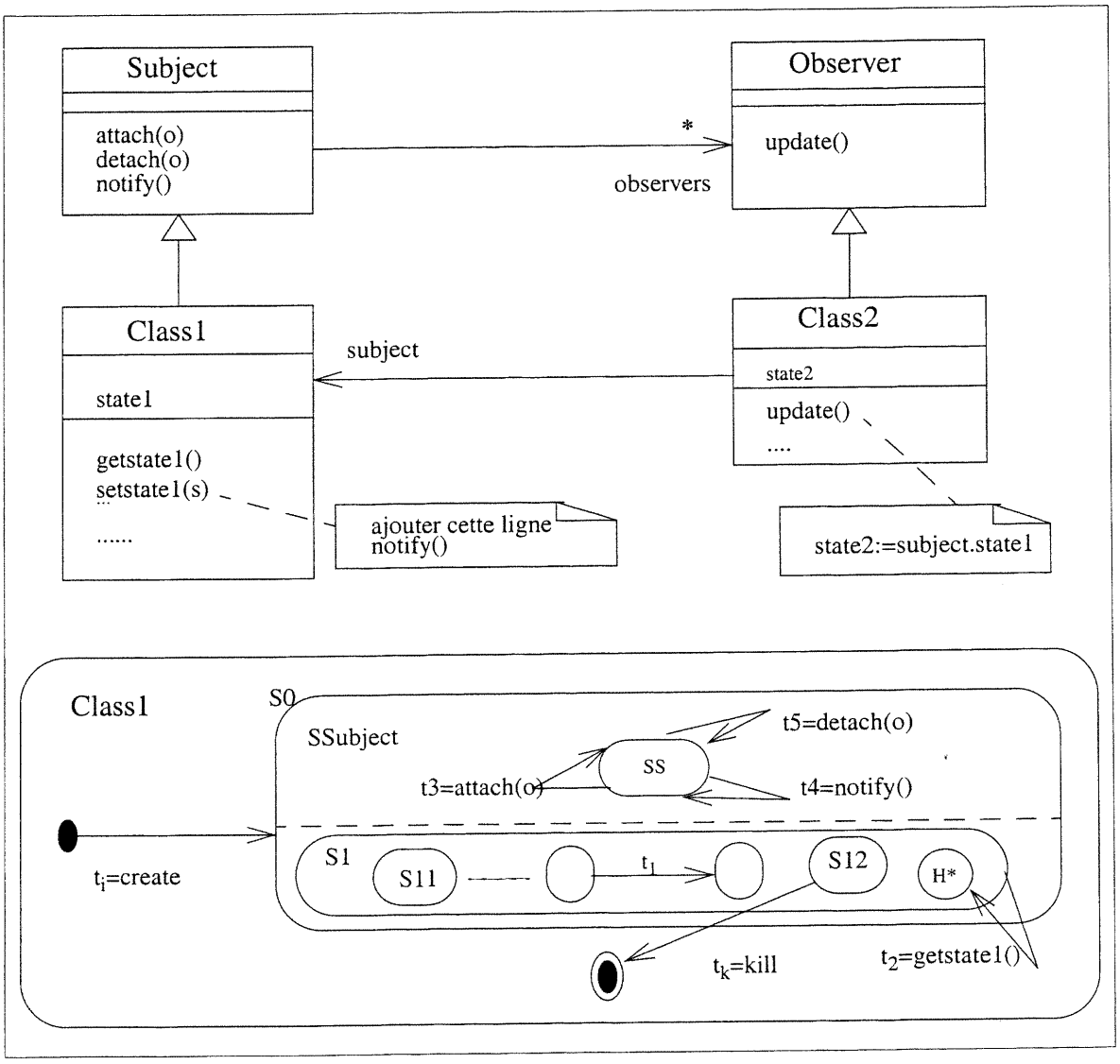


Figure 6.9: Le résultat du quatrième schéma de micro-raffinement *notification automatique* utilisé par Observer

1. *ext(X)* est un attribut défini pour chaque classe ou état X. Il exprime l'ensemble des instances existantes de X. ext(X) appartient à F(X), c'est-à-dire, l'ensemble des ensembles finis de X.
2. Si α est un symbole d'une action et P un prédicat alors $[\alpha] P$ est un prédicat disant que P est la post-condition de α.
3. $\alpha \supset \beta$ ("α appelle β", α et β sont des actions) est défini tel que "α appelle toujours β quand elle s'exécute".
4. les opérateurs temporels telles que $\Diamond$ ("quelques fois dans le futur"), $\Box$ ("toujours dans le

futur") et ○ ("suivant"). Notons que les opérateurs temporels sont très utiles dans la description formelles des diagrammes comportementaux tels que les CollDs et les StateDs.

Si D hérite de C alors Γ_D est construite par l'inclusion de Γ_C en ajoutant les symboles et axiomes des nouveaux éléments de D ainsi que les deux axiomes $D \subseteq C$ et $ext(D) \subseteq ext(C)$.

Un StateD d'une classe C est formalisé par l'extension de sa théorie Γ_C comme suit:

1. chaque sous-état S est représenté de la même manière qu'une sous-classe de C.
2. chaque transition du StateD et chaque événement acceptés par le StateD donnent lieu à un symbole d'action distinct. Aussi l'occurrence d'un événement e est équivalant à l'occurrence de l'une de ses transitions: $t_1 \supset e \wedge \ldots \wedge t_n \supset e$.
3. l'axiome qui définit l'effet de la transition t avec l'événement e avec la condition de garde G de l'état S1 à l'état S2 est: $\forall c: C.\ c \in ext(S1) \wedge a.G \Rightarrow [c!t]\ (c \in ext(S2))$ ("!" pour indiquer que t est une action de c alors que pour un attribut on utilise le point comme dans les langages OO).
4. une transition est déclenchée seulement dans le cas où l'événement correspondant survient alors que le StateD est dans le bon état: $\forall c: C.\ c \in ext(S1) \wedge a.G \Rightarrow (c!e \supset c!t)$.
5. Les actions liées à une transition t doivent s'exécuter à un futur proche: $a!t \Rightarrow \bigcirc \Diamond c.Action$.

Par exemple, la théorie correspondante à la classe Écran est $\Gamma_{Écran}$ et elle est décrite ci-dessous.

```
Theory Γ_Écran
types Écran, Pompe, Idle, Init,
attributes
 ext(Écran): F(Écran), ext(Idle): F(Idle), ext(Init): F(Init)
 self: Écran → Écran
 volume: Screen → R
actions
 créer_Écran(c: Écran)
 kill_Screen(c: Écran)
 on(c: Écran), of(c: Écran), getVolume(c: Écran)
```

```
 t_i(c:Écran), t_1(c:Écran), t_2(c:Écran), t_3(c:Écran)
axioms
 Idle ⊆ Écran ∧ ext(Idle) ⊆ ext(Écran)
 Init ⊆ Écran ∧ ext(Init) ⊆ ext(Écran)
 t_i ⊃ créer_Écran ∧t_1 ⊃ on ∧t_2 ⊃ off ∧t_3 ⊃ getVolume
 ∀ c: Écran.
   self(c) = c ∧ [c!t_i](c ∈ ext(Idle)) ∧
   (c ∈ ext(Idle)) ⇒[c!t_1(s)](c ∈ ext(Init)) ∧
   (c ∈ ext(Init)) ⇒ ([c!t_2](c ∈ ext(Idle)) ∧
   (c ∈ ext(Init)) ⇒ c!t_3 ∧
   c!créer_Écran ⊃ c!t_i ∧
   (c ∈ ext(Idle)) ⇒ (c!allumer ⊃ c!t_1) ∧
   (c ∈ ext(Init)) ⇒ (c!éteindre ⊃ c!t_2)
   (c ∈ ext(Screen)) ⇒ (c.volume=c.pump.volume)
```

Définition formelle des raffinements

Un système CR raffine un système C s'il existe un morphisme σ de la théorie Γ_C de C à la théorie Γ_{CR} de CR tel que chaque axiome φ de Γ_C sous σ est démontrable dans Γ_{CR}.

6.2.2 Preuve de validité du schéma d'Observer

Comme nous avons indiqué dans la section 6.1.3, le schéma de raffinement pour le patron Observer est composé de quatre petits raffinements. Le premier raffinement utilisant le schéma de micro-raffinement héritage est valide comme il est montré dans la section G.3. Le deuxième raffinement utilise le schéma de raffinement ajout d'une action dans une transition. Ce raffinement est aussi valide (voir la preuve dans la section G.4).

Il est évident que le troisième raffinement est correcte puisque la classe `Class1` hérite de la classe `Subject` l'association unidirectionnelle avec la classe `Observer` (et par conséquent avec la classe Class2). Pour le quatrième raffinement nous avons à démontrer l'axiome suivant:

c: Class2. c ∈ ext(Class2) ⇒ c.state2 = c.subject.state1

Dans le but de prouver que cet axiome reste valide dans le modèle raffiné et sur la condition que

`state1` n'est mis à jour que par `setstate1`, nous avons à prouver que:

$\forall$ c: Class2. c $\in$ ext(Class2) $\wedge$ c.subject!setstate1(s)$\Rightarrow$ $\bigcirc$ $\Diamond$ (c.state2 = s)

Dans le modèle raffiné, nous avons l'axiome:

$\forall$ c: Class2. c $\in$ ext(Class2) $\wedge$ c.subject!setstate1(s) $\Rightarrow$ $\bigcirc$ $\Diamond$ (for all o in c.subject.observers do o!update())

$\Rightarrow$ $\bigcirc$ $\Diamond$ (c!update()) puisque c $\in$ observers

$\Rightarrow$ $\bigcirc$ $\Diamond$ (c.state2 = c.subject!getstate1())

$\Rightarrow$ $\bigcirc$ $\Diamond$ (c.state2 = c.subject!getstate1())

$\Rightarrow$ $\bigcirc$ $\Diamond$ (c.state2 = s)

6.3 Travaux reliés

Dans cette section nous allons présenter des travaux qui ont pour objectif l'automatisation de l'application des patrons de conception. Ces travaux différent de notre approche dans la mesure qu'ils s'intéressent à la génération du code à partir d'une spécification du patron de conception. À l'opposé, nous utilisons les patrons de conception dans notre travail pour l'intégration de deux niveaux d'abstraction de la conception. Notons que la principale faiblesse de ces travaux réside dans le fait qu'ils sont obligés de fixer le choix des variantes d'implantation des patrons de conception. Cependant, ils ont l'avantage de s'intéresser aux moyens pour spécifier les patrons de conception dans des langages qui permettent la génération du code.

6.3.1 Lano et al.

Lano et al. utilisent les patrons de conception pour la réingénierie des systèmes procéduraux [Lano et al., 1996]. La réingéniere est réalisée par des transformations des spécifications écrites en langage VDM++ [IFAD, 1998]. Des preuves de validité sont aussi données. En outre, les auteurs fournissent un catalogue de petites transformations réutilisables. Deux de ces transformations, *abstraction* et *indirection* sont aussi identifiées par notre présent travail comme schémas de

micro-raffinement MR2 et MR5 (voir figure 6.10).

À l'opposé des sources (les modèles abstraits) de nos schémas de raffinement qui doivent être à un haut niveau d'abstraction, la source et la cible de leurs transformations sont à un niveau très détaillé. Ceci rend leur travail peu général et par conséquent va limiter le champ de son application. Finalement, il faut noter que dans ce travail, les auteurs ne considèrent pas les aspects dynamiques d'une conception.

6.3.2 O'Cinnéide et Nixon

O'Cinnéide et Nixon [O'Cinnéide et Nixon, 1999] présentent une méthodologie pour l'automatisation de l'application des patrons de conception dans la réingénierie. Ce travail s'inspire du travail d'Opdyke qui s'intéresse au "refactoring" des programmes en C++ [Opdyke, 1992]. En effet, ils définissent de la même manière les pre- et post-conditions des transformations pour démontrer leur préservation du comportement des programmes.

Les transformations sont décomposées en mini-transformations qui sont exprimées en terme d'un "refactoring" à un niveau plus bas. D'après leur article, il apparaît que jusqu'à maintenant, ils n'ont travaillé qu'avec un seul patron (le patron Factory [Gamma et al., 1995]). Comme nous l'avons déjà noté avec le travail de Lano et al. décrit ci-dessus, le point initial d'une transformation est peu général et va donc limiter l'application de l'approche de O'Cinnéide et Nixon à grande échelle.

6.3.3 Budinsky et al.

Budinsky et al. [Budinsky et al., 1996] fournissent un outil qui donne un accès rapide à la description des patrons de conception, organisé sous forme de page HTML. Grâce à un browser, le concepteur navigue entre ces pages jusqu'à ce qu'il trouve la solution qui lui convient. Ensuite, il peut choisir certains compromis d'implémentation prédéfinis. Puis le concepteur peut copier les morceaux du code C++ engendrés et les insérer à l'intérieur du code source de son application en développement.

En fait, l'outil est composé de trois modules: le Presenter, un browser Web qui sert d'interface entre l'utilisateur et l'outil; l'interpréteur COGENT (Code Generator Template) qui interprète les spécifications et génère le code C++; et le Mapper qui gère la coopération entre les deux premiers.

6.3.4 Eden et al.

Eden et al. [Eden et al., 1997] proposent Pattern Wizard, un outil capable de produire du code en Eiffel correspondant à un patron de conception. Un patron est décrit dans un méta-langage appelé PSL (Pattern Specification Language), une combinaison entre Smalltalk et une syntaxe abstraite. Ils présentent aussi dans [Eden et Yehuday, 1997] un catalogue de micro-patrons, appelés astuces (tricks), qui peuvent être réutilisés pour l'implantation des patrons de conception.

6.3.5 Meijers

Meijers présente PatternTool [Meijers, 1996], un outil qui permet l'intégration des patrons dans du Code Smalltalk. Cet outil utilise un modèle en fragements pour représenter des patrons de conception. Ces fragements sont, par l'intermédiaire de rôles, liés aux classes qui vont réaliser concrètement une instance d'un patron.

En fait, les fragements sont une hiérarchie d'objets, où chaque objet est une agrégation d'un ou plusieurs "slots". Il existe trois types de slots: fragement, primitif et code. Les slots de fragements maintiennent une référence à un fragement d'un type particulier pour représenter des rôles. Les slots primitifs peuvent contenir une référence à un objet Smalltalk quelconque, permettant ainsi aux instances des patrons d'avoir leurs propres variables. Enfin, les slots de code contiennent un bloc de "byte code" correspondant à un comportement.

6.3.6 Rapicault et Blay-Fornarino

Rapicault et Blay-Fornarino proposent un méta-protocole pour la définition, l'instanciation et la vérification des patrons de conception [Rapicault et Fornarino, 2000]. Ce protocole se présente sous forme d'une architecture de trois couches: Meta Pattern Protocol, Meta Pattern et Design Pattern. Le Meta Pattern Protocol est une entité composée de modèles de méta-classes, de méta-relations, de méta-méthodes, de méta-variables et de méta-contraintes. Le Meta Pattern est une entité composée de modèles de classes, de relations, de méthodes, de variables et de contraintes. Il sert à la définition d'un patron de conception. Un Design Pattern est l'instanciation d'un Meta Pattern.

L'utilisation d'un méta-protocole pour la définition des patrons de conception rend sa description non compréhensible et difficile à lire. En outre, nous pensons que le méta-modèle d'UML [UML,

1998] est largement suffisant, avec une adéquate utilisation du concept des stéréotypes, pour la définition d'un Meta Pattern sans avoir à passer par le Meta Pattern Protocol.

6.3.7 Reiss

Reiss présente le système PEKO qui fournit un ensemble d'outils pour utiliser les patrons de conception durant le processus de développement [Reiss, 2000]. Ces outils permettent au programmeur de maintenir un ensemble d'occurrences de patrons de conception dans le système et de s'assurer que ces patrons restent valides quand le système évolue. Pour cela, Reiss utilise un langage de définition de patrons de conception construit sur le langage OO de requêtes (OQL pour le *object-oriented query language* [Cattell et Barry, 1997]).

Grâce à la définition d'un patron de conception dans ce langage, le système PEKO peut générer son code correspondant, identifier ses instances dans un code et vérifier la cohérence d'un code qui a évolué avec sa définition.

6.3.8 Sunyé

Sunyé décrit PatternGen, un prototype d'outil de génération automatique de code à l'aide des patrons de conception qui prend en compte les variantes d'implémentation [Sunyé, 1999]. Le prototype contient un éditeur de diagrammes de classes d'OMT [Rumbaugh et al., 1991] pour modéliser une instance d'un patron de conception. En outre, il fournit des boîtes de dialogue pour que le concepteur puisse spécifier lui-même les compromis d'implémentation. Grâce à une base de règles de production, exprimant des connaissances de conception et d'implémentation, PatternGen peut analyser les compromis et les participants de chaque instance d'un patron, suggérer des modifications dans le diagramme de classes et générer le code spécifique aux variantes d'implémentation d'un patron.

PatternGen accepte la modification des caractéristiques d'un patron ainsi que l'ajout de nouveaux patrons grâce à un méta-modèle qui représente le langage MAC (pour Méthode, Attribut et Classes) où les patrons apparaissent comme des entités atomiques.

6.4 Discussion

Ci-dessous, nous allons discuter de certains points liés à notre approche et soient: décomposition et composition des raffinements, intérêts de l'approche pour les méthodes OO de développement et pour les outils CASE.

Décomposition/Composition

Nous avons vu que les schémas de raffinement des patrons de conception peuvent être décomposés en petits raffinements valides. Une importante facette de cette approche est que non seulement nous permettons la réutilisation du code mais aussi nous garantissons leur validité. Ces schémas de micro-raffinement peuvent être composés pour produire des schémas de raffinement valides. La figure 6.10 présente les schémas de micro-raffinement utilisés dans les schémas de raffinement correspondants aux patrons de conception étudiés. Les deux colonnes (en couleur grise) montrent que deux schémas de micro-raffinement sont réutilisés par trois schémas de raffinement.

	MR1	MR2	MR3	MR4	MR5	MR6	MR7	MR8	MR9	MR10	MR11	MR12
Observer		X		X		X	X					
Mediator	X				X	X			X	X		
Facade					X			X				
Proxy	X				X	X						
Forwarder-Receiver			X								X	X

Légende:

MR1 abstraction
MR2 héritage
MR3 ajout d'un comportement concurrent
MR4 ajout d'une action dans une transition
MR5 indirection
MR6 changement d'association
MR7 notification automatique
MR8 unification des interfaces
MR9 centralisation du flot de contrôle
MR10 suppressions des interaction non nécessaires
MR11 "forwarding" un message
MR12 "forwarding" un message "acknowledged"

Figure 6.10: Les schémas de micro-raffinement utilisés par les schémas de raffinement de cinq patrons de conception

En outre, ces schémas de micro-raffinement peuvent être utilisés seuls puisqu'ils aussi fournissent des solutions à certaines tâches de développement. Par exemple, le schéma de micro-raffinement *héritage* (MR2), qui utilise le schéma de micro-raffinement *ajout d'un comportement concurrent* (MR3), synthétise le nouveau StateD d'une classe qui possède une super-classe. Le nouveau StateD devient un état composite de type ET contenant deux sous-états concurrents: l'ancien

StateD et le StateD de la super-classe. Remarquons que cette solution n'est pas générale (à notre connaissance, il est impossible d'obtenir une solution générale) mais reste suffisant pour résoudre plusieurs problèmes, comme nous avons vu avec le schéma de raffinement du patron Observer.

Intérêts de l'approche pour les méthodes OO

En général, les méthodes de développement OO suivent un processus itératif et incrémental et sont composés de cinq étapes: analyse, conception générale, conception détaillée, codage et test. Durant la conception générale, une stratégie globale est développée pour la solution du problème capturé dans le modèle du monde réel (le résultat de l'étape d'analyse). Ceci consiste à organiser le système en sous-systèmes, allouer des sous-systèmes aux processeurs, choisir une approche pour la gestion des données persistantes, etc. Dans la conception détaillée, la définition complète des classes, interfaces, associations, et opérations est réalisée.

Avec notre approche, la phase de conception devient un raffinement successif du modèle du monde réel. En effet, les patrons de conception offrent des solutions pour différents problèmes qu'on peut rencontrer durant la conception. Par exemple, le patron Observer que nous avons déjà introduit fournit une bonne solution pour l'implantation d'une contrainte qui peut exister entre deux objets du monde réel.

Intérêts de l'approche pour les outils CASE

Les outils CASE actuels supportent plusieurs notations graphiques pour la modélisation des vues multiples d'un système. Cependant, ils n'offrent pas des possibilités de transformation automatique et de traçabilité entre les différents modèles. Ceci complique la tâche des développeurs pour s'assurer de la cohérence des modèles durant les étapes du cycle de développement. L'incorporation de notre travail dans de tel outil CASE va permettre d'avoir cette traçabilité. En outre, les outils peuvent conserver les choix de conception grâce aux transformations réalisées sur les conceptions.

Avant de son intégration dans un outil CASE, notre approche doit avoir un catalogue plus large de patrons de conception supportés. Nous sommes conscients que l'ensemble des schémas de micro-raffinement ne peut être complet sans l'étude d'un grand éventail de patrons de conception. C'est pourquoi nous comptons continuer ce travail dans le but d'extraire d'autres schémas de micro-raffinement et par conséquent enrichir notre catalogue de patrons supportés.

Chapitre 7 Applications et Outils

Dans ce chapitre [1], nous allons commencer par décrire les différentes étapes de l'algorithme de génération d'un prototype de l'IU à partir de la spécification des scénarios. Rappelons que cet algorithme correspond à la dernière activité de notre approche qui offre des outils automatiques pour le support du processus de l'ingénierie des besoins à l'aide des scénarios (voir figure 3.2). Ensuite nous allons esquisser notre vision d'un environnement logiciel supportant une approche transformationnelle pour le développement des systèmes OO.

7.1 Prototypage de l'interface usager à partir des scénarios

7.1.1 Description de l'algorithme

L'algorithme de génération d'un prototype de l'IU est composé de cinq étapes:

1. Génération d'un graphe de transitions.
2. Masquage des transitions non interactives.
3. Identification des blocs de l'IU.
4. Composition des blocs de l'IU.
5. Génération des fenêtres et widgets de l'IU.

Génération d'un graphe de transitions

Cette étape consiste à dériver un graphe orienté GT des transitions pour chaque StateD d'un objet d'interface qui intègre les scénarios d'un cas d'utilisation. Les transitions du StateD vont représenter les noeuds du graphe GT alors que les arcs vont indiquer la précédence d'exécution entre les transitions. Par exemple, si la transition T_1 précède la transition T_2 dans l'exécution alors un arc va relier le noeud représenté par T_1 à celui de T_2.

1. Une partie des résultats de ce chapitre sont publiés dans [Elkoutbi et al., 1999a] et décrits dans les rapports techniques [Elkoutbi et al., 1999b; Khriss et al., 2000].

Un graphe GT possède une liste de noeuds nodeList, une liste d'arcs edgeList, et une liste de noeuds initiaux initialNodeList (les noeuds d'entrées du graphe). La liste des noeuds nodeList d'un GT est facilement obtenue puisqu'elle correspond à la liste de transition du StateD. La liste des arcs edgeList d'un GT est obtenue par l'identification, pour chaque transition T, de toutes les transitions qui entrent dans l'état à partir duquel T peut être déclenchée. Toutes ces transitions précèdent la transition T et définissent donc chacune un arc vers le noeud représenté par T (voir [Elkoutbi et al., 1999a] pour plus de détail).

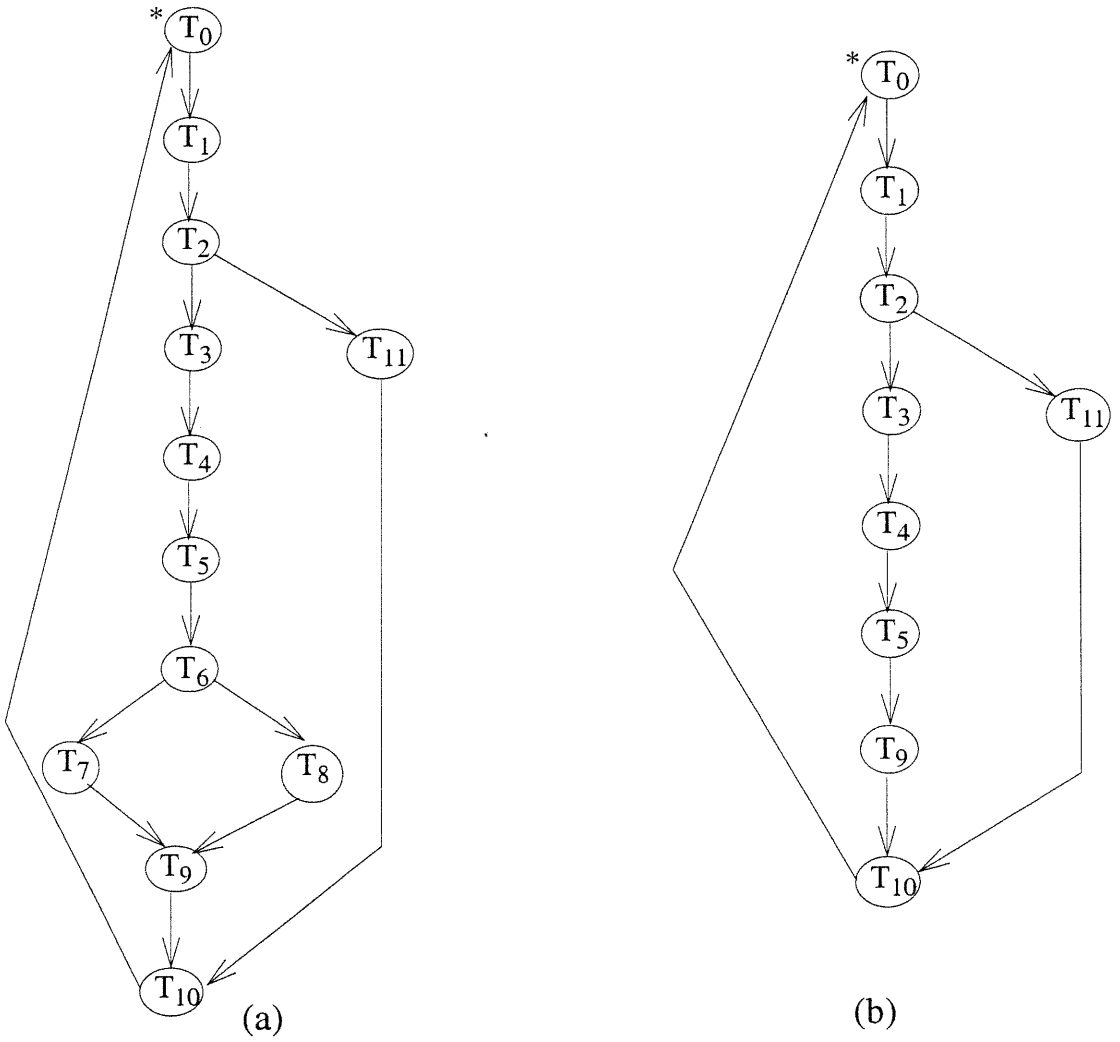


Figure 7.1: (a) Le graphe de transitions pour l'objet GAB et le cas d'utilisation Retrait (GT)
(b) Le graphe de transitions après masquage des transitions non interactives (GT')

Pour le StateD de la classe GAB pour le cas d'utilisation Retrait (voir figure 3.12), le graphe de transitions généré est décrit dans la figure 7.1(a). Le caractère étoile "*" est utilisé pour indiquer

les noeuds initiaux dans le graphe.

Masquage des transitions non interactives

Cette opération consiste à supprimer toutes les transitions qui ne concernent pas directement l'IU (c'est-à-dire qu'elles ne correspondent pas aux messages interactifs). Ces transitions sont appelées des transitions non interactives. Ces dernières vont être supprimées de la liste des noeuds `nodeList` et de la liste des noeuds initiaux `initialNodeList`.

En plus, tous les arcs qui sont définis par ces transitions sont aussi supprimés de la liste des arcs `edgeList`. En effet, lorsqu'une transition `T` est supprimée de `nodeList`, tous les arcs qui relient la transition `T` vont aussi être supprimés. Puis de nouveaux arcs sont ajoutés dans le but de relier les noeuds qui étaient au préalable reliés via les noeuds supprimés. Si `initialNodeList` contient des transitions non interactives alors ils vont être remplacées par les noeuds successeurs. Le résultat de cette opération sur le graphe de la figure 7.1(a) est le graphe `GT'` présenté dans la figure 7.1(b).

Identification des blocs de l'interface usager

Cette opération consiste à construire un graphe orienté où les noeuds représentent des blocs de l'IU (UIB). Un UIB est un sous-graphe de `GT'` comprenant une séquence de noeuds ayant un seul arc en entrée et un seul arc en sortie. Chaque `UIB` est délimité dans le graphe `GT'` par les règles suivantes:

Règle 7.1. Un noeud initial est le début d'un UIB.

Règle 7.2. Un noeud avec plus d'un arc en entrée est le début d'un UIB.

Règle 7.3. Un successeur d'un noeud avec plus qu'un arc en sortie est le début d'un UIB.

Règle 7.4. Un prédécesseur d'un noeud avec plus qu'un noeud en entrée termine un UIB.

Règle 7.5. Un noeud avec plus d'un arc en sortie termine un UIB.

En appliquant ces règles au graphe de la figure 7.1(b), nous obtenons le graphe `GB` tel que décrit dans la figure 7.2 (a). Dans cet exemple, la règle 7.1 détermine le début de B_1 et la règle 7.5 la fin de B_1. Les règles 7.3 et 7.4 délimitent les UIBs B_2 et B_3. L'UIB B_3 est obtenu par l'application de la règles 7.2.

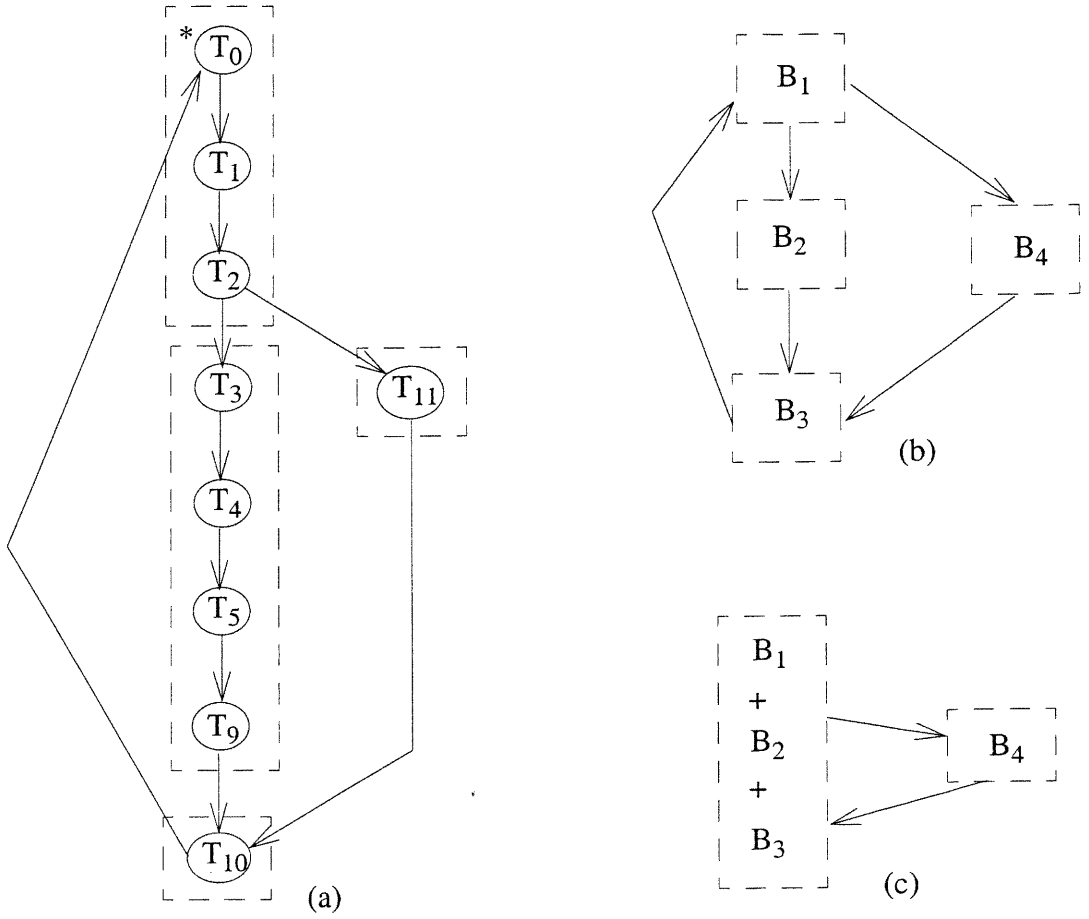


Figure 7.2: Le graphe GB résultant de l'identification des UIBs sur le graphe GT' de la figure 7.1(b): (a) vue étendue, (b) vue réduite et (c) GB' résultant de la composition des UIBs

Composition des blocs de l'interface usager

Généralement, les blocs de l'IU obtenus de l'opération précédente ne contiennent qu'un petit nombre d'objets d'interaction et ne représentent qu'une petite partie de la fonctionnalité du cas d'utilisation. C'est pourquoi nous supportons aussi le regroupement des UIBs dans le but d'obtenir des blocs plus intéressants qui vont permettre d'obtenir des fenêtres graphiques plus adéquates. Pour cela nous utilisons les heuristiques décrites dans les règles suivantes:

Règle 7.6 . Les UIBs adjacents appartenant au même scénario et non encore composés sont fusionnés (appartenance à un scénario).

Règle 7.7. L'opération de composition commence avec les scénarios ayant la plus grande fré-

quence (classification des scénarios).

Règle 7.8. Deux UIBs peuvent être regroupés si et seulement si le total de leurs objets d'interaction ne dépasse pas le chiffre vingt (critère ergonomique).

En appliquant ces règles sur le graphe GB de la figure 7.2(b), nous obtenons le graphe GB' tel que décrit dans la figure 7.2(c).

Génération des fenêtres et widgets de l'interface usager

Dans cette opération, nous générons une fenêtre graphique pour chaque UIB. Les fenêtre générées contiennent les objets d'interaction de toutes les transitions appartenant au UIB. Les arcs reliant les différent UIBs dans GB' sont transformés en appel de fonctions dans les classes des fenêtres générées. Dans notre implantation actuelle, le code Java généré est compatible avec le constructeur d'interfaces de Visual Café [Symantec, 1997]. Ceci donne la possibilité à l'analyste de personnaliser l'aspect visuel des fenêtres générées. Le deux fenêtres générées des blocs de la figure 7.2(c) sont montrées dans la figure 7.3.

L'aspect dynamique de l'IU est contrôlé par le comportement dynamique du StateD de l'objet d'interface en question. L'exécution du prototype générée revient à faire une exécution symbolique du StateD, ou dans notre cas, le parcours du graphe de transition GT'. Le prototype répond à toutes les interactions de l'utilisateur qui sont capturées dans les événements du graphe GT', et il ignore tous les autres événements.

Pour le support de l'exécution du prototype, une fenêtre de simulation est générée (voir figure 7.3, fenêtre d'en bas), ainsi qu'une boîte de dialogue pour le choix de scénarios à suivre (voir figure 7.3, fenêtre d'en haut à droite). Par exemple, après la sélection du cas d'utilisation `Retrait` à partir du menu de cas d'utilisation (voir figure 7.3, fenêtre d'en haut à gauche), un message est affiché dans la fenêtre de simulation qui confirme que le cas d'utilisation `Retrait` a été choisi et qui demande à l'utilisateur d'appuyer sur le bouton `insérer_carte`. Quand le bouton est appuyé, le champs `Mot de passe` est activé, et le simulateur attend les entrées de l'utilisateur. Lorsque l'exécution atteint un noeud dans le graphe GT' à partir duquel plusieurs chemins sont possibles, le prototype affiche la boîte de dialogue pour la sélection du scénario.

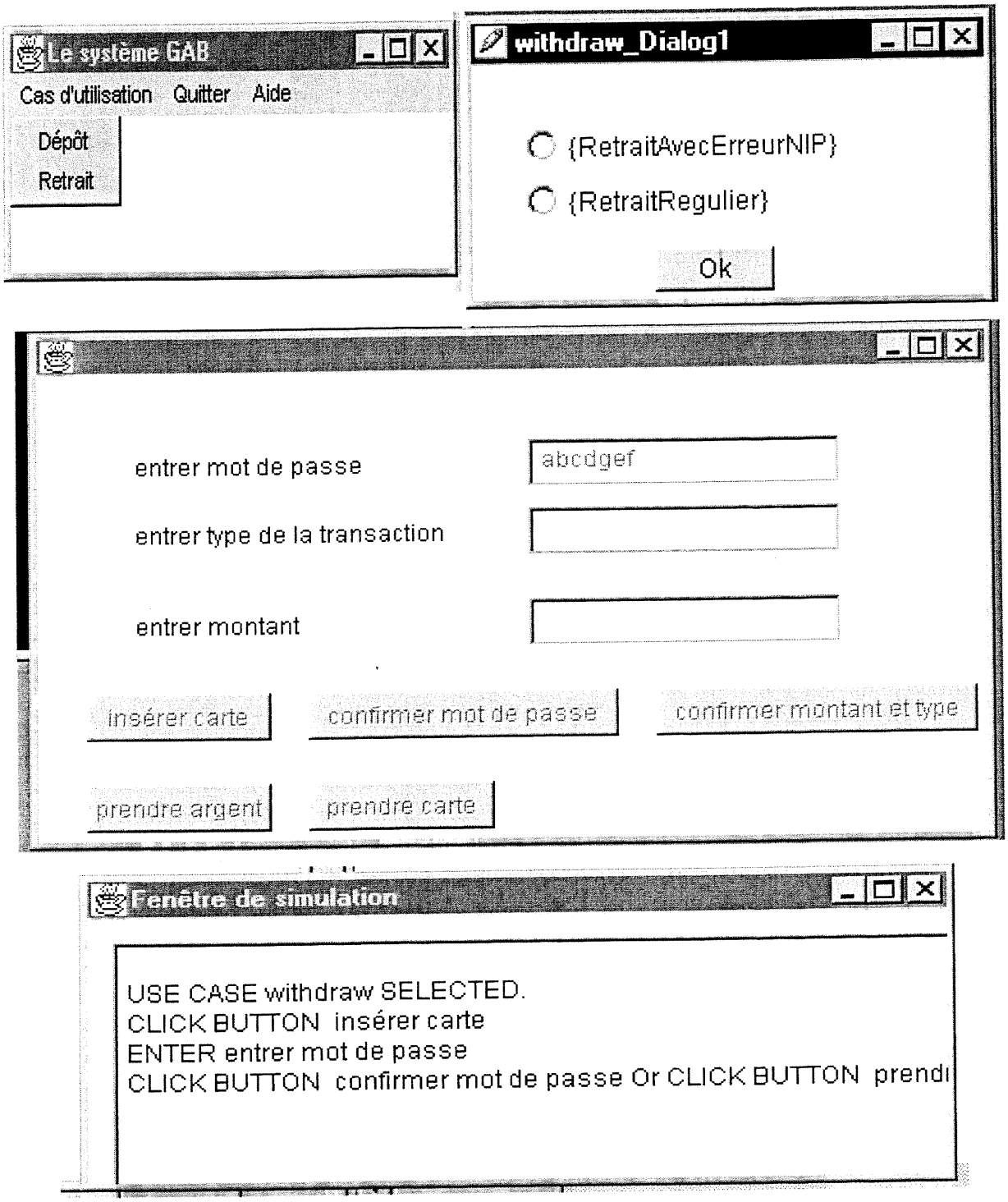


Figure 7.3: Exécution du prototype

Dans l'exemple de la figure 7.3, nous avons le choix entre les scénarios `RetraitAvecErreurNIP` et `RetraitRégulier`. Une fois le chemin est choisi, le prototype poursuit le parcours du graphe `GT'`.

7.1.2 Travaux reliés

Il existe plusieurs travaux qui se sont intéressés à la dérivation de l'IU à partir de la spécification du domaine d'application. En général, les attributs de données servent comme entrées pour la sélection des objets d'interaction et ceci en conformité avec des règles basées sur les différents guides de style de l'IU tels que CUA (Common User Access) [IBM, 1991], OpenLook [Sun Microsystems, 1990], et Motif [Open Software Foundation, 1990]. Dans ce qui va suivre nous allons présenter des exemples de travaux que nous considérons comme représentatifs des approches qu'on trouve dans la littérature (voir [Elkoutbi et al., 1999a] pour la description de certains autres travaux).

Genius

Dans Genius [Janssen et al., 1993], le domaine d'application est capturé dans un modèle de données décrit sous forme d'une extension du modèle entité-relation [Chen, 1976]. L'analyse définit un nombre de vues où chaque vue est composée d'un sous-ensemble d'entités, de relations, et d'attributs. Puis il spécifie le comportement du dialogue de ces vues par l'intermédiaire d'un réseau de Petri. À partir de ces vues et de la spécification du dialogue, Genius génère un prototype de l'IU. Nous remarquons donc que Genius, en dehors de la sélection automatique des objets d'interaction, reste un processus manuel.

Janus

Janus [Balzert, 1996] dérive les différentes fenêtres de l'IU à partir du modèle des objets de Coad et Yourdon [Coad et Yourdon, 1991]. Les classes non abstraites sont transformées en fenêtres contenant des objets d'interaction obtenus à partir des attributs et des opérations de ces classes. Janus ne supporte pas l'aspect dynamique des interfaces usagers.

À l'opposé de notre approche, Genius et Janus utilisent les spécifications des structures de données et ignorent l'analyse des tâches des utilisateurs. Par conséquent, de telles méthodes sont peu utiles pour les systèmes autres que les applications orientées données.

TRIDENT

Le point de départ de TRIDENT [Bodart et al., 1994] est l'analyse des tâches des utilisateurs ainsi que l'analyse fonctionnelle des besoins. L'analyse des tâches consiste en premier lieu à décompo-

ser l'application en tâches interactives, puis à déterminer les attributs des différentes tâches telles que l'importance et le stéréotype de l'utilisateur final du système (ses expériences dans le domaine d'application et dans l'utilisation des systèmes informatiques). L'analyse fonctionnelle des besoins a pour rôle de construire un modèle entité-relation pour les données et d'extraire de l'analyse des tâches les tâches qui doivent être considérées comme fonctions internes.

Un graphe d'enchaînement des activités est construit pour relier les tâches interactives aux fonctions du système. Ce graphe sert comme entrée pour la sélection des différentes fenêtres appelées unités de présentation. TRIDENT prétend fournir trois types d'assistance pour la définition des unités de présentation mais reste vague sur le type d'assistance apporté. En outre, le graphe d'enchaînement des activités n'est pas utilisé pour supporter l'aspect dynamique des interfaces usagers.

7.1.3 Discussion

Nous allons discuter certains points importants de notre algorithme de génération du prototype de l'IU: contexte et limitations de l'approche, prototypage rapide, et implantation de l'algorithme et expériences.

Contexte et limitations de l'approche

Notre approche a pour objectifs de (1) proposer un processus d'ingénierie des besoins compatibles avec UML, (2) fournir un support automatique pour la construction automatique de la spécification des objets, et (3) supporter la génération automatique de l'IU. Cette approche a deux limitations principales.

Premièrement, notre approche propose un processus de développement en avant du fait que la génération commence à partir des scénarios, alors que les modifications dans les spécifications résultantes et dans les prototypes de l'IU ne peuvent pas être propagées de façon automatique dans les scénarios. De ce fait une modification automatique devrait aussi être supportée.

Deuxièmement, notre approche peut être appliquée à la grande classe des systèmes réactifs exhibant des interfaces à fenêtres et objets d'interaction. Cependant, dans sa forme actuelle, il ne supporte pas de paradigmes alternatifs de l'IU.

Prototypage rapide

Notre approche supporte le prototypage rapide afin d'impliquer, dès le départ, les utilisateurs finaux dans la validation des scénarios. Le prototype généré constitue un outil pour l'évaluation et l'amélioration de la spécification sous-jacente.

En plus, le prototype généré peut évoluer, à travers son code source, pour couvrir tous les autres objets du système et ainsi atteindre l'application cible. Remarquons que les classes Java correspondantes aux objets peuvent être partiellement obtenus à l'aide du ClassD du système et des StateDs de ces objets à l'aide d'un outil CASE tel que Rational/Rose [Rational, 1998].

Implantation de l'algorithme et expériences

L'algorithme de génération de l'IU a été implanté avec un système de 10 classes et quelques 1500 lignes de code en langage Java (commentaires non inclus). L'algorithme a été testé avec les mêmes exemples de système qui ont servi à tester les autres algorithmes de notre approche (cf. section 4.4).

7.2 Environnement logiciel pour les transformations

Les outils CASE actuels tel que Rational/Rose [Rational, 1998] offrent des éditeurs graphiques pour supporter les notations des méthodes les plus utilisées dans le monde OO comme OMT [Rumbaugh, 1991], Booch [Booch, 1994] et UML. Ils fournissent aussi des outils pour la vérification de la validité des modèles de développement de point de vue syntaxe ou pour la génération d'un squelette de classes en C++ ou en Java à partir de la description des classes. Cependant, ils n'offrent pas des outils qui automatisent des tâches de développement plus pointues. Or, c'est de ce genre d'outils dont les concepteurs ont le plus besoin.

Dans l'état actuel, les algorithmes que nous avons développés ont été implantés avec le langage Java et testés sur des exemples de systèmes de petites tailles. À défaut d'un éditeur de diagrammes, nous avons utilisé une description textuelle interne pour la saisie et la visualisation des différents diagrammes. Nous avons aussi préparé un site web pour une diffusion plus large du logiciel qui englobe les algorithmes que nous avons proposés pour supporter le processus de l'ingénierie des besoins. Nous avons baptisé ce logiciel SUIP pour Scenario-based User Interface Prototyping. Le site est logé à l'adresse <http://www.iro.umontreal.ca/labs/gelo/suip>.

Dans ce qui suit, nous allons donner une idée sur la façon avec laquelle ce genre d'outil peut supporter les transformations que nous avons présentées. Nous allons conclure ce chapitre par une proposition d'architecture d'un outil CASE supportant notre travail.

7.2.1 Support de l'ingénierie des besoins

Pour le support d'ingénierie des besoins, un outil CASE doit fournir des éditeurs graphiques pour les ClassDs, CollDs et StateDs. Il doit incorporer un constructeur d'interface comme Visual Café [Symantec, 1997]. En outre, l'outil doit fournir trois options pour accéder et exécuter nos algorithmes.

La première option est l'acquisition d'un nouveau scénario. À travers son éditeur de CollD, le concepteur peut saisir son scénario tout en mettant à jour son modèle de classes à travers l'éditeur de ClassD.

La deuxième option est l'intégration d'un nouveau scénario dans la spécification existante. L'expérience d'utilisation de notre outil nous a montré que l'intégration d'un scénario ne réussit normalement pas d'un seul coup mais seulement après plusieurs corrections et ceci pour diverses raisons: incohérence du scénario, incohérence du nouveau scénario avec les scénarios déjà traités, une opération ou un attribut manquant dans la description des classes, etc. C'est pourquoi il est primordial que l'outil montre bien le ou les éléments des modèles (par exemple avec une couleur rouge) qui ont provoqué l'erreur.

La troisième option est la génération du prototype de l'IU. À ce niveau, la simulation du prototype telle que décrite dans la section 7.1 peut encore être améliorée par une visualisation du StateD de l'objet d'interface correspondant en montrant l'état en cours et la transition déclenchée.

7.2.2 Application automatique des patrons de conception

En plus des éditeurs graphiques pour les ClassDs, les CollDs et les StateDs, un outil CASE doit permettre la gestion de deux catalogues: un catalogue pour les schémas de raffinement et un autre pour les schémas de micro-raffinement. La gestion d'un catalogue de schéma de raffinement ou de schémas de micro-raffinement implique l'ajout de nouveaux schémas et la visualisation de la description d'un schéma.

La définition d'un nouveau schéma est faite à deux niveaux: une définition graphique du schéma et une autre textuelle correspondant au code à appeler en cas d'application du schéma de raffinement. En fait, le code ne fait que décrire le schéma de raffinement en terme de composition de schémas de micro-raffinement.

L'application du schéma de raffinement doit bien montrer les changements sur un modèle de conception. Le concepteur doit avoir la possibilité de récupérer le modèle initial s'il ne confirme pas les changements.

L'objectif ultime est d'avoir un outil intelligent pour le raffinement des modèles de conception. Pour cela, il faut travailler dans deux directions. Premièrement, il faut explorer les techniques qui utilisent le raisonnement basé sur les connaissances afin d'automatiser la sélection des patrons de conception pour raffiner un modèle abstrait de la conception. Par exemple, pour le développement du système de gestion d'une station de service présenté dans le chapitre 6 (voir figure 6.1), un outil intelligent pourrait automatiquement choisir le patron Observer pour l'implémentation de la contrainte qui existe entre les classes `Pompe` et `Écran` à condition que l'outil a appris que ce patron est la solution adéquate pour ce genre de problèmes.

Deuxièmement, il faut automatiser le processus de définition de nouveaux schémas de raffinement (voir figure 3.16). Ceci veut dire qu'ayant une définition de la structure d'un patron de conception ainsi que le modèle abstrait de son futur schéma de raffinement, l'outil peut automatiquement générer le modèle détaillé approprié et identifier les schémas de micro-raffinement qui compose le schéma de raffinement. Ces schémas de micro-raffinement peuvent être ceux qui ont été déjà identifiés et répertoriés dans le catalogue ou ceux qui sont nouvellement synthétisés par l'outil. L'environnement SPOOL [Keller et al., 1999] peut être une plate-forme intéressante pour l'intégration de cet outil.

7.2.3 Support de la traçabilité

Nous avons vu que parmi les objectifs d'une approche transformationnelle est celui de supporter la traçabilité entre les éléments du modèle d'un système en développement. Or, UML définit entre autres deux mots clés prédéfinis *refine* et *trace* pour spécifier une relation de dépendance entre deux éléments *source* et *cible* du modèle. Le mot-clé «refine» spécifie que l'élément source est un raffinement de l'élément cible (la direction de la flèche est du côté de l'élément cible), alors que le mot clé «trace» spécifie que l'élément cible est un prédécesseur historique de l'élément source. Donc un outil CASE supportant nos transformations peut insérer automatiquement ces relations

de dépendance entre les éléments des modèles.

Nous aurons comme dépendance de type «trace» entre autres les éléments suivants:

- un cas d'utilisation avec chacun de ces CollDs,
- un CollD avec chaque StateD partiel généré par ce CollD,
- un cas d'utilisation avec chaque StateD généré qui intègre les StateDs partiels des CollDs du cas d'utilisation,
- le UseCaseD du système avec chaque StateD résultant de la synthèse des cas d'utilisation du UseCaseD,
- le StateD d'un objet d'interface avec le prototype de l'IU correspondant,
- un StateD d'une classe avec son StateD mis à jour après l'application d'un schéma de raffinement.

Et nous aurons comme dépendance de type «refine» entre autres les éléments suivants:

- une association dans un ClassD avec tous les éléments nouvellement créés ou mis à jour dans le ClassD après l'application d'un schéma de raffinement,
- une instance d'association dans un CollD avec tous éléments nouvellement créés ou mis à jour dans le CollD après l'application d'un schéma de raffinement.

7.2.4 Proposition d'architecture

Ces dernières années, la construction des outils CASE est devenue un champ de recherche très actif [Gray et al., 1999]. Elle couvre une variété d'activités telles que les approches meta-CASE [Alderson et al., 1999; Sunyé, 1999], qui s'intéressent à la génération d'outils CASE personnalisés à un langage de modélisation particulier, ou les techniques d'échange de données [Bowman et al., 1999; Plantec et Ribaud, 1999]. Nous avons déjà mentionné que l'environnement SPOOL [Keller et al., 1999] peut être une plate-forme intéressante pour le support de notre approche transformationnelle. Dans ce qui suit nous présentons une extension de l'architecture de SPOOL pour l'intégration des transformations que nous avons présentées.

L'environnement SPOOL suit une architecture trois-tier ("three-tier") [Keller et al., 2000] (voir figure 7.4). Le premier tier (le plus bas) correspond à un système de gestion de base de données OO qui fournit un dépôt de données persistent pour les modèles du système en développement et les schémas de raffinements et de micro-raffinements. Le deuxième tier (au milieu) est un schéma

de dépôt qui est une hiérarchie de classes OO qui décrivent les modèles du système en développement, le catalogue de schémas de raffinement et le catalogue de schémas de micro-raffinements. Ce schéma est basé sur le méta-modèle de UML [Rational et al., 1997].

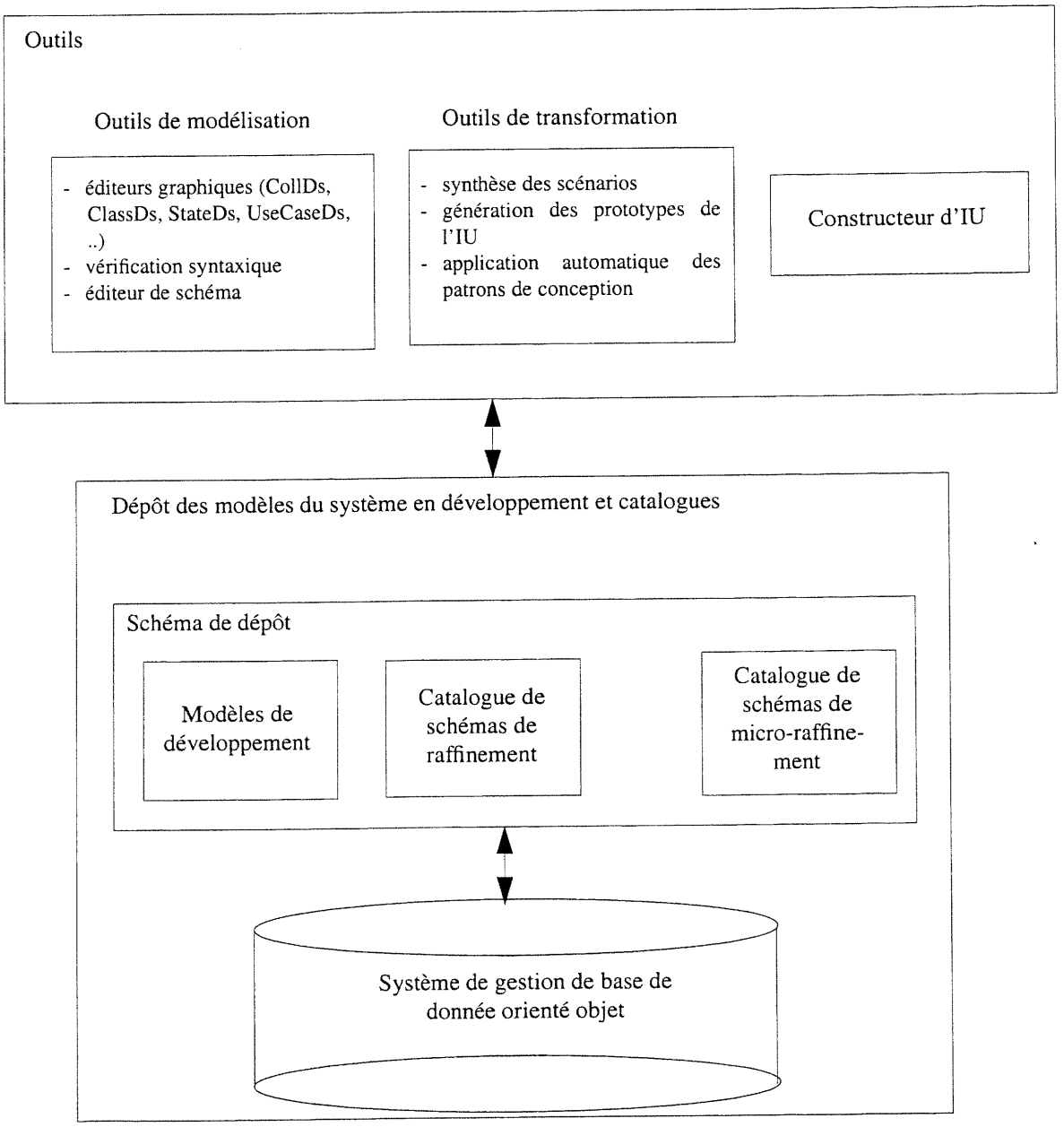


Figure 7.4: Architecture d'un outil CASE supportant les transformations proposées

Le troisième tier comprend les outils qui vont être utilisés par les développeurs. Trois catégories d'outils sont à envisager. La première catégorie correspond aux outils de modélisation tels que les

éditeurs graphiques pour les diagrammes d'UML (CollDs, ClassDs, StateDs, UseCaseDs,...), un vérificateur syntaxique des diagrammes et un éditeur de schémas de raffinement ou de micro-raffinements. La deuxième catégorie comprend les transformations que nous avons proposées à savoir la synthèse des scénarios, la génération des prototypes de l'IU l'application automatique des patrons de conception. La troisième catégorie ne contient que le constructeur d'IU utilisé pour encastrer les prototypes de l'IU.

Notons que l'avantage principale d'une architecture trois-tier consiste à permettre aux outils d'être décrites à partir de la sémantique du domaine sans se soucier de l'emplacement physique des données. Cette architecture a été adoptée dès 1978 [Tsichritzis et al., 1978] mais elle a été rarement utilisée à cause de la difficulté de son implantation [Fowler, 1997]. Ce n'est qu'avec la venue de la technologie OO que cette architecture a retrouvé les moyens adéquats pour son implantation.

Au fil des chapitres de cette thèse, nous avons présenté les éléments de notre approche transformationnelle pour supporter le processus d'ingénierie des besoins avec les scénarios et l'application automatique des patrons de conception. Cette approche respecte bien les quatre caractéristiques que nous avons introduites dans le section 1.4.

Premièrement notre approche supporte le langage UML qui est un langage de spécification en même temps expressif et facile à utiliser. Deuxièmement, elle intègre les nouvelles techniques de génie logiciel à savoir le paradigme OO, les scénarios comme moyen de capturer les besoins des utilisateurs et les patrons de conception comme véhicule de transmission et réutilisation de bonnes expériences dans la conception OO. Troisièmement, les règles de transformation sont à un niveau adéquat d'abstraction puisque les algorithmes d'intégration des scénarios, de prototypage et d'application des patrons de conception permettent un grand gain de productivité. Finalement, notre approche offre des algorithmes et des schémas de raffinements qui peuvent être réutilisés par un grand ensemble des systèmes OO.

Conclusion

Dans cette thèse, nous avons proposé une approche transformationnelle pour supporter le processus d'ingénierie des besoins avec les scénarios et l'application automatique des patrons de conception. Dans ce qui suit, nous allons résumer nos contributions et présenter les extensions possibles à notre recherche.

Support de l'ingénierie des besoins avec les scénarios

Dans l'ingénierie des besoins, nous supportons les trois principales activités du processus à savoir la synthèse des scénarios, la vérification des scénarios et la génération d'un prototype du système. Pour cela, nous avons fourni deux algorithmes.

Le premier algorithme synthétise incrémentalement la spécification dynamique des objets à partir des scénarios. Il prend comme entrée un ensemble de diagrammes de collaboration d'UML et produit en sortie les diagrammes d'états-transitions d'UML de tous les objets collaborant dans ces scénarios. En outre il permet la vérification de la cohérence et de la complétude des scénarios. Les principales caractéristiques de cet algorithme peuvent être résumées en deux points. Le premier point est qu'il accepte non seulement des scénarios séquentiels mais aussi des scénarios itératifs ou ceux qui exhibent un comportement concurrent. Le deuxième point est qu'il résoud le problème de chevauchement des scénarios sans changer ni la syntaxe ni la sémantique des diagrammes d'états-transitions.

Le deuxième algorithme a pour but la génération d'un prototype de l'IU à partir de la spécification des scénarios. Les scénarios sont acquis sous forme de diagrammes de collaboration enrichis par des informations de l'IU. L'intérêt de cet algorithme est qu'il automatise complètement le processus de génération de l'IU. Les deux aspects, statique et dynamique, sont tous les deux dérivés ce qui permet de simuler le prototype dans le but de valider les scénarios avec les utilisateurs. En outre le prototype peut être accepté par un constructeur d'interface tel que Visual Café ce qui facilite sa personnalisation et son raffinement éventuel.

Application automatique des patrons de conception

L'application automatique des patrons de conception a pour objectif de réduire le fossé existant entre les deux niveaux d'abstraction de l'étape de conception, à savoir la conception générale et celle détaillée. Avec notre approche, la conception est devenue un processus de raffinement successif des modèles de conception. Chaque raffinement consiste en une application d'un schéma de raffinement basé sur un patron de conception. L'intérêt de cette approche est double. Premièrement, elle rend systématique l'étape de conception par l'utilisation des patrons de conception. Ceci contribue à la réutilisation de bonnes expériences de conception. Deuxièmement, les schémas de raffinement s'assurent que les différents modèles, statique et dynamique, soient automatiquement mis à jour ce qui permet une cohérence et une traçabilité entre ces modèles de conception.

En plus, nous avons proposé un ensemble de schémas de micro-raffinement avec leur preuves de validité. Un schéma de micro-raffinement est aussi décrit par un schéma. Un schéma de raffinement est spécifique pour un seul patron de conception alors qu'un schéma de micro-raffinement est plus général. Par conséquent, ce dernier peut être réutilisé pour la description de plusieurs schémas de raffinement. Il joue un rôle similaire à celui d'une fonction dans une librairie. Cependant, la différence ici est que non seulement son code peut être réutilisé mais aussi la preuve de sa validité.

Extensions possibles

Cette thèse peut faire l'objet d'au moins deux types d'extension. Le premier type d'extension consiste à améliorer les travaux actuels et le second à supporter tous les diagrammes proposés par UML.

Dans l'ingénierie des besoins, notre approche peut être améliorée pour supporter une traçabilité dans le sens inverse, c'est-à-dire propager des éventuelles modifications sur la spécification ou sur le prototype vers la description des scénarios. Nous pouvons aussi supporter différents paradigmes de l'IU pour accepter des systèmes autres que ceux qui exhibent des interfaces à fenêtres et objets d'interaction.

En ce qui concerne les schémas de raffinement, un travail supplémentaire reste indispensable pour étudier d'autres patrons de conception. Cette étude aura aussi comme conséquence d'étendre le catalogue de schémas de micro-raffinement. Une autre piste possible est d'explorer des moyens pour aider le concepteur dans son choix de patrons de conception à utiliser.

Le langage UML offre une variété de diagrammes pour la description d'un système en développement (cf. section 2.1.2). Dans cette thèse, nous n'avons utilisé que quatre des neuf types de diagrammes fournis. Donc un autre défi est d'offrir une approche transformationnelle qui supporte tous ces diagrammes. À titre d'exemple, nous pouvons citer un algorithme qui transforme un diagramme de séquencement en un diagramme de collaboration et inversement. Remarquons que ces deux diagrammes sont équivalents et par conséquent, la conception d'un tel algorithme ne devrait pas poser de gros problèmes. Nous pouvons aussi chercher à assurer une cohérence entre les différentes vues d'un système à savoir les vues logique, réalisation, processus, déploiement et cas d'utilisation.

Références bibliographiques

Alderson, A., Cartnell, J.W. et Elloit, A. (1999). Toolnuilder: From Case Tool Components to Method Engineering. *In Proceedings of the International Workshop on Constructing Software Engineering Tools (CoSET'99)*, collocated with ICSE'99, Los Angeles, CA, pp. 9-18.

Balzer, R. (1981). Transformational Implementation: An Example. *IEEE Transactions on Software Engineering*, 7(1): 3-14.

Balzer, R. (1977). Correct and Efficient Software Implementation via semi-automatic transformations. *USC/ISI Internal Report*, Information Science Institute, University of Southern California, Marina del Rey, CA.

Balzer, R. (1973). A Global View of Automatic Programming. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Stanford, CA, pp. 494-499.

Balzer, R., Goldman, N. et Wile, D. (1980). Informality in Program Specification. *IEEE Transactions on Software Engineering*, 4(2): 94-103.

Balzer, R., Goldman, N. et Wile, D. (1976). On the Transformational Implementation Approach to Programming. In *Proceedings of 2nd International Conference on Software Engineering*, San Francisco, CA, IEEE, New York, pp. 337-344.

Balzert, H. (1996). From OOA to GUIs: The Janus System. *IEEE Software*, 8(9): 43-47.

Basili, V. (1990). Viewing maintenance as reuse-oriented software development. *IEEE Software*, 7(1):19–25.

Batory, D. (1999). When are Transformations Necessary? Or Scaling Transformations To Larger Units of Encapsulation. In *Proceedings of the International Workshop on Software Transformation Systems (STS'99)*, collocated with ICSE'99, Los Angeles, CA, pp. 69-70.

Biermann, A.W. et Krishnaswamy, R. (1976). Constructing Programs from Example Computations. *IEEE Transactions on Software Engineering*, 2(3): 141–153.

Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Provot, I. et Vanderdonckt, J. (1994). A Model-based Approach to Presentation: A Continuum from Task Analysis to Prototype. In *Proceedings of the Eurographics Workshop on Design, Specification, and Verification of Interactive Systems*, Carrara, Italy, pp.77-94.

Boehm, B. (1988). A Spiral Model for Software Development and Enhancement. *IEEE Computer*, 21(5):61–72.

Booch, G. (1994). *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings Publishing Company Inc., Redwood City, CA. Second edition.

Booch, G. (1991). *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings Publishing Company Inc., Redwood City, CA. First edition.

Bowman, I., Godfrey, M. et Holt, R. (1999). Connecting Architecture Reconstruction Frameworks. *In Proceedings of the International Workshop on Constructing Software Engineering Tools (CoSET'99)*, collocated with ICSE'99, Los Angeles, CA, pp. 43-54.

Budinsky, F.J.M, Finnie, M.A., Vlissides, J.M. et Yu, P.S. (1996). Automatic code generation from design patterns. *Object Technology*, 35(2): 172-191.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. et Stal, M. (1996). *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons.

Cattell, R.G.G. et Barry, D. K. (1997). *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann.

Chen, P. (1976). The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1): 9-36.

Coad, P., North, D. et May, M. (1995). *Object Models, Strategies, Patterns & Applications*. Yourdon Press, Englewod Cliffs.

Coad, P. et Yourdon, E. (1991). *Object-oriented Analysis*. Prentice Hall, Englewod Cliffs, NJ.

Coleman, D., Arnold, P., Bodoff, S., Dollin, Ch., Gilchrist, H., Hayes, F. et Jeremaes, P. (1994).

Object-Oriented Development: The Fusion Method. Prentice-Hall, Inc.

Dano, B., Briand, H. et Barbier, F (1997). An Approach based on the Concept of Use Cases to Produce Dynamic Object-Oriented Specifications. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, Annapolis, pp. 56-64.

Desharnais, J., Frappier, M., Khédri, R. et Mili, A. (1998). Integration of sequential scenarios. *IEEE Transactions on Software Engineering*, 24(9): 695–708.

DOD-STD-2167A (1988). Military Standard. *Defense Systems Software Development*. Department of Defense, Washington, DC 2031, USA.

Eden, A.H., Jil, J. et Yehuday, A. (1997). Precise Specification and Automatic Application of Design Patterns. In *Proceedings of the IEEE International Conference on Automated Software Engineering Conference*, Incline Villagem Nevada, USA, pp.143-152.

Eden, A.H. et Yehuday, A. (1997). Tricks Generate Patterns. *Technical report 324/97*, The Department of Computer Science, Schriber School of Mathematics, Tel Aviv University.

Elkoutbi, M., Khriss, I. et Keller, R.K. (1999a). Generating User Interfaces from Scenarios. In *Proceedings of the Fourth IEEE Int. Symposium on Requirements Engineering*, Limerick, Ireland, pp.150-158.

Elkoutbi, M., Khriss, I. et Keller, R.K. (1999b). User Interface Prototyping using UML Specifications. *Technical Report GELO-98*, Université de Montréal, Montréal, Québec, Canada.

Eriksson, H.E. et Penker, M. (1998). *UML-Toolkit*. John Wiley and Sons, 1998.

Fiadeiro, J. et Maibaum, T. (1991). Sometimes "Tomorrow" is "Sometime". In *Temporal Logic*, Lecture Notes in Artificial Intelligence (Vol. 827), Springer-Verlag, pp. 48-66.

Fowler, M. (1997). *Analysis Patterns: Reusable object models*. John Wiley and Sons.

Gamma, E., Helm, R., Johnson, R. et Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Glinz, M. (1995). An Integrated Formal Model of Scenarios based on Statecharts. In *Fifth European Software Engineering Conference,* Lecture Notes in Computer Science (vol. 989), Springer-Verlag, pp. 254-271.

Gray, J., Liu, A., Scott, L. et Harvey, J. (1999). International Workshop on Constructing Software Engineering Tools (CoSET'99). *In Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, pp. 707-708.

Harel, D. (1988). Statecharts: On Visual Formalisms. *Communications of the ACM*, 31(5): 514–530.

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A. et Trakhtenbrot, M.. (1990). STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Transactions on Software Engineering*, 16(4): 403-414.

Heimdahl, M.P.E. et Leveson, N.G. (1996). Completenes and Consistency in Hierarchical State-Based Requirements. *IEEE Transactions on Software Engineering*, 22(6): 363–377.

Hsia, R., Samuel, J., Gao, J., Kung, D., Toyoshima, Y. et Chen, C.(1994). Formal Approach to Scenario Analysis. *IEEE Software*, 11(2): 33-41.

IBM (1991). *Systems Application Architecture: Common User Access - Guide to User Interface Design - Advanced Interface Design Reference*, IBM.

IEEE/Std.1219 (1992). *IEEE Standard for Software Maintenance*. Institute of Electrical and Electronic Engineers, New York, USA.

IEEE/Std.982.1 (1989). *IEEE Standard and Dictionary of Measures to Produce Reliable Software*. Institute of Electrical and Electronic Engineers, New York, USA.

IEEE/Std.830 (1984). *IEEE Guide to Software Requirements Specification*. Institute of Electrical and Electronic Engineers, New York, USA.

IFAD (1998). User Manual for the IFAD VDM++ Toolbox. *IFAD-VDM-50*, the VDM Tool Group, IFAD, Odense, Denmark.

ISO/9000-3 (1991). *Quality Management and Quality Assurance Standards*. International Organization for Standardization, Geneva, Switzerland.

Jacobson, Booch, G. et I. Rumbaugh, J. (1999). The *Unified Software Development Process*. Addison Wesley, Inc.

Jacobson, I., Christerson, M., Jonsson, P. et Oevergaard, G. (1992). *Object-Oriented Software*

Engineering: A Use Case Driven Approach. Addison-Wesley.

Jahanian, F. et Mok, A.K. (1986). Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions of Software Engineering*, 12(9): 890-904.

Janssen, C., Weisbecker, C. et Ziegler, U. (1993). Generating User Interfaces from Data Models and Dialogue Net Specifications. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'93)*, Amsterdam, The Netherlands, pp. 418-423.

Kawashita, I. (1996). *Spécficiation formelle de systèmes d'information interactifs par la technique des scénarios*, Mémoire de maîtrise, Université de Montréal, Montréal, Québec, Canada.

Keller, R.K., Knapen, G., Laguë, B., Robitaille, S., St-Denis, G. et Schauer, R. (2000). The SPOOL design repository: Architecture, schema, and mechanisms. *In Hakan Erdogmus and Oryal Tanir, editors, Advances in Software Engineering. Topics in Evolution, Comprehension, and Evaluation*. Springer-Verlag, 28 pages. À parître.

Keller, R.K., Schauer, R., Robitaille, S. et Patrick, P. (1999). Pattern-based Reverse Engineering of Design Components, In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, IEEE, pp. 226-235.

Khriss, I., Keller, R.K. et Hamid, I. (2000). Pattern-based Refinement Schemas for Design Knowledge Transfer. *Technical Report GELO-117*, Université de Montréal, Montréal, Québec, Canada. Soumis sur invitation au Journal of Knowledge-Based Systems.

Khriss, I., Elkoutbi, M. et Keller, R.K. (1999a). Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams, In *<<UML>>'98: Beyond the Notation*, Lecture Notes in Computer Science (vol. 1618), Jean Bezivin and Pierre Alain Muller, editors, Springer-Verlag, pp. 132-147. Version étendue et révisée de Khriss et al. (1998).

Khriss, I., Elkoutbi, M. et Keller, R.K. (1999b). Automatic Synthesis of Behavioral Specifications from Scenarios, *Technical Report GELO-96*, Université de Montréal, Montréal, Québec, Canada.

Khriss, I. , Keller, R.K. et Hamid, I. (1999c). Supporting Design by Pattern-Based Tranformations. In *Proceedings of the 2nd International Workshop on Strategic Knowledge and Concept Formation (IWSKCF'99)*, Iwate, Japan, pp. 157-167.

Khriss, I., Elkoutbi, M. et Keller, R.K. (1998). Automating the Synthesis of UML Statechart Dia-

grams from Multiple Collaboration Diagrams. In *Proceedings of the Workshop <<UML>>'98: Beyond the Notation*, Mulhouse, France, pp. 115-126.

Khriss, I. et Keller, R.K. (1999a). Transformations for Pattern-Based Forward-Engineering. In *Proceedings of the International Workshop on Software Transformation Systems (STS'99)*, collocated with ICSE'99, Los Angeles, CA, pp. 50-58.

Khriss, I. et Keller, R.K. (1999b). Integration between High-level and Low-level Design with Refinement Schemas based on Design Patterns. *Technical Report GELO-91*, Université de Montréal, Montréal, Québec, Canada.

Koskimies, K. et Mäkinen, E. (1994). Automatic synthesis of state machines from trace diagrams. *Software — Practice & Experience*, 24(7): 643–658.

Koskimies, K., Systa, T., Tuomi, J. et Mannisto, T. (1998). Automatic Support for Modeling OO Software. *IEEE Software*, 15(1): 42–50.

Kruchten, P. (1995). The 4 + 1 Model of Architecture. *IEEE Software*, 12(6): 42–50.

Lano, K. et Bicarregui, J. (1999). Semantics and Transformations in UML Models, In *<<UML>>'98: Beyond the Notation*, Lecture Notes in Computer Science (vol. 1618), Jean Bezivin and Pierre Alain Muller, editors, Springer-Verlag, pp. 107-119.

Lano, K., Bicarregui, J. et Goldsack, S. (1996). Formalising Design Patterns. *RBCS-FACS Northern Formal Methods Workshop.*

Lee, W.J., Cha, S.D. et Kwon, Y.R. (1998). Integrating and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(12): 1115–1130.

Lindvall, M. et Snadhal, K. (1996). Practical Implications of Traceability. *Software — Practice & Experience*, 26(10):1161–1180.

Maher, T.P. (1994). *Automated Generation of User Interfaces*, Thèse de doctorat, Department of Computer Science and Computer Engineering, La Trobe University, Melbourne, Australia.

Meijers, M. (1996). *Tool Support for Object-Oriented Design Patterns*, Technical report, Utrecht University, Utrecht.

Moriconi, M., Qian, X. et Riemenschneider, R.A (1995). Correct architecture refinement. *IEEE Transactions on Software Engineering* , 21(4):356-372.

Objective Systems (1993a). *Objectory Analysis and Design 3.3 Process.* Objective Systems SF AB, Kista, Sweden.

Objective Systems (1993b). *Objectory Analysis and Design 3.3 Tool.* Objective Systems SF AB, Kista, Sweden.

O'Cinnéide, M. O. et Nixon, P. A. (1999). Methodology for the Automated Introduction of Design Patterns. In *Proceedings of the IEEE International Conference of Software Maintenance (ICSM'99)*, Oxford, England, pp. 463-472.

Opdyke, W.F. (1992). *Refactoring Object-Oriented Frameworks*, PhD Thesis, University of Illinois, USA.

Open Software Foundation (1990). *OSF/Motif Style Guide*, Prentice Hall, Englewood Cliffs, NJ, USA.

Ostroff, J.S. (1989). *Temporal Logic for Real-Time Systems*, John Wiley.

Partsch, H. et Steinbrüggen, R. (1986). Program Transformation Systems. In *New Paradigms for Software Development*, William W. Agresti, IEEE Press, pp. 189-226.

Plantec, A. et Ribaud, V. (1999). Using and Re-using Application Generators. *In Proceedings of the International Workshop on Constructing Software Engineering Tools (CoSET'99)*, collocated with ICSE'99, Los Angeles, CA, pp. 55-60.

Pressman, R. S. (1997). *Software Engineering. A Practitioner's Approach.* McGraw-Hill, fourth edition.

Rapicault, P. et Blay-Fornarino, M. (2000). Instanciation et vérification de design patterns: un méta protocole. In Proceedings du colloque sur les Langages et Modèles à Objets-2000, Mont Saint-Hilaire, QC, Canada, pp. 43-58.

Rational Software Corporation (1998). *Rational Rose.* Rational Software Corporation, Santa Clara, CA.

Rational Software Corporation, Microsoft, Hewlett-Packard, Oracle, Sterling, MCI, Unisys,

ICON, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum, Ptech, Taskon, Reich Technologies, Softeam (1997). *UML Notation Guide, version 1.1.* Rational Software Corporation, Santa Clara, CA.

Reiss, S.P. (2000). Working with Patterns and Code. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-2000*, Maui, HI. Sur CD-ROM, 10 pages.

Royce, W.W. (1970). Managing the Development of Large Software Systems, In *Proceeding of IEEE WESCON.*

Rumbaugh, J., Jacobson, I. et Booch, G. (1999). *The Unified Modeling Language Reference Manual.* Addison Wesley, Inc.

Rumbaugh, J. (1996).To Form a More Perfect Union: Unifying the OMT and Booch Methods. *Journal of Object-Oriented Programming*, 8(8):14-18.

Rumbaugh, J. (1995a). OMT: The Development Process. *Journal of Object-Oriented Programming*, 8(9):8–76.

Rumbaugh, J. (1995b). OMT: The Dynamic Model. *Journal of Object-Oriented Programming*, 8(2):6–12.

Rumbaugh, J. (1995c). OMT: The Functional Model. *Journal of Object-Oriented Programming*, 8(3):10–14.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. et Lorensen, W. (1991). *Object-oriented Modeling and Design.* Prentice-Hall, Inc.

Sant'Anna, M., Leite, J., Baxter, I., Wile, D., Biggerstaff, T., Batory, D., Devanbu, P. et Burd, L. (1999). International Workshop on Software Transformation Systems (STS'99). In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, pp. 701-702.

Schönberger, S., Keller, R. K. et Khriss, I. (2000). Algorithmic Support for Model Transformation in Object-Oriented Software Development. *Theory and Practice of Object Systems (TAPOS)*, John Wiley, à paraître.

Shlaer, S. et Mellor, S.J. (1993). A Deeper Look at the Transition form Analysis to Design. *Journal of Object-Oriented Programming*, 5(9):16–21.

Shlaer, S. et Mellor, S. J. (1992). *Object Lifecycles. Modeling the World in States.* Yourdon Press

Computing Series.

Somé, S., Dssouli, R. et Vaucher, J. (1996). Toward an Automation of Requirements Engineering using Scenarios. *Journal of Computing and Information* (JCI), (2) 1: 1110–1132.

Sun Microsystems et AT&T (1990). *OPEN LOOK GUI Application Style Guidelines*, Addison-Wesley, USA.

Sunyé, G. (1999). *Mise en oeuvre de patterns de conception: un outil*, Thèse de doctorat, Université PARIS VI, France.

Symantec (1997). *Visual Café for Java: User Guide*, Symantec, Inc., 1997.

UML (1998). *Unified Modeling Language Specificiation*. Object Management Group, Framingham, Mass. Internet: www.omg.org.

Tsichritzis, D.C. et Klug, A. (1978). The ANSI/X3/SPARC DBMS Framework: report of the study group on database management systems, *Information Systems*, 3(3):173-191. Pergamon Press Ltd.

Wile, D. (1983). Program Developments: Formal Explanation of Implementations. *Communications of the ACM*, 26(11): 902–911.

Wile, D., Balzer, R. et Goldman, N. (1977). Automated Derivation of Program Control Structure from Natural Language Program Descriptions. In *Proceedings of Symposium on Artificial Intelligence and Programming Languages*, Rochester, N.Y., pp. 77-84.

Annexe A: Grammaire des diagrammes des classes

Dans ce qui suit nous allons donner la grammaire d'un ClassD. Elle comprend seulement les aspects dont dépend notre travail. En outre, nous avons précisé le format que nous acceptons pour les pre- et post-conditions des opérations d'une classe. Ce format constitue un sous-ensemble d'OCL. Le chapitre 5 donne plus d'informations sur les raisons de notre choix.

La grammaire ci-dessous ainsi que celles des annexes B et C suivent le format EBNF qui utilisent les conventions suivantes:

- [élément] élément est optionnel,
- {élément} élément peut survenir zéro ou plusieurs fois,
- | exprime des alternatives ("ou"),
- () désigne les précédences.

```
classD =
   [classDName] class {class} {relationship}.
class =
   [packageName] ["::"] className
   {attribute} {operation}.
attribute =
   [visibility] attributeName ["[" multiplicity "]"]
   ":" typeExpression "=" initialValue.
operation =
   [visibility] operationName ["(" operationParameter {","
   operationParameter} ")"] [":" returnTypeExpression]
   "pre:" preCondition
    "post:" postCondition.
relationship =
   association | generalisation | dependency |
   constraintRelation | attachNote.
association =
   [associationName] [nameDirectionArrow] [constraint]
   associationEnd associationEnd {associationEnd}.
associationEnd =
   linkedClass multiplicity navigability ["{ordered}"] [qualifier]
```

```
   [aggregationIndicator] [roleName] [visibility].
linkedClass =
   < ref. to a class >.
navigability =
   true | false.
aggregationIndicator =
  filledDiamond | unfilledDiamond.
multiplicity =
   lowerBound ".." upperBound {"," lowerBound ".." upperBound}.
lowerBound =
   integerLiteral.
upperBound =
   integerLiteral | "*".
operationParameter =
   [kind] parameterName ":" typeExpression "=" defaultValue.
kind =
   "in" | "out" | "inout".
visibility =
   public | private | protected.
public =
   "+".
private =
   "-".
protected =
   "#".
qualifie =
   attribute {attribute}.
generalisation =
   superClass subClass {subClass}.
sperClass, subClass =
   < ref. to a class>.
dependency =
   modelElement modelElement {modelElement}
   "{" dependencyKind {"," dependencyKind}"}".
constraintRelation =
   modelElement modelElement constraint.
attachNote =
   modelElement note.
modelElement =
   class | attribute | operation | relationship.
dependencyKind =
   predefindKeyword | anyString.
predefindKeyword =
   "trace" | "refine" | "uses" | "bind".
note =
   comment | constraint | methodBody.
constraint =
   "{" expression {"," expression }"}".
expression =
  <suit le langage OCL>.
preCondition =
     orExpression.
postCondition =
     ifExpression {"OR" ifExpression}
```

```
     | orExpression.
ifExpression =
     "IF" orExpression "THEN" orExpression "ENDIF".
orExpression =
     andExpression {"OR" andExpression}.
andExpression =
     basicExpression {"AND" basicExpression}.
basicExpression =
     identifier ( ( "=" | "<" | ">" "<=" | ">=" ) ( string_literal |
     character_literal | integer_literal | floating_point_literal ) )
     | ("=" ( "true" | "false" ) ).
typeExpression, returnTypeExpression =
   "string" | "character" | "integer" | "float"
     | "true" | "false"
initialValue =
   string_literal | character_literal | integer_literal |
   floating_point_literal | "true" | "false".
classDName, packageName, className, attributeName, operationName,
associationName, returnValue, parameterName , linkedClass, roleName = identi-
fier.
```

Un diagramme de classes est un réseau de classes et de relations. La description d'une classe comprend:

- le nom du module (package) auquel la classe appartient,
- le nom de la classe,
- une liste d'attributs,
- une liste d'opérations.

Un attribut peut être constitué de:

- sa visibilité publique (public), privée (private) ou protégée (protected),
- son nom,
- sa multiplicité,
- son type,
- sa valeur de départ.

Une opération peut être décrite par:

- sa visibilité publique (public), privée (private) ou protégée (protected),
- son nom,

- ses paramètres,
- le type de sa valeur de retour,
- sa pre-condition,
- sa post-condition.

Une relation peut être une association, une généralisation, une dépendance, une contrainte, ou un attachement à une note. Une association relie au moins deux classes. Chaque côté de l'association (association end) est déterminé par:

- la référence de la classe reliée,
- une multiplicité,
- une navigabilité déterminant le fait que l'association est uni-directionnelle ou non,
- une contrainte `"ordered"` indiquant l'ordonnancement ou nom des objets reliés,
- un ensemble de clés candidates (qualifier) qui permet d'identifier uniquement un objet dans le cas d'une association un à plusieurs ou plusieurs à plusieurs,
- un indicateur d'agrégation pour indiquer que l'association est une relation d'agrégation,
- un rôle,
- une visibilité publique (public), privée (private) ou protégée (protected).

Une généralisation est une relation d'héritage entre une super-classe et un ensemble de sous-classes. Une dépendance est une relation de dépendance entre les éléments du modèle. Un élément du modèle peut être une classe, un attribut, une opération ou une relation. Une dépendance peut être soit d'un type prédéfini tel que `"trace"`,`"refine"`,`"uses"` ou `"bind"`, soit de n'importe quel type qui sera défini par le concepteur.

Une relation de contrainte relie deux éléments du modèle avec une contrainte écrite en suivant le langage OCL (Object Contraint Language). Un attachement à une note est une relation qui lie un élément du modèle à une note qui peut être un commentaire, le corps d'une méthode ou une contrainte

Annexe B: Grammaire des diagrammes de collaboration

La grammaire des CollDs est donnée ci-dessous.

```
CollD =
   ["frequency=" integer_literal] startMessage {object}.
startMessage = message object.
object =
   ["{new}" | "{destroyed}" | "{transient}"]
   [objectName] [":" packageName]
   [":" className]
   {attributeName "=" value}
   {link}.
link =
   linkedObject [role] [linkType] {message}.
role =
   ["{new}" | "{destroyed}" | "{transient}"] roleName.
linkType =
   "{association}" | "{global}" | "{local}" | "{parameter}" | "{self}".
message =
   controlFlowType [predecessor] [sequenceExpression]
   [returnValue ":="] messageName ["("[ argument {"," argument} ] ")"]
   [constraint].
controlFlowType =
   procedureCall | flatFlow | asynchronousFlow.
procedureCall =
   →.
flatFlow =
   →.
asynchronousFlow =
   ⇀.
predecessor =
   sequenceNumber {"," sequenceNumber } "/".
sequenceExpression =
   sequenceNumber [recurrence] ":".
sequenceNumber =
   integer_literal {("." integer_literal) | character_literal }.
recurrence =
   "*" iteration_clause | condition_caluse.
constraint =
     "{" ( ( "inputData(" className "." ( operationName | attributeName )
     ")" )
     | ( "outputData(" ( ( className "." attributeName ) |
     <string_Literal> ) ")" ) "}".
```

```
argument =
     identifier | string_literal | integer_literal |
     character_literal | "true" | "false".
iteration_clause =
     ( identifier "=" integer_literal ".." integer_literal ) | condition_clause
condition_clause =
     ifExpression {"OR" ifExpression}
     | orExpression.
ifExpression =
     "IF" orExpression "THEN" orExpression "ENDIF".
orExpression =
     andExpression {"OR" andExpression}.
andExpression =
     basicExpression {"AND" basicExpression}.
basicExpression =
     identifier ( ( "=" | "<" | ">" "<=" | ">=" ) ( string_literal |
     character_literal | integer_literal | floating_point_literal ) )
     | ("=" ( "true" | "false" ) ).

ObjectName, className, packageName, className, attributeName, roleName,
returnValue, messageName , linkedObject, roleName = identifier.
```

Le vocabulaire fourni par les CollDs pour la spécification d'une opération ou d'un scénario est composé de trois éléments de base: objets, liens, et messages. Les objets sont reliés entre eux par des liens. Les messages sont attachés aux liens pour montrer la communication entre les objets. Lorsqu'un CollD spécifie un scénario, il peut contenir l'information sur la fréquence d'utilisation du scénario (cf. section 3.1.1).

Un objet peut contenir les éléments suivants:

- le nom de l'objet,
- une contrainte sur l'objet: `new` (nouveau) pour un objet crée avec la première interaction, `destroyed` (détruit) pour un objet qui va être détruit avec la dernière interaction, `transient` (transitoire) pour un objet crée avec la première interaction puis détruit avec la dernière interaction,
- la classe de l'objet,
- la liste des attributs.

Un lien peut contenir les éléments suivants:

- l'objet avec lequel le lien est fait,
- le rôle,
- le type de lien,

- la liste des messages.

Un lien est une connexion unidirectionnelle entre deux objets. Plusieurs messages peuvent être attachés à ce lien. Ces messages sont envoyés à l'objet lié. Si la modélisation du problème exige une connexion bi-directionnelle entre deux objets alors deux liens, un dans chaque direction, doit être établi.

Un message peut contenir les éléments suivants (voir figure B.1):

- le type du flot de contrôle (→ appel procédual, → synchrone, ou ⇀ asynchrone),
- la liste des prédécesseurs pour les besoins de synchronisation entre les messages concurrents,
- le numéro de séquencement,
- la récurrence (pour indiquer les messages répétitifs ou les messages conditionnels),
- la valeur de retour,
- le nom du message et les arguments,
- une contrainte pour la liaison avec des éléments d'un ClassD.

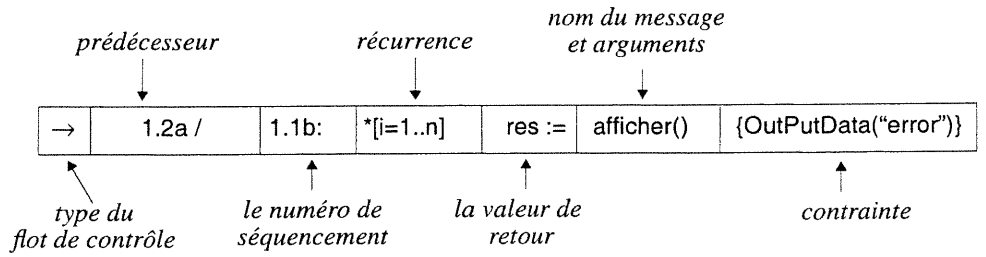


Figure B.1: Structure d'un message dans un CollD

Dans nos travaux, nous nous sommes limités aux concepts fondamentaux des CollDs tels que définis dans [Rational et al., 1997]. Nous avons donc introduit de petites simplifications. Premièrement, nous ne supportons pas les indicateurs de temps. Deuxièmement, l'indicateur de multiplicité pour les objets à l'intérieur d'une classe composite n'est pas supporté. Troisièmement, les problèmes relatifs à la distribution tels que la localisation, la migration, et la classification dynamique ne sont pas non plus supportés.

En outre, nous avons fait quelques extensions au niveau des CollDs pour les besoins de la génération d'un prototype de l'interface usager (cf. chapitres 3 et 7). Nous avons ajouté la notion de fréquence d'utilisation d'un scénario dans les CollDs. Nous avons également ajouté dans la

définition des messages une contrainte pour la liaison avec des éléments d'un ClassD.

Finalement, nous avons précisé le format que nous acceptons pour les conditions de récurrence comme dans le cas des pre- et post-conditions des opérations d'une classe.

Annexe C: Grammaire des diagrammes d'états-transitions

La grammaire des StateDs est donnée ci-dessous. Elle comprend seulement les aspects dont dépend notre travail. En outre, comme dans le cas des pre- et post-conditions des opérations d'une classe, nous avons précisé le format que nous acceptons pour les conditions de garde.

```
StateD =
   [name] {stateVariableDecl}
   {transition}.
stateVariableDecl =
   stateVariable ":" [className]
   ["=" initValue].
node =
   initialState | regularState | terminalState | merge bar |
   split bar | orstate | andstate.
orstate, andstate =
   [name] {substate}.
substate =
   StateD.
transition =
   fromNode
   [event]
   [guardCondition]
   {"/" action}
   {sendClause}
   ["^" returnValue]
   toNode.
fromNode, toNode = <ref to Node>.
event =
   eventName "("
   [parameter {"," parameter}] ")".
guardCondition =
   "[" conditionExpression "]".
sendClause =
   syncIndicator
   [result ":="]
   target "." event.
syncIndicator =
   procedureCall | flatFlow | asynchronousFlow.
procedureCall =
   →.
flatFlow =
   →.
```

```
asynchronousFlow =
   →.
target =
   className | objectName.
action =
   actionName "(" [ argument { "," argument } ] ")".
condition_clause =
   ifExpression {"OR" ifExpression}
   | orExpression.
ifExpression =
   "IF" orExpression "THEN" orExpression "ENDIF".
orExpression =
   andExpression {"OR" andExpression}.
andExpression =
   basicExpression {"AND" basicExpression}.
basicExpression =
   identifier ( ( "=" | "<" | ">" "<=" | ">=" ) ( string_literal |
   character_literal | integer_literal | floating_point_literal ) )
   | ("=" ( "true" | "false" ) ).
stateName, stateVariable, className, objectName, parameter, resutlt,
argument =
   identifier.
```

Les variables locales peuvent être utilisées dans le plus haut niveau de hiérarchie du StateD. Elles sont aussi appellées: les variables d'états. Donc, un StateD est composé de son nom, un ensemble de variables d'états, un ensemble d'états et un ensemble de transitions. Un StateD peut être de type `intialState` (état initial), `regularState` (état régulier), `terminalState` (état final), `splitBar` (état de synchronisation, point d'ouverture des flots d'exécution concurrents), `merge-Bar` (état de synchronisation, point de fermeture des flots d'exécution concurrents), `orState` (état de type OU), `andState` (état de type ET).

Une transition peut contenir les éléments suivants:

- le noeud de départ,
- l'événement de déclenchement,
- la condition de garde,
- une liste d'action,
- une liste d'envoi d'événements (événements à envoyer aux autre objets),
- le noeud d'arrivée.

Annexe D: Description détaillée de l'algorithme de transformation d'un diagramme de collaboration en diagrammes d'états-transitions

Dans ce qui suit, nous allons donner une description détaillée de l'algorithme de transformation d'un CollD en StateDs. Pour des question de lisibilité, nous allons présenter les étapes 1 à 4 dans un français structuré alors que toutes les parties essentielle de l'étape 5 vont être présentées en détail et ceci dans un langage proche de Modula-2.

Étape 1: Création des StateDs vides

Pour chaque `object` dans `CollD.{object}` faire 1.1:

1.1 si `StateD` pour `object` n'existe pas
alors
créer un nouveau `StateD` pour `object`.
`StateD.name := object.className`

Étape 2: Création des variables d'états pour les StateDs

Pour chaque `link` dans `CollD.{object}.{link}` faire 2.1, 2.2 et 2.3:

2.1 si `link.linktype = "local"`
alors
créer une nouvelle variable d'état `stVar` dans `StateD` de `object`.
`stVar.variableName := link.role.roleName.`
`stVar.className := link.linkedObject.className`

-- *Créer les variables d'états pour les liens de type `local`. Tous les autres types ne nécessite pas de déclaration de variables d'états.*

2.2 Pour chaque `message` dans `link.{message}` faire:
si `message.returnValue` n'est pas déclarée dans (`ClassD` ou `StateD`) de `object`
alors
créer une nouvelle variable d'état `stVar` dans `StateD` de `object`.
`stVar.variableName := returnValue`

-- *Créer les variables d'états pour les objets résultats non déclarés.*

2.3 Pour chaque `message` dans `link.{message}` faire:

Pour chaque `argument` dans `message.{argument}` faire:
si `argument` n'est pas une constante ou pas déclarée dans (`ClassD` ou `StateD`) de `object`
alors
créer une nouvelle variable d'état `stVar` dans `StateD` de `object`.
`stVar.variableName := argument`

-- Créer les variables d'états pour les paramètres des messages qui ni sont pas ni une constante ni déclarés.

2.4 Pour chaque `var` survenant dans `(link.{message}.sequence-Expression.recurrence` faire:
si `var` n'est pas déclarée dans (`ClassD` ou `StateD`) de `object`
alors
créer une nouvelle variable d'état `stVar` dans `StateD` de `object`.
`stVar.variableName := var.variableName`

-- *Créer les variables d'états pour les variables non déclarées dans les itérations et les conditions.*

Étape 3: Création des transitions pour les objets émetteurs

Pour chaque `message` dans `CollD.{object}.{link}.{message}` faire 3.1 à 3.11:

3.1 créer une nouvelle transition `trans` dans `StateD` de `object`.

3.2 si `object ≠ link.linkedObject` alors faire 3.3 à 3.9:

-- *Ne pas traiter les messages où l'émetteur est identique au récepteur (même objet).*

La partie événement de la transition `trans` (3.3 et 3.4):

3.3 si le nombre des éléments dans `message.predecessor = 1`
alors
trouver le message `msg` dans `CollD` où
`msg.sequenceNumber = message.predecessor.sequenceNumber.`
Si `msg ∉ object.{link}.{message}`
alors
`trans.event.eventName := msg.messageName.`
`trans.event.{parameter} := msg.{argument}.`

-- *trouver un message avec le même numéro de séquencement que le prédécesseur. Son nom devient l'événement qui déclenche la transition.*

3.4 si le nombre des éléments dans `message.predecessor > 1`
alors
pour chaque `sN` dans `message.predecessor.{sequenceNumber}`
faire:
trouver le message `msg` dans `CollD` avec `msg.sequenceExpression.sequenceNumber = sN`.
si `msg ∉ object.{link}.{message}`
alors
créer une nouvelle transition `auxTrans` pour `object`.
`auxTrans.event.eventName := msg.messageName + msg.sequenceNumber.`
`auxTrans.event.{parameter} := msg.{argument}.`
`auxTrans.tempSeqNumber := message.sequenceNumber + "-".`

-- *trouver des messages avec le même numéro de séquencement que les prédécesseurs. Pour chacun de ces messages (dans le cas où ils ne sont pas envoyés par `object`) une transition auxiliaire `auxTrans` est créée. La transition auxiliaire va être relié à une barre de synchronisation (`mergeBar`) laquelle en retour var être suivie par la transition correspondante au `message`.*
-- *Le champ `eventName` de `auxTrans` devient égal à la concaténation de la valeur de `messageName` du message duquel il est sorti, avec le numéro de séquencement de ce message. De cette façon, `nous garantissons l'unicité du champ eventName de auxTrans`.*
-- *"-" est ajouté à `tempSeqNumber` pour indiquer que ces transitions doivent être ordonnées dans une façon qu'elles précédent immédiatement la transition avec le même numéro de séquencement, mais sans le "-", c'est-à-dire, la transition corresponde au message.*

La partie envoi d'événement (sendClause) de la transition `trans` (3.5 à 3.9):

3.5 `trans.sendClause.syncIndicator := message.controlFlowType.`

3.6 si `message.returnValue` est spécifié
alors
`trans.sendClause.result := message.returnValue.`

3.7 si ((`link.role."{new}"` est spécifié) ou (`link.role."{transient}"` est spécifié)) et `minSeqNum(object, message, linkedObject)`
alors
`trans.sendClause.target := link.linkedObject.className`
sinon si `link.role.roleName` est spécifié
alors
`trans.sendClause.target := link.role.roleName.`

-- *Envoyer le premier événement à la classe `linkedObject` (constructeur), et tous les événements suivants à l'objet désigné par `rolename`.*
-- *`minSeqNum(obj1,msg,obj2)` est une fonction booléenne pour vérifier si le message `msg` a le plus petit numéro de séquencement de tous les messages envoyés de `obj1` à `obj2`.*
-- *Si `roleName` n'est pas spécifiée, `target` reste vide (doit être complété par la suite par le concepteur).*

3.8 s'il existe un message `msg` dans `link.linkedObject.{link}.{message}` pour lequel le nombre d'élément dans `msg.predecessor > 1` et
`(message.sequenceExpression.sequenceNumber ∈ msg.predecessor)`
alors
```
trans.sendClause.event.eventName := message.messageName +
   message.sequenceExpression.sequenceNumber
```
sinon
```
trans.sendClause.event.eventName := message.messageName.
```

-- *Si `sendClause.event` est l'un des événements qu'attend `link.linkedObject` (spécifié pr un message à multiple prédécesseur),le numéro de séquencement de `message` doit être ajouté à `eventName`. De cette façon, nous garantissions que les événements, pour lesquels `link.linkedObject attend, vont avoir un nom unique` (cf. le deuxième commentaire dans l'étape 3.4).*

3.9 `trans.sendClause.event.{parameter}:= message.{argument}`

Données temporaires (3.10 à 3.12):

3.10 trans.tempSeqNumber := message.sequenceNumber

3.11 trans.tempRecurrence := message.sequenceExpression.recurrence

Pour chaque `link` dans `CollD.{object}.{link}` faire:

3.12 si `link.role."{new}"` est spécifié alors `trans.tempIsNew := "new"`

-- *Les informations traitées dans 3.10 à 3.12 sont utilise dans l'étape 5 pour un séquencement adéquat. Une fois l'opération de séquencement est terminée, ces données vont être supprimée.*

Étape 4: Création des transitions pour les objets récepteurs

4.1 créer une nouvelle transition `trans` dans `StateD` de `startMessage.object`.
```
trans.event.eventName := startMessage.messageName.
trans.event.{parameter} := startMessage.{argument}.
trans.tempSeqNumber := startMessage.sequenceNumber.
```
si `startMessage.returnValue` est spécifié
alors
```
   trans.returnValue := startMessage.returnValue
```

-- *Créer une transition pour l'objet à qui le message de départ `startMessage` est envoyé. Seuls le nom, les arguments, le résultat et le numéro de séquencement de `startMessage` sont considérés; toutes les autres informations sont ignorées.*

-- *En retournant `returnValue` dans cette transition consiste une décision de conception faite par notre algorithme (cf. section 4.4 5).*

4.2 pour chaque `message` dans `CollD.{object}.{link}.{message}` faire:
si (`message.sequenceNumber` ∉ `link.linkedObject.{link}.{message}.predecessor}` and `object` ≠ `link.linkedObject`)
alors
créer une nouvelle transition `trans` dans `StateD` de `object.link.linkedObject.`
`trans.event.eventName := message.messageName.`
`trans.event.{parameter} := message.{argument}.`
`trans.tempSeqNumber := message.sequenceNumber.`
si `message.returnValue` est spécifié
alors
`trans.returnValue := message.returnValue.`

-- *Créer des transitions pour les messages qui sont envoyés à `object.link.linkedObject`. Ces messages doivent être séquentiels, c'est-à-dire, des messages sans prédécesseurs. En outre ils ne doivent pas être des messages envoyés par un objet à lui-même. les transitions pour les objets récepteurs en ce qui concerne les messages concurrents ont été déjà traité dans l'étape 3.4.*

Étape 5: Séquencement des transitions

Pour chaque `StateD` faire

`5.1 buildStateD(StateD)`

```
VAR   transList: ARRAY OF Transition;  (* premier élément a pour indice 1 *)
      nodeList: ARRAY OF Node;
      threadList: ARRAY OF INTEGER;
      threadLevel, level: INTEGER;
```

PROCEDURE buildStateD(VAR stateD: StateD) =

```
VAR   tr: Index;
      initialState, startState: Node;
TYPE TransitionType = {sequential, concurrent, iteration'};
BEGIN
   threadLevel := 0;
   transList := createOrderedTransitionList(stateD);
   IF (size(transList)>0) THEN
      startState := createRegularState();
      tr := 1;
      IF isSpecified(transList[1].TempIsNew) THEN
         initialState=createInitialState();
         connect(initialState, startState, transList[1]);
         tr:=tr+1
         END;
      sequence(startState, tr, size(transList));
      compressStateD(stateD)
      END
END buildStateD;
```

PROCEDURE sequence(fromNode: Node; from, to: Index) =

```
VAR   tr: Index;
      newNode: Node;
```

```
      transType: TransitionType;
      oldThreadLevel, count, tl: INTEGER;
BEGIN
   tr := from;
   WHILE tr <= to DO
      level := 0;
      IF ((tr<>from) and isSpecified(transList[tr].sendClause))
         THEN
         newNode := createRegularState();
         transList[tr-1].toNode := newNode
         END
      ELSIF tr=from THEN
         newNode := fromNode;
      ELSE
         newNode := createRegularState();
         fromNode := newNode;
         tl := tl + 1;
         threadList[tl] := tr - 1 (* transitions between threadList[tl-1] and
                     threadList[tl-1] belong to one concurrent substate *)
         END;
      transType := determineTransitionType(transList[tr]);
      CASE transType OF
         | sequential:  tr := tr + sequential(count, newNode, fromNode, tr);
                        to := to + count;
         | iteration:   tr := tr + iteration(count, newNode, fromNode, tr);
                        to := to + count;
         | condition:   oldThreadLevel := threadLevel;
                        threadLevel := threadLevel + 1;
                        tr := tr + condition(count, newNode, fromNode, tr);
                        to := to + count;
         | concurrent:  oldThreadLevel := threadLevel;
                        threadLevel := threadLevel + 1;
                        tr := tr + thread(count, newNode, fromNode, tr);
                        to := to + count;
                        threadLevel := oldThreadLevel;
         END
      END;
   tl := tl + 1;
   threadList[tl] := tr - 1;
   END sequence;
```

PROCEDURE sequenceInSide(fromNode, toNode: Node; from, to: Index): INTEGER=

```
VAR   tr, oldTo: Index;
      newNode: Node;
      transType: TransitionType;
      oldThreadLevel, count: INTEGER;
BEGIN
   tr := from;
   oldTo := to;
   level := level + 1;
   WHILE tr <= to DO
      newNode := createRegularState();
      transType := determineTransitionType(transList[tr]);
```

```
      CASE transType OF
         | sequential:  connect(fromNode, newNode, transList[tr]);
                        tr := tr + 1;
         | iteration:   tr := tr + iteration(count, fromNode, newNode, tr);
                        to := to + count;
         | condition:   oldThreadLevel := threadLevel
                        threadLevel := threadLevel + 1;
                        tr := tr + condition(count, fromNode, newNode, tr);
                        to := to + count;
                        threadLevel := oldThreadLevel;
         | concurrent:  oldThreadLevel := threadLevel
                        threadLevel := threadLevel + 1;
                        tr := tr + thread(count, fromNode, newNode, tr);
                        to := to + count;
                        threadLevel := oldThreadLevel;
         END;
      fromNode := newNode;
      END;
   deleteNode(fromNode);
   transList[tr-1].toNode := toNode;
   RETURN (to - oldTo)
   END sequenceInSide;
```

PROCEDURE connect(fromNode, toNode: Node; trans: Transition) =

```
BEGIN
   trans.fromNode := fromNode;
   trans.toNode := toNode;
   END connect;
```

PROCEDURE sequential(VAR count: Number; fromNode, toNode: Node; tr: Index): INTEGER=

```
VAR   newNode: Node;
      seq, size: INTEGER;
BEGIN
   size := determineNumberOfConstituentTransitions(tr);
   seq := 0;
   IF size>1 THEN
      newNode := createRegularState();
      connect(fromNode, newNode, transList[tr]);
      seq := sequenceInSide(newNode, toNode, tr+1, tr+size-1)
      END;
   ELSE   connect(fromNode, toNode, transList[tr]);
   size := size + seq;
   count := count + seq;
   RETRUN size
   END sequential;
```

PROCEDURE iteration(VAR count: INTEGER; fromNode, toNode: Node; tr: Index): INTEGER =

```
VAR   startState, checkState, endState: Node;
      loopTrans, incTrans, endTrans: Transition;
```

```
      loopVariable, upperBound: String;
      size, loopIndex: INTEGER;
      initLoop: BOOLEAN;
BEGIN
   size := determineNumberOfConstituentTransitions(tr);
   loopIndex := 0;
   initLoop := false;
   loopVariable := getLoopVariableFromTempRecurrence(transList[tr].
                     tempRecurrence);
   IF isSpecified(loopVariable) THEN
      loopIndex := loopIndex + 1;
      initLoop := true;
      checkState := createRegularState();
      transList[tr].action.actionName := "initLoop";
      connect(fromNode, checkState, transList[tr]);
      loopTrans := createTransition();
      upperBound := getUpperBoundFromTempRecurrence(transList[tr].
         tempRecurrence);
      END
   ELSE
      loopTrans := transList[tr];
      IF fromNode is SplitBar THEN checkState := fromNode
      ELSE
         checkState := createRegularState();
         trans := createTransition();
         connect(fromNode, checkState, trans);
         END
      END;
   startState := createRegularState();
   IF initLoop THEN
      loopTrans.guardCondition := "[" + loopVariable + "<=" + upperBound + "]";
   loopTrans.guardCondition := loopTrans.guardCondition +
      getConditionFromTempRecurrence(transList[tr].tempRecurrence);
   connect(checkState, startState, loopTrans);
   IF initLoop THEN
      endState := createRegularState()
   ELSE
      endState := checkState;
   seq := sequenceInSide(startState, endState, tr+loopIndex+1,
      tr+size+loopIndex-1);
   size := size + seq;
   IF initLoop THEN
      loopIndex := loopIndex + 1;
      incTrans := createTransition();
      incTrans.action.actionName := "increment";
      connect(endState, checkState, incTrans)
      END;
   endTrans := createTransition();
   endTrans.guardCondition := "not (" + loopTrans.guardCondition+")";
   connect(checkState, toNode, endTrans);
   count := count + seq + loopIndex + 1;
   RETURN size + loopIndex + 1
   END iteration;
```

PROCEDURE condition(VAR count: INTEGER; fromNode, toNode: Node; tr: Index): INTEGER =

```
VAR    threads: ARRAY OF INTEGER;
       oldTr: Index;
       sumSeq, seq, count2: INTEGER;
BEGIN
   oldTr := tr;
   threads := determineLengthsOfThreads(tr);
   IF not exclusiveGuardConditions(threads) THEN
      RETURN thread(count, newNode, fromNode, tr);
   FOR i := 1 TO size(threads) DO
      cond := cond + "AND NOT" + transList[tr].condition;
      IF not isSpecified(getLoopVariableFromTempRecurrence(
         transList[tr].tempRecurrence)) THEN
         transList[tr].condition := transList[tr].tempConditional;
         sequence(fromNode, toNode, tr, tr+threads[i]-1);
         tr := tr + threads[i];
         END;
      ELSE
         count2 := 0;
         seq := iteration(count2, sB, mB, tr);
         seq := count2
         END
      tr := tr + threads[i] + seq;
      sumSeq := sumSeq + seq;
      END;
   nullTrans := createTransition();
   nullTrans.condition := cond;
   connect(fromNode, toNode, nullTrans);
   count := count + sumSeq + 1;
   RETURN tr - oldTr + 1;
   END condition;
```

PROCEDURE thread(VAR count: INTEGER; fromNode, toNode: Node; tr: Index): INTEGER =

```
VAR    sB: SplitBar;
       mB: MergeBar;
       nextState: Node;
       threads: ARRAY OF INTEGER;
       oldTr: Index;
       sumSeq, seq, count2: INTEGER;
BEGIN
   oldTr := tr;
   threads := determineLengthsOfThreads(tr);
   IF size(threads) > 1 THEN
      sB := createSplitBar();
      mB := createMergeBar();
      trans := createTransition();
      connect(fromNode, sB, trans);
      tr := tr + 1;
      FOR i := 1 TO size(threads) DO
         IF not isSpecified(getLoopVariableFromTempRecurrence(
```

```
                transList[tr].tempRecurrence)) THEN
            nextState := createRegularState();
            connect(sB, nextState, transList[tr]);
            seq=sequenceInSide(nextState, mB, tr+1, tr+threads[i]-1);
            END
         ELSE
            count2 := 0;
            seq := iteration(count2, sB, mB, tr);
            seq := count2
            END
         tr := tr + threads[i] + seq;
         sumSeq := sumSeq + seq;
         END;
      trans := createTransition();
      connect(mB, toNode, trans);
      count := count + sumSeq + 2;
      RETURN tr - oldTr + 1
      END
   ELSIF determineTransitionType(transList[tr]) = concurrent THEN
      threadLevel := threadLevel + 1;
      RETURN thread(count, fromNode, toNode, tr)
      END
   ELSIF isSpecified(getLoopVariableFromTempRecurrence
         (transList[tr].tempRecurrence)) THEN
      RETURN iteration(count, fromNode, toNode, tr)
   ELSE
      connect(fromNode, toNode, transList[tr]);
      RETURN 1
      END;
   END thread;
```

PROCEDURE createOrderedTransitionList(stateD: StateD): ARRAY OF Transition =

```
   (* construire un tableau avec tous les éléments de stateD.{transition}
      ordonnés par tempSeqNumber. Les transitions avec tempSeqNumber ayant un
      '-' à la fin sont ordonnées dans une façon qu'elles précédent
      immédiatement la transition correspondante à tempSeqNumber. Ces
      transitions sont des transitions artificielles qui reflètent les
      prédécesseurs d'un message.
   *)
```

PROCEDURE removeElementFromTransitionList(VAR transitionList: ARRAY OF Transition; seqNum: SequenceNumber; tr: Index) =

```
   (* supprimer de transitionList, de l'indice tr à taille(transitonList), les
      transitions que leur tempSeqNumber est égal à seqNum *)
```

PROCEDURE determineTransitionType(trans: Transition): TransitionType =

```
BEGIN
   IF hasThreadAtLevel(trans.tempSeqNumber, threadLevel+1) = FALSE
      AND isSpecified(getLoopVariableFromTempRecurrence(trans.tempRecurrence))
            = FALSE
      THEN RETURN sequential
```

```
   ELSIF hasThreadAtLevel(trans.tempSeqNumber, threadLevel+1) = TRUE
      THEN RETURN concurrent
   ELSIF hasThreadAtLevel(trans.tempSeqNumber, threadLevel+1) = FALSE
      AND isSpecified(getLoopVariableFromTempRecurrence
         (trans.tempRecurrence)) = TRUE
      THEN RETURN iteration;
   END determineTransitionType;
```

PROCEDURE numberOfThreadIndicators (seqNumber: SequenceNumber): INTEGER =

```
   (* compte le nombre d'indicateurs de concurrence seqNumber. *)
```

PROCEDURE hasThreadAtLevel
(seqNumber: SequenceNumber; threadLevel: INTEGER): BOOLEAN=

```
BEGIN
   IF numberOfThreadIndicators(seqNumber)=threadLevel THEN
      RETURN true
   ELSE
      RETURN false
   END hasThreadAtLevel;
```

PROCEDURE isSpecified(symbol: Symbol): BOOLEAN =

```
   (* retourner true si symbol a une valeur.
      Si aucune valeur n'est assignée, retorune false. *)
```

PROCEDURE determineNumberOfConstituentTransitions(tr: Index): INTEGER =

```
   (* retourne le nombre de transitions qui correspondent à tempSeqNumber de
      la transition référencée par l'indice tr.
      Dans le cas où loopVariable de tempRecurrence de cette transition est
      spécifiée, la transition correspond si son tempSeqNumber a comme préfixe
      tempSeqNumber de la transition référencée par l'indice tr. Dans le cas
      d'un flot d'exécution concurrent, la correspondance ignore le numéro de
      la plus petite séquence de la transition référencée par l'indice tr. *)
```

PROCEDURE determineLengthsOfThreads(tr: Index): ARRAY OF INTEGER =

```
   (* retourne un tableau des longueurs des flots d'exécution concurrents dans
      un groupe de flots à qui la transition référencée par l'indice tr
      appatient. *)
```

PROCEDURE exclusiveGuardConditions(indexList: ARRAY OF INTEGER): boolean=

```
   (* retourne true si les condition des transitions correspondantes aux indi-
ces de indexList sont exclusive. Sinon retourne false. *)
```

PROCEDURE compressStateD(VAR stateD: StateD)=

```
VAR   newNode: Node;
      listOfNodes: ARRAY OF Node;
BEGIN
   IF size(threadList)<2 THEN
      mergeSuccessiveActions(transList, nodeList);
      mergeSuccessiveSendClauses(transList, nodeList);
```

```
      mergeActionsAndSendClauses(transList, nodeList);
      mergeEventAndActions(transList, nodeList);
      eliminateDuplicateTransitionsBetweenTwoNodes(transList, nodeList);
      stateD.{transition} := transList;
      stateD.{node} := nodeList
      END
   ELSE
      newNode := createRegularState();
      newNode.{concurrentSubstate} := buildAndCompressConcurrentSubstates();
      listOfNodes[0] := n;
      stateD.{node} := listOfNodes;
      END
   END compressStateD;
```

PROCEDURE mergeSuccessiveActions(VAR theTransList: ARRAY OF Transition; VAR theNodeList: ARRAY OF Node)=

```
(* fusionne en une transition deux transitions séquentielles ne contenant
   que des actions. *)
```

PROCEDURE mergeSuccessiveSendClauses(VAR theTransList: ARRAY OF Transition; VAR theNodeList: ARRAY OF Node)=

```
(* fusionne en une transition deux transitions séquentielles ne contenant
   que des envois d'événements. *)
```

PROCEDURE mergeActionsAndSendClauses(VAR theTransList: ARRAY OF Transition; VAR theNodeList: ARRAY OF Node)=

```
(* fusionne en une transition deux transitions séquentielles dont la
   premiére ne contient que des actions et la deuxième des envois
   d'événements. *)
```

PROCEDURE mergeEventAndActions(VAR theTransList: ARRAY OF Transition; VAR theNodeList: ARRAY OF Node)=

```
(* fusionne en une transition deux transitions séquentielles où la première
   transition contient seulement un événement et/ou une condition de garde,
   et la seconde transition contient seulement des envois d'événements et/
   ou des actions et/ou une valeur de retour. *)
```

PROCEDURE eliminateDuplicateTransitionsBetweenTwoNodes(VAR theTransList: ARRAY OF Transition; VAR theNodeList: ARRAY OF Node)=

```
(* éliminer les transitions dupliquées. Deux transitions sont dupliquées si
   elles relient les mêmes états et contienent les mêmes événements,
   conditions de garde, action, envois d'événements et valeurs de retour. *)
```

PROCEDURE buildAndCompressConcurrentSubstates(): ARRAY OF Node =

```
(* À l'aide des numéros dans threadList, cette procédure retourne une liste
   d'états concurrent pour un état. Chaque sous-état retourne une liste
```

d'état concurrents. Chaque sous-état est aussi comprimée en utilisant la procédure de compression. *)

PROCEDURE getSequenceNumberFromEventName(String eventName): String =

(* retourne un numéro de séquencement du message qui a généré la transition. Ce numéro de séquencement est sauvegardé dans le nom de l'événement de la transition (voir étape 3 de l'algorithme). *)

PROCEDURE getLoopVariableFromTempRecurrence(String tempRecurrence): String =

(* retourne le nom de la variable de boucle si elle existe dans tempRecurrence, sinon retourne une chaîne vide. *)

PROCEDURE getUpperboundFromTempRecurrence(String tempRecurrence): String =

(* Si elle existe, retourne la borne supérieure de tempRecurrence, sinon retourne une chaîne vide. *)

PROCEDURE createInitialState(): InitialState

(* crée un état initial et insère le dans *nodeList*. *)

PROCEDURE createTerminalState(): TerminalState

(* crée un état final et insère le dans *nodeList*. *)

PROCEDURE createRegularState(): RegularState

(* crée un état régulier et insère le dans *nodeList*. *)

PROCEDURE createTransition(): Transition

(* crée une transition et insère la dans *transList*. *)

PROCEDURE createSplitBar(): SplitBar

(* crée un splitBar et insère le dans *nodeList*. *)

PROCEDURE createMergeBar(): MergeBar

(* crée un état mergeBar et insère le dans *nodeList*. *)

PROCEDURE deleteNode(node: Node)

(* supprime un noeud indiquée par node. Supprime le *nodeList*. *)

Annexe E: Description détaillée de l'algorithme d'analyse d'un diagramme d'états-transitions

Dans ce qui suit, nous allons donner toutes les parties essentielles de l'algorithme d'analyse d'un StateD dans un langage proche de Modula-2.

PROCEDURE analysePartialSpecification(aClass: Class; VAR stateD: StateD): boolean=

```
VAR   i, j, mbcount, sbcount: integer;
      bool: boolean;
      pre, post: Condition;
      transList: ARRAY OF Transition;
      indexBT: ARRAY OF integer;
BEGIN
   bool := classDescriptionConsistencyChecking(aClass);
   IF not bool THEN RETURN FALSE;
   transList := stateD.{transition};
   pre := ""; post := "";
   i := 1; mbcount := 0; sbcount := 0;
   FOR i:=1 TO transList.size DO
      indexBT[i] := 0;
   WHILE (i<transList.size) DO
      labelOneTransition(aClass, i, mbcount, sbcount, stateD, indexBT);
      IF (i<transList.size-1) THEN
         bool := transitionConsistencyChecking(transList[i+1], aClass);
         IF not bool  THEN
            i := lookForIndexOfBacktracking(indexList, transList);
         END;
      i := i + 1;
      END;
   stateD.{transition} := transList;
   IF not bool THEN RETURN FALSE;
   bool := stateDConsistencyChecking(aClass, stateD);
   RETURN bool;
END analysePartialSpecification;
```

PROCEDURE labelOneTransition(aClass: Class; index: integer; VAR mbcount, sbcount: integer; VAR stateD: StateD; VAR indexBT: ARRAY OF integer)=

```
VAR   i:  integer;
      bool: boolean;
      c1, c2, attributePart: Condition;
      Sc: Array of Condition;
      transList: ARRAY OF Transition;
      trans: Transition;
```

```
      st: StateD;
BEGIN
   transList := stateD.{transition};
   trans := transList[index];
   computec1andc2andSc(index , aClass, stateD, c1, c2, Sc, indexBT);
   IF (trans.fromNode is a merge Bar or trans.fromNode is a splitBar) THEN
      handleSplitBarOrMergeBar(trans.fromNode, mbCount, sbcount);
   ELSE
      attributePart := attributePart(c1, aClass);
      st := lookForStateWithName(attributePart, stateD);
      IF (st=NIL) THEN trans.fromNode.name := attributePart;
      ELSE trans.fromNode := st;
      END;
   IF (trans.toNode is a mergeBar or trans.toNode is a splitBar) THEN
      handleSplitBarOrMergeBar(trans.toNode, mbCount, sbcount);
   ELSE
      attributePart := attributePart(c2, aClass);
      st := lookForStateWithName(attributePart, stateD);
      IF (st=NIL) THEN trans.toNode.name := attributePart;
      ELSE trans.toNode := st;
      END;
   IF equalConditions(aClass, Sc) THEN
      trans.guardCondition := conjonctionOf(trans.guardCondition,
                                            cpList[1]);
   ELSE
      trans.guardCondition := conjonctionOf(trans.guardCondition,
                                            preCond(c1));
END labelOneTransition;
```

PROCEDURE classDescriptionConsistencyChecking(aClass: Class): boolean=

```
VAR   i:  integer;
      bool, bool1: boolean;
      op: Operation;
BEGIN
   bool := TRUE;
   opList := aClass{operation};
   FOR i=1 TO size(opList) DO
      op := opList[i];
      bool1 := conditionConsistencyChecking(op.pre);
      IF (not bool1) THEN
         output("The pre-condition of ", op, "is not consistent");
      bool := bool AND bool1;
      bool1 := conditionConsistencyChecking(op.post);
      IF (not bool1) THEN
         output("The post-condition of ", op, "is not consistent");
      bool1 := conditionConsistencyCheckingWithRespectTo(op.post, op.pre);
      IF (not bool1) THEN
         output("The post-condition of ", op, "is not consistent with
                  its pre-condition");
      bool := bool AND bool1;
      END;
   RETURN bool;
END classDescriptionConsistencyChecking;
```

PROCEDURE stateDConsistencyChecking(aClass: Class; VAR stateD: StateD): boolean=

```
VAR   bool: boolean;
BEGIN
   bool := guardConditionsConsistencyChecking(aClass, stateD) and
         isNondeterminst(aClass, stateD) and stateRelationsChecking(aClass,
                                                    stateD);
   RETURN bool;
END stateDConsistencyChecking;
```

PROCEDURE stateRelationsChecking(aClass: Class; stateD: StateD): boolean=

```
BEGIN
   IF stateD is an orstate THEN
      RETURN orStateChecking(aClass, stateD);
   IF stateD is an andstate THEN
      RETURN andStateChecking(aClass, stateD);
   RETURN true;
END stateRelationsChecking;
```

PROCEDURE orStateChecking(aClass: Class; stateD: StateD): boolean=

```
VAR   bool1, boo: boolean;
      i: integer;
      substateList: Array of StateD;
BEGIN
   substateList := stateD.{substate};
   IF (stateD.name = aClass.name) THEN bool2 := false;
   FOR i=1 TO size(substateList) DO
      bool1 := bool1 and stateRelationsChecking(aClass, substateList[i]);
      IF (bool2 and substateList[i] is not a split bar or merge bar) THEN
         IF (not refines(stateD.name, substateList[i].name, aClass)) THEN
            bool1 := false;
            output("The state ", stateD, "is not an orstate of ",
                  substateList[i]);
            END;
      END;
   RETURN bool1;
END orStateChecking;
```

PROCEDURE andStateChecking(aClass: Class; stateD: StateD): boolean=

```
VAR   bool1, boo: boolean;
      i: integer;
      substateList: Array of StateD;
BEGIN
   substateList := stateD.{substate};
   IF (stateD.name = aClass.name) THEN bool2 := false;
   FOR i=1 TO size(substateList)-1 DO
      bool1 := bool1 and stateRelationsChecking(aClass, substateList[i]);
      IF (bool2 and substateList[i] is not a split bar or merge bar) THEN
         FOR j=i+1 TO size(substateList) DO
            IF (substateList[j] is not a split bar or merge bar) THEN
               IF (not isConcurrentWith(substateList[i].name,
                  substateList[j].name, aClass))
```

```
                THEN
                   bool1 := false;
                   output("The states ", substateList[i], "and ",
                         substateList[j], " are no concurrent");
                   END;
      END;
   RETURN bool1;
END andStateChecking;
```

PROCEDURE isNondeterminist(aClass: Class; VAR stateD: StateD): boolean=

```
BEGIN
   return (checkForTransitionsHavingOnlyDifferentActionParts(aClass, stateD)
         and checkForTransitionsHavingOnlyDifferentToNodes(aClass, stateD);
END isNondeterminist;
```

PROCEDURE checkForTransitionsHavingOnlyDifferentActionsPartsA(aClass: Class; stateD: StateD): boolean=

```
VAR    i, j: integer;
       bool: boolean;
       transList: Array of Transition;
BEGIN
   transList := stateD.{transition}; bool := true;
   FOR i=1 TO size(transList)-1 DO
      FOR j=i+1 TO size(transList) DO
         IF (transList[i].fromNode is not a split bar and
            transList[i].fromNode = transList[j].fromNode and
            transList[i].toNode = transList[j].toNode and
            transList[i].event = transList[j].event and
            intersection(transList[i].guardCondition,
                  transList[j].guardCondition, aClass) THEN
            IF (not equalListOfActions(transList[i], transList[j]) or
               not equalListOfActions(transList[i], transList[j])) THEN
               bool := false;
               output("the action parts of ", transList[i], " and ",
                     transList[j], " are different");
               END;
   RETURN bool;
END checkForTransitionsHavingOnlyDifferentActionParts;
```

PROCEDURE checkForTransitionsHavingOnlyDifferentToNodes(aClass: Class; stateD: StateD): boolean=

```
VAR    i, j: integer;
       bool: boolean;
       trans: Transition;
       transList1, transList2: Array of Transition;
       lsb: Array of StateD;
       cond: Condition;
       s: string;
BEGIN
   transList1 := stateD.{transition}; bool := true; i := 1;
   WHILE (i<= size(transList1)) DO
```

```
      trans := transList1[i];
      transList2 :=
      lookForTransitionsHavingOnlyDifferentToNodesAfter(aClass, trans, i,
      stateD);
      IF (size(transList2)>0) THEN
         IF allToNodesAreAtTheSameLevel(transList2) THEN
            st := createAndState();
            IF (trans.toNode is not a split bar or a merge bar) THEN
               cond := trans.toNode.name;
            lsb[size(lsb)+1] := trans.toNode;
            removeStateWithName(trans1.toNode, stateD);
            s := getTransScenarioVariable(trans, stateD);
            FOR j=1 TO size(transList2) DO
               IF (transList2[j].toNode is not a split bar or a merge bar) THEN
               cond := conjunctionOf(cond, transList2[j].toNode.name);
               lsb[size(lsb)+1] := transList2[j].toNode;
               removeStateWithName(transList2[j].toNode, stateD);
               s := mergeTwoSetsOfString(s,
                  getTransScenarioVariable(transList2[j], stateD);
               removeTransition(transList2[j], stateD);
            END
            st.{substate} := lsb;
            substateList[size(substateList)+1] := st;
            trans.toNode := st;
            changeTransScenarioVariable(i, s, stateD);
         END
         ELSE
            bool := false; i := size(transList1);
            END;
      i++;
   END
   RETURN bool;
END checkForTransitionsHavingOnlyDifferentToNodes;
```

PROCEDURE computec1andc2andSc(index: integer; aClass: Class; stateD; StateD; VAR c1, c2: Condition; VAR Sc: Array of Condition; VAR indexBT: ARRAY OF integer)=

```
VAR   trans: Transition;
      transList: ARRAY OF Transition;
BEGIN
   transList := stateD.{transition};
   trans := transList[index];
   IF postTrans(trans) is an ORexpression THEN
      IF (trans.fromNode.name=="") THEN c1 := preTrans(trans);
      ELSE c1 := trans.fromNode.name;
      IF (trans.toNode.name=="") THEN c2 := post(trans);
      ELSE c2 := trans.fromNode.name;
      Sc := parameterList(preTrans(trans));
      END
   ELSE
      IF (trans.fromNode.name=="") THEN
         indexBT[index] := indexBT[index] + 1;
         c1 := expressionListOfOR(postTrans(trans))[indexBT[index]];
         END
```

```
      ELSE c1 := trans.fromNode.name;
      IF (trans.toNode.name=="") THEN
         c2 := postCond(postTrans(trans), c1, aClass);
      ELSE c2 := trans.fromNode.name;
      Sc := expressionListOfOR(
            expressionListOfOR(pos Trans(trans))[indexBT[index]]);
      END
END computec1andc2andSc;
```

PROCEDURE handleSplitOrMergeBar(VAR st: Node; VAR mbcount, sbcount: Integer)=

```
BEGIN
   IF st is a split Baris THEN
      st.name := "sB" + sbcount; sbcount ++;
   END
   ELSE
      IF st is a mergeBar THEN
         st.name := "mB" + sbcount; mbcount ++;
      END
END handleSplitOrMergeBar;
```

PROCEDURE allNodesAreAtTheSameLevel(transList: Array of Transition): boolean

```
(* retourne true si tous les fromNode des transitions de transList appartien-
nent au même niveau hiérarchique. Sinon elle retourne false. *)
```

PROCEDURE lookForTransitionsHavingOnlyDifferentToNodesAfter(aClass: Class; trans: Tansition; i: integer; stateD: StateD): Array of Transition

```
(* retourne toutes les transitions de stateD{transition} à partir de l'indice
i+1. Chacune de ces transitions trans2 doit en outre respecter la condition
suivante: (trans.fromNode raffine trans2.fromNode ou trans2.fromNode raffine
trans.fromNode) et (trans.event=trans2.event) et (trans.guardCondition et
trans2.guardCondition ne sont pas exclusives) et (trans.{action}=
trans2.{action}) et (trans{sendClause}=trans2.{sendClause}) et trans.toNode
est différent de trans2.toNode). *)
```

PROCEDURE transitionsConsistencyChecking(trans: Transition; aClass: Class;): boolean

```
(* retourne true si trans est cohérente avec son fromNode et son toNode (voir
définition 5.20). Sinon elle retourne false. *)
```

PROCEDURE guardConditionsConsistencyChecking(aClass: Class; stateD: StateDt): boolean

```
(* retourne true si toutes les conditions de garde des transitions de stateD
sont cohérents (voir définitions 5.17 et 5.18). Sinon elle retourne false. *)
```

PROCEDURE refines(cd1, cd2: Condition; aClass: Class): boolean

```
(* retourne true cd1 raffine cd2 (voir définition 5.5.Sinon elle retourne
false. *)
```

PROCEDURE isConcurrentWith(cd1, cd2: Condition; aClass: Class): boolean

```
(* retourne true cd1 et cd2 respecte la condition de concurrence de la défini-
tion 5.8.Sinon elle retourne false. *)
```

PROCEDURE intersection(cd1, cd2: Condition; aClass: Class)): boolean

```
(* retourne true si eval(cd1) ∩ eval(cd2) ≠∅ (voir définition 5.1).Sinon elle
retourne false. *)
```

PROCEDURE lookForIndexBackTracking(indexBT: ARRAY OF integer; transList: ARRAY OF Transition): integer

```
(* retourne le dernier élément de indexBT qui est inférieure aux nombre d'élé-
ments retourné par expressionList de la post-condition de la transition corre-
spondante. *)
```

PROCEDURE conditionConsistencyChecking(cd: Condition; aClass: Class)): boolean

```
(* retourne true cd est cohérent (voir définitions 5.17 et 5.18).Sinon elle
retourne false. *)
```

PROCEDURE conditionConsistencyCheckingWithRespectTo(cd1, cd2: Condition; aClass: Class)): boolean

```
(* retourne true cd1 est cohérent avec cd2 (voir définition 5.19).Sinon elle
retourne false. *)
```

PROCEDURE preTrans(trans: Transition; aClass: Class): Condition

```
(* retourne preTrans(trans) de la classe aClass (voir définition 5.9) *)
```

PROCEDURE postTrans(trans: Transition; aClass: Class): Condition

```
(* retourne postTrans(trans) de la classe aClass (voir définition 5.9) *)
```

PROCEDURE postCond(cd1, cd2: Condition, aClass: Class): Condition

```
(* retourne postCond(cd1, cd2) (voir définition 5.11) *)
```

PROCEDURE expressionListOfOR(cd: Condition): Condition

```
(* retourne expressionListOfOR(cd) (voir définition 5.16) *)
```

PROCEDURE equalConditions(aClass: Class; Sc: Array of Condition): boolean

```
(* retourne true si tous les éléments de Sc sont, sinon elle retourne false.*)
```

PROCEDURE conjunctionOf(cd1, cd2: Condition): Condition

```
(* retourne une condition cd qui fait la conjonction de deux conditions cd1 et
cd2. cd est dans une forme canonique disjunctive. *)
```

PROCEDURE createTransition(VAR stateD: StateD): Transition

```
(* crée une nouvelle transition dans stateD. *)
```

PROCEDURE lookForStateWithName(name: string; stateD: StateD): StateD

```
(* retourne un état s'il existe, ayant comme nom name, de stateD. Sinon elle
retourne NIL. *)
```

PROCEDURE removeStateWithName(name: string; stateD: StateD)

```
(* supprime un état ayant name comme nom de stateD. *)
```

PROCEDURE removeTransition(trans: Transition; VAR stateD: StateD)

```
(* supprime la transition trans dans stateD. *)
```

PROCEDURE createAndState(): StateD

```
(* crée un état de type ET. *)
```

Annexe F: Description détaillée de l'algorithme d'intégration de deux diagrammes d'états-transitions

Dans ce qui suit, nous allons donner toutes les parties essentielles de l'algorithme d'intégration de deux StateDs dans un langage proche de Modula-2.

PROCEDURE integrationOfStateDs(stateD1, stateD2: StateD; VAR stateD: StateD)

```
VAR   b:  boolean;
BEGIN
   b := stateChecking(stateD1, stateD2);
   IF (b) THEN
      stateD.name := stateD1.name;
      incorporationOfCompositionVariables(StateD1, StateD2);
      stateVariableMerging(stateD1, stateD2, stateD);
      stateMerging(stateD1, stateD2, stateD);
      transitionMerging(stateD1, stateD2, stateD);
      stateDConsistencyChecking(aClass, stateD);
      END
END integrationOfStateDs;
```

PROCEDURE stateChecking(stateD1, stateD2: StateD): boolean=

```
VAR   b1, b2: boolean;
BEGIN
   b1 := checkDuplicateStates(stateD1);
   b2 := checkDuplicateStates(stateD2);
   IF (b1 AND b2) THEN
      b1 := checkConflictsBetweenStateDs(stateD1, stateD2, stateD1);
      END;
   ELSE b1:= false;
   return b1 and b2;
END stateChecking;
```

PROCEDURE checkDuplicateStates(stateD: StateD): boolean=

```
VAR   b, b1:BOOLEAN;
      i: integer;
      substateList: ARRAY OF StateD;
BEGIN
   b := false;
   IF (getNumberOfStateWithName(st.name, stateD.name)<>1) THEN
```

```
      b := false;
      output("State ", st.name, " exist more than once")
      END;
   substateList := st.{substates};
   FOR i=1 TO size(substateList) DO
      b1 := checkDuplicateStates(substateList[i], stateD);
      b := b and b1
      END;
   return b
END checkDuplicateStates;
```

PROCEDURE getNumberOfStateWithName(name: string; stateD: StateD): INTEGER=

```
(* returns the number of states with a name name in stateD *)
```

PROCEDURE getPath(name: string; stateD: StateD; VAR path: ARRAY OF NameAndType): BOOLEAN=

```
VAR   i:  integer;
      b: boolean;
      substateList: ARRAY OF StateD;
      nt: NameAndType;
BEGIN
   IF (name=stateD.name) THEN RETURN TRUE;
   ELSE
      substateList := stateD.{substate};
      i := 1;
      WHILE (i<=size(substateList)) DO
         b := getPath(name, substateList[i], path);
         IF (b) THEN i := size(substateList) ELSE i := i + 1;
         END;
      IF (b) THEN
         nt.name := stateD.name;
         nt.type := type of stateD;
         insertElementsInPathAt(nt, 1, path)
         END
      return b;
   END;
END getPath;
```

PROCEDURE checkConflictsBetweenStates(stateD1, stateD2, stateD: StateD): BOOLEAN=

```
VAR   b, b1: boolean;
      i: integer;
      path1, path2: ARRAY OF NameAndType;
      substateList: ARRAY OF StateD;
BEGIN
   IF (lookForStateWithName(stateD.name, stateD2)<>NIL) THEN
      getPath(stateD.name, stateD1, path1);
      getPath(stateD.name, stateD2, path2);
      IF (not equalsPaths(path1, path2)) THEN
         b := false;
         output("State ", stateD.name, " exist in the two stateDs with
```

```
                different paths")
        END
      END;
   substateList := stateD.{substate};
   FOR i=1 TO size(substateList) DO
      b1 := checkConflictsBetweenStates(stateD1, stateD2,
            substate[i]);
      b := b and b1
      END;
   return b;
END checkConflictsBetweenStates;
```

PROCEDURE stateMerging(stateD1, stateD2: StateD; VAR stateD: StateD) =

```
VAR   b: boolean;
      i, pos:integer;
      substateList1, substateList2: ARRAY OF StateD;
      startStateList1, startStateList2: ARRAY OF string;
      st, stateD3: stateD;
BEGIN
   IF ((stateD1 is an andState) OR (stateD1 is an orState)
      OR (stateD2 is an andState) OR (stateD2 is an orState))
      THEN
         substateList1 := stateD1.{substate};
         substateList2 := stateD2.{substate};
         startStateList1 := getListOfStartStates(stateD1);
         startStateList2 := getListOfStartStates(stateD2);
         FOR i=1 TO size(startStateList2) DO
            pos := positionOfElementInStartStateList(startStateList2[i],
                                                     startList1);
            IF (pos<>-1) THEN
               IF (stateD1 is not an andState) THEN
                  IF (stateD2 is not an andState) THEN
                     orMerging(stateD1, stateD2, stateD);
                  ELSE
                     stateD3 := lookForStateWithName(stateD2.name, stateD);
                     orMerging(stateD1, substateList2[i], stateD);
                  END;
               ELSE
                     stateD3 := lookForStateWithName(stateD1.name, stateD);
                     IF (stateD2 is not an andState) THEN
                        orMerging(substateList1[pos], stateD2, stateD);
                     ELSE
                        orMerging(substateList1[pos], substateList2[i], stateD);
                     END;
               END;
            ELSE
                  IF (stateD2 is not an andState) THEN
                     andMerging(stateD1, stateD2, stateD);
                  ELSE
                     andMerging(stateD1, substateList2[i], stateD)
            END;
            IF (size(startStateList2)=0) THEN
                  andMerging(stateD1, stateD2, stateD);
```

```
            FOR i=0 TO size(substateList1) DO
               stateD3 = lookForStateWithName(substateList1[i].name, stateD2);
               st = lookForStateWithName(stateD3.name, stateD2);
               IF (st<>NIL) THEN
                  stateMerging(stateD3, st, stateD);
               END
         END
END stateMerging;
```

PROCEDURE transitionMerging(stateD1, stateD2: StateD; VAR stateD: StateD) =

```
VAR   b: boolean;
      i, j, pos: integer;
      c: string;
      s: SET OF string;
      st: StateD;
      lsb: ARRAY OF StateD;
      transList1, transList2: ARRAY OF Transition;
      la1, la2: ARRAY OF string;
      lsc1, lsc2: ARRAY OF SendClause;
BEGIN
   transList1:= stateD1.{transition};
   transList2:= stateD2.{transition};
   FOR i=1 TO size(transList2) DO
      trans2 := transList2[i];
      pos := lookForTransitionHavingTheSameFirstTriplet(trans2, transList1);
      IF (pos=-1) THEN transList1[size(transList1)+1] := trans2
      ELSE
         trans1 := transList1[pos];
         la1 := trans1.{action};
         lsc1 := trans1.{sendClause};
         la2 := trans2.{action};
         lsc2 := trans2.{sendClause};
         IF (equalsListOfAction(la1, la2) and equalsListOfSendClause(lsc1,
            lsc2)) THEN
            trans1.guardCondition := disjunction(trans1.guardCondition,
            trans2.guardCondition);
            transList1[pos] := trans1;
            s := mergeTwoSetsOfString(getTransScenarioVariable(trans1,
                     stateD1), getTransScenarioVariable(trans2, stateD2));
            changeTransScenarioVariable(pos, s, stateD)
         END
         ELSE
            transList1[size(transList1)] := trans2;
            s := getTransScenarioVariable(trans2, stateD2);
            changeTransScenarioVariable(size(transList1), s, stateD)
         END
      END
   END
   stateD.{transition} := transList1;
END transitionMerging;
```

PROCEDURE orMerging(stateD1, stateD2: StateD; VAR stateD: StateD) =

```
VAR    b: boolean;
       i: integer;
       substateList1, substateList2, lsb: ARRAY OF StateD;
       st, auxstateD: stateD;
BEGIN
   substateList1 := stateD1.{substate};
   substateList2 := stateD2.{substate};
   auxstateD := stateD1;
   lsb := auxstateD.{substate};
   FOR i=0 TO size(substateList2) DO
      IF (lookForStateWithName(substateList2[i].name, stateD1)=NIL) THEN
         lsb[size(lsb)+1] := substateList2[i];
   st := lookForStateWithName(auxstateD.name, stateD);
   IF (st<>NIL) THEN
      st := auxstateD
END orMerging;
```

PROCEDURE andMerging(stateD1, stateD2: StateD; VAR stateD: StateD) =

```
VAR    b: boolean;
       i: integer;
       substateList1, substateList2, lsb: ARRAY OF StateD;
       st, auxstateD: stateD;
BEGIN
   substateList1 := stateD1.{substate};
   substateList2 := stateD2.{substate};
   IF ((type of stateD1<>orState) AND
      (type of stateD1 <>andState))THEN
      auxstateD := stateD2;
      auxstateD.name := stateD1.name;
      END
   ELSE
      IF (type of stateD1=orState) THEN
         IF ((type of stateD2 <>orState) AND
               (typeo of stateD2<>andState))THEN
               auxstateD := stateD1
            ELSE
               auxstateD := stateD1;
               st1 := lookForStateWithName(substateList1[1].name, stateD1);
               st2 := lookForStateWithName(substateList2[1].name, stateD2);
               st := createAndState();
               st.name := conjunction(st1.name, st2.name);
               lsb := substateList1;
               insertElementInListOfSubstatesAt(st, lsb, 1);
               FOR i=1 TO size(substateList2) DO
                  IF (lookForStateWithName(st1.name, stateD1)=NIL) THEN
                     lsb[size(lsb)+1] := st1;
               auxstateD.{substate} := lsb;
               removeStateWithName(substateList1[1], auxstateD);
               removeStateWithName(substateList2[1], auxstateD);
               END
```

```
            END
         ELSE
            IF ((type of stateD2<>orState) AND
               (type of stateD2<>andState))THEN
               auxstateD := stateD1
            ELSE
               b := TRUE;
               i := 1;
               WHILE (i<size(substateList2)) DO
                  IF (lookForStateWithName(substateList[i].name,
                     stateD1)<>NIL) THEN
                     i := size(substateList2);
                     b := FALSE;
                     END
                  ELSE i := i + 1
                  END;
               IF (b) THEN
                  auxstateD := stateD1;
                  lsb := auxstateD.{substate};
                  lsb[size(lsb)+1] := stateD2;
                  auxstateD.{substate} := lsb
                  END
               ELSE output("Error in integration");
               END
      END;
      st := lookForStateWithName(auxstateD.name, stateD);
      IF (st<>NIL) THEN
         st := auxstateD
END andMerging;
```

PROCEDURE incorporationOfCompositionVariables(Var stateD1, stateD2: StateD)

(* ajoute les variables de composition dans les deux StateDs dans le cas où elles n'existent pas. Ceci consiste à ajouter pour chaque StateD les trois variables de composition *scenarioList*, *dynamicScenarioList* et *transScenarioList* dans la liste des variables d'états. Puis ajouter une condition spéciale *sc* idans chaque transition. Une action spéciale *sa* est aussi ajoutée dans toutes les transitions sauf pour celles qui terminent un scénarii où une acrion de ré-initialisation est introduite.*)

PROCEDURE getTransScenarioVariable(trans: Transition; stateD: StateD): SET OF string

(* retourne l'élément de la variable d'état *TransScenarioVariable* de stateD correspondant à la transition trans. *)

PROCEDURE mergeTwoSetsOfString(s1, s2: SET OF string): SET OF string

(* fusionne deux ensembles de chaînes en une seule. *)

PROCEDURE changeTransScenarioVariable(pos: integer; s: SET OF string; VAR stateD: StateD)

```
(* change la variable d'état TransScenarioVariable de stateD à la position pos
avec la chaîne s. *)
```

PROCEDURE integrationOfStateVariables(stateD1, stateD2: StateD; VAR stateD: StateD)

```
(* fusionne les listes de variables d'états des deux StateDs. *)
```

PROCEDURE lookForStateVariable(stv: StateVariable; stateD: StateD): BOOLEAN

```
(* retourne true si stateD a une variable d'état qui s'appelle stv. Sinon elle
retourne false. *)
```

PROCEDURE getListOfStartStates(stateD: StateD): ARRAY OF string

```
(* retourne les états initiaux de stateD. *)
```

PROCEDURE positionOfElementInStartStateList(name: string; startStateList: ARRAY OF string): integer

```
(* retourne la position de name dans startStateList. Si name n'existe pas, la
procédure retourne -1. *)
```

PROCEDURE insertElementInPath(nt: NameAndType; pos: integer; VAR path: ARRAY OF NameAndType)

```
(* insére nt dans path à la position pos. *)
```

PROCEDURE equalsPaths(path1, path2: ARRAY OF NameAndType): BOOLEAN

```
(* retourne true si path1 et path2 sont les mêmes. Sinon elle retourne false.
*)
```

PROCEDURE disjunctionOf(cd1, cd2: Condition): Condition

```
(* retourne une condition cd qui fait la disjonction de deux conditions cd1 et
cd2. cd est dans une forme canonique disjunctive. *)
```

Annexe G: Les schémas de micro-raffinement proposés

Dans cet annexe, nous allons présenter les cinq schémas de micro-raffinement les plus utilisés par les schémas du raffinement que nous avons définis. L'objectif ici est de fournir un ensemble de petits raffinements valides qui offre un grand potentiel de réutilisation dans une description des schémas de raffinement des patrons de conception autres que ceux étudiés dans le cadre de cette thèse.

G.1 Abstraction

Description

Ce schéma de micro-raffinement consiste à créer une classe abstraite pour une classe (voir Figure G.1). Cette transformation reste valide puisque le morphisme σ est *l'identité*.

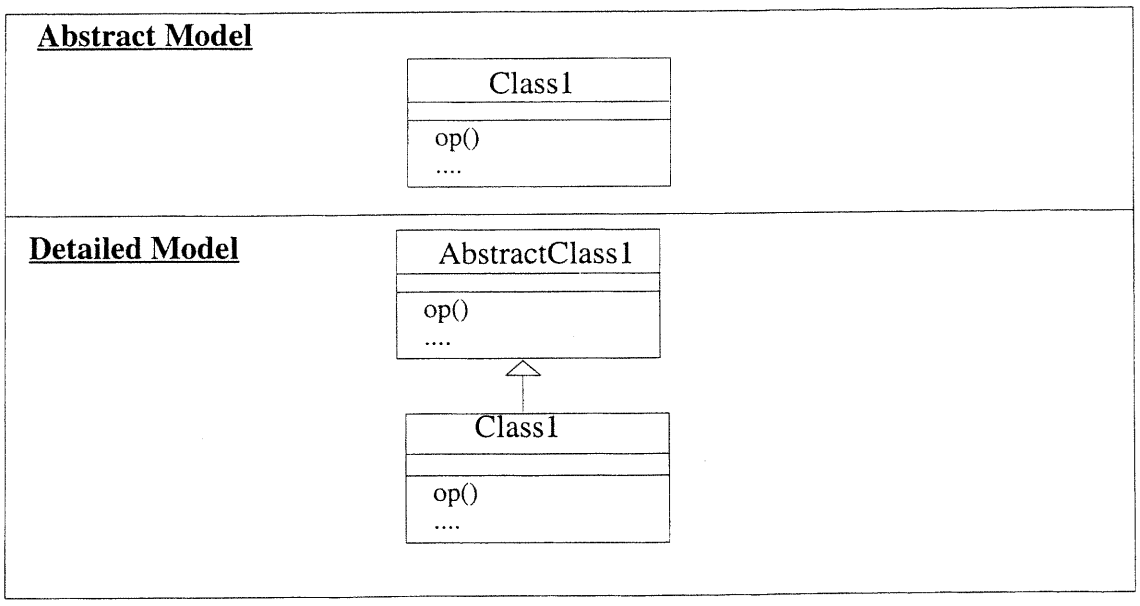


Figure G.1: Abstraction

Pseudo-code

PROCEDURE abstraction(className, abstractclassName: string; VAR classD: ClassD) =

```
VAR   class, abstractclass: Class;
BEGIN
   class := lookForClass(className, classD);
   IF not hasSuperClass(className, classD) THEN
      abstractclass := createClass(classD);
      abstractclass.packageName := class.packageName;
      abstractclass.className := abstractclassName;
      abstractclass.{attribute} := class.{attribute};
      abstractclass.{operation} := class.{operation};
      END
   END abstraction;
```

PROCEDURE lookForClass(className: string; classD: ClassD): class =

```
   (* retourne la classe qui a pour nom className si elle existe. Dans le cas
contraire elle retourne nil. *)
```

PROCEDURE hasSuperClass(className: string; classD: ClassD): boolean =

```
   (* retourne true si className possède une super-classe dans classD. Sinon
retourne false. *)
```

PROCEDURE createClass(VAR classD: ClassD): Class =

```
   (* crée une nouvelle classe dans classD. *)
```

G.2 Ajout d'un comportement concurrent dans un StateD

Description

Ce schéma de micro-raffinement ajoute un comportement concurrent `S2` à un StateD `S1` d'une classe `Class1` (voir figure G.2). Cette transformation revient à créer un état composite de type ET `S0` qui contient deux sous-états concurrents `S1` et `S2`. Toutes les transitions du StateD, telle que la transition `t`, reste inchangées exceptées les deux transitions t_i et t_k qui respectivement initialise et détruit un objet de la classe `Class1`. Ces transitions sont mises-à-jour tel que décrit dans la figure G.2.

Le morhpisme σ qui traduit le modèle abstrait en modèle détaillé ajoute de nouvelles axiomes de `S2`, change chaque sous-état `S` de `Class1` comme suit: $ext(S) \longrightarrow ext(S) \cap ext(S2)$, et change les axiomes liés aux transitions t_i et t_k. Tous les autres axiomes du modèle abstrait reste valide dans le modèle détaillé.

Pour les axiomes des transitions t_i et t_k dans le modèle abstrait, nous avons:

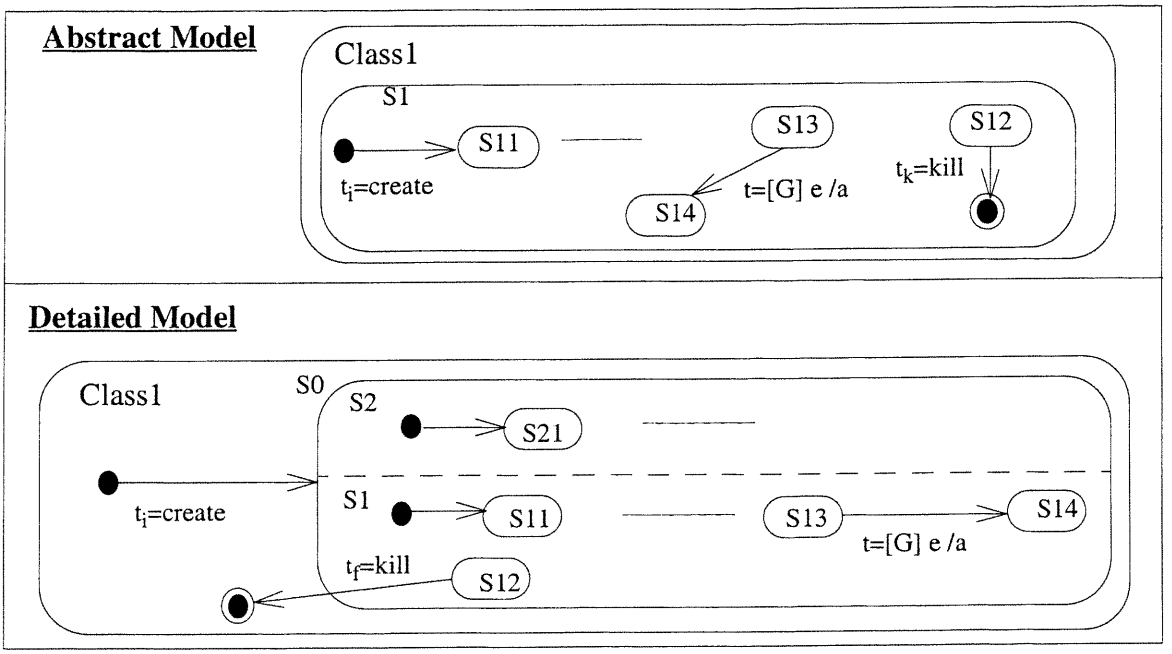


Figure G.2: Ajout d'un comportement concurrent pour un StateD d'une classe

$t_i \supset create \wedge \forall c: Class1. [c!t_i](c \in ext(S11)) \wedge (c!create \supset c!t_i)$

$t_k \supset kill \wedge \forall c: Class1. (c \in ext(S12)) \Rightarrow [c!t_k](c \notin ext(Class1)$

Ces axiomes sous σ deviennent:

$t_i \supset create \wedge \forall c: Class1. [c!t_i](c \in ext(S11) \cap ext(S2)) \wedge (c!create \supset c!t_i)$

$t_k \supset kill \wedge \forall c: Class1. (c \in ext(S12) \cap ext(S2)) \Rightarrow [c!t_k](c \notin ext(Class1))$

qu'on peut prouver facilement à partir des axiomes du modèle détaillé puisque les transitions t_i et t_k après raffinement ont comme axiomes ($ext(S21) \subseteq ext(S2)$ car S21 est un sous-état de S2):

$t_i \supset create \wedge \forall c: Class1. [c!t_i](c \in ext(S11) \cap ext(S21)) \wedge (c!create \supset c!t_i)$

$t_k \supset kill \wedge \forall c: Class1. (c \in ext(S12) \cap ext(S2)) \Rightarrow [c!t_k](c \notin ext(Class1))$

Pseudo-code

PROCEDURE addConcurrentBehaviour(S2, S1: StateD; S0Name: string): StateD =

```
VAR   stateD, S0, si: StateD;
      t: Transition;
      ti, transList: ARRAY of Transition;
      substateList1, substateList2, sf := ARRAY of StateD;
      i: integer;
BEGIN
   stateD := createOrState();
   stateD.name := S1.name;
   S0 := createAndState();
   S0.name := S0Name;
   si := createInitialState();
   substateList1[1] := si;
   substateList1[2] := S0;
   substateList2[1] := S2;
   substateList2[2] := S1;
   S0.{substate} := substateList2;
   ti := lookForInitialTransitions(S1);
   FOR i:=1 TO size(st) DO
      t := ti[i];
      t.fromNode := si;
      t.toNode := S0;
      transList[i] := t;
      ti[i].event := not specified;
      END;
   sf := lookForFinalStates(S1);
   FOR i:=1 TO size(st) DO
      substateList1[i+2] := sf[i];
      removeStateFrom(sf[i], S1.{substate});
      END;
   stateD.{substate} := substateList1;
   END addConcurrentBahaviour;
```

PROCEDURE lookForInitialTransitions(stateD: StateD): ARRAY of Transition =

```
   (* retourne les transitions qui ont comme fromNode l'état initial de stateD.
*)
```

PROCEDURE lookForFinalStates(stateD: StateD): ARRAY of StateD =

```
   (* retourne les états finaux de stateD. *)
```

G.3 Héritage

Description

Ce schéma de micro-raffinement ajoute une nouvelle classe `Class2` qui devient une super-classe

d'une classe `Class1` (voir figure G.3). Ce raffinement peut être effectué à la condition qu'il n'y a pas de problèmes entre les éléments de Class1 avec ceux de Class2. Par exemple, il ne doit pas y avoir un même attribut avec deux types différents dans les deux classes. Cette transformation utilise le schéma de micro-raffinement décrit dans la section précédente dans le but de mettre à jour le nouveau comportement de `Class1` par son extension pour tenir compte du nouveau comportement apporté par sa super-classe `Class2`.

Le modèle détaillé de `Class1` contient de nouvelles axiomes prises de `Class2` plus les axiomes de son modèle abstrait excepté ceux de son StateD qui subissent quelques changements tel que décrit dans la section G.2 Par conséquent, ce schéma de micro-raffinement est valide d'après la démonstration faite sur la validité du schéma de micro-raffinement de la section G.2.

Pseudo-code

PROCEDURE inheritance(subclassName: string; superclass: Class; VAR classD: ClassD) =

```
VAR   subclass: Class;
      classList: ARRAY of Class;
      generaLisation: Generalisation;
BEGIN
   classList := classD.{class};
   classList[size(class)+1] := superClass;
   subclass := lookForClass(subclassName, classD);
   superclass.packageName := subclass.packageName;
   generalisation := createGeneralisation(classD);
   generalisation.superClass := superclass;
   generalisation.subClass := subclass;
   addConcurrentBehaviour(stateD(superclass), stateD(subclass),
                          subclass.className+"AND"+superclass.className);
   END inheritance;
```

PROCEDURE createGeneralisation(VAR classD: ClassD): Generalisation =

```
   (* crée une généralisation dans classD. *)
```

G.4 Ajout d'une action dans une transition entre deux états

Description

Ce schéma de micro-raffinement consiste à ajouter une action dans une transition entre deux états `S1` et `S2` dans un StateD comme nous pouvons voir dans la figure G.4. Le morphisme σ du modèle abstrait vers le modèle détaillé est: $t_1 \longrightarrow t_2$. Les axiomes de t_1 dans le modèle abstrait est (1), sous σ est (2) et dans le modèle détaillé est (3):

(1) $t_1 \supset e \wedge \forall c: S1. ((c \in ext(S1) \wedge c.G) \Rightarrow [c!t_1] (c \in ext(S2))) \wedge$

$((c \in ext(S13) \wedge c.G) \Rightarrow (c!e \supset c!t_1) \wedge (c!t_1 \Rightarrow \bigcirc \Diamond c!a_1)$

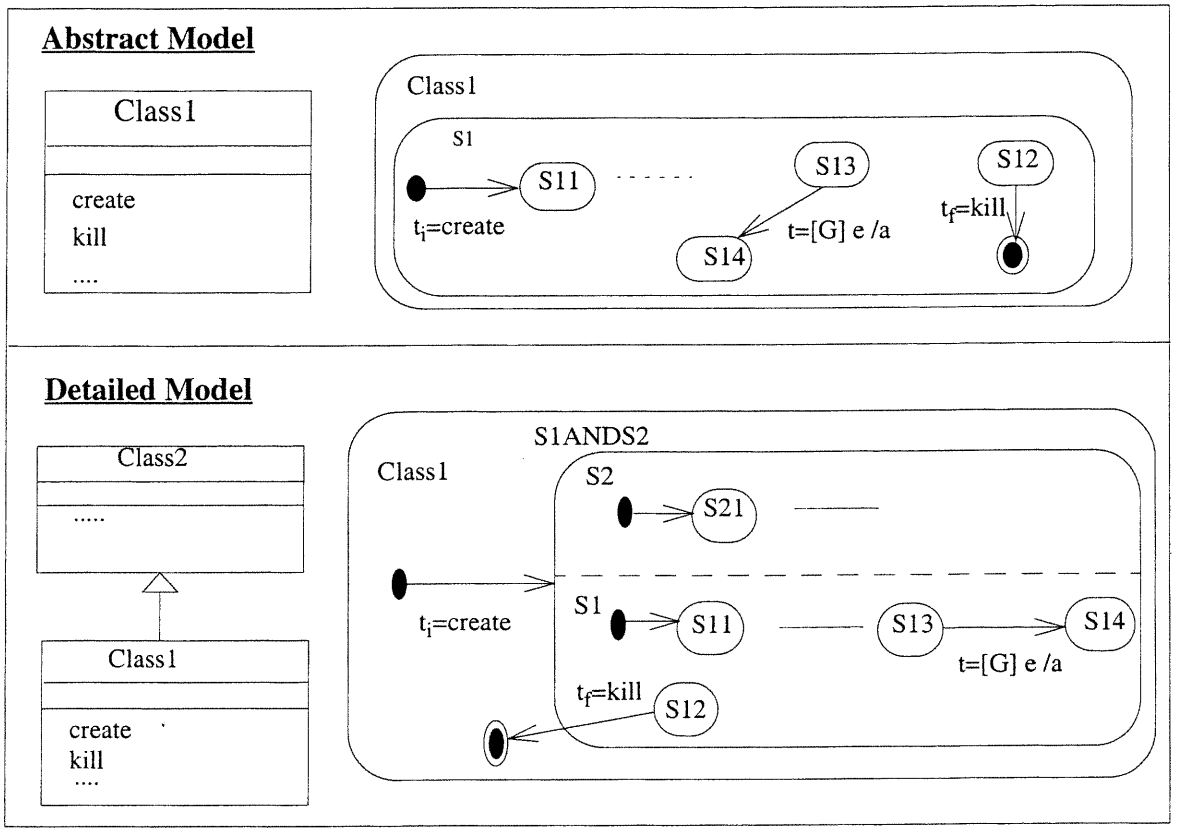


Figure G.3: Héritage

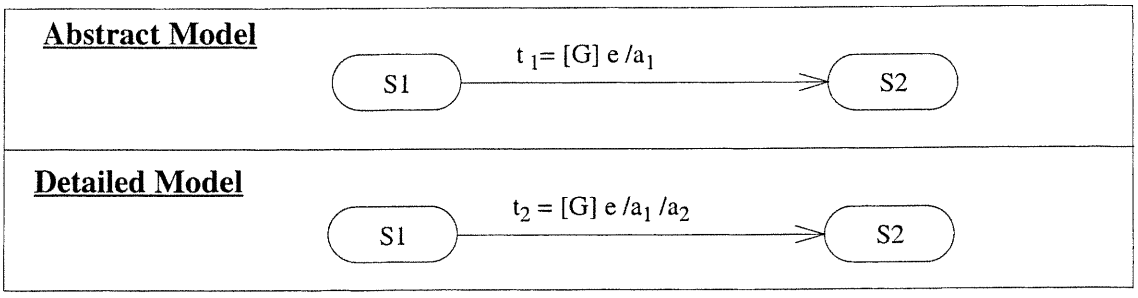


Figure G.4: Ajout d'une action dans une transition entre deux états

(2) $t_2 \supset e \wedge \forall c: S1. ((c \in ext(S1) \wedge c.G) \Rightarrow [c!t_2] (c \in ext(S2))) \wedge$

$((c \in ext(S13) \wedge c.G) \Rightarrow (c!e \supset c!t_2)) \wedge (c!t_2 \Rightarrow \bigcirc \Diamond c!a_1)$

(3) $t_2 \supset e \wedge \forall c: S1. ((c \in ext(S1) \wedge c.G) \Rightarrow [c!t_2] (c \in ext(S2))) \wedge$

$$((c \in ext(S13) \wedge c.G) \Rightarrow (c!e \supset c!t_2)) \wedge (c!t_2 \Rightarrow \bigcirc \Diamond (c!a_1; c!a_2))$$

L'axiome (2) peut se démontrer à partir de (3) puisque:

$$(c!t2 \Rightarrow \bigcirc \Diamond (c!a1; c!a2)) \Rightarrow (c!t2 \Rightarrow \bigcirc \Diamond (c!a1))$$

Pseudo-code

PROCEDURE addActionInATransition(action: ActionORSendClause; type: string; VAR trans:Transition) =

```
VAR    actionList: ARRAY of Action;
       sendClauseList: ARRAY of SendClause;
       i: integer;
BEGIN
   IF type ="action" THEN
      actionList := trans.{action};
      actionList[size(actionList)+1] := action.action;
      END
   ELSE
      sendClauseList := trans.{sendClause};
      sendClauseList[size(sendClauseList)+1] := action.sendClause;
      END
   END addActionInATransition;
```

G.5 Indirection

Description

Dans un ClassD, ce schéma de micro-raffinement introduit une classe intermédiaire `Class3` à la place d'une association unidirectionnelle entre deux classes `Class1` et `Class2` (voir Figure G.5). Pour chaque CollD concerné par cette association, comme celui qui spécifie l'opération `op1()`, doit être mis à jour. Ceci consiste à supprimer le lien qui est une instance de l'association, à ajouter un objet de la classe `Class3`, et à créer deux nouveaux liens.

Le premier lien relie l'objet de la classe `Class1` et celui de la classe `Class3` et va contenir tous les messages du lien supprimé. Le second lien relie l'objet de la classe `Class3` et celui de la classe `Class2` et va comprendre un nouveau sous-message, avec même nom et arguments, pour chaque message du premier lien. Par exemple, le message avec le numéro de séquencement `1` va avoir un sous-message avec le numéro de séquencement `1.1`. Notons que tous les sous-messages de chaque message du premier lien (comme pour le cas du message avec le numéro de séquencement `1`) deviennent des sous-messages du sous-message nouvellement créé.

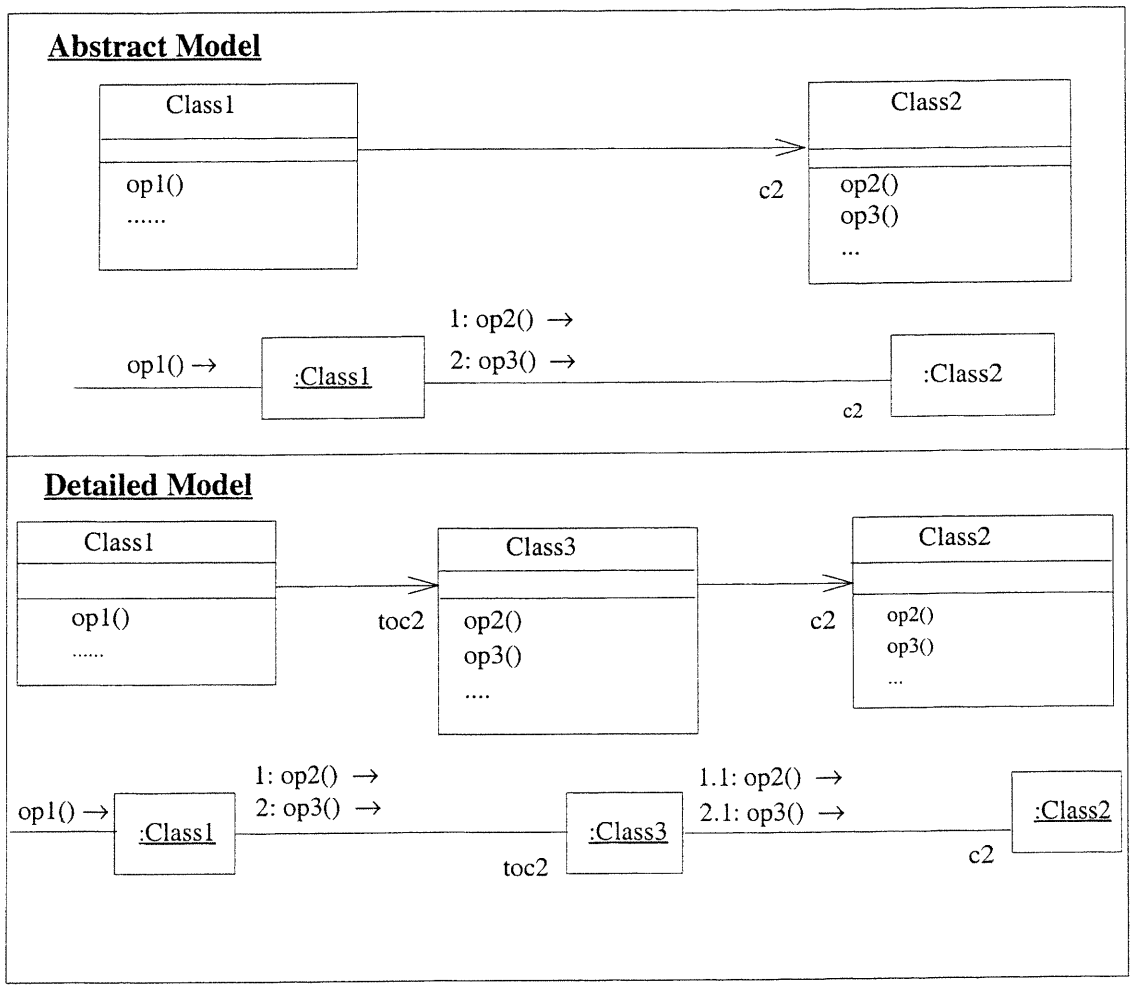


Figure G.5: Indirection

Le morphisme σ du modèle abstrait au modèle détaillé *l'identité* pour tous les symboles sauf pour `c2` qui est traduit à `toc2.c2` Excepté les axiomes des opérations mises-à-jour comme celle de `op1()`, tous les autres axiomes du modèle abstrait reste inchangé dans le modèle détaillé qui contient en plus de nouveaux axiomes de la classe `Class3`.

La preuve de validité des axiomes des opérations comme ceux de op1() est basé sur le fait que ⊃ est transitive (voir l'axiome (1) ci-dessous) et que les constructeurs comme la séquentialité ou les actions conditionnelles sont monotones en respect de l'opérateur ⊃ [Lano, 1998] (voir l'axiome (2) avec la séquentialité ci-dessous).

(1) ⊃ est transitive ⇔ ∀ a_1, a_2, a_3. (a_1 ⊃ a_2) ∧ (a_2 ⊃ a_3) ⇒(a_1 ⊃ a_3)

(2) la séquentialité est monotone avec $\supset \Leftrightarrow \forall a_1, a_2, b_1, b_2. (a_1 \supset a_2) \wedge (b_1 \supset b_2) \Rightarrow (a_1; b_1 \supset a_2; b_2)$

Dans le CollD de l'opération `op1()`, nous avons dans le modèle abstrait:

(3) $\forall$ c: Class1. c!op1() $\supset$ (c.c2!op2(); c.c2!op3())

(3) sous σ est (4) et dans le modèle détaillé, `op1()` a comme axiome (5).

(3) $\forall$ c: Class1. c!op1() $\supset$ (c.toc2.c2!op2(); c.toc2.c2!op3())

(4) $\forall$ c: Class1. c!op1() $\supset$ (c.toc2!op2(); c.toc2!op3())

Nous avons c.toc2 $\in$ Class3 donc:

(5) (c.toc2!op2() $\supset$ c.toc2.c2!op2()) $\wedge$ (c.toc2!op2() $\supset$ c.toc2.c2!op3())

Finalement, nous obtenons:

(4) $\wedge$ (5) $\wedge$ (2) $\wedge$ (1) $\Rightarrow$ (3)

Pseudo-code

PROCEDURE indirection(assoc: Association; class3Name: String; VAR system: System) =

```
VAR   collDList: ARRAY of CollD;
      link: Link
      i: integer;
BEGIN
   IF size(assoc.{associationEnd)=2) THEN
      classDIndirection(assoc, class3Name, system.classD);
      collDList := system.{collD};
      FOR i:=1 TO size(collDList) DO
         link := lookForCorrrespondingLink(assoc, collDList[i]);
         IF link <> NIL THEN
            collDIndirection(link, class3Name, collDList[i]);
         END
   END
   END indirection;
```

PROCEDURE classDIndirection(assoc: Association; class3Name: String; VAR classD: ClassD) =

```
VAR   assocEndList, assocEndList2: ARRAY of AssociationEnd;
      class1, class2, class3: Class;
      assoc3: Association;
BEGIN
```

```
   assocEndList := assoc.{associationEnd};
   class3 := createClass(classD);
   class3.className := class3Name;
   class1 := lookForClass(assocEndList[1]);
   class2 := lookForClass(assocEndList[2]);
   class3.{operation} := class2.{operation};
   assoc2 := createAssociation(classD);
   assocEndList2[2] := assocEndList[2];
   assocEndList[2].linkedClass := <ref. to class3>;
   IF (assocEndList[2].roleName is specified) THEN
      assocEndList[2].roleName := "to" + assocEndList[2].roleName;
   assocEndList2[1].linkedClass := <ref. to class3>;
   assocEndList2[1].navigability := false;
   assoc2.{associationEnd} := assocEndList2;
   END classDIndirection;
```

PROCEDURE collDIndirection(link: Link; class3Name: String; VAR collD: CollD) =

```
VAR   assocEndList, assocEndList2: ARRAY of AssociationEnd;
      object2, object3: Object;
      link3: Link;
      linkList: ARRAY of Link;
      messageList: ARRAY of Message;
BEGIN
   object3 := createObject(collD);
   object.className := class3Name;
   link3 := link;
   link.linkedObject := <ref. to object3>;
   linkList[1] := link3;
   object3.{link} := linkList;
   IF (link.role.roleName is specified) THEN
         link3.role.roleName := "to" + link.role.roleName;
   messageList := link3.{message};
   FOR i:=1 TO size(messageList) DO
      messageList[i].sequenceExpression.sequenceNumber :=
      messageList[i].sequenceExpression.sequenceNumber + ".1";
      messageList[i].sequenceExpression.recurrence := not specified;
      END
   END collDIndirection;
```

PROCEDURE lookForCorrespondingLink(assoc: Association; collD: CollD): Link =

```
   (* retourne un lien de collD correspondant à assoc.S'il n'exsite pas
retourne NIL. *)
```

PROCEDURE createAssociation(VAR classD: ClassD): Association =

```
   (* crée une association dans classD. *)
```

PROCEDURE createObject(VAR collD: collD): Object =

```
   (* crée un objet dans collD. *)
```

Annexe H: Description des schémas de raffinement pour Mediator et Observer

Dans la première section de cet annexe, nous allons présenter le schéma de raffinement que nous avons défini pour le patron Mediator. Dans la deuxième section, nous allons donner le pseudo-code du schéma de raffinement du patron Observer. Notons que ce schéma est décrit dans la section 6.1.

H.1 Le schéma de raffinement de Mediator

H.1.1 Exemple d'illustration

Comme exemple d'illustration, nous utilisons une partie de la conception d'un système électronique d'archivage [Derr, 1996]. L'objectif du système est la gestion des documents. La figure H.1 montre quatre classes du ClassD du système: `VecteurDeRecherche`, `RépertoireArchivage`, `Requête` et `ListeDocuments`. La classe `VecteurDeRecherche` est utilisée durant la recherche de documents. La classe `RépertoireArchivage` gère tous les fichiers de documents par la conservation dans une table de correspondance entre les noms de fichiers avec leurs indices. La classe `Requête` fournit des méthodes pour la gestion de requêtes. Finalement, la classe `ListeDocuments` est une liste de noms de documents qui correspond au résultat d'une recherche. La figure H.2 décrit un CollD pour l'opération `chercheDocument`.

Après l'analyse du CollD de l'opération checheDocument, nous trouvons qu'il est plus intéressant de créer un objet qui devient responsable de l'opération de recherche. Le patron Mediator (cf. chapitre 2) offre une bonne solution à ce problème. En effet, il définit un objet qui coordonne les interactions entre un ensemble d'objets. Nous allons voir par la suite comment ce patron peut être appliqué à cet exemple.

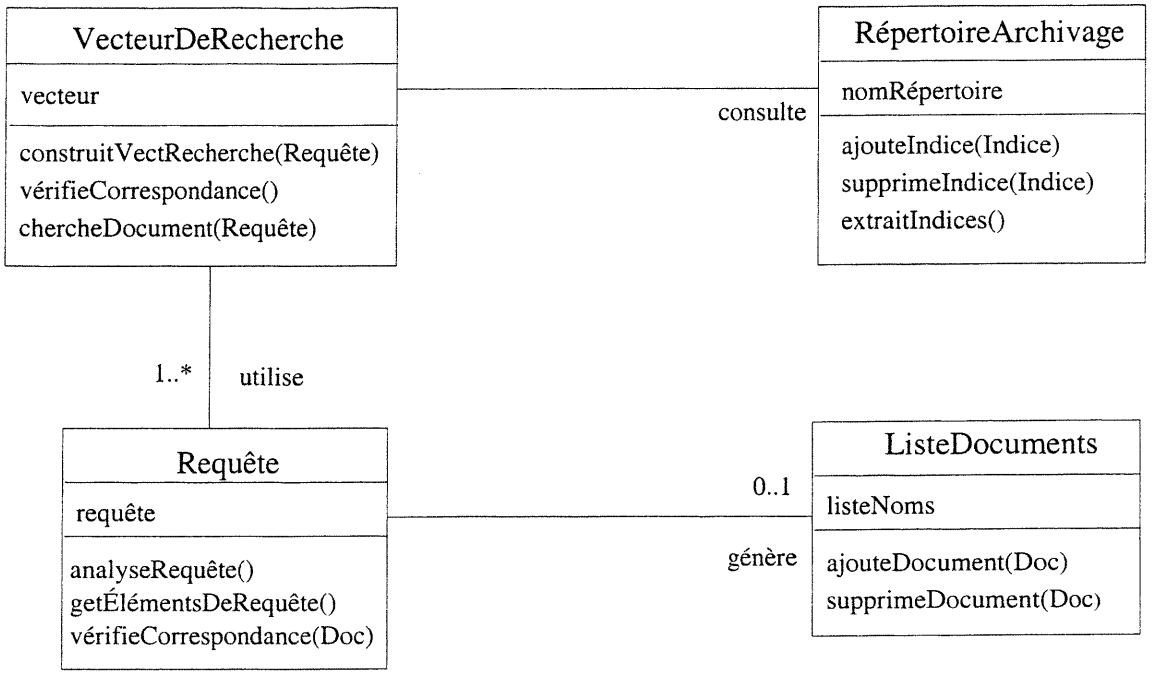


Figure H.1: Partie du ClassD du système électronique d'archivage

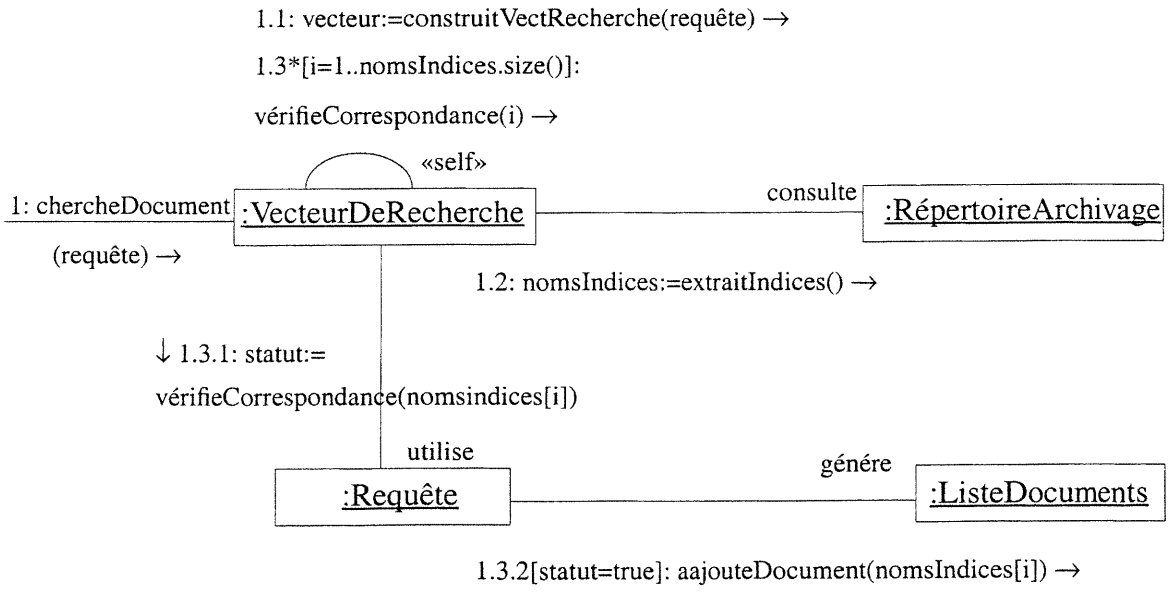


Figure H.2: CollD de l'opération `chercheDocument`

H.1.2 Description du schéma de raffinement de Mediator

La figure H.3 montre le schéma de raffinement que nous avons défini pour le patron Mediator. Ce schéma possède trois paramètres (voir pseudo-code dans la section G.4): un ensemble d'association unidirectionnelles entre classes (pour des questions de clarté, le schéma est décrit par un exemple de six association entre quatre classes), le nom à donner à la classe Mediator, et le modèle du système.

Le schéma de raffinement de Mediator met à jour le ClassD et tous les CollD, tel que celui de l'opération `operation()`, concernés par une de ces associations. Remarquons que l'algorithme correspondant à ce schéma peut traiter n'importe quel CollD (cf. section H.4). Pour l'exemple du système électronique d'archivage, les figures H.4 et H.5 montrent comment le modèle de conception du système est transformé après l'application du schéma de raffinement de Mediator. Cette transformation introduit un objet `EnginDeRecherche` dans le but de centraliser le flot de contrôle de l'opération `chercheDocument`.

Il faut noter que les nouveaux StateDs des objets collaborant dans le CollD affecté ainsi que le StateD du nouvel objet `EnginDeRecherche` peuvent être automatiquement obtenus par notre algorithme de synthèse décrit dans le chapitre 5.

H.1.3 Schémas de micro-raffinement utilisés par Mediator

Ce schéma de raffinement est composé d'une séquence de cinq petits raffinements. Le premier raffinement utilise le schéma de micro-raffinement *indirection* (cf. section G.5) dans le but d'introduire la classe `Mediator` à la place de chaque association unidirectionnelle du modèle abstrait (voir figure H.6). Chaque CollD concerné avec une de ces associations est mis-à-jour. Un exemple de ces CollDs est celui qui spécifie l'opération `operation()`. Le second raffinement (*abstraction*, cf. section G.2) crée une classe abstraite pour la classe Mediator et une classe abstraite unique pour tous les autres classes (voir figure H.7). Le troisième raffinement (*changement d'association*) modifie les associations entre les classes du modèle abstrait par une association qui relie leur classes abstraits respectifs (voir figure H.8).

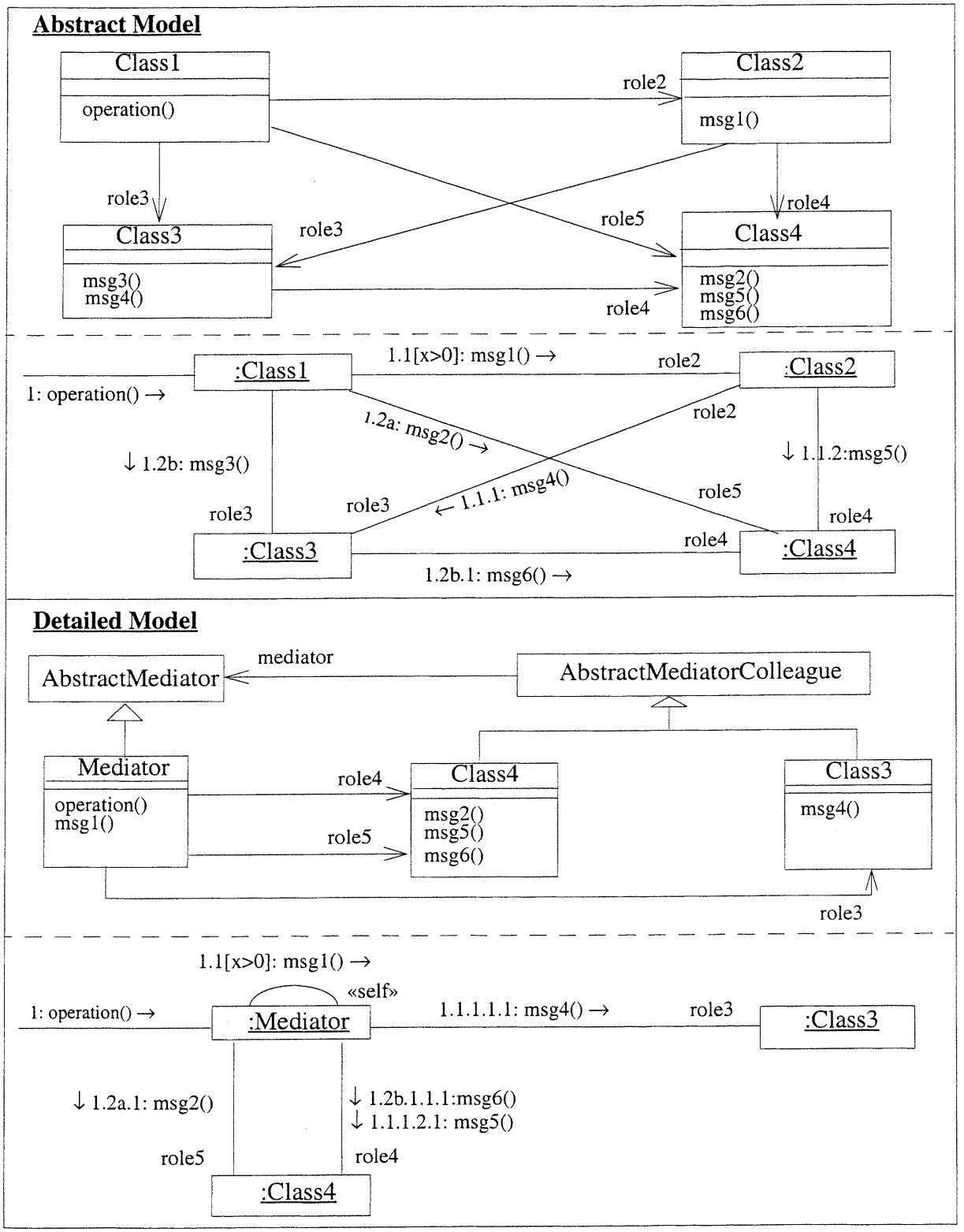


Figure H.3: Schéma de raffinement du patron Mediator

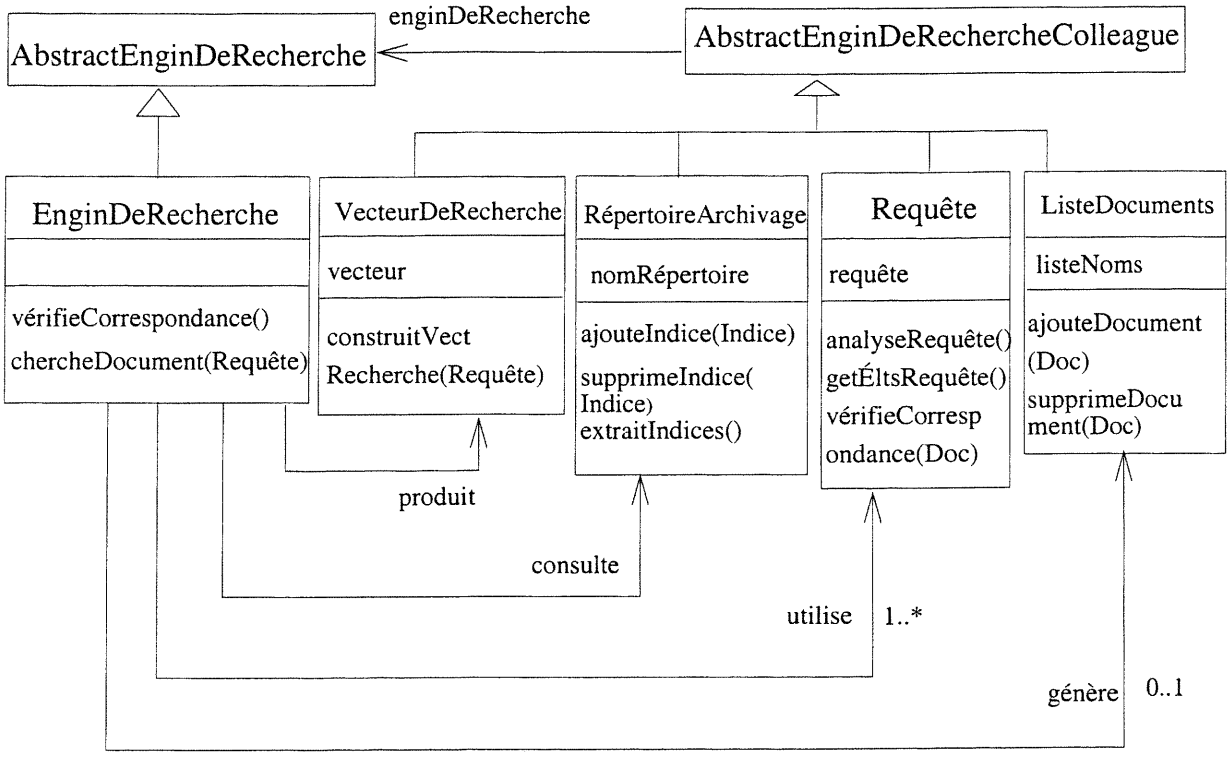


Figure H.4: ClassD après application du schéma de raffinement de Mediator

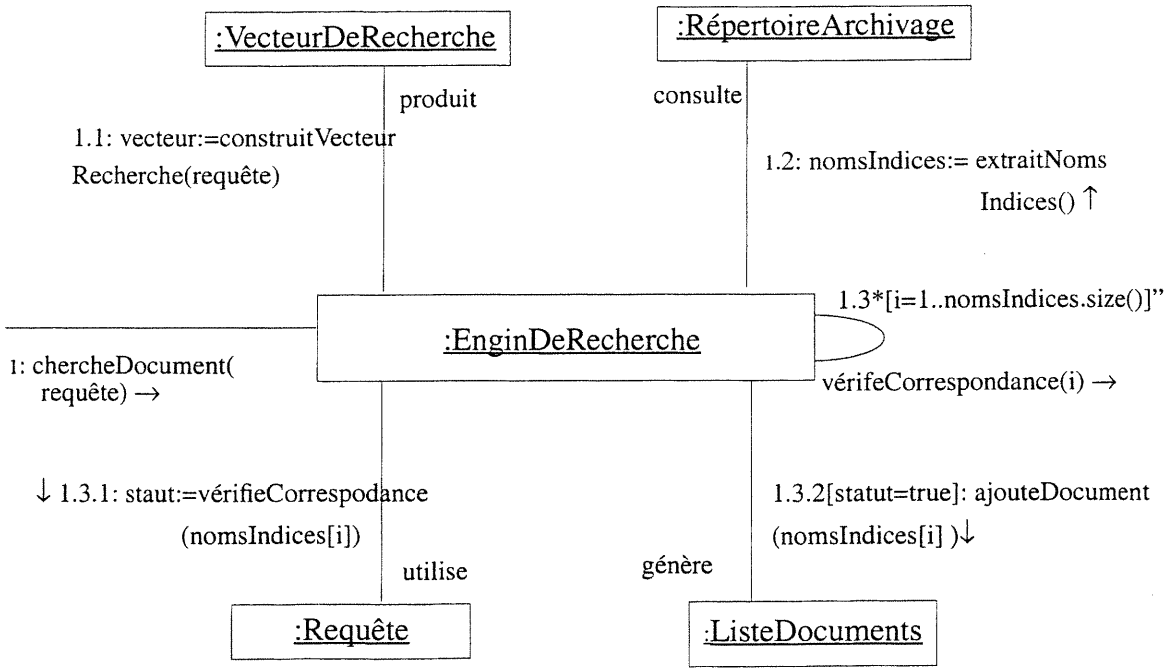


Figure H.5: CollD après application du schéma de raffinement de Mediator

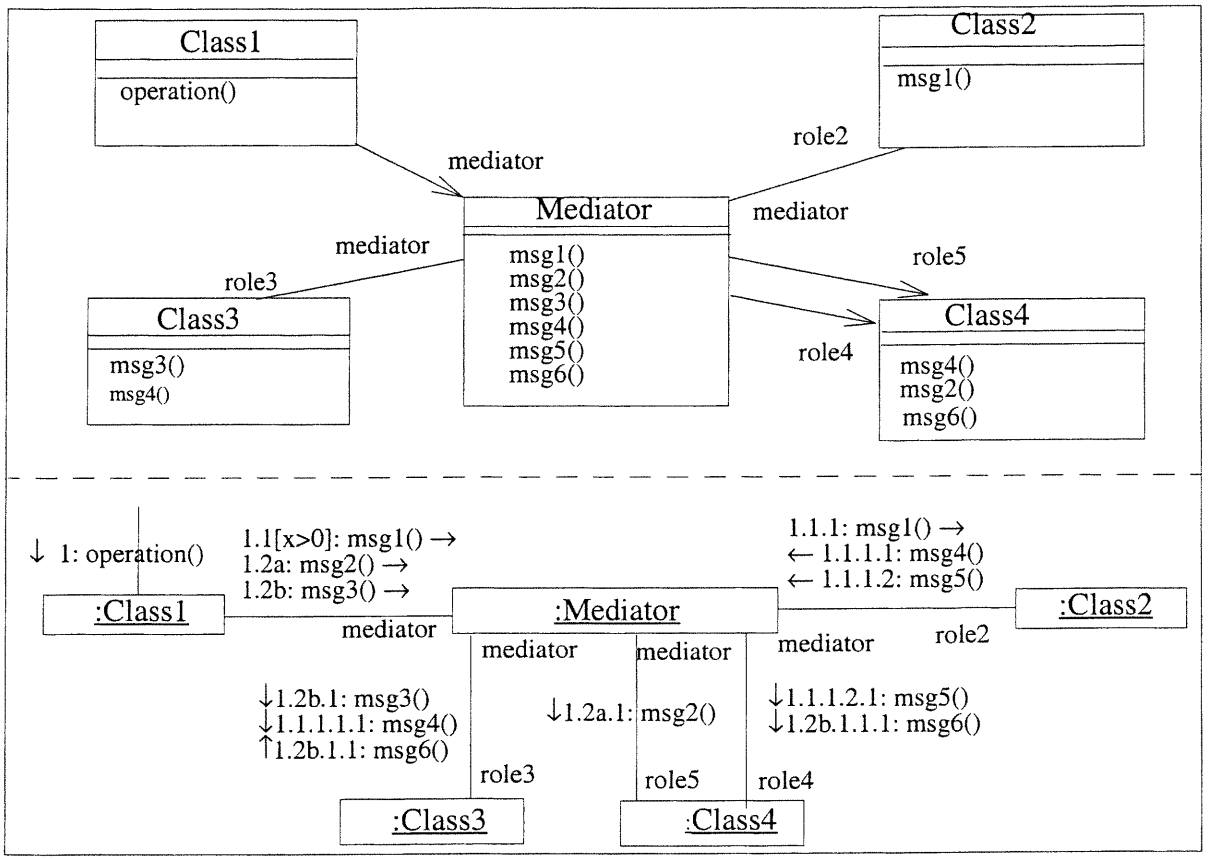


Figure H.6: Le résultat du premier schéma de micro-raffinement *indirection* utilisé par Mediator

Le quatrième raffinement (*centralisation du flot du contrôle*) centralise le flot de contrôle de chaque CollD sur l'objet `Mediator` comme dans le cas de l'opération `operation()` (voir figure H.9). Ce dernier objet devient le récepteur de tous les messages du CollD avec sous-messages. Il devient aussi l'émetteur de leurs sous-messages directs. Par exemple, l'objet `Mediator` est maintenant celui qui reçoit le message avec numéro de séquencement `1.2b.1` et initie le message avec le numéro de séquencement `1.2b.1.1`. Finalement, le cinquième raffinement (*suppression des interactions non nécessaires*) consiste à supprimer tous les interactions non nécessaires (voir le modèle détaillé de la figure G.3). Les interactions non nécessaires sont tous les messages qu'on peut s'en passer sans changer la fonctionnalité du CollD. Ces messages sont ceux qui ne sont ni conditionnels, ni itératifs, ni à prédécesseurs multiples tout en étant des messages que l'objet `Mediator` envoie à lui même et envoie leurs sous-messages respectifs. Le message `1.2b` est un exemple de message non nécessaire donc il est supprimé alors que le message `1.1` est nécessaire et par conséquent reste dans le modèle détaillé.

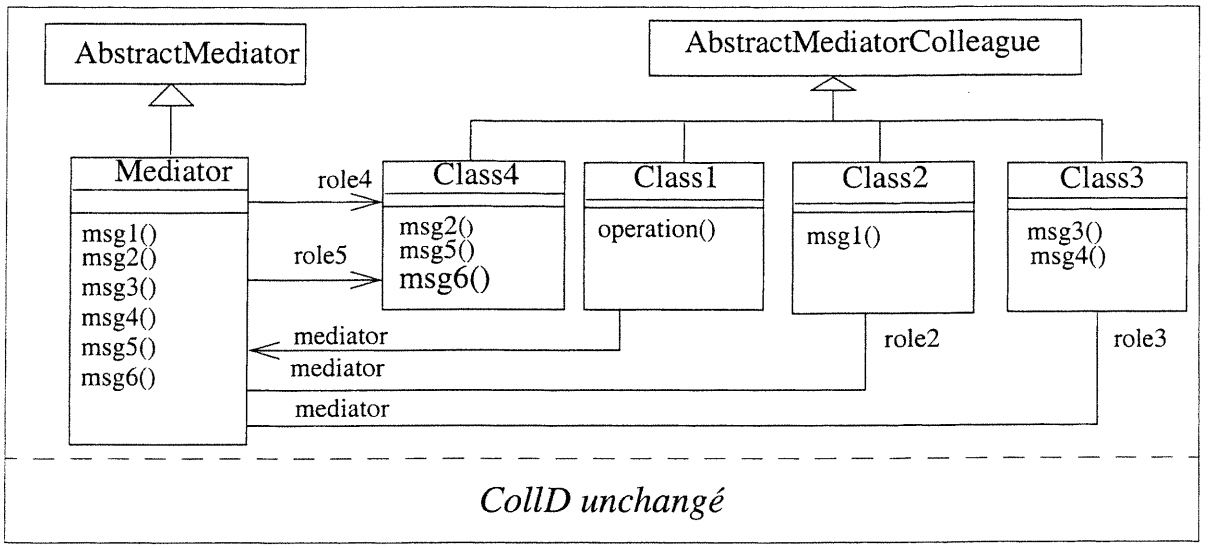


Figure H.7: Le résultat du deuxième schéma de micro-raffinement *abstraction* utilisé par Mediator

Le premier et le second raffinement sont valides car ils réutilisent les schémas de micro-raffinement indirection et abstraction. Il est évident que le troisième raffinement est valide puisque la nouvelle association entre les classes abstraites est héritée par leurs sous-classes respectives et par conséquent elle remplace les associations supprimées. Si nous ne tenons pas des objets qui émettent ou reçoivent les messages alors le quatrième raffinement ne change pas autre chose. C'est pourquoi nous pouvons dire que la fonctionnalité du CollD est conservée puisque la différence est que maintenant le flot de contrôle est centralisé par l'objet Mediator.

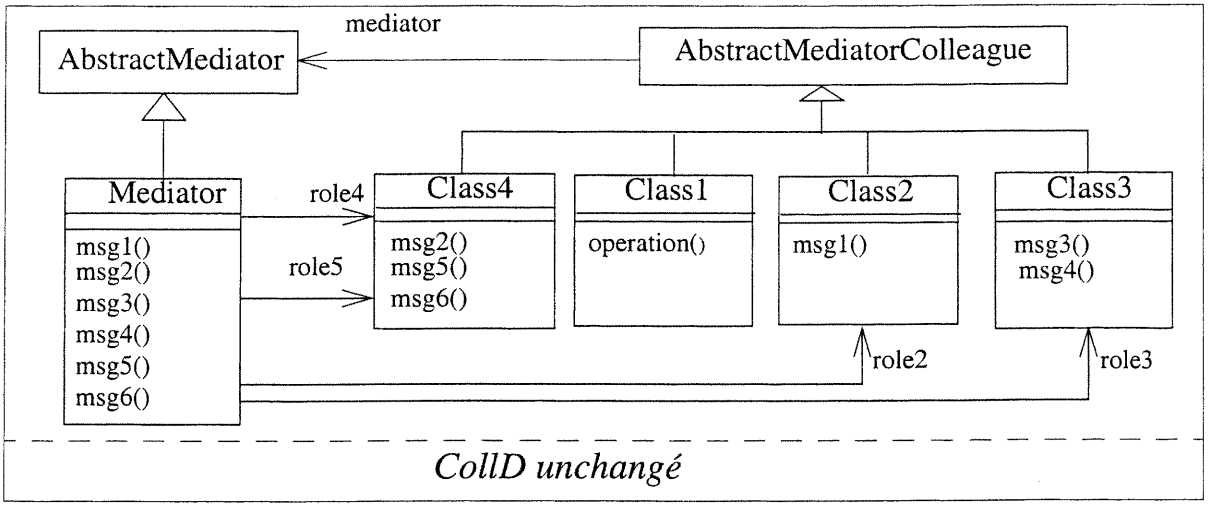


Figure H.8: Le résultat du troisième schéma de micro-raffinement *changement d'association* utilisé par Mediator

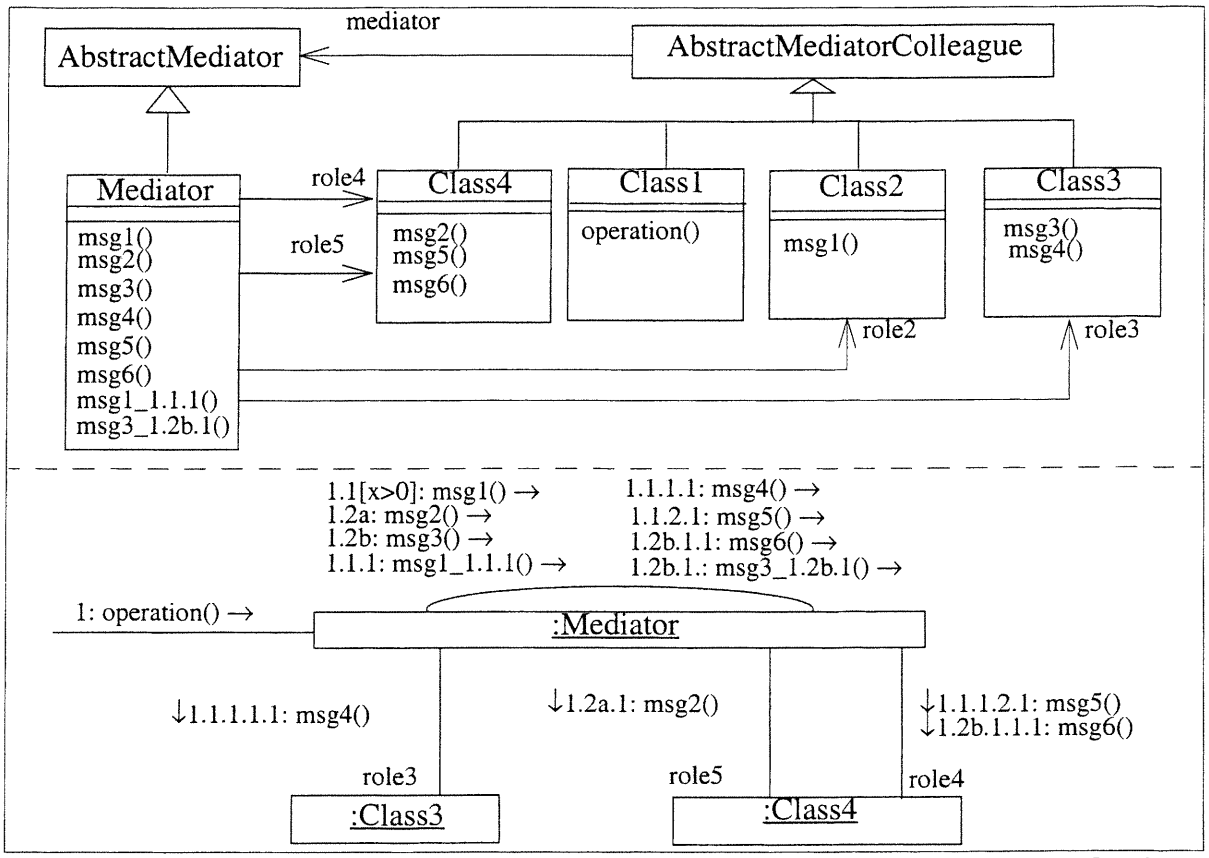


Figure H.9: Le résultat du quatrième schéma de micro-raffinement *centralisation du flot de contrôle* utilisé par Mediator

En fin, le cinquième raffinement est aussi valide puisque les messages supprimés n'ont aucune incidence sur la fonctionnalité du CollD.

H.1.4 Pseudo-code du schéma de raffinement de Mediator

PROCEDURE mediatorRefinementSchema(assocList: ARRAY of Association; mediatorName, : string; VAR system: System) =

```
VAR   classList: ARRAY of Class;
      mediator: Class;
      assoc: Association;
      i: integer;
BEGIN
   FOR i:=1 TO size(assocList) DO
      assocEndList := assocList[i].{associationEnd};
      addClassIn(lookForClass(assocEndList[1].linkedClass, system.classD),
                 classList);
      addClassIn(lookForClass(assocEndList[2].linkedClass, system.classD),
                 classList);
```

```
      indirection(assocList[i], mediatorName, system);
      END;
   mediator := lookForClass(mediatorName, system.classD);
   abstraction(mediatorName, "abstract" + mediatorName);
   FOR i:=1 TO size(classList) DO
      abstraction(classList[i].className, "abstract" + mediatorName
                  + "Colleague");
   FOR i:=1 TO size(classList) DO
      assoc := lookForAssociation(classList[i].className, mediatorName,
                                  system.classD);
      changeAssociation(assoc, "abstract" + mediatorName + "Colleague",
                        "abstract" + mediatorName);
      END;
   centralizingControlFlow(assocList, mediator, system);
   removingUncessaryInteractions(assocList, mediator, system);
   END mediatorRefinementSchema;
```

PROCEDURE centralizingControlFlow(assocList: ARRAY OF Association; mediator: Class; VAR system: System) =

```
VAR   collDList: ARRAY OF CollD;
      i: integer;
BEGIN
   collDList := lookForCollDs(assocList, system);
   FOR i:=1 TO size(collDList) DO
      centralizingControlFlowForOneCollD(assocList, mediator, collD[i]);
   END centralizingControlFlow;
```

PROCEDURE removingUncessaryInteractions(assocList: ARRAY OF Association; mediator: Class; VAR system: System) =

```
VAR   collDList: ARRAY OF CollD;
      i: integer;
BEGIN
   collDList := lookForCollDs(assocList, system);
   FOR i:=1 TO size(collDList) DO
      removingUncessaryInteractionsForOneCollD(mediator, collD[i]);
   END removingUncessaryInteractions;
```

PROCEDURE centralizingControlFlowForOneCollD(assocList: ARRAY OF Association; mediator: Class; VAR collD: CollD) =

```
VAR   mediatorObject: Object;
      newLink: Link
      linkList: ARRAY OF Link;
      msg: Message;
      messageList: ARRAY OF Message;
      i, j, k: integer;
BEGIN
   mediatorObject := createObject(mediator, collD);
   FOR i:=1 TO size(assocList) DO
      linkList := lookForLinks(assocList[i], collD);
      FOR j:=1 TO size(linkList) DO
         oldLinkedObject := linkList[j].linkedObject
```

```
            linkList[j].linkedObject := mediatorObject;
            messageList := linkList[j].{message};
            FOR k:=1 TO size(messageList) DO
               msg := messageList[k];
               msg.sequenceExpression.sequenceNumber :=
                  msg.sequenceExpression.sequenceNumber + ".1";
               newLink := createLink(mediatorObject, oldLinkedObject, collD);
               addMessage(msg, newLink);
               END;
            END;
         END;
      END centralizingControlFlowForOneCollD;
```

PROCEDURE removingUncessaryInteractionsForOneCollD(mediator: Class; VAR collD: CollD) =

```
VAR   mediatorObject: Object;
      linkList: ARRAY OF Link;
      messageList: ARRAY OF Message;
      i, j: integer;
BEGIN
   mediatorObject := lookForObjectInACollD(mediator, collD);
   linkList := mediatorObject.{link};
   FOR i:=1 TO size(linkList) DO
      messageList := linkList[i].{message};
      FOR j:=1 TO size(messageList) DO
         IF isAnUncessaryMessage(messageList[j], collD) THEN
            removeMessage(messageList[j], collD);
      END;
   END removingUncessaryInteractionsForOneCollD;
```

PROCEDURE lookForCollDs(assocList: ARRAY OF Association; system: System): ARRAY OF CollD=

```
   (* retourne tous les CollDs de system qui contient des liens correpondant à
une des associations de assocList. *)
```

PROCEDURE lookForObjectInACollD(class: Class; collD: CollD): Object =

```
   (* retourne un objet collaborant dans collD appartenant à la classe class.
*)
```

PROCEDURE isAnUncessaryMessage(msg: Message; collD: CollD): boolean =

```
   (* retourne true si msg est non nécessaire dans collD. Sinon retourne false.
Un message non nécessaire est un message non conditionnel, non itératif, ne
contenant pas de prédécesseur et que l'objet qui le reçoit envoie ses sous mes-
sages directs. *)
```

PROCEDURE lookForLinks(assoc: Association; collD: CollD): ARRAY OF Link=

```
   (* retourne tous les liens qui correspondent à assoc. *)
```

PROCEDURE createLink(obj1, obj2: Object; VAR collD: CollD): Link =

```
   (* retourne le lien entre obj1 et obj2 s'il existe dans collD. Sinon crée le
et ajouter le dans collD. *)
```

PROCEDURE createObject(class: Class; VAR collD: CollD): Object =

```
   (* crée un objet appartenant à class dans le collD. *)
```

PROCEDURE addMessage(msg: Message; link: Link) =

```
   (* ajoute msg dans link. *)
```

PROCEDURE removeMessage(msg: Message; VAR collD: CollD) =

```
   (* supprime msg dans collD. *)
```

H.2 Pseudo-code du schéma de raffinement de Observer

Dans cette section, nous présentons le pseudo-code correspondant au schéma du raffinement du patron Observer.

PROCEDURE observerRefinementSchema(assoc: Association; state1, state2: Attribute; getstate1, setstate1: Operation; subject, observer: Class; VAR classD: ClassD) =

```
VAR    class1, class2: Class;
       transi, transf: ARRAY of Transition;
       attachSendClause, detachSendClause: SendClause;
       assocEndList: ARRAY of AssociationEnd;
       action: ActionOrSendClause;
       parameterList: ARRAY of Parameter;
       i: integer;
BEGIN
   assocEndList := assoc.{associationEnd};
   IF size(assocEndList)=2 THEN
      IF assoc.constraint is specified and assocEndList[1].multiplicity="1"
         and assocEndList[1].multiplicity="0..*" THEN
         class1 := lookForClass(assocEndList[1]);
         class2 := lookForClass(assocEndList[2]);
         inheritance(class1.className, subject, classD);
         inheritance(class2.className, observer, classD);
         associationChangeDistributionInInheritanceHierarchy(assoc, subject,
                                                    observer, classD);
         transi := lookForInitialTransition(stateD(class2));
         attachSendClause := createSendClause();
         attachSendClause.target := subject;
         parameterList[1] := "this";
         attachSendClause.event := createEvent("attach", parameterList);
         action.sendClause := attachSendClause;
         FOR i:=1 TO size(transi) DO
            addActionInATransition(action, "sendClause", transi[i]);
         transf := lookForFinalTransitions(stateD(class2));
```

```
        detachSendClause := createSendClause();
        detachSendClause.target := subject;
        detachSendClause.event := createEvent("detach", parameterList);
        action.sendClause := detachSendClause;
        FOR i:=1 TO size(transf) DO
           addActionInATransition(action, "sendClause", transf[i]);
        automatedNotification(state1, state2, getstate1, setstate1,
                              class1, class2, classD);
     END
  END observerRefinementSchema;
```

PROCEDURE associationChangeDistributionInHierarchy(assoc: Association; subject, observer: Class; VAR classD: ClassD) =

```
VAR   assocEndList, assocEndList2: ARRAY of AssociationEnd;
BEGIN
   assocEndList := assoc.{associationEnd};
   assoc2 := createAssociation(classD);
   assocEndList2 := assocEndList;
   assocEndList[2].navigability := true;
   assocEndList2[2].navigability := true;
   assocEndList2[1].linkedClass := <ref. to Subject>;
   assocEndList2[2].linkedClass := <ref. to Observer>;
   END associationChangeDistributuionInHierarchy;
```

PROCEDURE automatedNotification(state1, state2: Attribute; getstate1: Operation; VAR setstate1: Operation; VAR class1, class2: Class; VAR classD: ClassD) =

```
VAR   note: attachNote;
      update: Operation;
      trans: Transition;
      parameterList: ARRAY of Parameter;
BEGIN
   attachNote := createAttachNote("add this line. notify()", setstate1,
                 classD);
   update := lookForOperation("update");
   attachNote := createAttachNote(state2+":="+subject+"."+getstate1", update,
                 classD);
   trans.event:= createEvent(getstate1, paramerList);
   addTransitionInAllStatesOf(trans, stateD(class1));
   END automatedNotfication;
```

PROCEDURE createEvent(eventName: EventName; parameterList: ARRAY of Parameter): Event =

```
   (* crée un événement ayant comme nom eventName et liste des paramètres para-
meterList. *)
```

PROCEDURE createAttachNode(note: Note; VAR modelElement: ModelElement; VAR classD: ClassD): AttachNote =

```
   (* crée un attachNote reliant modelElement avec note dans classD. *)
```

PROCEDURE addTransitionInAllStatesOf(trans: Transition; stateD: StateD) =

```
(* ajoute trans dans tous les états de stateD. *)
```