

Vers une génération de séquences de test fonctionnel des classes pour les applications Java : Une approche basée sur le comportement des objets

Mourad BADRI & Linda BADRI

Département de Mathématiques et d'Informatique
Université du Québec à Trois-Rivières
C.P. 500, Trois-Rivières, Québec, Canada
G9A 5H7

Téléphone : +1 (819) 376 5011 postes 3834 – 3819
Télécopie : +1 (819) 376 5185

Email : Mourad.Badri@uqtr.quebec.ca, Linda.Badri@uqtr.quebec.ca

Résumé :

Les langages de programmation orientée objet actuels, malgré leur utilisation de plus en plus importante durant cette dernière décennie, présentent toujours, à notre avis, une insuffisance en ce sens qu'ils ne permettent que l'implémentation directe du seul modèle objet. Ce modèle offre, en fait, une représentation statique des objets d'un système et ne prend pas en compte la description de leur comportement, en particulier, les aspects temporels dans le système, les séquences d'événements et de transitions, l'évolution de l'état des objets durant leur cycle de vie, ainsi que leur restriction comportementale liée à un de leurs états impliquant la restriction portant sur l'utilisation de leurs méthodes. C'est dans ce contexte, et dans un objectif d'assurance qualité, que nous proposons dans cet article, d'une part, une approche formelle de description et d'analyse du comportement des objets d'un système et, d'autre part, une stratégie globale pour le test fonctionnel orienté objet incluant, en particulier, une technique de génération automatique de séquences de test fonctionnel pour les classes, basée sur le comportement des objets, et son utilisation pour les applications Java. L'approche que nous proposons, pour le test fonctionnel des classes, prend en compte les aspects dynamiques d'un système à travers la description formelle du comportement de ses objets. Cette stratégie suit un processus de test incrémental et se base sur les diagrammes d'états des objets. Les séquences ainsi que les données de test sont, alors, construites pour parcourir les différents chemins fonctionnels, à travers les différentes transitions, des diagrammes d'états des objets du système.

Mots clés : Assurance Qualité Logiciel, Technologie Objet, Modèle Objet, Modèle Dynamique, État, Transition, Langage de Description, Test Fonctionnel, Séquence de Test, Classes d'Équivalence, Domaine de Valeurs.

1 Introduction :

Les concepts introduits par la technologie objet ont apporté, durant cette dernière décennie, plusieurs solutions à de nombreux problèmes. Cette approche est, en fait, remarquable par un de ces avantages qui est de permettre, en particulier, de maîtriser la complexité des systèmes logiciels, d'améliorer leur développement et de faciliter la réutilisation de leurs composants.

Les premières grandes phases du processus de développement orienté objet ont fait l'objet de nombreux travaux contrairement aux différentes étapes du processus de test. En effet, les méthodes d'analyse et de conception orientées objet [7, 8, 9, 11, 12, 16, 26, 27, 30] ont connu, durant cette dernière décennie, une grande évolution, pour la plupart d'entre elles, une convergence et, récemment, une unification [10, 17, 28], pour les plus importantes d'entre elles [8, 9, 16, 27].

La plupart des méthodes importantes de développement orienté objet se sont basées de plus en plus, durant leur évolution, sur l'élaboration d'un modèle dynamique en complément au modèle objet.

Cependant, les langages de programmation supportant le paradigme objet, malgré le fait qu'ils aient connu une grande évolution eux aussi, durant cette dernière décennie, et une utilisation, de plus en plus importante, présentent toujours, à notre avis, une insuffisance importante.

En effet, ils ne permettent que l'implémentation directe du seul modèle objet. Ce modèle, tel qu'il est proposé dans les méthodes de développement orienté objet actuelles, offre uniquement une représentation statique des objets d'un système. Il ne décrit le comportement des objets d'un système, à travers les méthodes de classes, que partiellement. En fait, il ne permet de décrire que la structure des objets dans un système mettant en valeur leur composition, leurs attributs, leurs méthodes et leurs relations.

Cette description ne prend pas en compte le comportement dynamique réel et global des objets, en particulier, les aspects temporels dans le système, l'évolution de l'état des objets durant leur cycle de vie, les séquences d'événements et de transitions d'un état à un autre, ainsi que leur restriction comportementale dans le temps impliquant la restriction portant sur l'utilisation de leurs méthodes à un de leurs états [19]. Ces différents aspects, très importants dans la modélisation des systèmes et complémentaires aux aspects statiques décrits dans le modèle objet, sont décrits dans le modèle dynamique. Par conséquent, une bonne partie des propriétés dynamiques des systèmes, développées dans le modèle dynamique, sera, durant la phase de programmation diluée dans le code et, à cause de l'insuffisance des langages de programmation orientée objet à les décrire formellement, risque de ne pas être complètement prise en compte.

Par ailleurs, ce n'est que depuis quelques années que les chercheurs se sont intéressés sérieusement aux problèmes posés par le test orienté objet, selon des approches fonctionnelles ou structurelles, en apportant les premiers éléments de réponse tenant compte des concepts du paradigme objet. Cependant, bien qu'il y ait eu plusieurs travaux intéressants dans ce domaine très complexe [4, 5, 6, 13, 15, 18, 20, 21, 22, 23, 25], on est loin, à notre avis, d'avoir apporté une solution complète et satisfaisante à la multitude de problèmes posés par le test des applications orientées objet.

En effet, nous pensons que toute méthode de test des classes d'une application, basée uniquement sur le modèle structurel objet, donc sur le code de l'application, ne sera pas efficace en ce sens où elle ne permettra qu'un test partiel des classes. L'ensemble des cas de test que l'on peut construire avec une telle méthode sera incomplet. Il ne permettra, en fait, de couvrir que la structure interne des classes, à travers la structure de leurs méthodes, ainsi qu'une partie du comportement des objets qu'elles décrivent. Une bonne partie du comportement global des objets risque d'être ignorée. En fait, une grande partie des aspects temporels dans le système risque d'être ignorée, en particulier, on ne pourra pas prendre en compte, durant le processus de test des classes de l'application, l'évolution des différents états des objets durant leur existence, les séquences d'événements et de transitions d'un état à un autre, ainsi que la restriction portant sur l'utilisation des méthodes des objets dans un de leurs états. Ces caractéristiques nous permettent, en fait, d'avoir une idée précise sur l'ensemble des conditions d'utilisation des différentes méthodes d'une classe, donc des requêtes possibles que l'on peut adresser à un de ses objets.

Par ailleurs, et d'une manière générale, l'importance du test dans le processus de développement des logiciels n'est plus à démontrer, d'une part, en raison de son importance économique croissante [24] et, d'autre part, parce qu'il constitue une tâche essentielle dans l'élaboration de leur qualité [1, 29]. Il est également important, lors du test d'une application, en particulier orientée objet, d'utiliser des méthodes de test tenant compte, dans un souci de complémentarité et d'efficacité, aussi bien de ses aspects structurels que de ses aspects dynamiques.

C'est dans ce contexte, et dans un objectif d'assurance qualité, que nous avons décidé de nous intéresser à ces problèmes. Nos différents travaux, dans ce domaine, nous ont permis de développer et de proposer, d'une part, une stratégie pour les tests unitaires et d'intégration des classes, appliquées aux applications SmallTalk, basée sur le modèle structurel et les graphes de dépendances entre classes de l'application [1] et, d'autre part, une stratégie de test fonctionnel des classes, appliquée aux applications C++, basée sur le comportement des objets [2].

Nous proposons, dans cet article, l'approche globale que nous préconisons pour le test fonctionnel orienté objet, d'une manière générale, basée sur une approche formelle de description et analyse du comportement des objets, et une technique pour la génération automatique de séquences de test

fonctionnel des classes ainsi que son utilisation pour les applications Java, prenant en compte la description formelle du comportement des objets, d'une manière particulière. La stratégie que nous préconisons pour le test fonctionnel orienté objet suit un processus incrémental et se base, essentiellement, sur les diagrammes d'états des objets.

2 Principe général de la stratégie :

La plupart des méthodes de développement orienté objet actuelles [8, 10, 17, 27] utilisent la notion d'état pour modéliser le comportement dynamique des instances de classes. Par ailleurs, plusieurs styles de diagrammes d'états ont été proposés dans la littérature [14]. L'approche que nous proposons, pour le test fonctionnel des classes, est orientée sur la vue externe, c'est-à-dire observable, du comportement des objets. Elle se base, en fait, sur une analyse des diagrammes d'états des objets du système. Ces derniers sont décrits d'une manière formelle à l'aide d'un langage de description que nous avons développé et enrichi.

L'approche adoptée nous permet de tester une classe, à travers le test de toutes ses méthodes, grâce à une séquence de test appropriée générée automatiquement en se basant sur la description du comportement dynamique des objets qu'elle décrit. Ceci nous permet d'exercer, au moins une fois, chaque méthode définie dans la classe dans l'ensemble des cas de figure de son utilisation décrits dans le diagramme d'états des objets, et permet de vérifier sa conformité à ses spécifications. Ceci nous permet, par conséquent, d'exercer, au moins une fois, chaque méthode de la classe dans chacun des états de l'objet où elle est permise. De cette façon, notre approche se base, pour le test fonctionnel d'une classe donnée, aussi bien sur le modèle objet que sur le modèle dynamique (Figure 1).

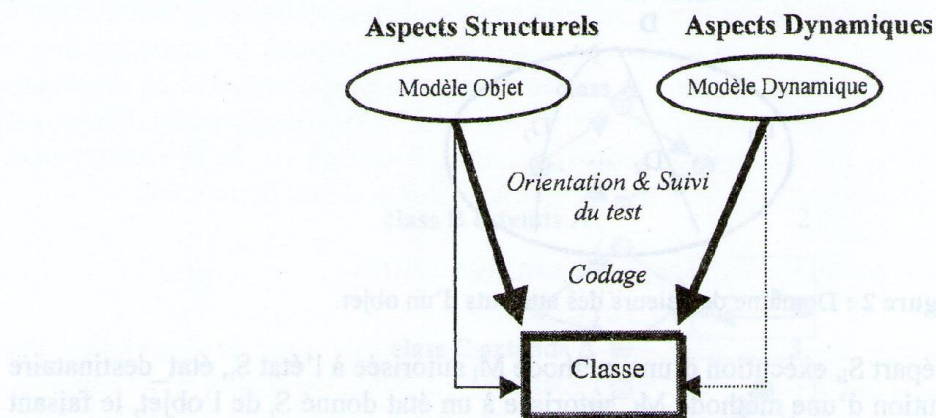


Figure 1 : Principe général de la stratégie.

L'analyse de la description formelle des diagrammes d'états des objets nous permet, dans un premier temps, d'effectuer, en plus de plusieurs contrôles sémantiques statiques, en particulier, de vérifier la cohérence entre les deux modèles objet et dynamique, un affinement et enrichissement de la description du comportement des objets. Dans un second temps, l'analyse de la description des diagrammes d'états nous permet de générer, pour chaque classe de l'application, une séquence de test fonctionnel tenant compte des aspects dynamiques décrits dans le diagramme d'états correspondant et de générer automatiquement un moniteur de test le supportant. Ce moniteur permet de tester la classe en question en fonction de la séquence générée.

Cependant, nous supposons, dans le cadre de notre approche, que les différents diagrammes d'états définis dans le cadre de l'application ont été validés par un expert du domaine. En effet, l'objectif principal de notre approche est de générer automatiquement, pour chaque classe de l'application, une séquence de test et de construire les données correspondantes, à partir de la description des aspects dynamiques des objets qu'elle décrit pour exécuter, au moins une fois, chaque transition dans les diagrammes d'états des objets et de vérifier la conformité des méthodes à leurs spécifications.

3 Diagrammes d'états des objets : Concepts et Définitions

Le comportement global, ou partiel, d'un objet peut être décrit à l'aide d'un diagramme d'états. Le diagramme d'états d'un objet est un graphe orienté dont l'ensemble des nœuds est constitué de l'ensemble des différents états $S = \{S_1, S_2, \dots, S_n\}$ que peut avoir l'objet durant son existence, et dont l'ensemble des arcs est constitué de l'ensemble des différentes transitions $T = \{T_1, T_2, \dots, T_m\}$ possibles entre ces états, suite à des événements donnés.

Les différents états d'un objet représentent des comportements distincts que peut avoir l'objet durant son cycle de vie. Les transitions d'un état à un autre du diagramme correspondent à l'exécution de méthodes autorisées à un état donné de l'objet et sont causées par des événements. Chaque état dans le diagramme doit être accessible par, au moins, une séquence d'une ou plusieurs transitions. Les états initiaux d'un objet, un ou plusieurs selon les cas, sont accessibles par les méthodes de créations d'instances de sa classe.

A un état S_i d'un objet correspond un sous domaine D_i de valeurs auquel les attributs de l'objet peuvent appartenir durant une période de sa vie. L'union des différents sous-domaines $D_1, D_2, \dots, D_n$ des valeurs des attributs d'un objet, correspondant à l'ensemble des états $S_1, S_2, \dots, S_n$ qu'il peut avoir durant sa vie, forme l'ensemble global D des différentes valeurs que peuvent prendre les attributs de l'objet (Figure 2). Quand un objet est dans un de ses états possibles, à un moment de son existence, son comportement se spécialise dans la mesure où il ne peut répondre qu'à un sous-ensemble M_{S_i} de l'ensemble total M des méthodes faisant partie de son interface.

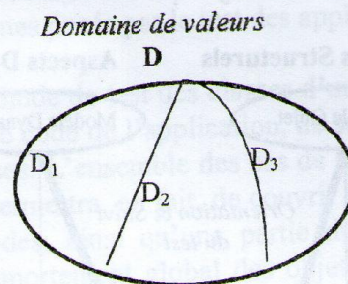


Figure 2 : Domaine de valeurs des attributs d'un objet.

Une transition $T_k = \langle \text{état_départ } S_i, \text{ exécution d'une méthode } M_i \text{ autorisée à l'état } S_i, \text{ état_destinataire } S_j \rangle$ est le résultat de l'exécution d'une méthode M_i , autorisée à un état donné S_i de l'objet, le faisant passer de cet état à un état destinataire S_j dans le diagramme. Cela se traduit par un changement du comportement de l'objet (observable), donc par un changement du sous-domaine d'appartenance des valeurs de ses attributs (Figure 3).

Un Chemin Fonctionnel Réduit (CFR) est une séquence composée d'une ou plusieurs transitions successives $\langle T_1, T_2, \dots, T_m \rangle$ faisant passer l'objet d'un état de départ à un état destinataire dans le diagramme, en passant par une série d'états intermédiaires. Cette séquence de transitions est causée par une succession d'événements. Un Chemin Fonctionnel Complet (CFC) est une séquence composée d'une ou plusieurs transitions faisant passer l'objet d'un état initial à un état final dans le diagramme (s'il existe).

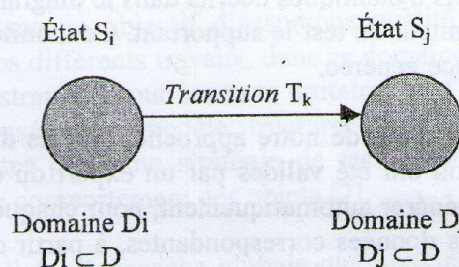


Figure 3 : Transition et domaines de valeurs.

4 Processus de test fonctionnel des classes et héritage :

Le processus des tests unitaires que nous préconisons pour l'ensemble des classes d'une application, est basé sur le graphe d'héritage et suit une stratégie incrémentale descendante. Ce processus est itéré autant de fois qu'il y a de graphes d'héritage dans l'application [1].

On commence par tester les classes les plus générales, la racine de la hiérarchie en premier, classe qui décrit les propriétés générales communes à tous les objets, et suivant un processus incrémental descendant on teste les classes qui la spécialisent. Les dernières classes à tester seront les feuilles du graphe d'héritage. Suivant ce processus, les propriétés héritées par une classe donnée à partir de ses classes de base, et non transformées, ne seront pas retestées.

La stratégie que nous proposons optimise le processus de test des classes d'une application et permet de faire une économie importante en réduisant, non seulement, les propriétés à tester au niveau de chaque classe, d'une manière générale, mais aussi le nombre de cas de test nécessaires pour tester l'ensemble des classes de l'application.

En effet, durant le test d'une classe donnée, l'ensemble des propriétés qu'elle a héritées, structurelles ou comportementales, de ses classes de base et non transformées ne sera pas testé. Seules les parties qui correspondent à la spécialisation de la classe (enrichissement et substitution) seront testées. L'exemple de l'application donnée par la figure 4 montre que seule la partie 2 du diagramme d'états des objets de la classe B et la partie 3 du diagramme d'états des objets de la classe C seront testées. Le reste des diagrammes d'états des classes B et C, correspondant, en fait, aux propriétés comportementales qu'elles ont héritées de la classe A et non transformées, ne sera pas testé.

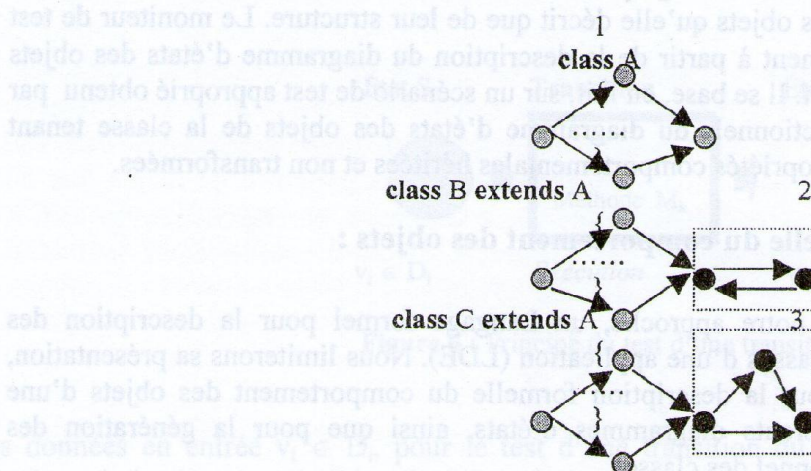


Figure 4 : Test fonctionnel et héritage.

Les différentes parties du comportement des objets correspondant aux propriétés comportementales héritées et non transformées, seront déterminées par analyse des descriptions formelles du comportement des objets de chaque classe de la hiérarchie (Figure 5). Cette analyse nous permet, donc, de déterminer de façon précise, pour chaque classe d'objets, quelles sont les propriétés comportementales, dans la description formelle, qui correspondent exactement à la spécialisation du comportement des objets. De cette façon, le test sera concentré sur, seulement, les propriétés comportementales correspondant à la spécialisation du comportement des objets (enrichissement et substitution du comportement).

Au début du processus de test fonctionnel unitaire d'une classe, le diagramme d'états des objets qu'elle décrit sera, suite à une analyse de la description formelle correspondante, marqué. Les différentes transitions dans le diagramme d'états, qui correspondent aux différentes propriétés comportementales héritées par la classe de ses classes de base et non redéfinies seront marquées testées. Seules les parties du diagramme d'états non marquées, qui correspondent, en fait, à la spécialisation du comportement des objets de la classe (par enrichissement, substitution, ou les deux) seront à tester.

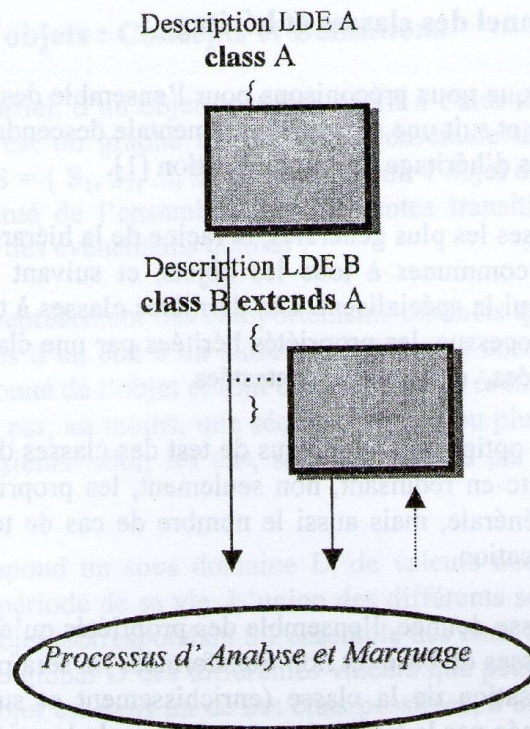


Figure 5 : Marquage du comportement hérité.

Le processus de test d'une classe donnée est dirigé par un moniteur de test spécifique, tenant compte aussi bien des aspects dynamiques des objets qu'elle décrit que de leur structure. Le moniteur de test d'une classe est généré automatiquement à partir de la description du diagramme d'états des objets décrits par la classe et du modèle objet. Il se base, en fait, sur un scénario de test approprié obtenu par parcours des différents chemins fonctionnels du diagramme d'états des objets de la classe tenant compte du marquage, éventuel, des propriétés comportementales héritées et non transformées.

5 Langage de description formelle du comportement des objets :

Nous proposons, dans le cadre de notre approche, un Langage formel pour la description des Diagrammes d'États des objets des classes d'une application (LDE). Nous limiterons sa présentation, dans cet article, à son utilisation pour la description formelle du comportement des objets d'une application, modélisé dans les différents diagrammes d'états, ainsi que pour la génération des moniteurs et séquences de test fonctionnel des classes.

Toutes les informations contenues, ou que l'on peut déduire, du diagramme d'états d'un objet (attributs, états, transitions, restrictions, ...) seront, donc, décrites de façon formelle et serviront de base pour guider le processus de test des classes.

Le langage LDE permet, dès l'étape de modélisation, plusieurs contrôles sémantiques statiques (conformité du diagrammes d'états au modèle objet, états à conflits, élimination des ambiguïtés, chemins fonctionnels impossibles, ... etc). La description du comportement des objets d'une classe se fait, essentiellement, à travers la description des différents états qu'ils peuvent avoir durant leur cycle de vie, des différentes transitions entre ces états, et des différents événements et conditions liés aux parcours de ces transitions.

Description des états : Les différents états que peut avoir un objet durant son existence sont décrits de façon formelle. Cette description comprend à la fois le nom de l'état, sa signature ainsi que l'ensemble des requêtes auxquelles l'objet peut répondre à cet état (transition).

Description des transitions : La description d'une transition d'un état de départ vers un état destinataire du diagramme comprend à la fois la pré-condition de la transition, le nom de la méthode qui sera exécutée en réponse à une requête donnée, l'état destinataire éventuel suite à cette exécution

ainsi que sa post-condition. Cette description peut, cependant, suite à une analyse (section 7), être affinée et contenir la description des sous-états éventuels de l'état courant.

Par ailleurs, d'autres travaux ont porté sur l'extension de ce langage pour supporter, d'une manière formelle, l'héritage du comportement entre objets [3]. Le langage LDE*, ainsi défini, permet de supporter, en fait, dans un premier temps lors de l'étape de modélisation, une construction incrémentale et formelle du diagramme d'états des objets d'une classe à partir des diagrammes d'états des objets de ses classes de base. Ceci permet, par conséquent, lors du processus de test, pour chaque classe de l'application et tenant compte des propriétés qu'elles hérite, en particulier comportementales, de ses classes de base, un marquage des transitions en principe déjà testées qui correspondent au comportement hérité et non redéfini par les objets de la classe, d'une part, et une génération incrémentale des séquences de test, d'autre part.

6 Test d'une transition :

Lors du test d'une transition, on s'intéresse aux données d'entrée (état de départ), c'est-à-dire aux valeurs prises par les différents attributs de l'objet avant l'exécution de la requête, d'une part, et aux résultats de l'exécution de la requête (état destinataire), c'est-à-dire aux nouvelles valeurs prises par les différents attributs de l'objet après l'exécution de la requête, d'autre part. L'objectif est de vérifier, en plus des valeurs prises par les différents attributs de l'objet, en particulier les attributs d'états, après exécution de la requête grâce à la signature de l'état (ou sous-état) destinataire définie dans la description LDE, la conformité de la requête à ses spécifications. La transition qui correspond, en fait, à l'exécution d'une requête particulière permise par l'objet dans l'état où il se trouve, sera considérée comme une boîte noire (Figure 6).

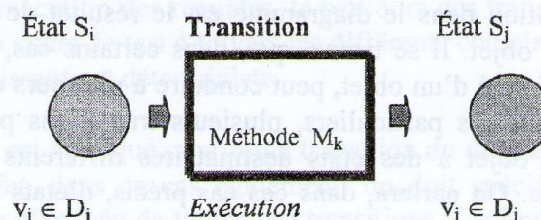


Figure 6 : Principe du test d'une transition.

Les données en entrée $v_i \in D_i$, pour le test d'une transition du diagramme, seront construites en fonctions de la signature de l'état de départ S_i de la transition. Il s'agira, alors, d'affecter des valeurs aux attributs de l'objet, vérifiant la signature de l'état de départ de la transition spécifiée dans la description du diagramme d'états, de lancer l'exécution de la requête correspondant à la transition testée et, enfin, de vérifier la conformité des résultats obtenus par rapport aux résultats attendus (post-condition).

Par ailleurs, il faut également s'assurer que les valeurs des différents attributs de l'objet, après l'exécution de la requête en question, sont conformes à la signature spécifiée pour l'état destinataire dans le diagramme (Figure 7). Ces vérifications sont effectuées par évaluation des signatures de l'état de l'objet, avant et après l'exécution de la requête, et comparaison avec les signatures spécifiées dans la description du diagramme correspondant. Par ailleurs, tout écart par rapport à la signature de l'état destinataire spécifiée dans la description du diagramme sera interprété comme une défaillance de la requête et nécessitera, donc, une mise au point de la méthode en question.

Le domaine des valeurs des attributs, correspondant à un état donné de l'objet, sera subdivisé en classes d'équivalence. L'objectif est de couvrir l'ensemble des classes d'équivalence du domaine des valeurs des attributs de l'objet, si elles existent, en exerçant la transition avec au moins un cas de test de chaque classe d'équivalence en entrée et d'observer les résultats en sortie (Figure 8).

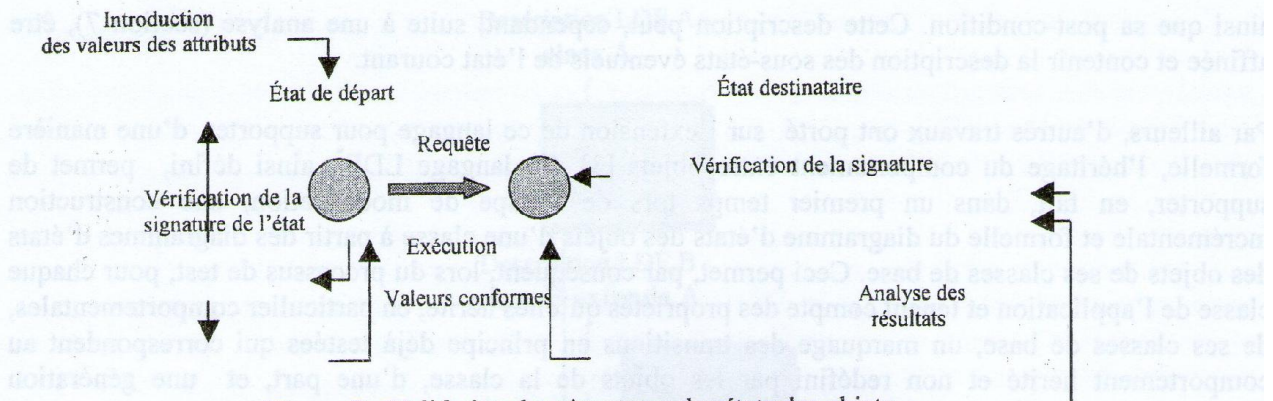


Figure 7 : Validation des signatures des états des objets.

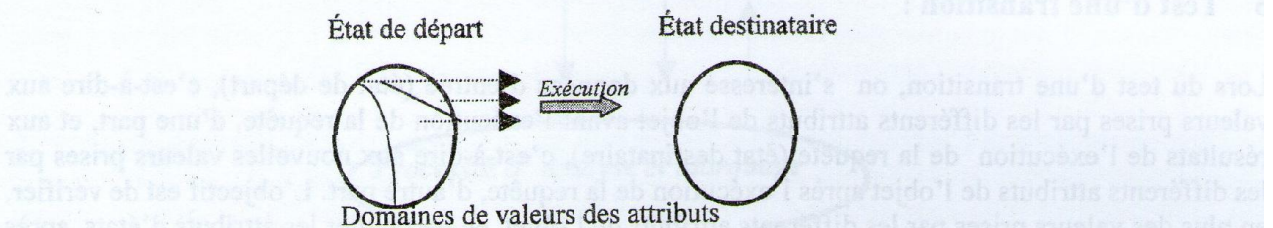


Figure 8 : Domaine de valeurs des attributs et création des cas de test.

7 États à conflits :

7.1 Définition

L'état destinataire d'une transition dans le diagramme est le résultat de l'exécution d'une requête autorisée à un état donné de l'objet. Il se trouve que, dans certains cas, l'exécution d'une requête (méthode) autorisée, à un état donné d'un objet, peut conduire à plusieurs états destinataires dans son diagramme (Figure 9). Dans ces cas particuliers, plusieurs transitions partant d'un même état du diagramme et faisant passer l'objet à des états destinataires différents correspondent, en fait, à l'exécution de la même requête. On parlera, dans ces cas précis, d'états à conflits en ce sens où il existe certaines ambiguïtés dans le parcours des transitions.

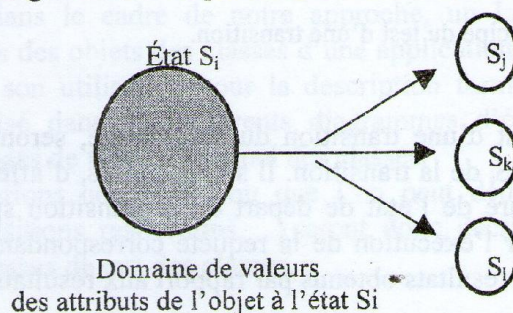


Figure 9 : État à conflits.

7.2 Résolution des conflits

L'analyse du domaine D des valeurs des attributs de l'objet, correspondant à un état à conflits, nous conduit à conclure qu'il existe, dans ces cas particuliers, plusieurs sous-domaines du domaine global D de valeurs des attributs de l'objet, donc plusieurs sous-états de l'état en question. C'est en fonction du sous-domaine auquel les valeurs des attributs de l'objet appartiennent avant l'exécution de la requête (sous-état) que se détermine l'état destinataire dans le diagramme. Ces différents sous-domaines correspondent, en fait, à des sous-états de l'état dans lequel se trouvait l'objet avant l'exécution de la requête.

D'une manière générale, si à partir d'un état S d'un objet dans le diagramme, l'exécution de la même requête R peut conduire l'objet à différents états destinataires $S_1, S_2, \dots, S_n$, on peut conclure que le domaine D des valeurs des attributs de l'objet est subdivisé en, au moins, n sous-domaines $D_1, D_2, \dots, D_n$, correspondant chacun à un sous-état de l'objet, tels que : $D \supseteq D_1 \cup D_2 \cup \dots \cup D_n$.

A l'exécution de la requête R à l'état S de l'objet correspondra, dans la description de son diagramme d'états, n transitions différentes. Chacune d'entre elles possèdera son propre sous-domaine de départ. De cette façon, on lève les ambiguïtés de départ et le contrôle de l'exécution de R pourra se faire de manière déterministe, c'est-à-dire suivant le sous-domaine de valeurs dans lequel se trouveront les attributs de l'objet, on saura de façon déterministe l'état destinataire dans le diagramme à la suite de l'exécution de R .

L'analyse des différents conflits, correspondant à un état donné du diagramme, nous permet de déterminer ses différents sous-états et, par conséquent, les différents sous-domaines de valeurs de ses attributs.

Les différents conflits, qui peuvent exister dans le diagramme d'états d'un objet, peuvent être détectés automatiquement par simple analyse de la description LDE et, en particulier, par analyse des transitions qui partent de chaque état. Pour résoudre ces différents conflits, on doit spécifier de façon précise, pour chaque état à conflits, l'ensemble des différents sous-domaines (sous-états), en introduisant leur signature dans la description des transitions correspondantes. Ceci permet, en fait, de lever les ambiguïtés dans le parcours des différentes transitions correspondant à l'exécution de la même requête à un état à conflits, d'une part, et d'affiner la description du comportement des objets, d'autre part.

Le contrôle de l'exécution des différentes transitions repose entièrement sur la richesse et la précision de la description du comportement des objets du système. En effet, plus la description est complète et précise, plus le contrôle de l'exécution des requêtes, le parcours des transitions, ainsi que l'évolution du comportement de l'objet en cours de test à travers les différents chemins de son diagramme d'états, peut se faire d'une manière puissante et déterministe.

Par ailleurs, le même principe est appliqué pour toute transition du diagramme dont l'état destinataire est un état à conflits. En effet, dans ce cas également, on doit spécifier, dans la description du diagramme, les sous-domaines d'arrivée de toutes ces transitions. Ceci permet d'affiner, encore plus, la description du comportement des objets et permet de suivre, donc de pouvoir contrôler de façon déterministe l'exécution de tous les chemins fonctionnels.

7.3 États à conflits et combinaisons entre transitions

Après avoir déterminé l'ensemble des états à conflits d'un diagramme, les différents sous-états de chacun d'entre eux et précisé leur signature, on peut, par analyse de ces dernières, déterminer pour chaque état à conflits les combinaisons impossibles entre les transitions qui aboutissent à cet état et les transitions qui partent de cet état.

En effet, d'une manière générale, si pour un état donné S d'un objet, il existe deux transitions $\langle S_p, T_p, S \rangle$ qui part d'un état S_p prédécesseur de S et aboutit à l'état S et $\langle S, T_s, S_s \rangle$ qui part de l'état de S et aboutit à un état S_s successeur de S , et la signature de l'arrivée de T_p est différente de la signature de départ de T_s , correspondant respectivement à deux sous-domaines différents inclus dans le domaine global D_s de S , on peut conclure que le CFR de longueur 2 ou la combinaison $\langle T_p, T_s \rangle$ est impossible à exécuter. Dans le cas où un état donné du diagramme ne présente aucun conflit, on peut supposer que toutes les combinaisons, entre les transitions qui arrivent à cet état et les transitions qui partent de cet état, sont possibles.

L'exemple de la figure 10 montre que seules les combinaisons (T_1, T_4) , (T_2, T_5) et (T_3, T_6) sont possibles. Toutes les autres combinaisons sont impossibles. Ceci permet, d'une part, d'avoir une idée plus précise sur le comportement des objets et, d'autre part, de piloter au mieux le processus de test.

Par ailleurs, l'analyse des différents sous-domaines correspondant aux sous-états d'un objet ainsi que l'ensemble des transitions qui arrivent et qui partent de chaque sous-domaine, permet de détecter l'ensemble des transitions qui ne sont jamais exécutées, ce qui constitue une anomalie dans la modélisation du comportement des objets. La transition T_7 , de l'exemple donné dans la figure 10, est une transition impossible à exécuter. En effet, elle part de l'un des quatre sous-domaines de l'objet, seulement, aucune transition n'arrive à ce sous-domaine. Par ailleurs, cette analyse nous permet d'écarter, parmi l'ensemble des chemins fonctionnels théoriques possibles d'un diagramme d'états, tous les chemins impossibles à exécuter.

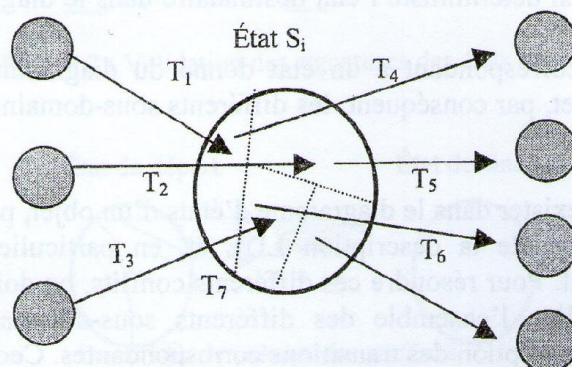


Figure 10 : État à conflits et combinaison entre transitions.

L'impossibilité d'exécuter un chemin donné du diagramme est due à la présence de prédicats contradictoires, prédicats qui conditionnent le parcours de ses transitions. Ces résultats nous permettent, en fait, de déterminer les différents chemins fonctionnels exécutables et, par conséquent, les différentes séquences de test, appropriées pour une classe, à générer automatiquement, et déterminer également celles qui ne le sont pas. Ces dernières doivent, en principe, être rejetées par l'objet car elles ne correspondent pas à son comportement.

8 Stratégie incrémentale de test d'une classe :

La stratégie que nous proposons pour le test fonctionnel d'une classe suit un processus incrémental. Le principe de cette stratégie est de tester la classe, à travers le test de l'ensemble de ses méthodes, en se basant, essentiellement, sur le diagramme d'états de ses objets.

L'objectif principal de cette stratégie est d'exercer, au moins une fois, chaque transition du diagramme, ce qui permet d'exercer chaque méthode de la classe dans tous les cas de figure de son utilisation prévus dans la description du comportement des objets.

Le processus de test d'une classe donnée démarre par les états initiaux du diagramme correspondant, et se poursuit progressivement, transition par transition, le long des différents chemins fonctionnels. L'objectif est de parcourir, à partir de chaque état initial du diagramme, l'ensemble des chemins fonctionnels réduits en exerçant chaque transition au moins une fois et en évitant de repasser par les transitions déjà testées. Le parcours du diagramme s'arrête lorsque la dernière transition non encore testée est réduite du diagramme.

Le chemin fonctionnel réduit parcouru depuis un état initial est sauvegardé. A la fin d'une opération de test d'une transition et après validation, la transition en question est réduite du diagramme et enregistrée dans le chemin fonctionnel réduit parcouru. Cette opération nous permet de garder la trace des différents chemins parcourus, pour les éventuels retour en arrière de la stratégie. L'objectif est de parcourir, à partir de tout état du diagramme, et en fonction des transitions qui partent de cet état, l'ensemble des chemins fonctionnels réduits non testés. Ce traitement est réalisé à l'aide de l'algorithme récursif `Parcours_CFR( Etat )` de parcours des différents chemins fonctionnels réduits à partir d'un état quelconque du diagramme.

A la fin de l'opération de test d'une transition, la transition sera marquée testée et réduite du diagramme d'états. Lorsque, pour un état donné du diagramme, toutes les transitions qui partent de cet

état sont marquées testées et avant de revenir, grâce à la sauvegarde du chemin fonctionnel parcouru, à l'état prédécesseur dans le diagramme, le moniteur de test de la classe procède au test des requêtes non autorisées à cet état. On doit s'assurer, lors de cette opération, que toutes les requêtes non autorisées, à cet état, selon la description LDE du comportement, seront rejetées par l'objet.

Par ailleurs, tout état du diagramme dont l'ensemble des transitions de départ ainsi que l'ensemble des transitions d'arrivée sont marquées testées sera, à son tour, réduit du diagramme. Avant la réduction d'un tel état, le moniteur procède au test des différentes combinaisons possibles entre les transitions qui arrivent à cet état et celles qui partent de cet état. L'opération consiste à tester toutes les combinaisons théoriques possibles et à s'assurer que les combinaisons impossibles, détectées lors de l'analyse des descriptions dynamiques, seront rejetées par l'objet, c'est-à-dire la requête correspondant à la seconde transition de la combinaison ne sera pas exécutée par l'objet.

La réduction des transitions testées permet, d'une part, d'éviter lors du parcours des chemins fonctionnels du diagramme de parcourir des transitions déjà testées et, d'autre part, de réduire de façon incrémentale la complexité du diagramme et de renseigner les transitions et chemins fonctionnels qui restent à tester (Figure 11).

L'exemple donné par la figure 11 montre que le chemin fonctionnel $\langle T_1, T_2, T_3 \rangle$ est marqué parcouru et que les transitions T_4 , T_5 et T_6 ne sont pas encore testées, ce qui fait qu'il reste, en principe, deux chemins fonctionnels à parcourir : $\langle T_4, T_5, T_3 \rangle$ et $\langle T_4, T_6 \rangle$.

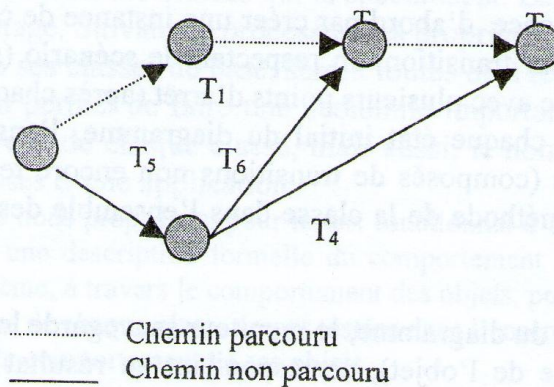


Figure 11 : Marquage des transitions parcourues.

La stratégie adoptée permet, grâce à la sauvegarde du chemin fonctionnel parcouru, de revenir en arrière. L'objectif général est de revenir en arrière, vers les états prédécesseurs, et de s'arrêter au premier état à partir duquel on peut parcourir d'autres transitions (non testées).

Dans le cas de l'exemple, et sachant qu'il n'y a plus de transitions après T_3 , le moniteur de test effectue un retour en arrière, après le test de la transition T_3 , et suivant le même principe revient au début, c'est-à-dire à l'état initial et permet de parcourir les chemins fonctionnels qui restent à partir de l'état initial, c'est-à-dire : $\langle T_5, T_3 \rangle$ et $\langle T_4, T_6 \rangle$.

Le processus de test fonctionnel des classes d'une application se fait d'une manière incrémentale, classe par classe, en fonction de la hiérarchie. Le test démarre par la classe racine du graphe d'héritage et se poursuit, progressivement, jusqu'aux classes feuilles de la hiérarchie suivant un ordre bien déterminé.

Le test de chaque classe de l'application nécessite le développement d'un moniteur de test spécifique. Ce dernier sera généré automatiquement à partir de la description du diagramme d'états des objets qu'elles décrit et du code de la classe. Cette opération nécessite une instrumentation des classes de l'application. L'objectif de cette opération est de rajouter des instructions particulières (méthodes) permettant au moniteur de piloter et contrôler le test d'une classe (contrôle de l'objet, vérification des états de l'objet, vérification des pré-conditions et des post-conditions, séquences de messages à l'objet, etc).

Le moniteur de test d'une classe se base sur un scénario généré à partir de la description du comportement de ses objets. Ce scénario correspond, en fait, à un ordonnancement d'une séquence de

messages adressés à l'objet, ce qui provoquera (ou non selon le cas) l'exécution des différentes méthodes correspondantes. Cette séquence de messages correspond au parcours établi, suivant la stratégie décrite, des différentes transitions du diagramme.

Les différents numéros marquant les transitions du diagramme de l'exemple donné par la figure 12, indiquent un exemple possible dans lequel elles seront parcourues, donc de l'ordre dans lequel les méthodes correspondantes seront exécutées (messages adressés à l'objet).

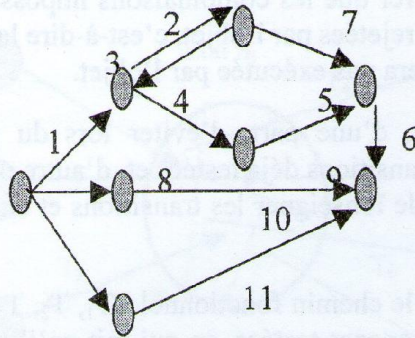


Figure 12 : Ordonnancement de l'exécution des transitions.

Le moniteur de test d'une classe commence d'abord par créer une instance de cette classe. Le test se fait de façon incrémentale, transition par transition, en respectant le scénario (séquence de requêtes adressées à l'objet) établi par la stratégie avec plusieurs points d'arrêt (après chaque test de transition). Ceci permet de parcourir, à partir de chaque état initial du diagramme, l'ensemble des différents chemins fonctionnels réduits possibles (composés de transitions non encore testées), ce qui permet d'exercer, au moins une fois, chaque méthode de la classe dans l'ensemble des cas de figure de son utilisation.

A la fin du test d'une transition donnée du diagramme, le moniteur sauvegarde les valeurs des attributs de l'objet dans son état actuel (image de l'objet), correspondant au résultat de l'exécution de la transition.

Lors du test d'une transition, le moniteur adresse le message correspondant à l'objet et vérifie par la suite la validité des nouvelles valeurs des attributs de l'objet, en vérifiant leur conformité par rapport à la signature de l'état destinataire spécifié dans la description. On peut revenir, éventuellement, plusieurs fois pour relancer l'exécution de la transition avec de nouvelles valeurs qu'on peut affecter aux attributs de l'objet. Cette opération est effectuée dans le cas où le domaine des valeurs des attributs de l'objet, avant l'exécution de la requête, est vaste (plusieurs classes d'équivalence). A la fin de chaque opération de test d'une transition, le testeur doit valider l'opération. La transition testée sera réduite du diagramme d'états et la couverture de test des transitions du diagramme sera augmentée. Les différentes valeurs des attributs de l'objet, que se soit avant ou après l'exécution d'une transition sont archivées, ce qui facilite les différents retour en arrière de la stratégie.

Avec cette approche, et grâce aux enchaînements dans l'exécution des transitions (scénario – séquence de messages), le nombre de cas de test, pour exercer l'ensemble des transitions du diagramme, est réduit considérablement. En effet, les valeurs prises par les attributs d'un objet, à la suite de l'exécution d'une transition donnée durant le test, serviront de données d'entrée pour le test de la transition suivante du chemin fonctionnel réduit, si toutefois l'exécution s'est bien déroulée. D'une manière générale, on peut, lors du parcours d'un chemin fonctionnel, enchaîner l'exécution de ses différentes transitions en récupérant les résultats de l'une comme données d'entrée de la suivante.

A la fin du processus de test des différentes transitions du diagramme d'états d'une classe, et avant de passer à la classe suivante, suivant l'ordre établie à partir de la hiérarchie, un récapitulatif sera donné indiquant :

- pour chaque transition du diagramme le nombre de fois où elle a été exercée,
- pour chaque transition du diagramme, dont l'état de départ est subdivisé en classes d'équivalence, le pourcentage des classes exercées, au moins une fois, sur le total des classes possibles,
- pour chaque transition du diagramme, dont l'état destinataire est subdivisé en classes d'équivalence, le pourcentage des classes exercées, au moins une fois, sur le total des classes possibles,
- pour chaque état du diagramme, le pourcentage des requêtes non autorisées testées sur le total des requêtes non autorisées,
- pour chaque état du diagramme, le pourcentage des combinaisons testées sur le total des combinaisons possibles entre les transitions qui arrivent et les transitions qui partent de cet état,
- enfin la couverture de test des différentes transitions du diagramme évaluée par le rapport entre le nombre de transitions exercées, au moins une fois, sur le nombre total des transitions du diagramme d'états.

9 Conclusions :

Le processus de test fonctionnel que nous préconisons, pour l'ensemble des classes d'une application, est basé sur son graphe d'héritage et suit une stratégie incrémentale descendante. On commence par tester les classes les plus générales, la racine de la hiérarchie en premier et, suivant un processus incrémental descendant, on teste les classes qui la spécialisent. Les dernières classes à tester seront les feuilles du graphes d'héritage. Suivant ce processus, les propriétés héritées et non transformées par une classe donnée, à partir de ses classes de base, seront toutes déjà testées. De cette manière, on optimise le processus de test et on permet de faire une économie importante en réduisant, non seulement, les propriétés à tester au niveau de chaque classe, mais aussi, le nombre de cas de test nécessaires pour tester l'ensemble des classes d'une application.

Par ailleurs, la stratégie que nous proposons, pour le test fonctionnel d'une classe, suit également un processus incrémental et se base sur une description formelle du comportement de ses objets. La prise en compte des aspects dynamiques du système, à travers le comportement des objets, permet d'orienter aux mieux le test d'une classe. En effet, les méthodes de chaque classe seront testées dans l'ensemble des cas de figure de leur utilisation prévus dans la description du comportement de ses objets.

Références

- [1] M. Badri, L. Badri et S. Layachi, « Vers une stratégie de tests unitaires et d'intégration des classes dans les applications orientées objet », Revue Génie Logiciel, N° 38, Décembre 1995.
- [2] M. Badri et L. Badri, « Test fonctionnels des classes : Une approche incrémentale basée sur le comportement des objets », 9^{èmes} Journées Internationales sur le « Génie Logiciel et ses Applications », Numéro spécial de la Revue Génie Logiciel, N° 42, Décembre 1996.
- [3] M. Badri et L. Badri, « Génération incrémentale de séquences de tests fonctionnels des classes : Vers une approche formelle basée sur l'héritage du comportement des objets », 12^{èmes} Journées Internationales sur le « Génie Logiciel & Ingénierie des Systèmes et leurs Applications », Paris, Décembre 1999 (à paraître).
- [4] R.V. Binder, « Design for testability in object-oriented systems », Communications of the ACM, Vol. 37, N° 9, pp. 87-101, September 1994.
- [5] R.V. Binder, « Testing Object-Oriented Systems : A Status Report », American Programmer, Vol. 7, N° 4, April 1994.
- [6] R.V. Binder, « Testing Object-Oriented Software : A Survey », Journal of Software Testing, Verification and Validation, Vol. 6, 1996.
- [7] G. Booch, « Object-Oriented Development », IEEE Transactions on Software Engineering, SE-12, February 1986.
- [8] G. Booch, « Object-Oriented Analysis and Design with Applications », Benjamin/Cummings, 1994.
- [9] G. Booch, « Conception orientée objet et applications », Addison-Wesley, 1994.