CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

Thèse présentée pour l'obtention du titre de

DOCTEUR DU CNAM
Spécialité : **INFORMATIQUE**

par
Amel MAMMAR

Un environnement formel pour le développement d'applications bases de données

soutenue le 21 Novembre 2002 devant le jury composé de :

Président :	**M. Michel SCHOLL** (CNAM Paris)
Rapporteurs :	**M. Marc FRAPPIER** (Université de Sherbrooke, Quebec) **M. Jean Pierre GIRAUDIN** (Université de Grenoble)
Directrices de thèse :	**Mme. Régine LALEAU** (CEDRIC-I.I.E-CNAM) **Mme. Véronique DONZEAU-GOUGE** (CEDRIC-CNAM Paris)
Examinateur :	**M. Mamoun FILALI** (IRIT Toulouse)

CENTRE D'ÉTUDES ET DE RECHERCHE EN INFORMATIQUE DU CNAM (CEDRIC)

Remerciements

Je tiens à remercier, en tout premier lieu, Régine Laleau, ma directrice de recherches. Tout au long de cette thèse, Régine m'a guidée, conseillée, aidée et soutenue. Sa disponibilité et sa bonne humeur exceptionnelle m'ont permis de travailler dans une ambiance fort agréable et inoubliable. Qu'elle trouve ici l'assurance de ma profonde gratitude.

Monsieur Jean Pierre Giraudin, Professeur à l'université de Grenoble, et Monsieur Marc Frappier, Professeur à l'université de Sherbrooke, m'ont fait l'honneur et le plaisir d'être rapporteurs de cette thèse. Je tiens à leur exprimer ma gratitude pour l'intérêt qu'ils portent à mes activités et pour toutes leurs remarques pertinentes qui ont permis d'améliorer la qualité de ce manuscrit.

Je remercie vivement Monsieur Michel Scholl, Professeur au CNAM, et Monsieur Mamoun Filali, chargé de recherche CNRS l'IRIT, pour l'honneur qu'ils me font de participer à mon jury de thèse.

Je tiens à exprimer ma gratitude à Madame Véronique Donzeau-Gouge, Professeur au CNAM, de m'avoir accueillie dans son équipe "conception de programmation raisonnée". Tout au long de ces mois, elle a porté un grand intérêt à mon travail. Ses relectures de ce rapport, ainsi que ces remarques pertinentes m'ont permis d'améliorer sa qualité.

Mes vifs remerciements vont également à Madame Catherine Dubois, Professeur à l'IIE (CNAM PARIS), pour le temps qu'elle a consacré à la relecture de ce rapport, à Madame Mireille Jouve, directrice des études à l'IIE, pour les échanges très fructueux qu'on a pu avoir sur les enseignements.

Un grand merci à Karima, Zina, Samira, Assia, Farida, Olivier (Boite), Olivier (Pons), Frédéric pour leur aide, leur soutien et les agréables moments qu'on a passés ensemble.

Je remercie également toutes les personnes que j'ai côtoyées : chercheurs, enseignants, techniciens, ingénieurs et administratifs. Chacun a contribué à créer un cadre de travail convivial et efficace.

Je réserve mes derniers remerciements à mes chers parents, à mes frères, à mes sœurs ainsi qu'à mes neveux et nièces (que j'aime tant) qui n'ont épargné aucun effort pour m'aider et m'encourager durant toute ma scolarité. Qu'ils sachent que je leur suis très reconnaissante.

À la mémoire de Philippe Facon

Liste des publications

Aucune publication n'a été retenue sur résumé. Cette liste exclut les rapports de recherche et les publications en cours de soumission.

Revues nationales avec comité de lecture

A. Mammar "Une approche formelle par raffinement pour le développement d'applications bases de données" 19 ème congrès INFORSID, section Jeunes Chercheurs, Genève, 2001, pp. 319–339.

L'article a été retenu avec 4 autres articles et augmenté pour un numéro spécial de la revue Ingénierie des Systèmes d'Information de Hermès, "*Formalismes et modèles pour les systèmes d'information*" Volume 6, n° 2, pp. 41–72, 2001,

Conférences internationales avec comité de sélection et actes

J'ai distingué les conférences de bonne sélectivité et/ou d'audience internationale des workshops.

a) Conférences de bonne sélectivité et/ou d'audience internationale

R. Laleau, A. Mammar, A Generic Process to Refine a B Specification into a Relational Database Implementation, Conférence Internationnale, ZB2000, Springer-Verlag LNCS no 1878, pp. 20–41, York. Aug–Sep 2000.

R. Laleau, A. Mammar, An Overview of a Method and its Support Tool for Generating B Specifications from UML Notations, In Proceeding 15th IEEE conf on Automated Software Engineering (ASE2000), pp. 269–272, Grenoble, France, Septembre 2000.

b) Conférences nationales/ Workshops

A. Mammar, Un Environnement Formel pour le Développement d'Applications Bases de Données, Poster, Forum Jeunes Chercheurs, INFORSID 2000.

R. Laleau, A. Mammar, An Automatic Generation of B Specification from Well-Defined UML Notations for Database Applications, Fifth International Symposium on Programming and Systems (ISPS), pp. 27–38, Algiers, May 12-14 2001.

Présentation des travaux de recherche

A. Mammar, R. Laleau "Définition d'un environnement formel pour le développement d'applications bases de données", Journées du Groupe B du GDR ALP <<Outil pour et autour de B>> organisées par l'INRETS, villeneuve-d'Ascq, Octobre 2001.

Table des matières

Table des figures

Chapitre 1

Introduction

Au cours de cette dernière décennie, la sécurité dans le développement d'applications bases de données est devenu un besoin de plus en plus primordial et crucial. L'utilisation des bases de données s'est répandu à divers domaines : e-business, systèmes financiers, cartes à puces [37], etc. Bien que les applications de ce type de domaine ne soient pas réellement critiques (pas de risque humain), néanmoins, un degré de sûreté est exigé. En effet, par exemple, le coût de correction d'un code portable sur des cartes à puces est relativement lourd à supporter : retour et correction de toutes les cartes distribuées, remise en cause de l'image du concepteur, etc.

Les applications bases de données sont caractérisées par une forte composante données. Ces données sont souvent interdépendantes et fortement liées. Des contraintes d'intégrité peuvent être exprimées sur ces dernières. Dans ce type d'applications, la difficulté pour les concepteurs réside essentiellement dans l'écriture de programmes qui respectent l'intégrité de l'ensemble des données. Cette difficulté est de plus en plus importante avec la croissance continue de la complexité de ce type d'applications.

Les processus traditionnels de développement d'applications bases de données sont centrés particulièrement sur la définition des structures de données. Le développement des programmes (transactions) est souvent informel. Le passage du niveau conceptuel de la spécification des traitements au développement des programmes est basé complètement sur l'expérience et le savoir-faire du programmeur. La validation des programmes obtenus est réalisée par jeux de tests. La sûreté de cette validation dépend fortement de la quantité et de la qualité de ces jeux de tests.

Les méthodes utilisées pour le développement d'applications bases de données peuvent être classées, suivant le niveau de formalisation de leurs concepts, en trois familles: les méthodes informelles, semi-formelles et formelles. Les méthodes informelles utilisent la langue naturelle. Étant facilement accessibles, ces méthodes sont chronologiquement les premières à être utilisées. Elles facilitent la communication avec les différentes équipes de développement. Néanmoins, leur manque de formalisation et donc de précision conduit souvent à des erreurs d'interprétation des besoins du système. Les systèmes développés par ce type de méthodes ne peuvent être validés de manière sûre.

Les méthodes semi-formelles utilisent souvent des notations graphiques. Elles permettent une représentation intuitive et synthétique du système à étudier. Ces méthodes représentent des avantages indéniables pour la modélisation. Elles constituent un support idéal pour la communication entre les différents acteurs du système. Néanmoins, ces méthodes souffrent encore d'un manque de sémantique précise de leurs concepts ce qui réduit considérablement toute possibilité de raisonnement et de preuve sur les modélisations ainsi obtenues.

Les méthodes formelles, basées sur des notations mathématiques, fournissent des concepts précis non ambigus. Elles permettent la construction de modélisations non seulement précises et concises mais surtout analysables par des outils (contrôleurs de types, prouveurs, simulateurs, ...). De plus, ces méthodes offrent souvent des techniques de raffinement prouvées permettant la transition de la spécification abstraite d'un système vers son implémentation (code exécutable). En dépit de leurs avantages, l'utilisation des méthodes formelles, en industrie, est restreinte au développement de systèmes critiques (dans ce cas, elles sont inévitables). La faible étendue de l'utilisation des méthodes formelles est due aux raisons suivantes. L'utilisation des méthodes formelles requiert des connaissances mathématiques non négligeables, particulièrement dans la phase la plus importante qui est la preuve. Ce degré et type de connaissances sont difficiles à exiger de développeurs habitués aux méthodes informelles ou semi-formelles basées surtout sur l'intuition et l'expérience du domaine. De plus, les méthodes formelles sont plutôt des démarches (notations + philosophie de développement) et n'offrent pas de guides méthodologiques au développement. Néanmoins, nous pensons que l'intégration des méthodes formelles en industrie peut être facilitée et améliorée en proposant des méthodologies supportées par des outils qui permettent l'utilisation conjointe des notations graphiques et mathématiques.

Dans l'optique de couplage des notations graphiques et mathématiques pour le développement d'applications bases de données, plusieurs travaux ont été réalisés [22, 53, 17, 51]. Le principe de base de ces travaux consiste à définir un ensemble de règles de traduction de ces notations graphiques en spécifications formelles **Z** ou **B** (notations mathématiques). Ce couplage permet de tirer profit, d'une part, de l'aspect intuitif et visuel qu'offrent les notations graphiques, et d'autre part, de l'aspect formalisation des notations mathématiques : précision et possibilité de prouver les spécifications ainsi obtenues. Afin de faciliter la mise en pratique de ce couplage, quelques outils support ont été développés.

L'objectif de tout développement base de données est l'établissement de programmes (aspect exécution) traitant des données. La plupart des travaux existants sur l'intégration des méthodes formelles dans le développement des systèmes d'information ne traitent que le niveau conceptuel. Les approches proposées ne couvrent qu'une partie du processus de développement d'un projet, d'autant plus que les spécifications obtenues sont trop abstraites pour être exécutables.

1.1 Objectifs et contributions du travail de recherche

Dans le cadre de notre travail de recherche, nous nous intéressons à la définition d'un environnement formel pour le développement d'applications bases de données sûres. Notre travail utilise la technique du raffinement de la méthode **B**. L'utilisation de la méthode **B**, plutôt qu'une autre méthode comme **Z** ou **VDM**, est justifiée comme suit. En effet en **Z**, seuls un contrôleur de type et un assistant à la preuve interactive (non automatique) sont disponibles, le processus de raffinement n'est pas outillé. Les mécanismes de structuration de **VDM** ne correspondent pas à nos besoins.

Le raffinement [1] désigne la dérivation formelle de programmes à partir de la spécification abstraite de ces derniers. Le raffinement concerne aussi bien les données que les traitements. Il permet la construction graduelle de programmes par une série de transformations successives prouvées. L'utilisation de la technique de raffinement exige, à chaque étape, l'établissement d'un ensemble de preuves permettant de vérifier que le programme transformé satisfait bien (ne contredit pas) le programme initial. La correction de chaque étape de raffinement permet l'établissement, de proche en proche, de la correction du code généré par rapport à la spécification abstraite initiale.

Le point de départ de notre approche est une spécification **B** obtenue par traduction d'un ensemble de diagrammes **UML**[54] dédiés à la description des applications bases de données. Le processus de traduction que nous utilisons a été défini dans [53, 22]. Les diagrammes **UML** concernés sont le diagramme de classes, les diagrammes états/transitions et les diagrammes de collaborations. Ces diagrammes n'ayant pas de sémantique précise, une étape préalable à cette traduction a été leur formalisation. Les spécifications **B** que nous considérons sont caractérisées par une forme générique standard. On distingue trois composantes principales : un ensemble de données abstraites qui représente l'état du système, un ensemble d'opérations qualifiées de base qui permet la manipulation de chaque entité du système indépendamment des autres, et enfin, un ensemble d'opérations d'interface utilisant les opérations de base et qui correspondent aux transactions utilisateurs.

L'approche de raffinement que nous proposons consiste en la définition d'un ensemble de règles génériques qui permettent le raffinement automatique et conjoint des données abstraites et des opérations de base. La définition de ces règles est rendue possible grâce à l'aspect générique et standard des spécifications abstraites **B** générées et de la modélisation formelle en **B** du modèle relationnel. Les règles de raffinement, définies dans ce travail, se divisent en deux catégories : les règles de raffinement de données et les règles de raffinement des substitutions de base. À chaque règle de raffinement de données et forme de substitutions **B**, correspond une règle de raffinement des substitutions qui permet de réécrire des substitutions abstraites en fonction des données concrètes. Nos règles de raffinement permettent la génération des différentes tables relationnelles ainsi que les ordres de base **SQL** [50] comme l'insertion d'un nouveau tuple et la suppression d'un ensemble de tuples existants. Certaines règles de raffinement des données proviennent des algorithmes classiques de transformation d'un diagramme Entités/Associations en un schéma relationnel [9]. L'apport de notre processus au regard de ces algorithmes est la prise en compte des traitements. De plus, ces algorithmes ne traitent pas les concepts élaborés comme les attributs de types complexes qui, pour notre part, nous ont demandé une réflexion et un travail importants. Notons que nos règles ne prennent pas en compte l'implémentation des transactions. En effet, cette dernière nécessite l'intervention de l'utilisateur et reste donc un processus manuel.

Un autre point que nous avons abordé, dans le cadre de cette thèse, est la traduction des composants terminaux **B** en langage SQL imbriqué dans le langage hôte **JAVA**. Cette traduction a permis de mettre en évidence une architecture à deux niveaux. Le premier niveau, dont une grande partie peut être générée automatiquement, est constitué d'un ensemble de classes **JAVA** qui correspondent aux données du système (tables relationnelles) et aux opérations de base associées. Le second contient le code associé aux transactions.

Pour établir la correction de notre processus de raffinement, nous avons établi la preuve de chacune des règles de raffinement proposées. Les preuves ont été réalisées en utilisant l'AtelierB. La preuve interactive de certaines règles a nécessité l'ajout de règles utilisateur que nous avons vérifiées à leur tour. Le caractère générique de ces règles de raffinement permet d'une part l'automatisation du processus de raffinement ainsi défini et, d'autre part, la réutilisation par instanciation de leurs preuves de correction. En effet, pour chaque règle, un scénario de preuve est défini, il suffit par la suite de l'instancier avec les données effectives de l'application à raffiner. Notons néanmoins que la correction de notre processus ne concerne que les étapes de raffinement effectuées dans l'AtelierB, la phase de traduction des composants terminaux en **SQL** n'est pas prouvée. En effet, cette dernière est réalisée en dehors de tout environnement formel. La fiabilité de cette phase de traduction peut être améliorée en formalisant les règles de traduction comme des règles de réécriture d'un système formel.

Un deuxième niveau de réutilisation des preuves que nous avons étudié est de définir comment il est possible de réutiliser les preuves de raffinement des opérations de base, préalablement établies, pour prouver le raffinement des transactions. Les transactions sont des substitutions construites à partir de substitutions correspondant aux opérations de base via les différents opérateurs **B** (Précondition, IF, Parallèle, etc). Nous avons donc procédé en deux temps. Dans un premier lieu, nous avons raisonné par induction sur la structure de la transaction à raffiner. Pour chaque opérateur **B**, nous avons défini les conditions suffisantes à la réutilisation des preuves de raffinement des substitutions qu'il relie pour l'établissement de la substitution globale. Les résultats obtenus pour le cas d'un opérateur ont été généralisés pour le cas de raffinement de substitutions composées de plusieurs opérateurs. Enfin, nous avons montré que dans le domaine des systèmes d'information les conditions de réutilisation de preuves sont pratiquement toujours vérifiées, d'où l'intérêt de notre approche.

Afin de rendre notre approche opérationnelle, nous avons développé un prototype qui permet sa mise en oeuvre. Cet outil, réalisé en OCaml, procède en deux étapes:

– génération de spécifications **B** à partir des diagrammes **UML**. Les diagrammes **UML** sont définis en utilisant l'atelier de modélisation graphique **ROSE**[58],

– raffinement des spécifications **B** obtenues.

1.2 Plan de la thèse

Cette thèse est organisée en 8 chapitres. La première partie du deuxième chapitre présente un état de l'art général sur les différentes techniques de synthèse de programmes. La deuxième partie est consacrée à une étude détaillée et commentée des différents travaux existants dans le développement d'applications base de données.

Le chapitre 3 précise le type de spécifications abstraites **B** que nous considérons. Ce chapitre décrit la traduction en spécifications **B** des diagrammes **UML** spécialisés pour le domaine des systèmes d'information.

Les chapitres 4, 5 et 6 constituent le noyau de la thèse. Le chapitre 4 présente un ensemble de règles de raffinement permettant de s'approcher petit à petit d'une implémentation relationnelle. Un bilan sur la phase de preuve de ces règles est également présenté. Le chapitre 5, suite du chapitre 4, présente un ensemble de règles d'implémentation qui permettent la génération d'une implémentation relationnelle. Ce chapitre aborde aussi la traduction de l'implémentation obtenue en **SQL** imbriqué dans le langage hôte **JAVA**.

Le chapitre 6 étudie la stratégie de réutilisation de preuves dans le processus de correction de raffinements **B**. Le but de ce chapitre est la définition des conditions suffisantes qui permettent de réutiliser les preuves de raffinements, établies aux deux précédents chapitres, pour l'établissement de la correction du raffinement des transactions utilisateur.

Le chapitre 7 présente l'implémentation de l'outil qui permet l'utilisation effective de l'approche que nous avons développée.

Enfin, nous terminons par une conclusion sur notre contribution au développement formel d'applications base de données et esquissons les principales perspectives.

Chapitre 2

État de l'art sur la synthèse de programmes

2.1 Introduction

La programmation désigne le processus d'élaboration de programmes pour répondre à des problèmes donnés. Cette discipline est apparue avec l'arrivée de l'informatique. Dès le départ, le souci des informaticiens était de fournir des langages de programmation qui seraient les plus indépendants possible de la machine. Les langages de programmation ont évolué des suites de 0 et 1 jusqu'au développement des langages évolués. Bien que qualifiés de haut niveau, ces langages sont limités par l'aspect calculatoire ou décidable de leurs concepts. Les programmes produits ne peuvent être validés que par jeux de tests et/ou l'instrumentation des programmes par des instructions dites "code mort" qui ne doivent jamais être exécutées pour un programme correct. Or, de telles approches ne permettent pas d'affirmer de manière formelle et sûre la correction du programme à valider. Outre le problème de fiabilité des programmes produits, le coût de conception de ces programmes est souvent très élevé, leur évolution et maintenance sont ardues, longues et coûteuses.

L'une des principales réponses au problème de fiabilité des programmes est la synthèse automatique de programmes. Elle désigne le processus de construction de programmes à partir de la spécification formelle de ces derniers. De manière générale, la synthèse de programmes désigne l'ensemble des approches visant l'automatisation totale ou partielle du processus de programmation. L'idée de base de cette discipline est d'élever au plus haut niveau possible le degré d'abstraction des langages de programmation. Ce haut niveau d'abstraction permet d'une part, de rendre la connaissance de l'utilisateur pour les détails d'implémentation la moins importante possible, et d'autre part, elle offre l'opportunité de développement d'outils d'analyse et de traitement des programmes ainsi obtenus.

La synthèse automatique de programmes est basée sur des langages formalisés utilisant des notions mathématiques souvent élémentaires. Contrairement aux langages classiques de programmation, ces langages sont complètement indépendants de toute considération relative à la machine. Le but de la programmation automatique est de fournir une expression purement mathématique de la solution du problème à résoudre. Dans un premier temps, on s'intéresse plus à décrire ce que le programme doit accomplir sans préciser de détails sur la façon dont cette solution va être concrètement mise en œuvre. Dit autrement, on spécifie d'abord le "Quoi" du programme, le problème du "Comment" qui évoque plus des détails sur la stratégie de calcul est reporté à une étape ultérieure du développement. De plus, les notions mathématiques étant bien définies et de sémantique précise et claire, des générateurs de programmes peuvent être réalisés. Ces générateurs assimilables à des super-compilateurs, sont évidement plus difficiles à concevoir que les compilateurs des langages de programmation. En effet, leur implémentation nécessite l'intégration de nombreuses connaissances du domaine.

Dans la littérature, plusieurs approches de synthèse de programmes ont été développées. Ces approches peuvent être classées selon trois critères :

– Généralité du système : le langage de spécification utilisé et le générateur de programmes associé sont-ils dédiés ou non à un domaine d'applications spécifique?

– Degré d'automatisation du système : le générateur de programmes est-il automatique ou nécessite t-il l'intervention de l'utilisateur?

  En pratique, les systèmes dédiés sont évidement les plus fréquents étant plus faciles à construire. Le domaine d'application de ces systèmes étant bien ciblé, ils englobent une grande partie des connaissances du domaine qui leur permet souvent une génération de code optimale et automatique. On trouve ce type de systèmes en particulier dans les domaines des bases de données, des interfaces graphiques (générateur d'écrans, ...), etc. Par opposition à ces systèmes, les systèmes généraux sont plus complexes et donc peu fréquents. Les programmes générés par de tels systèmes ne sont pas connus à l'avance, et sont obtenus principalement par interaction avec l'utilisateur. On parle dans ce cas plus d'*assistants à la programmation* que de générateurs de programmes. Le système *Descartes* [27], développé à la Direction des Études et Recherche de l'Électricité de France, est l'un des rares générateurs de programmes non dédiés à un domaine spécifique d'applications, et le premier qui soit entièrement automatique.

– Méthode ou technique de développement utilisée : les méthodes de construction de systèmes de programmation automatiques peuvent être groupées en trois familles :

  1. les méthodes déductives (constructives, approches par résolution) : le programme est construit par l'établissement de la preuve de correction de sa spécification. L'idée de base de cette technique est que construire un programme est équivalent à la détermination constructive de la preuve de sa spécification. La spécification représentant les propriétés que le programme doit remplir est représentée par une formule logique. Un démonstrateur de théorèmes est utilisé pour l'élaboration de sa preuve. Bien que cette méthode ait l'avantage d'être extrêmement générale et applicable à n'importe quel domaine, celle-ci est limitée par la qualité des démonstrateurs utilisés. En effet, d'une part le processus de recherche de la preuve est, par nature, infini et d'autre part, même si une solution est engendrée, rien ne garantit que celle-ci soit la meilleure (optimale). Cette technique a été, entre autres, utilisée dans les travaux de Smith [69] et Qian [57]. Le travail de Smith concerne la construction d'un système interactif de développement de programmes corrects à partir d'une spécification formelle (*KIDS*). Ce système implémente plusieurs opérations telles la conception algorithmique, le raffinement, l'optimisation, la déduction etc. À l'exception de la première opération qui nécessite l'intervention de l'utilisateur, toutes les autres opérations sont automatiques. Le travail de Qian concerne la synthèse de transactions base de données sûres à partir d'une spécification logique de la propriété qui lui est associée. Ce travail est présenté en détails dans la section 2.5.
  2. les méthodes inspectionnelles : ces méthodes consistent à identifier des schémas formels de spécification dits "clichés' et à sélectionner, dans un catalogue d'implémentation, l'implémentation correspondant au cliché. L'avantage de ces méthodes est qu'elles sont proches du mécanisme de raisonnement du programmeur qui consiste à se ramener à des cas plus connus pour lesquels on dispose déjà d'une solution. Ces techniques sont une sorte de réutilisation des connaissances antérieurement acquises. Elles sont particulièrement efficaces pour le cas de problèmes relativement simples. Leur application à des cas complexes nécessite leur combinaison avec des méthodes déductives ou transformationnelles. Le principal système utilisant cette technique est "The Programmer Apprentice" de Rich et Waters au laboratoire *MIT* d'intelligence artificielle [59]. Ce système est un assistant intelligent au développement de programmes utilisant une base de connaissances du domaine d'applications. Cette base comporte, par exemple, un ensemble d'algorithmes ou de structures de données couramment utilisés dans le domaine en question. La technique inspectionnelle est aussi largement utilisée dans le domaine de la rétro-conception et de la compréhension de programmes.
  3. les méthodes transformationnelles (méthodes par raffinement) : ces méthodes consistent à effectuer une série de transformations graduelles sur une spécification afin de la rapprocher le plus possible

du langage d'implémentation choisi. Ces méthodes sont très générales et de plus en plus utilisées actuellement. Dans ces méthodes, deux types de transformations sont à distinguer : transformations horizontales et transformations verticales. Le premier type vise à optimiser le programme généré, le second permet de décroître d'un cran le niveau d'abstraction de la spécification. La correction des programmes générés par ces méthodes est basée sur la correction de chaque étape de transformation mais aussi sur la correction de la spécification abstraite initiale.

Dans le cadre de ce travail, on se propose d'utiliser cette technique de synthèse de programmes pour le développement d'applications base de données à partir de spécifications formelles **B**. Des travaux similaires ont été développés en **Z** [5, 6, 19, 20] et **B** [29, 66, 77].

4. la théorie des catégories : l'idée principale est d'utiliser les catégories (théorie mathématique d'ordre supérieur) comme langage de spécification. Un ensemble d'étapes de raffinement et de composition est alors appliqué pour la génération du code associé. L'utilisation des catégories permet une description minimale des entités du système en mettant en évidence les propriétés externes les plus importantes. Cette description est souvent graphique représentée par un ensemble d'objets (nœuds) et de liens (arcs entre objets). Cette technique a été utilisée dans le système *SPECWARE* [71, 72] de l'institut de *Kestrel*.

Le travail présenté dans ce manuscrit rentre dans le cadre de la synthèse d'applications base de données sûres. Nous utilisons une approche par raffinement de spécifications formelles **B**. Afin de situer notre contribution par rapport aux différents travaux réalisés dans ce contexte, nous présentons dans les sections qui suivent une étude détaillée des travaux représentatifs existants dans la littérature.

La suite de ce chapitre est organisée comme suit. La section suivante décrit une approche par raffinement pour la dérivation d'une implémentation **SQL** à partir d'une spécification formelle **Z**. La section 2.3 présente un processus formel similaire au précédent qui permet la dérivation d'une implémentation **DBPL** à partir d'une spécification formelle **TDL**. Un processus de traduction en **DBPL** de spécifications **Z** représentant la description des différents concepts relationnels est étudié dans la section 2.4. La section 2.5 étudie une approche déductive de développement de transactions base de données sûres. Pour chaque travail, nous donnons les points forts et les limites. Nous terminons par des commentaires généraux sur les différents travaux étudiés et une conclusion.

2.2 Dérivation formelle d'une implémentation relationnelle à partir d'une spécification Z

Dans [19, 20] une approche de dérivation d'une implémentation relationnelle à partir d'une spécification **Z** a été présentée. L'approche proposée est basée sur la technique de raffinement. Le système étudié est représenté par deux spécifications **Z** correspondant aux deux niveaux abstrait et concret de description, un troisième schéma **Z** spécifiant la liaison de ces deux niveaux est établi. Ce schéma de liaison permet principalement d'établir la correction du schéma concret par rapport au schéma abstrait.

La figure 2.1 montre le lien entre les vues abstraite et concrète que l'utilisateur peut avoir d'un système donné. Elle illustre la démarche usuelle de conception d'un système. L'utilisateur commence par une description conceptuelle qu'il fait évoluer jusqu'à l'obtention d'une spécification exécutable. Cette spécification exécutable peut être perçue comme une simulation de la vue conceptuelle préalablement définie.

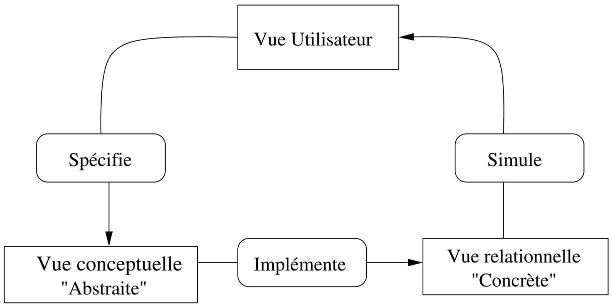


FIG. 2.1 – *Vues conceptuelle et relationnelle d'un système*

Les étapes constituant cette approche sont les suivantes :

– établissement des schémas abstraits **Z** représentant la vue conceptuelle du système. Cette vue concerne aussi bien les données que les opérations traduisant les fonctionnalités du système (cf. section 2.2.1),
– établissement du schéma concret de données **Z** représentant la vue relationnelle du système. Ce schéma constitue le raffinement du schéma abstrait établi à la phase précédente (cf. section 2.2.2).
– établissement du schéma d'abstraction qui permet l'expression du lien existant entre les données abstraites et concrètes (cf. section 2.2.3).
– réécriture des opérations abstraites en fonction des données concrètes (cf. section 2.2.4).
– établissement des preuves de correction des opérations concrètes par rapport aux opérations abstraites (cf. section 2.2.5).

Les sous-sections suivantes illustrent cette démarche de développement à travers un système de gestion d'inscription et de notation d'un ensemble d'étudiants.

2.2.1 Vue conceptuelle du système : spécification abstraite

Le système à implémenter concerne un ensemble d'étudiants. Un étudiant est décrit par un identifiant, un nom et un prénom. Un étudiant peut s'inscrire à plusieurs thèmes. Une inscription à un thème est décrite par une date. Une note est éventuellement attribuée à chaque thème pour lequel l'étudiant est inscrit.

Une vue assez globale de cette description informelle, est de percevoir un étudiant comme un objet complexe décrit par les caractéristiques citées ci-dessus. En **Z**, on obtient le schéma suivant (on suppose que la notation est faite sur une échelle de 0 à 100) :

$$\begin{array}{|l}
\hline
\textit{étudiant} \\
\textit{id} : \textit{Étudiant} \\
\textit{nom}, \textit{prénom} : \textit{Nom} \\
\textit{inscrit} : \textit{Thème} \pfun \textit{Date} \\
\textit{note} : \textit{Thème} \pfun 0..100 \\
\hline
\end{array}$$

où :

– *Étudiant* : désigne l'ensemble des étudiants possibles,
– *Nom* : désigne l'ensemble des noms et prénoms possibles,
– *Date* : désigne l'ensemble des dates valides possibles,
– *Thème* : désigne l'ensemble des thèmes possibles.

De la même manière, un thème est décrit par un titre, une date limite d'inscription et un coefficient. L'ensemble des thèmes est spécifié par le schéma **Z** suivant :

$$
\begin{array}{|l}
\textit{enseignement} \\
\hline
titre : Thème \rightarrowtail\!\!\!\!\!+ Nom \\
date\_lim : Thème \rightarrowtail\!\!\!\!\!+ Date \\
coef : Thème \rightarrowtail\!\!\!\!\!+ 0..100 \\
\hline
dom\ coef = dom\ date\_lim = dom\ titre \\
\hline
\end{array}
$$

Les deux schémas spécifiés ci-dessus sont utilisés pour définir l'état de notre système :

$$
\begin{array}{|l}
\textit{Système} \\
\hline
étudiants : \mathbb{P}(étudiant) \\
enseignement \\
\hline
\forall s,t : étudiants \bullet s.id = t.id \Rightarrow s = t \\
\forall s : étudiants \bullet dom\ s.inscrit \subseteq dom\ titre \\
\forall s : étudiants \bullet dom\ s.note \subseteq dom\ inscrit \\
\hline
\end{array}
$$

Les trois prédicats spécifient que :
- le champ id identifie chaque étudiant de manière unique : deux étudiants différents ne peuvent avoir les mêmes identifiants,
- un étudiant ne peut s'inscrire qu'aux thèmes répertoriés dans le système,
- un étudiant ne peut être noté que pour les thèmes auxquels il est inscrit.

Sur ce système est spécifiée l'opération *Affecte_Note* qui affecte une note à un étudiant (non encore noté) pour un thème donné. Cette opération est spécifiée par le schéma d'opération suivant :

$$
\begin{array}{|l}
\textit{Affecte\_Note} \\
\hline
\Delta Système \\
\Xi enseignement \\
e? : Étudiant \\
t? : Thème \\
n? : 0..100 \\
\hline
\exists\, e : étudiants \bullet \\
\qquad e? = e.id \\
\qquad t? \in dom(e.inscrit) \\
\qquad t? \notin dom(e.note) \\
\qquad \exists\, e' : étudiant \bullet \\
\qquad\quad e'.id = e.id \\
\qquad\quad e'.nom = e.nom \\
\qquad\quad e'.prénom = e.prénom \\
\qquad\quad e'.inscrit = e.inscrit \\
\qquad\quad e'.note = e.note \cup \{(t?,n?)\} \\
\qquad\quad étudiants' = étudiants - \{e\} \cup \{e'\} \\
\hline
\end{array}
$$

Cette opération reçoit trois paramètres en entrée : un identifiant ($e?$), le thème ($t?$) et la note ($n?$). Après avoir vérifié qu'il existe un étudiant e ayant pour identifiant $e?$ ($e? = e.id$), inscrit au thème $t?$ ($t? \in$

$dom(e.inscrit)$) et n'ayant pas encore eu de note pour ce thème ($t? \notin dom(e.note)$), le système doit se trouver dans un état tel que :

- il existera un étudiant e' ayant les mêmes identifiant, nom, prénom et inscrit aux mêmes thèmes que e,
- la note de l'étudiant e' pour le thème t est n,
- l'ensemble des étudiants existants privé de e inclut e'

2.2.2 Vue relationnelle du système : spécification concrète

La spécification concrète d'un système est sa description d'un point de vue réalisation ou implémentation. Ce point de vue est donc fortement lié au langage d'implémentation choisi. Ayant choisi une implémentation relationnelle de ce système d'information, il faudrait donc exhiber les différentes tables permettant la réalisation du système. Deux types de tables sont à distinguer :

1. les tables décrivant les entités du système ayant une existence propre. Ces tables correspondent aux classes **UML**,
2. les tables modélisant les liens entre les entités exhibées précédemment. Ces tables correspondent aux associations **UML**.

Le système étudié met en œuvre trois tables. Les deux premières tables permettent de décrire les informations relatives à un étudiant et à un thème respectivement, la troisième représente les informations relatives à un étudiant pour un thème donné. Ces tables sont spécifiées en **Z** comme suit :

- *ÉtTab* : cette table représente un étudiant décrit par un identifiant, un nom et un prénom. Seules les informations liées directement à l'étudiant sont reprises dans cette table.

$$
\begin{array}{|l}
\hline
\textit{ÉtTab} \\
Id : \textit{Étudiant} \\
Nom, Prénom : Nom \\
\hline
\end{array}
$$

- *ThTab* : cette table représente un thème. En plus, des informations décrivant un thème, un champ numéro (Num) est ajouté. En effet, dans le modèle relationnel, chaque table doit posséder un champ identifiant chaque tuple de la table. Cette contrainte de clé est spécifiée dans le schéma global du système (*Système*$_{DB}$).

$$
\begin{array}{|l}
\hline
ThTab \\
Num : Thème \\
Titre : Nom \\
Date\_lim : Date \\
Coef : 0..100 \\
\hline
\end{array}
$$

- *ResTab* : cette table représente les informations relatives à un étudiant et à un thème donné : la date d'inscription de l'étudiant au thème et sa note. La note d'un étudiant non noté est $null$.

$$
\begin{array}{|l}
\hline
ResTab \\
Id : \textit{Étudiant} \\
Num : Thème \\
Inscrit : Date \\
Note : 0..100 \mid null \\
\hline
\end{array}
$$

Ces trois définitions de tables sont utilisées dans le schéma décrivant l'état du système :

$$
\begin{array}{|l}
\textit{Système}_{DB} \\
\hline
\textit{Étudiants} : \mathbb{P}(\textit{ÉtTab}) \\
\textit{Thèmes} : \mathbb{P}(\textit{ThTab}) \\
\textit{Résultats} : \mathbb{P}(\textit{ResTab}) \\
\hline
\forall S,T : \textit{Étudiants} \bullet S.Id = T.Id \Rightarrow S = T \\
\forall A,B : \textit{Thèmes} \bullet A.Num = T.Num \Rightarrow A = B \\
\forall R,S : \textit{Résultats} \bullet R.Id = S.Id \wedge R.Num = S.Num \Rightarrow R = S \\
\{R : \textit{Résultats} \bullet R.Id\} \subseteq \{S : \textit{Étudiants} \bullet S.Id\} \\
\{R : \textit{Résultats} \bullet R.Num\} \subseteq \{S : \textit{Étudiants} \bullet S.Num\} \\
\hline
\end{array}
$$

La partie prédicat de ce schéma spécifie les contraintes suivantes :

- le champ Id du schéma *ÉtTab* est une clé.
- le champ Num du schéma *ThTab* est une clé.
- le couple de champs (*Id*, *Num*) constitue une clé pour le schéma *ResRab*.
- chaque valeur du champ *Id*, de la table *ResTab*, désigne une valeur du champ *Id* de la table *ÉtTab*.
- chaque valeur du champ Num, de la table *ResTab*, désigne une valeur du champ *Num* de la table *ThTab*.

Les deux dernières contraintes correspondent aux contraintes référentielles.

2.2.3 Schéma de liaison des vues concrète et abstraite

Le but de cette étape est de définir la relation existante entre les deux vues abstraite et concrète du système. Cette relation, exprimée dans un schéma d'abstraction, est utile pour l'établissement de la correction de la spécification concrète par rapport à la spécification abstraite. Par exemple, le lien entre les spécifications *Système* et *Système*$_{DB}$ est décrit par le schéma d'abstraction suivant :

$$
\begin{array}{|l}
\textit{Liaison} \\
\hline
\textit{Système} \\
\textit{Système}_{DB} \\
\hline
\textit{étudiants} = \\
\qquad\qquad \{e : \textit{étudiant} \mid \\
\qquad\qquad\quad \exists E : \textit{Étudiants} \bullet \\
\qquad\qquad\quad e.id = E.Id \\
\qquad\qquad\quad e.nom = E.Nom \\
\qquad\qquad\quad e.prénom = E.Prénom \\
\qquad\qquad\quad e.inscrit = \{R : \textit{Résultats} \mid R.Id = E.Id \bullet (R.Id, R.Inscrit)\} \\
\qquad\qquad\quad e.note = \{R : \textit{Résultats} \mid R.Id = E.Id \wedge R.Note \neq null \bullet (R.Id, R.Inscrit)\}\} \\
titre = \{T : \textit{Thèmes} \bullet (T.Num, T.titre)\} \\
date\_lim = \{T : \textit{Thèmes} \bullet (T.Num, T.Date\_lim)\} \\
coef = \{T : \textit{Thèmes} \bullet (T.Num, T.Coef)\} \\
\hline
\end{array}
$$

Le schéma de liaison exprime la façon dont les variables de la spécification abstraite sont réalisées par les variables de la spécification concrète. Les quatre prédicats du schéma précédent expriment les variables abstraites *étudiants*, *titre*, *date_lim* et *coef* en fonction de l'espace des variables concrètes présentes dans le schéma concret *Système*$_{DB}$.

2.2.4 Réécriture de l'opération *Affecte_Note*

Les variables abstraites de la spécification étant remplacées par de nouvelles variables concrètes, les opérations de la spécification abstraite doivent être à leur tour remplacées par des opérations concrètes. En effet, les variables abstraites sur lesquelles agissaient ces opérations n'existent plus. Chaque opération abstraite est donc réécrite en fonction des variables concrètes. L'opération abstraite *Affecte_Note* est réécrite comme suit :

$$
\begin{array}{|l}
\multicolumn{1}{l}{Affecte\_Note_{EXE}} \\
\Delta Syst\grave{e}me_{DB} \\
e? : \acute{E}tudiant \\
t? : Th\grave{e}me \\
n? : 0..100 \\
\hline
\exists\, R : R\acute{e}sultats \bullet \\
\qquad R.Id = e?\ R.Num = t?\ R.Note = null \\
\qquad \exists\, R' : ResTab \bullet \\
\qquad\qquad R'.Id = R.Id\ R'.Num = R.Num\ R'.Date\_lim = R.Date\_lim \\
\qquad\qquad R'.Note = n? \\
\qquad\qquad R\acute{e}sultats' = R\acute{e}sultats - \{R\} \cup \{R'\} \\
\acute{E}tudiants' = \acute{E}tudiants \\
Th\grave{e}me' = Th\grave{e}me \\
\hline
\end{array}
$$

2.2.5 Preuve du développement

La correction d'un développement par réécriture (raffinement) est assurée par l'établissement d'un ensemble d'obligations de preuves. Dans ce travail, l'auteur utilise la sémantique relationnelle du raffinement. Un raffinement est correct si les trois conditions suivantes sont satisfaites :

- **Condition d'applicabilité** : si la précondition de l'opération abstraite et le prédicat du schéma de liaison sont vérifiés alors la précondition de l'opération concrète doit être aussi vérifiée :

  $$\forall\, C\acute{E}tat;\ A\acute{E}tat;\ x? : X \bullet pre\ Aop \wedge Abs \Rightarrow pre\ Cop$$

  où :
  - $A\acute{E}tat$: désigne le schéma abstrait,
  - $C\acute{E}tat$: désigne le schéma concret,
  - $x?$: désigne les paramètres d'entrée de l'opération abstraite,
  - $pre\ Aop$: désigne la précondition de l'opération abstraite,
  - $pre\ Cop$: désigne la précondition de l'opération concrète,
  - Abs : désigne le schéma d'abstraction définissant la liaison du schéma concret au schéma abstrait.

  Cette condition correspond à l'affaiblissement de la précondition par raffinement.
- **Correction** : si la précondition de l'opération abstraite est satisfaite et que la post-condition de l'opération concrète l'est aussi alors la post-condition de l'opération abstraite doit être aussi satisfaite :

  $$\forall\, A\acute{E}tat;\ A\acute{E}tat';\ C\acute{E}tat;\ C\acute{E}tat';\ x? : X \bullet pre\ Aop \wedge Abs \wedge Cop \wedge Abs' \Rightarrow Aop$$

  avec Aop et Cop désignant les post-conditions des opérations abstraite et concrète respectivement.
- **Faisabilité** : les états initiaux abstrait et concret doivent vérifier le schéma d'abstraction :

  $$\forall\, A\acute{E}tat;\ C\acute{E}tat \bullet Cinit \wedge Abs \Rightarrow Ainit$$

Afin de pouvoir établir ces preuves de raffinement, l'auteur a défini un ensemble de règles de réécriture lui permettant le passage de la spécification abstraite à la spécification concrète [20].

2.2.6 Commentaires

Ce travail présente une démarche de développement de systèmes d'information par raffinement. À un système donné, sont associées deux vues équivalentes correspondant aux niveaux abstrait (conceptuel) et concret (exécutable) de la modélisation. Le niveau abstrait, plus simple à construire, sert de référence de correction pour le niveau concret. Un ensemble de prédicats, exprimés dans un schéma de liaison, spécifie la relation existante entre les deux vues abstraite et concrète du système. Cette approche permet une génération prouvée des structures des tables ainsi que les transactions utilisateur représentant les fonctionnalités du système.

Néanmoins, cette approche présente quelques limites :

- cette approche ne peut servir que de guide méthodologique pour le développement d'applications bases de données. Elle ne spécifie pas comment les schémas abstrait et concret sont construits. Le degré d'abstraction du niveau abstrait n'est pas caractérisé. La vue aussi bien abstraite que concrète que peut avoir l'utilisateur d'un système n'est pas unique. De plus, l'exhibition des tables nécessaires à l'implémentation d'un système peut nécessiter un travail conséquent de conception.
- l'approche ne précise pas la façon dont le schéma de liaison est calculé. Ce schéma de liaison constitue le fondement de la correction de l'approche.
- la réécriture des opérations abstraites est basée sur l'intuition du concepteur. Aucune règle de réécriture des opérations abstraites n'est précisée.
- n'ayant pas de règles définies permettant la détermination du schéma concret et du schéma de liaison, la correction de cette démarche ne peut être établie de manière générale et générique. Les preuves de correction du raffinement doivent être établies pour chaque application.
- les systèmes d'information étant caractérisés par de gros volumes de données, l'exhibition des tables nécessaires à leur réalisation requiert plusieurs pas de raffinement. Une telle approche de raffinement en une passe conduit à un invariant de liaison très complexe qui rend les preuves de correction difficiles à établir.
- l'approche ne considère que des informations atomiques et monovaluées. Les concepts avancés tels que l'héritage, les attributs multivalués ne sont pas considérés.

2.3 Développement par raffinement de spécifications DBPL à partir de spécifications TDL

Cette approche de développement a été proposée dans les travaux complémentaires de Schewe [66] et Gunther [29]. Ces travaux ont été réalisés dans le cadre du projet *DAIDA* [77] qui avait pour objectif l'application de méthodes formelles pour le développement d'applications bases de données. Ce type de travaux rentre dans l'optique de réduction de l'écart, souvent important, entre les langages utilisés au niveau conceptuel et ceux utilisés au niveau implémentation du développement d'un projet. Le but des travaux présentés dans cette partie, est la dérivation d'une implémentation **DBPL** [68] à partir d'une spécification formelle **TDL** [11]. L'intérêt d'utiliser la méthode **B** est le processus de raffinement prouvé qu'elle offre.

La démarche proposée dans ces travaux se décompose en trois phases représentées par la figure 2.2.

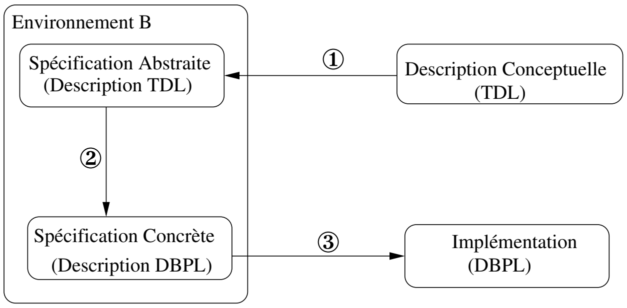


FIG. 2.2 – *Processus de développement*

1. traduction de la spécification **TDL** en une spécification **B** (cf section 2.3.2),
2. raffinement de la spécification **B** jusqu'à obtenir un niveau d'abstraction correspondant à une description **DBPL** (cf section 2.3.3),
3. traduction de la spécification concrète **B** en des spécifications **DBPL**. Cette traduction est réalisée via l'utilisation du langage formel algébrique de réécriture **OBJ** [28] (cf section 2.3.4).

2.3.1 Aperçu des langages TDL et DBPL

En **TDL**, les données sont décrites par un ensemble de classes liées par des attributs. **TDL** définit plusieurs types de classes : classes de base représentant les types de base (c'est la notion d'ensembles abstraits B), classes énumérées (c'est la notion d'ensembles énumérés de **B**), classes entités, etc. Les entités du système à étudier sont représentées par ces dernières. La catégorie d'un attribut précise si celui-ci est modifiable (*CHANGING*) ou non (*UNCHANGING*). Des contraintes additionnelles peuvent être spécifiées sur la classe : attribut à valeur unique (clé), attribut référençant une autre classe, etc.

L'aspect dynamique du système est décrit par un ensemble de transitions (opérations) défini sur les classes. Une transition est spécifiée dans un style de pré-post condition. Les paramètres d'entrée et de sortie sont introduits par les mots clés **IN** et **OUT** respectivement. La précondition et la post-condition de la transaction sont spécifiées par les mots clés **GIVEN** et **GOALS** respectivement. La figure 2.3 représente la description formelle **TDL** de la gestion d'un ensemble de projets. Nous distinguons dans ce schéma, une classe énumérée et trois classes entités : *Agence, Compagnie, Employé et Projet*. Chaque compagnie identifiée par son nom est engagée dans un ensemble variable de projets. Chaque employé, caractérisé par son nom, appartient à une seule compagnie et travaille sur plusieurs projets dans lesquels sa compagnie est engagée. Chaque projet est décrit par son nom et initié par une agence. Le couple d'attributs (*nom-du-projet, Agence-initiatrice*) permet d'identifier le projet. Un projet est suivi par les compagnies engagées dans ce dernier. On spécifie dans cette description l'opération de recrutement d'un employé.

```
TDLDESIGN GestionProjet IS
  ENUMERATED CLASS Agence={'ESPRIT,'DFG,'NFS ... }

  ENTITY CLASS Compagnie WITH
    UNIQUE, UNCHANGING nom: String;
    CHANGING engagée: SetOf Projet;
  END Compagnie
```

```
  ENTITY CLASS Employé WITH
    UNIQUE, UNCHANGING nom: String;
    CHANGING appartient: Compagnie; travaille: SetOf Projet;
    INVARIANT onEmpProj: True IS
        (THIS.travaille ⊆ THIS.appartient.engagée)
  END Employé

  ENTITY CLASS Projet WITH
    UNIQUE, UNCHANGING nom: String; initié: Agence;
    CHANGING consortium: SetOf Compagnie;
    INVARIANT onProjComp: True IS
        (THIS.consortium={EACH x ∈ Compagnie: THIS ∈ x.engagée} )
  END Projet
  TRANSACTION CLASS RecruterEmployé WITH
    IN no: String; appar: Compagnie, trav: SetOf Projet;
    OUT PRODUCES e′: Employé;
    GOALS (e′.nom=no) AND (e′.travaille=trav) AND (e′.appartient=appar);
  END RecruterEmployé
END GestionProjet
```

FIG. 2.3 – *Exemple de spécification TDL*

DBPL est un langage de programmation de bases de données relationnelles. Ce langage est basé sur une extension du modèle relationnel intégré dans le langage de programmation Modula-2. Il est plus puissant que le langage **SQL**. Le concept de relation défini dans ce langage permet la définition de relations imbriquées. Il permet aussi l'encapsulation des données dans des structures de modules tout en gérant l'accès concurrent à ces dernières. Les changements d'états de la base de données sont exprimés par des transactions atomiques. La figure 2.4 représente la spécification **DBPL** de l'exemple de gestion des projets.

```
DEFINITION MODULE GestionProjetModule
  IMPORT     Identifier, String;
  TYPE
    Agence = (ESPRIT, DFG,NFS,. . . );
    NomCom, NomEmp, NomPro=String.Type;
    IdEmployé=Identifier.Type
    ProjetIdRecType=RECORD nompro:NomPro, initié: Agence END;
    ProjetIdRelType=RELATION OF ProjetIdRecType ;
    CompagnieRelType=RELATION nomcom OF RECORD
                 nomcom:NomCom, engagée: ProjetIdRelType END;
    EmployéRelType= RELATION employé OF RECORD employé:IdEmployé, nomemp: NomEmp,
             appartient:NomCom, travaille: ProjetIdRelType END;
    ProjetRelType=RELATION projetId OF RECORD
             projId:ProjetIdRecType; consortium: RELATION OF NomCom END;
TRANSACTION RecruterEmployé(no:NomEmp; appar:NomCom; trav: ProjetIdRel): IdEmployé;
```

```
END GestionProjetModule

DATABASE IMPLEMENTATION MODULE GestionProjetModule
  IMPORT Identifier
  VAR comRel: CompagnieRelType; empRel: EmployéRelType; projRel:ProjetRelType;
  TRANSACTION RecruterEmployé
            (no:NomEmp; appar:NomCom; trav: ProjIdRelType): IdEmployé;
    VAR tIdEmp:IdEmployé;
    BEGIN
      IF SOME c IN compRel (c.nomcom=appar) AND
        ALL w IN trav(SOME p IN compRel[appar].engagée(w=p))
      THEN tIdEmp:=Identifier.new;
        empRel: + EmpRelType{{tIdEmp,no,appar,trav}};
        RETURN tIdEmp
      ELSE RETURN Identifier.Nil
      END
    END RecruterEmployé;
END GestionProjetModule
```

FIG. 2.4 – *Spécification DBPL correspondant à la gestion des projets*

2.3.2 Traduction d'une description TDL en une spécification abstraite B

Une spécification **TDL** est traduite en **B** comme suit : chaque classe de base ou énumérée est traduite par un ensemble abstrait ou énuméré **B**. Chaque classe entité e est traduite par un ensemble abstrait E et une variable d'état e qui est sous ensemble de E ($e \subseteq E$). Chaque attribut a d'une classe entité e est traduit par une variable d'état a définie comme une fonction totale de a dans le type de cet attribut.

Un attribut de catégorie **UNIQUE** est traduit par un invariant. Le caractère modifiable de l'attribut est ignoré en considérant que toute variable d'état est modifiable. La caractère non modifiable considéré comme une propriété dynamique n'est pas spécifié en **B**. Les invariants et transactions **TDL** sont traduits par des invariants et opérations **B** respectivement. La spécification **B** traduisant la description **TDL** de la figure 2.3 est représentée par la figure 2.5.

```
MACHINE    GestionProjet
SETS       Agence={ESPRIT, EDF, NFS, ...}; Compagnie; Employé; Projet;
           NomCom, NomEmp, NomPro=STRING
VARIABLES
  compagnie, nomcom, engagée,employé, nomemp, appartient, travaille, projet, nompro, initié, consortium
INVARIANT
  compagnie ⊆ Compagnie ∧ nomcom ∈ compagnie → NomCom ∧ engagée ∈ compagnie → P(projet) ∧

  employé ⊆ Employé ∧ nomemp ∈ employé → NomEmp ∧ appartient ∈ employé → compagnie ∧
  travaille ∈ employé → P(projet) ∧
```

projet $\subseteq$ *Projet* $\wedge$ *nompro* $\in$ *projet* $\rightarrow$ *NomPro* $\wedge$ *initié* $\in$ *projet* $\rightarrow$ *Agence*

$\forall x{,}y.x{,}y \in compagnie \Rightarrow (nomcom(x) = nomcom(y) \Rightarrow x = y)$
$\forall x.x \in$ *employé* $\Rightarrow (travaille(x) \subseteq$ *engagé(appartient(x))*
$\forall$ *x,y. x,y* $\in$ *projet* $\Rightarrow$ *(nompro(x)=nompro(y)* $\wedge$ *initié(x)=initié(y)* $\Rightarrow$ *x=y)*
$\forall$ *x. x* $\in$ *projet* $\Rightarrow$ *(consortium(x)={y* $\mid$ *y* $\in$ *compagnie* $\wedge$ *x* $\in$ *engagée(y)}*
OPERATION
emp $\longleftarrow$ **RecruterEmployé(no,appar,trav)**
PRE *no* $\in$ *STRING* $\wedge$ *appar* $\in$ *compagnie* $\wedge$ *trav* $\in$ *P(engagée(appar))*
THEN[1] 1
(@ *e* • *(e* $\in$ *(Employé -employé))* $\longrightarrow$
nomemp(e), travaille(e), appartient(e):=no, trav, appar $\|$
employé := employé $\cup$ *{e}* $\|$ *emp:=e)*
END

FIG. 2.5 – *Spécification abstraite B correspondante à la description TDL*

2.3.3 Étapes de raffinement

L'auteur propose quatre règles de raffinement pour le passage d'une spécification abstraite **B** à une spécification concrète relationnelle (**DBPL**).

– Étape 1 : Identification des objets
Cette étape permet d'exhiber un identificateur pour chaque classe et de passer d'un modèle orienté objets (qui est **B**) à un modèle orienté valeurs (qui est **DBPL**). Formellement, l'auteur définit cette règle comme suit : à chaque variable a, associée à l'ensemble abstrait A, s'il existe des variables $(y_1, \ldots, y_n)$ satisfaisant l'invariant :

$$\bigwedge_{i=1..n}(y_i \in a \rightarrow B_i) \wedge \forall x_1{,}x_2.(x_1 \in a \wedge x_2 \in a \Rightarrow (\bigwedge_{i=1..n}(y_i(x_1) = y_i(x_2)) \Rightarrow x_1 = x_2))$$

alors une variable a' vérifiant l'invariant de collage $a' \subseteq B_1 \times \ldots B_n \wedge \forall x.(x \in a' \Rightarrow y(x) = x)$ est introduite par le raffinement. De plus l'ensemble A est remplacé par un nouvel ensemble abstrait $(B_1 \times \ldots B_n)$.
Si une telle variable y n'existe pas (pas de clé définie sur la classe), on introduit un nouvel ensemble abstrait B et une variable abstraite y qui définit une clé de type B sur A. On se ramène ainsi au premier cas. L'application de cette étape à la spécification abstraite donne :

SETS *IdEmployé*; *Compagnie=NomCom, Employé=IdEmployé, Projet=NomPro* $\times$ *Agence*
VARIABLES *compagnie′, employé′, idemployé, projet′, …*
CHANGE *compagnie= compagnie′*; *employé′=employé*; *projet=projet′*
INVARIANT
compagnie′ $\subseteq$ *NomCom* $\wedge$ $\forall$ *x.(x* $\in$ *NomCom* $\Rightarrow$*(nomcom(x)=x))* $\wedge$
idemployé $\in$ *employé* $\rightarrow$ *IdEmployé* $\wedge$ *employé′* $\subseteq$ *IdEmployé* $\wedge$ $\forall$ *x. (x* $\in$ *employé* $\Rightarrow$ *idemployé(x)=x)*
Projet′ $\subseteq$ *Projet* $\wedge$
$\forall$ *x.(x* $\in$ *Projet′* $\wedge$ $\exists$ x_1, x_2. *x=*$(x_1{,}x_2)$ $\Rightarrow$ *nompro(x)=*x_1 $\wedge$ *initié(x)=*x_2*)*

1. (@ *x* • *P* $\longrightarrow$ *S*)$\equiv$ **ANY** x **WHERE** P **THEN** S.

– Étape 2 : Génération de l'identificateur
Cette étape a pour but d'éliminer l'indéterminisme introduit par la substitution abstraite **ANY**. Formellement, la substitution @ z • ($z \in Id \longrightarrow S(z)$) est remplacée par : ($z \longleftarrow NewId();S(z)$). *NewId* est une opération sans paramètres d'entrée générant l'identificateur z en sortie. Par cette règle, l'opération *RecruterEmployé* est raffinée par (après passage à la séquence) :

emp ⟵ **RecruterEmployé(no,appar,trav)**
PRE *no* ∈ *STRING* ∧ *appar* ∈ *compagnie* ∧ *trav* ∈ *P(engagée(trav))*
THEN
e ⟵ *NewEmpId*;
employé′=employé′ ∪ *{e}*; *Idemp(e)=e*;
nomemp(e):=no; *travaille(e):=trav*; *appartient(e):=appar*;
emp:=e
END

– Étape 3 : Définition des structures des tables
Cette étape permet le passage d'un modèle fonctionnel à un modèle ensembliste. Formellement, étant donné l'ensemble des variables $a,y_1,\ldots,y_n$ satisfaisant la contrainte :

$$a \subseteq A \wedge \bigwedge_{i=1..n} y_i \in a \rightarrow B_i$$

cette règle remplace les variables y_i par l'unique variable y définie par (le tuple de variables $(y_1,\ldots,y_m)$ désignant la clé) :

$$y \in b_1 \times \ldots \times b_m \rightarrow B_{m+1} \times \ldots \times B_n$$

avec : b_i=*ran*(y_i) et y_i= *prj*$^2{}_{i-m} \circ y$
L'application de cette règle aux variables *nomemp*, *appartient* et *travaille* donne :

VARIABLES *empClass*
CHANGE
$nomemp = prj_1 \circ empClass$; $appartient = prj_2 \circ empClass$; $travaille = prj_3 \circ empClass$
INVARIANT *empClass* ∈ *employé′* → *NomEmp* × *compagnie′* × *P(projet′)*

– Typage des données
Cette étape a pour but d'introduire les structures de données du langage d'implémentation choisi (relation). Cette règle affecte directement la structure des ensembles abstraits. À chaque fonction $y \in a \rightarrow b$ on rajoute un ensemble abstrait R défini par $R = A \mapsto B$. Une variable y' satisfaisant la contrainte ($y' \in R \wedge dom\ y' = a \wedge \forall x.(x \in a \Rightarrow y(x) = y'(x))$) est rajoutée.
L'application de cette règle sur les données du raffinement précédent donne :

SETS
$ProjetIdRecType = NomPro \times Agence$;
$ProjetRelType = P(ProjetIdRecType)$;
$ComRelType = NomCom \twoheadrightarrow ProjetIdRelType$;
$EmpRelType = IdEmp \twoheadrightarrow NomEmp \times NomCom \times ProjetRelType$;
$ProjRelType = ProjRecType \twoheadrightarrow P(NomCom)$
VARIABLES *comRel, empRel, projRel*
CHANGE *comRel=comClass*; *empRel=empClass*; *projRel=projClass*
INVARIANT *comRel* ∈ *ComRelType* ∧ *empRel* ∈ *EmpRelType* ∧ *projRel* ∈ *ProjetRelType*

De même, l'opération *RecruterEmployé*, de la page 30, est raffinée par :

emp ⟵ **RecruterEmployé(no,appar,trav)**

2. prj_n désigne la n^{ime} projection

PRE *appar* $\in$ *dom(comRel)* $\wedge \forall$ *w. w* $\in$ *trav* $\Rightarrow$ *w* $\in$ *prj1(comRel(appar)))*
THEN
e $\longleftarrow$ *NewEmpId*;
employé'=employé' $\cup$ *{e}*;
Idemp(e)=e;
nomemp(e):=no;
travaille(e):=trav;
appartient(e):=appar;
emp:=e
END

2.3.4 Traduction de la spécification concrète B en DBPL

Afin de pouvoir traduire en **DBPL** les spécifications concrètes **B**, obtenues au dernier raffinement, une formalisation en **B** des concepts clés de ce langage est nécessaire. Ceci permet d'instaurer une sorte d'équivalence entre des spécifications **DBPL** et un certain sous-langage de **B**. Elle permet aussi de caractériser le type de spécification **B** constituant le dernier niveau de raffinement.

2.3.4.1 Formalisation des concepts DBPL en B

Type enregistrement. Un enregistrement (record) **DBPL** est défini par :

D= **RECORD** $champ_1$: D_1; ... $champ_n$: D_n

avec $champ_i$ et D_i représentant le nom du champ et son type respectivement. En **B**, cet enregistrement est formalisé par :

SETS $D = D_1 \times \ldots D_n$
CONSTANT $champs_i \in D \rightarrow D_i$
PROPERTIES $champs_i = \lambda\, x.(x \in D \wedge x=(d_1, \ldots d_n) \mid d_i)$

Type relation. Une relation T de type D ayant pour clé ($clé_1, \ldots, clé_i$) définie en **DBPL** par :

T= **RELATION** $clé_1, \ldots, clé_i$ **OF** D

est formalisée en **B** par :

SETS $T=\{x \mid x \in D \wedge \forall\, d,e \in x.\ clé_1(d)= clé_2(e) \wedge \ldots \wedge clé_i(d)=clé_i(e) \Rightarrow d=e\}$

Sélection d'un élément. Soit T une relation de type D et de clé ($clé_1, \ldots, clé_i$). L'élément $r[k_1, \ldots, k_i]$ désigne l'enregistrement dont les valeurs des champs $clé_1, \ldots, clé_i$ sont $k_1, \ldots, k_i$ respectivement. Cet élément est formalisé en **B** par :

CONSTANTS $sel_T \in T \times D_1 \times \ldots D_n \rightarrow D$
PROPERTIES $sel_T = \lambda\, x.\ y_1, \ldots, y_n.(z \in x \wedge clé_1 = y_1 \wedge \ldots clé_i = y_i \mid z)$

L'élément $r[k_1, \ldots, k_i]$ désigne l'expression $sel_T(r, k_1, \ldots, k_i)$.

Expression d'accès sélectif. Un tel opérateur, utilisé dans l'expression des requêtes **DBPL**, permet de sélectionner un sous-ensemble de tuples d'une relation donnée. Soient e une variable symbolique, R une relation et P un prédicat. L'expression sélective : **EACH** *e* **IN** *R:P* désigne l'ensemble des tuples de la relation R vérifiant le prédicat P. En **B**, cette même expression est modélisée par :

$\{x \mid x \in R \wedge P(x)\}$

Opérations de bases. Soient r et R deux relations de type T. L'expression **DBPL** $r : +R$ désigne l'insertion dans la relation r de tout tuple de R dont la valeur de la clé n'appartient pas à r. En **B**, cette expression est modélisée par :

$$r{:=}id_T(r \cup \{x \mid x \in R \wedge \forall\, e \in r.(\mathit{clé}_1(e) \neq \mathit{clé}_1(x) \vee \ldots \mathit{clé}_n(e) \neq \mathit{clé}_n(x))\})$$

On définit de manière similaire les opérateurs de suppression et de surcharge $r : -R$ et $r : \&R$ respectivement.

2.3.4.2 Codage

La traduction du dernier niveau de raffinement **B** en spécifications **DBPL** est réalisée en utilisant le langage **OBJ**. Le langage **OBJ** est un langage algébrique bien adapté pour la spécification de langages. Les spécifications **OBJ** sont regroupées dans des modules ou objets. Un objet définit une signature et des axiomes. La signature permet de décrire des sortes (types) et les opérations qui leurs sont associées. Les sortes sont organisées en hiérarchies. Un axiome est une équation qui permet de réécrire un terme gauche en un autre terme droit. Les langages **B** et **DBPL** sont décrits par deux modules **OBJ**, puis on décrit dans un troisième module, qui inclut les deux premiers, la traduction d'expressions **B** en expressions **DBPL** à l'aide d'un ensemble d'axiomes de réécriture. Ces règles de réécriture correspondent aux règles de formalisation en **B** des concepts **DBPL**. Nous donnons ci-dessous, la formalisation en **OBJ** de quelques éléments du langage **B** et **DBPL** ainsi que la traduction en **DBPL** de l'opération d'insertion dans une relation exprimée en **B**.

Module OBJ décrivant le langage B.

```
sorts set.
subsort identifier enumset < set.
op INT : -> set.
op POW : set -> set.
op _ /\ _ : set set -> set.
sort substitution
op(_ := _) : lambda expression -> substitution.
op(_ || _) : substitution substitution -> substitution.
.........
```

Module OBJ décrivant le langage DBPL.

```
sorts Type IdentList FieldList.
op (_:_) : IdentList -> FieldList.
op (_ , _) : FieldList FieldList -> FieldList.
op RECORD _ END : FieldList -> Type
sort Designator Assignement.
op (_:=_) : Designator Expression -> Assignement.
op (_:+_) : Designator Expression -> Assignement.
.......
```

À partir de ces deux modules, un troisième permet de coder chaque terme du langage **B** en **DBPL**. Ce codage est exprimé par des équations de réécriture du langage **OBJ**. Par exemple, l'expression **B** spécifiant l'insertion dans une relation est traduite en **DBPL** par l'équation:

eq (F:= (Z{EACH Y IN F: TRUE, EACH Y1 IN E: NOT (Y1 IN F)}).Expression=(F:+E)

cette expression renvoie la relation Z dont les tuples sont ceux de F (sans conditions supplémentaires précisé par le terme TRUE dans l'équation) et ceux de E n'appartenant pas à F.

2.3.5 Commentaires

L'approche de développement proposée traite la génération d'une implémentation **DBPL** à partir d'une description **TDL**. Cette approche est basée sur la technique de raffinement **B**. La description initiale **TDL** est traduite en spécifications **B** puis ces dernières sont successivement raffinées jusqu'à l'obtention d'une spécification **B** proche d'une spécification **DBPL**. La traduction de ce dernier niveau de raffinement est réalisée en utilisant le système formel de réécriture **OBJ**. Comme les différentes étapes de développement de cette approche utilisent toutes des langages formels, la correction de ces dernières peuvent être établie de manière sûre.

Néanmoins, cette approche présente quelques limites :

- seul le raffinement des données est formalisé. L'auteur ne définit aucune règle concernant le raffinement des opérations.
- certaines étapes rajoutent des variables dont l'utilité n'est pas justifiée. Dans la première étape par exemple, la variable a' est superflue car $a' = a$. Ce rajout de variables inutiles peut rendre la spécification très lourde et donc illisible pour l'utilisateur.
- cette approche est basée sur une syntaxe **B** qui permet à deux ensembles abstraits de noms différents d'être égaux (étape 2 du processus de raffinement). Cette possibilité n'est plus supportée par la méthode **B** actuellement : le passage d'un modèle orienté objets à un modèle orienté valeurs ne peut se faire qu'à la dernière étape de raffinement.

Un travail similaire à cette approche a été proposé dans [65]. L'auteur propose un ensemble de règles de raffinement permettant la génération d'une implémentation (code) **DBPL** à partir de la spécification formelle d'un diagramme Entité/Association d'ordre supérieur (*HERM*)[73]. Le langage de spécification utilisé est *SAMT* [67]. Ce langage, basé sur les mêmes principes que le langage B, permet l'expression des concepts avancés du modèle relationnel tel que les vues. Les règles de raffinement définies dans ce travail traitent des aspects orientés programmation/ implémentation physique tels que la modularisation de l'implémentation obtenue, la définition des vues, des procédures, normalisation, etc.

2.4 Traduction DBPL de la spécification formelle Z des concepts relationnels

Dans [5], une méthode de dérivation de transactions bases de données à partir d'une spécification **Z** est proposée. Cette approche est basée sur la formalisation en **Z** des principaux concepts relationnels : domaines, relations et transactions. La génération de l'implémentation **DBPL** est formalisée par un ensemble de règles de traduction des spécifications **Z** ainsi obtenues. Le raffinement dans ce cas n'est pas nécessaire car les concepts relationnels et ceux de **DBPL** sont assez proches. Ce processus correspond à la phase de traduction de notre développement.

2.4.1 Modélisation abstraite Z des concepts relationnels

Le point de départ de cette approche est la spécification en **Z** des principaux concepts du modèle relationnel. Cette spécification concerne aussi bien les concepts de base (*domaines*, *relations*, *contraintes* et *instructions relationnelles*) que les concepts avancés comme les *transactions*, *traitement d'erreurs* et *vues*.

Domaines. Un domaine désigne l'ensemble de valeurs qu'un attribut peut prendre. En **Z**, un domaine est traduit par une définition. L'auteur distingue cinq types de domaines :

- domaines de base : *ENUM == N*,
- domaines intervalles : *AGE == 0..18*,

- domaines définis par compréhension : *DNUM == {n: N | n >100}*
- domaines énumérés : *SEXE == Homme| Femme*,
- domaines abstraits : [NOM].

Relations. Une relation représente un ensemble de tuples. La définition d'une relation est indépendante de tout détail d'implémentation. La définition formelle d'une relation comporte deux parties : l'intention et l'extension. L'intention d'une relation est représentée en **Z** par un type enregistrement dont les champs correspondent aux attributs. Par exemple, l'intention d'une relation *Employé* décrite par les attributs *NumEmp*, *Nom*, *Chef*, *Salaire* et *Age* est formalisée en **Z** par (l'attribut *Chef* désigne le chef éventuel d'un employé donné.) :

EMPLOYÉ $\widehat{=}$

[NumEmp: ENUM; Nom: NOM; Chef: ENUM ; Age: AGE; Salaire: SALAIRE; NumD: DNUM]

L'extension d'une relation représente le contenu de la relation à un instant donné. L'extension d'une relation est définie par un schéma **Z**. La partie prédicat de ce schéma spécifie les contraintes statiques que doit vérifier toute extension possible de la relation. Ces contraintes concernent, entre autres, la contrainte non nulle d'un attribut, la clé et les contraintes référentielles. Celles-ci sont exprimées par les mots réservés *REQUIRED*, *KEY_OF* et *FOR_KEY* respectivement. Par exemple, l'extension de la relation *Employé* est formalisée en **Z** par :

$$
\begin{array}{|l}
\textit{Employé} \\
\hline
\textit{employés} : \mathbb{P}\, \textit{EMPLOYÉ} \\
\hline
\textit{REQUIRED employés NumEmp} \land \\
\textit{REQUIRED employés Nom} \land \\
\quad \ldots \\
\textit{KEY\_OF employés NumEmp} \land \\
\textit{FOR\_KEY employés Chef employés NumEmp} \\
\hline
\end{array}
$$

avec *REQUIRED*, *KEY_OF* et et *FOR_KEY*correspondant à des prédicats définis par des schémas. Par exemple, le prédicat *KEY_OF* est décrit par :

$$
\begin{array}{|l}
[A,B] \\
\hline
\mathit{KEY\_OF} : \mathbb{P}\, A \rightarrow (A \rightarrow B) \rightarrow \mathit{BOOL} \\
\hline
\forall\, \mathit{rel} : \mathbb{P}\, A,\ \mathit{Att} : A \rightarrow B \bullet \\
\quad \mathit{KEY\_OF}\ \mathit{rel}\ \mathit{Att} \leftrightarrow \\
\qquad (\forall\, t_1, t_2 :\ \mathit{rel} \bullet \mathit{Att}\ t_1 = \mathit{Att}\ t_2 \leftrightarrow t_1 = t_2) \\
\hline
\end{array}
$$

avec :

- A désignant un schéma de relation, $\mathbb{P}\, A$ une extension possible de cette relation.
- Att désignant un attribut de la relation A de type B.

Schéma d'une base de données. Le schéma d'une base de données décrit l'ensemble des relations qui la composent et les différentes contraintes inter-relations. Ces contraintes concernent principalement les contraintes référentielles entre les différentes relations. Elles sont spécifiées par le mot réservé *FOR_KEY*. Par exemple, on considère une relation supplémentaire *Département* comportant les attributs *NumDep* (numéro du département), *NumChef* (le numéro de l'employé chef du département) et *NbEmp* (le nombre d'employés travaillant dans un département). Le schéma de la base de données DB regroupant ces deux relations est formalisé par le schéma **Z** suivant :

$$
\begin{array}{|l}
\hline
DB \\
Employé \\
Département \\
\hline
FOR\_KEY\ départements\ NumChef\ employés\ NumEmp\ \wedge \\
\quad \ldots \\
\forall d : départements \bullet d.NbEmp = \#\{e : employés \mid e.NumD = d.NumDep\} \\
\hline
\end{array}
$$

Le premier prédicat spécifie la contrainte référentielle de l'attribut *NumChef* vers l'attribut *NumEmp*. Le deuxième exprime le fait que l'attribut *NbEmp* est dérivé (peut être calculé à partir d'autres attributs). Comme pour les prédicats *REQUIRED* et *KEY_OF*, le prédicat *FOR_KEY* est défini par un schéma formel **Z**.

Instructions relationnelles. Les instructions de base du modèle relationnel se divisent en deux grandes catégories : instructions de lecture et instructions d'écriture. La première catégorie englobe la sélection (avec les différents agrégats), les différentes sortes de jointure et la projection. Les ordres d'écriture concernent l'insertion (*INSERT*), la suppression (*DELETE*) et la mise à jour (*UPDATE*). Pour chacun de ces ordres, l'auteur propose un schéma **Z**. Par exemple, l'ordre de suppression est spécifié formellement par le schéma **Z** suivant :

$$
\begin{array}{|l}
\hline
[A,B] \\
DELETE : \mathbb{P}\,A \rightarrow (A \rightarrow B) \rightarrow \mathbb{P}\,B \rightarrow \mathbb{P}\,A \\
\hline
\forall rel : \mathbb{P}\,A, Att : (A \rightarrow B), sv : \mathbb{P}\,B \bullet \\
\qquad DELETE\ rel\ Att\ sv\ = \{t : rel \mid Att\ t \notin\ sv\} \\
\hline
\end{array}
$$

Ce schéma spécifie que l'ordre *DELETE* prend en paramètres une extension $\mathbb{P}\,A$ d'une relation A, un attribut de type B $(A \rightarrow B)$, un ensemble de valeurs de cet attribut ($\mathbb{P}\,B$) et renvoie une nouvelle extension de la relation A. La partie prédicat spécifie l'effet de cet ordre.

Ces ordres de base sont utilisés afin d'exprimer les différentes opérations du système étudié. Par exemple, l'opération de suppression d'un ensemble de départements, désignés par leur clé, est exprimée par :

$$
\begin{array}{|l}
\hline
Delete\_Dep \\
\Delta DB \\
sd? : \mathbb{P}\,DNUM \\
\hline
départements'\ =\ DELETE\ départements\ NumEmp\ sd? \\
\hline
\end{array}
$$

Ces mêmes ordres de base sont aussi utilisés pour la spécification d'opérations plus élaborées. Par exemple, le schéma de suppression précédent est utilisé pour l'expression des différentes alternatives de suppression quand la relation considérée est référencée par d'autres relations. Par exemple, la suppression d'un ensemble d'employés par mise à jour à *NULL* de l'attribut *Chef* est spécifiée par :

$$
\begin{array}{|l}
\hline
Delete\_Emp\_NULL \\
\Delta DB \\
se? : \mathbb{P}\,ENUM \\
\hline
\mathbf{Let}\ emp\ =\ UPDATE\ employés\ Chef\ se?\ NULLNAT \bullet \\
\qquad\qquad\qquad employés' = DELETE\ emp\ ENUM\ se? \\
\hline
\end{array}
$$

Transaction. Une transaction est une opération complexe résultat de l'exécution de plusieurs opérations élémentaires. Une transaction réussit complètement ou échoue et le système retrouve son état avant l'exécution de la transaction considérée. Une transaction spécifie une exécution séquentielle d'opérations élémentaires. Par exemple, pour calculer la somme des salaires des employés d'un département *Salaire_Dep*, on exécute l'opération *Emp_Dep* renvoyant l'ensemble des employés d'un département, puis on exécute l'opération qui somme leurs salaires. Cette transaction est représentée par le schéma **Z** suivant (dans le cas où les préconditions sont vérifiées) :

$$\begin{array}{|l}
\hline Salaire_Dep_OK \\
\Xi DB \\
d? : DNUM \\
totale_sal! : SALAIRE \\
\hline
\exists\, dp : départements \bullet dp.DNum = d? \wedge \\
\quad \textbf{let } se == \{e : employés \mid e.NumD = d?\} \bullet \\
\qquad\qquad totale_sal! = SUM\ se\ Salaire \\
\hline
\end{array}$$

avec (*SUM se Salaire*) désignant la somme des valeurs de l'attribut salaire pour les tuples se.

Transactions sûres. Les schémas spécifiant les transactions contiennent dans leur partie prédicat des préconditions qui n'assurent l'exécution correcte de la transaction que si ces dernières sont vraies. Ces schémas spécifiés tels quels ne permettent pas la gestion des cas d'erreurs. Pour cela, un schéma supplémentaire est introduit. Par exemple, le schéma de traitement d'erreurs associé au schéma précédent est spécifié par :

$$\begin{array}{|l}
\hline Salaire_Dep_Erreur \\
\Xi DB \\
d? : DNUM \\
resultat! : STRING \\
\hline
(\neg(\exists\, dp : départements \bullet dp.Dnum = d?) \wedge \\
\qquad\qquad resultat! = \text{“}numéro\ de\ département\ invalide'') \\
\hline
\end{array}$$

Le schéma global de la transaction est donc défini par:

$$Salaire\_Dep \mathrel{\widehat{=}} (Salaire\_Dep\_OK \wedge OK) \vee Salaire\_Dep\_Erreur$$

2.4.2 Dérivation d'une implémentation DBPL à partir des schémas Z

La spécification **Z**, obtenue par la traduction des concepts relationnels, est directement traduisible en **DBPL**. Cette traduction est présentée sous forme de règles informelles. Les règles sont présentées à travers des exemples pratiques.

Domaine, relation, attribut obligatoire et clé Suivant les moyens offerts par le système de gestion des bases de données choisi, un domaine est soit traduit syntaxiquement en lui même ou traduit par l'un des types de base disponibles en ajoutant des contraintes de domaines. Par exemple, en **DBPL**, le domaine *AGE* est traduit par *AGE*== $\mathbb{N}$ plus une contrainte spécifiant que la valeur de chaque variable de ce type est comprise entre 0 et 18.

L'extension d'une relation est traduite en utilisant le type de données enregistrement. Par exemple, l'expression **Z** de l'intention et de l'extension de la relation *EMPLOYÉ* est traduite par :

TYPE *EMP*=
RECORD
NumEmp : ENUM; *Nom:NOM*; *Chef : ENUM*; *Chef : DNUM*; *Salaire: SALAIRE*; *NumD : DNUM*
END;

TYPE *employés* = **RELATION OF** *EMP*;

La contrainte d'attribut obligatoire est soit supportée par le langage comme par exemple dans **SQL92**, soit simulée en se réservant une valeur particulière du domaine qui représentera par la suite la valeur nulle. La traduction de la contrainte de clé est assez évidente car théoriquement tout SGBD offre la possibilité de définition de clé. Par exemple, en **DBPL**, la clé *NumEmp* de la relation *EMPLOYÉ* est spécifiée par :

TYPE *employés* = **RELATION** *NumEmp* **OF** *EMP*;

De la même manière, l'auteur définit, par des règles informelles, la traduction de tous les éléments de sa spécification **Z** : clé composée de plusieurs attributs, les fonctions d'agrégats, etc.

Opérations de base et transactions Les opérations de base sont traduites de manière assez naturelle. Les règles de traduction présentées de manière informelle sont illustrées à travers des exemples pratiques. Par exemple, l'opération de base *Delete_Dep* est traduite en **DBPL** par :

département :- *département*{**EACH** *t* **IN** *département: t.DNUM* **IN** *sd*}

En DBPL, une transaction est traduite par une procédure particulière spécifiée par le mot réservé **TRANSACTION**. En plus des paramètres d'entrée et de sortie du schéma **Z**, correspondant à la transaction, cette procédure contient un paramètre supplémentaire, de type chaîne de caractères, qui renseigne sur le succès ou l'échec de la transaction. Le schéma spécifiant un comportement correct de la transaction ainsi que les sous transactions utilisées sont traduits par des procédures. Les paramètres des sous-transactions qui ne sont pas des paramètres d'entrée de la transaction globale sont traduits par des paramètres locaux à la sous-transaction associée. Par exemple, la transaction *Salaire_Dep* est traduite en :

```
TRANSACTION Salaire_Dep (d : DNUM; VAR totale_sal : SALAIRE; VAR resultat : STRING);
    PROCEDURE Salaire_Dep_OK;
    VAR emp: employés ;
    BEGIN
        Emp_Dep(d,emp);
        Sum_Salaire(emp,totale_sal);
    END Salaire_Dep_OK;
BEGIN
            /* vérification des préconditions et traitement d'erreurs*/
    IF NOT (SOME dp IN département (dp.NumDep = d)) THEN
        résultat:="numéro de département invalide";
        RETURN;
    END
    Salaire_Dep_OK;
    resultat="Succès";
END Salaire_Dep
```

2.4.3 Commentaires

L'approche développée par Barros permet la génération d'une implémentation **DBPL** à partir d'une spécification **SQL**. Cette démarche consiste à formaliser en **Z** les principaux concepts relationnels. Une seconde étape permet de générer une implémentation **DBPL** par traduction des spécifications **Z** ainsi obtenues. Cette approche traite aussi bien les concepts élémentaires tels les domaines et les relations, que les concepts élaborés tels les transactions et la gestion des cas d'exception. Les deux modèles relationnel et **DBPL** étant assez proches (**DBPL** est une extension du modèle relationnel enrichi par des structures supplémentaires de données et de programmation), le passage de la spécification **Z** à une implémentation **DBPL** ne nécessite pas d'étapes de raffinement. L'approche proposée est complémentaire de notre démarche. Elle peut être redéfinie dans notre contexte (qui est **B**) pour la traduction en **SQL** de transactions plus élaborées. Un prototype implémentant partiellement cette approche a été réalisé. Néanmoins, quelques limites sont à noter :

- la génération de spécifications **DBPL** est une étape de traduction presque syntaxique.
- contrairement au travaux de [29] utilisant le système de réécriture formel **OBJ**, la phase de traduction des spécifications **Z** vers **DBPL** n'est pas prouvée.
- seule la phase de formalisation des concepts relationnels en **Z** est réalisée par des règles précises. La traduction des spécifications **Z** en **DBPL** est présentée de manière informelle. Aucune règle n'est définie.

2.5 Construction déductive de transactions sûres

Les travaux de Qian [57] traitent de la synthèse de transactions base de données sûres. L'auteur propose une démarche formelle de construction de transactions à partir de la spécification formelle de cette dernière. Cette démarche est basée sur l'utilisation des tableaux déductifs introduits par Manna [46, 47]. Les tableaux déductifs sont une technique d'extraction de programmes à partir de la preuve de la spécification formelle de ces derniers. Les tableaux déductifs définissent un système de preuve par déduction.

Le point de départ de l'approche proposée par Qian est constitué des éléments suivants :

1. un schéma de base de données représentant la définition des tables,
2. un ensemble de contraintes représentant les propriétés que doivent satisfaire les données contenues dans la base,
3. la spécification formelle de la transaction à réaliser.

Les éléments 2 et 3 sont exprimés dans un langage de spécification défini par l'auteur. Ce langage est appelé logique des transactions.

Nous commençons par une présentation rapide du langage des transactions, ensuite nous montrons à travers un exemple la façon dont les contraintes du système et ses transactions sont spécifiées. Nous terminons par l'illustration de la méthode des tableaux déductifs pour la génération du code de la transaction à partir de la spécification obtenue à l'étape précédente.

2.5.1 Le langage et la théorie de spécification des transactions

Le langage de transactions, utilisé dans ces travaux, est un sous-ensemble de la logique du premier ordre. Ce langage comprend les éléments suivants : les symboles de ponctuation : "(",")" et ",", les valeurs booléennes : vrai et faux, les variables, les connecteurs logiques ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\equiv$ et $if - then - else$), les quantificateurs ($\forall$ et $\exists$) et le symbole d'égalité. Une relation est représentée par un prédicat n-aire R. Les transactions sont définies inductivement par (R est une relation, p un prédicat, x et e sont des tuples de variables et d'expressions respectivement) :

1. l'insertion (resp. suppression, modification) $insert_R(e)$ (resp. $delete_R(e)$, $modify_R(x,e)$) est une transaction sur R.
2. si t_1 et t_2 sont des transactions alors la composition séquentielle ($t_1 ;; t_2$) est aussi une transaction.
3. si t_1 et t_2 sont des transactions alors (*if* p *then* t_1 *else* t_2) et (*if* p *then* t_1) sont aussi des transactions.

La sémantique d'une transaction est décrite par l'effet de son exécution. L'état d'une base de données est représenté par la valeur des différents éléments $\mathcal{U}$ qui la constituent : c'est l'extension des différentes relations de la base de données. L'ensemble des états d'une base de données est désigné par $état(\mathcal{U})$. La sémantique d'une transaction peut être définie comme une fonction de transition d'une état initial à l'état résultant de son exécution. Cette fonction est notée $\mathcal{K}_\mathcal{U}$ dans la suite. Formellement, la sémantique d'une transaction est définie inductivement par[3] :

1. Pour toute relation R, tuple de variables x et tuple d'expressions e :

$$\begin{aligned}
&\mathcal{K}_\mathcal{U}(insert_R(e)) = \{(I,J) \mid I \in état(\mathcal{U}) \wedge J = I[R/(R^I \cup \{e^I\})]\} \\
&\mathcal{K}_\mathcal{U}(delete_R(e)) = \{(I,J) \mid I \in état(\mathcal{U}) \wedge J = I[R/(R^I - \{e^I\})]\} \\
&\mathcal{K}_\mathcal{U}(modify_R(x,e)) = \{(I,J) \mid I \in état(\mathcal{U}) \wedge \\
&\qquad\qquad J = if\ x^I \in R^I\ then\ [R/(R^I - \{x^I\} \cup \{e^I\})]\}\ else\ I
\end{aligned}$$

2. Pour toute transaction t_1 et t_2 :
$$\mathcal{K}_\mathcal{U}(t_1 ;; t_2) = \mathcal{K}_\mathcal{U}(t_1) \circ \mathcal{K}_\mathcal{U}(t_2)$$
3. Pour tout prédicat p et transactions t_1 et t_2 :
$\mathcal{K}_\mathcal{U}$(*if* p *then* t_1 *else* t_2 *fi*= $\{(I,J) \mid I \in état(\mathcal{U}) \wedge J =$ *if* p^I *then* $\mathcal{K}_\mathcal{U}(t_1)$ *else* $\mathcal{K}_\mathcal{U}(t_2)\}$

Afin de pouvoir manipuler, effectuer des calculs et des raisonnements sur les objets de cette théorie, l'auteur définit un ensemble d'axiomes. Seuls les axiomes nécessaires à la compréhension de la suite sont repris.

$$\begin{aligned}
&R'(insert_R(w,e'),e') \\
&R'(w,e') \rightarrow \neg R'(delete_R(w,e'),e') \\
&R'(w,x) \rightarrow R'(modify_R(w,x,e'),e')
\end{aligned}$$

avec :

- w : désigne l'état courant de la base de données,
- e' : désigne l'évaluation du tuple d'expressions e à l'état w
- $R'(w,e')$: désigne un prédicat indiquant l'appartenance ou non à la relation R de l'évaluation de e.
- $t'(w,e')$: désigne l'état résultant de l'exécution à l'état w de la transaction t sur l'évaluation de e.

Le premier axiome spécifie que quelque soit l'état w de la base, la relation R contient le tuple e' à l'état résultant de l'insertion du tuple e' dans R. Le second indique que, si la relation R contient le tuple e' à l'état w de la base, alors cette relation ne le contient plus à l'état résultant de la suppression de ce tuple de R. Enfin le dernier indique que dans un état w, si une relation R contient un tuple x alors cette relation R contient le tuple e' dans l'état résultant de la modification du tuple x par e'.

2.5.2 Spécification des transactions

La spécification d'une transaction est une assertion logique qui exprime la relation existant entre l'état initial s_0 de la base (avant l'exécution de la transaction) et son état final (après exécution de la transaction). Trouver une solution (programme, preuve de l'assertion) à cette spécification consiste à définir les actions t

3. X^I désigne l'évaluation de X à l'état I. $I[X/Y]$ désigne un état identique à I à l'exception de X qui est substitué à Y.

à exécuter à partir de l'état initial qui, d'une part, réalisent le traitement que doit effectuer la transaction, et d'autre part, ne violent pas les contraintes. L'exécution de ces actions doit établir l'état final souhaité de la base de données. Formellement, la spécification d'une transaction revient à prouver la formule logique :

$$(\forall\, s_0)(\forall\, \overline{x})\, Q(s_0,\, s_0;t,\, \overline{x})$$

avec :

- s_0 : désigne l'état initial de la base de données,
- $\overline{x}$: désignent les paramètres de la transaction,
- $s_0;t$: désigne l'état de la base de données après exécution de la transaction t,
- Q : une formule logique spécifiant le traitement que doit effectuer la transaction.

Exemple : on considère l'exemple de la base de données suivant :

COMPAGNIE(co_nom, adresse)
EMPLOYÉ(em_nom, date-naiss, numtel)
CANDIDAT(ca_nom,date-naiss, numtel)
ENTRETIEN (candidat, compagnie, heure, rang)

avec les trois contraintes :

- l'ensemble des employés et des candidats sont disjoints.
- l'attribut $candidat$ de la relation $ENTRETIEN$ est lié par une contrainte référentielle à l'attribut ca_nom de la relation $CANDIDAT$.
- L'attribut ca_nom est une clé de la relation candidat.

Ces contraintes sont exprimées par les formules logiques suivantes (E, C, T désignent les relations *EMPLOYÉ*, $CANDIDAT$ et $ENTRETIEN$ respectivement) :

$$(\forall\, s)(\forall\, x)(E'(s,x) \rightarrow \neg\, C'(s,x))$$
$$(\forall\, s)(\forall\, x)(T'(s,x) \rightarrow (\exists\, y)(C'(s,y) \wedge candidat(x) = ca\_nom(y)))$$
$$(\forall\, s)(\forall\, x)(\forall\, y)(C'(s,x) \wedge C'(s,y) \wedge ca\_nom(x) = ca\_nom(y) \rightarrow x = y)$$

On voudrait spécifier la transaction qui permet l'insertion d'un nouvel employé e. Pour cela, on doit vérifier que l'employé que l'on veut insérer n'est plus candidat (le cas échéant, le supprimer) et ne participe à aucun entretien. La transaction t cherchée doit donc satisfaire la formule logique suivante :

$$(\forall\, s_0)(\forall\, e)(\exists\, t)$$

$$\left(\begin{array}{l}(\forall\, x)(E'(s_0,\, x) \rightarrow \neg\, C'(s_0,\, x))\wedge \\ (\forall\, x)(T'(s_0,\, x) \rightarrow (\exists\, y)(C'(s_0,\, y) \wedge candidat(x) = ca\_nom(y)))\wedge \\ (\forall\, x)(\forall\, y)(C'(s_0,\, x) \wedge C'(s_0,\, y) \wedge ca\_nom(x) = ca\_nom(y) \rightarrow x = y)\end{array}\right)$$

$$\rightarrow$$

$$\left(\begin{array}{l}E'(s_0;t,\, e)\wedge \\ (\forall\, x)(E'(s;t,\, x) \rightarrow \neg\, C'(s;t,\, x)) \\ (\forall\, x)(T'(s;t,\, x) \rightarrow (\exists\, y)(C'(s;t,\, y) \wedge candidat(x) = ca\_nom(y)))\wedge \\ (\forall\, x)(\forall\, y)(C'(s;t,\, x) \wedge C'(s;t,\, y) \wedge ca\_nom(x) = ca\_nom(y) \rightarrow x = y)\end{array}\right)$$

Cette formule logique exprime que quelque soit l'état initial s_0 de la base et les caractéristiques de l'employé e à insérer, il existe une transaction t telle que, si l'état s_0 vérifie les contraintes exprimées sur la base, son exécution engendre un état où la relation *EMPLOYÉ* contient l'employé e et les contraintes sont toujours vérifiées.

2.5.3 Construction déductive de transactions

La spécification d'une transaction est une assertion logique qui exprime la relation existante entre l'état initial de la base (avant l'exécution de la transaction) et son état final (après exécution de la transaction). La description procédurale d'une transaction décrit son implémentation. Le processus de construction de transactions désigne le mécanisme de transformation graduelle de la spécification associée en une procédure implémentable. L'approche utilisée dans ce travail est le raisonnement par déduction. Le problème peut être formulé de la manière suivante : il s'agit de déterminer une transaction t qui permet de prouver la formule :

$$\forall\, s_0, \forall\, \overline{x}. \exists\, t Q(s_0,\, s_0;t,\, \overline{x})$$

L'approche utilisée pour la détermination de t est les tableaux déductifs. Ils permettent l'extraction de programmes par construction de la preuve associée. Un tableau déductif est constitué de deux colonnes représentant les hypothèses (assertions) et les buts respectivement et d'une à plusieurs colonnes de sortie représentant les objets (programmes) cherchés. Chaque ligne de ce tableau représente exclusivement soit une assertion soit un but. Schématiquement un tableau déductif est de la forme suivante :

Assertions	Buts	Sortie$_1$	...	Sortie$_n$
α_1			...	γ
$\vdots$		μ		
α_n				
	β_1			
	$\vdots$		ν	
	β_m			

Ce tableau représente la formule logique suivante :

$$(\forall\, \overline{x})\alpha_1 \wedge \ldots \wedge (\forall\, \overline{x})\alpha_n \rightarrow (\exists\, \overline{x})\beta_1 \vee \ldots \vee (\exists\, \overline{x})\beta_m$$

Dans le cas de construction de transactions, une seule sortie est à distinguer. Cette sortie représente la transaction à définir. Le tableau initial est obtenu par skolémisation (élimination des quantificateurs les plus à gauche) de la formule logique représentant la spécification de la transaction. Les assertions sont les différents axiomes de la théorie des transactions. La transaction d'entrée est $(s_0;t)$. Le tableau initial correspondant à la formule logique représentant l'insertion d'un nouvel employé de la section 2.5.2 est le suivant :

Buts	Sortie : Transaction
G1.$Cont(s_0) \rightarrow E'(s_0;t,e) \wedge Cont(s_0;t)$	$s_0;t$

avec $Cont(s_0)$ et $Cont(s_0;t)$ désignant l'expression des contraintes à l'état initial et après exécution de la transaction respectivement.

La construction de la transaction t par utilisation de tableaux déductifs est formalisée par un ensemble de règles de transformation d'un tableau en un autre tableau équivalent. La condition d'arrêt d'un tel processus est l'obtention d'un but $true$ ou d'une assertion $false$. Les tableaux déductifs pouvant être utilisés dans divers domaines, il n'existe pas de règles fixes bien définies de transformation de tableaux. L'utilisateur peut définir ses propres règles de déduction (transformation) selon ces besoins. Pour cela dans [47], une condition suffisante de correction d'une règle de déduction a été définie.

Définition 2.5.1 *Une substitution est un ensemble de paires d'assignation de la forme $(x_i \leftarrow t_i)$, où x_i et t_i désignent des variables et termes respectivement.*

Définition 2.5.2 *Soient e une expression et θ une substitution. L'expression $e\theta$ désigne l'application de la substitution θ à l'expression e.*

Définition 2.5.3 *Une substitution θ est plus générale qu'une substitutions ϕ, s'il existe une substitution λ tel que :*

$$\theta\,\lambda = \phi$$

Définition 2.5.4 *Une substitution θ unifie deux expressions d et e si :*

$$e\theta = d\theta$$

Il existe un algorithme testant l'unification de deux expressions. Cet algorithme retourne soit l'unificateur le plus général (si les deux expressions sont unifiables) ou un objet nul.

Définition 2.5.5 *À toute sous-formule d'une formule α du tableau déductif, une polarité est assignée. La polarité d'une formule indique la valeur souhaitée à la formule qui rend la preuve triviale. Intuitivement, on aimerait qu'une hypothèse soit fausse et qu'un but soit vrai. Elle est définie inductivement comme suit :*

1. une assertion est négative, un but est positif.
2. si α est de forme $\neg\beta$ alors α et β ont des polarité opposées.
3. si α est de forme $\beta \wedge \gamma$ ou $\beta \vee \gamma$ alors β et γ ont la même polarité que α.
4. si α est de forme $\beta \rightarrow \gamma$ alors β a une polarité opposée à α et γ a la même polarité que α.
5. si α est de forme $(\forall\, x)\beta$ ou $(\exists\, x)\beta$ alors β a la même polarité que α.
6. si α est de la forme $\beta \equiv \gamma$ alors β et γ ont simultanément la polarité positive et négative.
7. si α est de la forme *if* β *then* γ *else* δ alors β a simultanément la polarité positive et négative, γ et δ ont la même polarité que α.

Dans la suite, les polarités positive et négative sont désignées par (+) et (-) respectivement.

2.5.4 Règles de transformation

Dans cette sous-section sont décrites quelques règles de transformation sur les tableaux déductifs. Ces règles sont illustrées à travers l'exemple d'insertion d'un nouvel employé dans la base de donnée de la section précédente.

Règle de décomposition. Cette règle a pour but de décomposer une conjonction d'hypothèses, une disjonction de buts ou un but de la forme $A \rightarrow B$. Formellement :

- une assertion de la forme $A \wedge B$ est décomposée en deux assertions A et B.
- un but de la forme $A \vee B$ est décomposé en deux sous buts A et B
- un but de la forme $A \rightarrow B$ donne lieu à une assertion A et à un but B (c'est le principe de déduction).

L'application de cette règle sur le tableau initial génère le tableau suivant :

Assertions	Buts	Transaction
	G1.$Cont(s_0) \rightarrow E'(s_0;t,e) \wedge Cont(s_0;t)$	$s_0;t$
A1.$Cont(s_0)$		$s_0;t$
	G2.$E'(s_0;t,e) \wedge Cont(s_0;t)$	$s_0;t$

Règle de résolution. Soient $\alpha[p^+]$ et $\beta[q^-]$ deux lignes du tableau n'ayant pas de variables libres communes. Soit θ l'unificateur le plus général de p et q, alors :

1. si α et β sont des assertions alors ajouter $\alpha\theta[true] \vee \beta\theta[false]$ à la liste des assertions.
2. si α et β sont des buts alors ajouter $\alpha\theta[true] \vee \beta\theta[false]$ à la liste des buts.
3. si α est une assertion et β est un but alors ajouter $(\neg\alpha)\theta[true] \wedge \beta\theta[false]$ à la liste des buts.
4. si α est un but et β est une assertion alors ajouter $\alpha\theta[true] \wedge (\neg\beta)\theta[false]$ à la liste des buts.

Le tableau suivant montre l'application de cette règle sur le but $G2$ et l'assertion (axiome) $A2$ avec comme unificateur le plus général $\theta \equiv \{w \leftarrow s_0;t_1,\ t \rightarrow t_1;;\ insert_E(e),\ x \rightarrow e\}$:

Assertion	But	Transaction
	G1. $Cont(s_0) \rightarrow E'(s_0;t,e) \wedge Cont(s_0;t)$	$s_0;t$
A1. $Cont(s_0)$		$s_0;t$
	G2. $\underline{E'(s_0;t,e)^+} \wedge Cont(s_0;t)$	$s_0;t$
A2.$\underline{E'(insert'_E(w,x),x)^-}$		
	G3.$Cont(s_0;t_1,insert_E(e))$	

Règle d'extraction des instructions de base. Cette règle permet la génération d'une instruction de base (insert, delete, modify). S'il existe une assertion $\alpha[p^-]$, un but $\beta[q^+]$ relative à la sotie γ, si θ est l'unificateur le plus général de p et q alors on ajoute le but $(\neg\alpha)\theta[false] \wedge \beta\theta[true]$ avec la sortie $\gamma\theta$. Cette règle est l'extension du cas 4. de la règle de résolution. L'application de cette règle associe au but $G3$ la transaction de sortie $(s_0;t_1;insert_E(e))$:

Assertion	But	Transaction
	G1. $Cont(s_0) \rightarrow E'(s_0;t,e) \wedge Cont(s_0;t)$	$s_0;t$
A1. $Cont(s_0)$		$s_0;t$
	G2. $\underline{E'(s_0;t,e)^+} \wedge Cont(s_0;t)$	$s_0;t$
A2.$\underline{E'(insert'_E(w,x),x)^-}$		
	G3. $Cont(s_0;t_1,insert_E(e))$	$s_0;t_1;insert_E(e)$

Règle d'extraction d'une transaction séquentielle. La génération d'une transaction séquentielle est réalisée par application successive de la règle de résolution. Le but $G3$ peut être réécrit en :

$$\text{G4. } \neg C'(s_0;t_1,e) \wedge Cont(s_0;t_1)$$

En effet, le but $G3$ exprime que l'ensemble des contraintes est vérifié après l'insertion de e dans un état $s_0;t_1$. Donc, ces contraintes devraient être aussi vraies avant cette insertion ($Cont(s_0;t_1)$). Le tuple e ne doit pas donc être dans la relation $CANDIDAT$ ($\neg C'(s_0;t_1,\ e)$). Une seconde application de la règle de résolution au but $G4$ et l'assertion $A3$ avec l'unificateur le plus général $\theta \equiv \{w \leftarrow s_0;t_2, t_1 \leftarrow t_2;;\ delete_C(e), x \leftarrow e\}$ génère le but $G5$ suivant :

Assertion	But	Transaction
	G4. $\underline{\neg C'(s_0;t_1,e)^+} \wedge Cont(s_0;t_1)$	$s_0;t_1;insert_E(e)$
A3. $C'(w,x) \rightarrow$ $\underline{\neg C'(delete'_C(w,x),x)^-}$		
	G5. $C'(s_0;t_2,e) \wedge Cont(s_0;t_2;delete_C(e))$	$s_0;t_2;delete_C(e);insert_E(e)$

Règle d'extraction d'une transaction conditionnée Cette règle permet la dérivation d'une transaction conditionnée. C'est l'extension du cas 2. de la règle de résolution présentée précédemment. S'il existe un but positif $\alpha[p^+]$ associé à la sortie μ, et un but négatif $\beta[q^-]$ associé à la sortie ν, si θ est l'unificateur le plus général de p et q alors ajouter le but $(\alpha\theta[true] \wedge (\neg\beta)\theta[true])$ associé à la sortie *if* $p\theta$ *then* $\mu\,\theta$ *else* $\nu\theta$.

L'application de cette règle sur les buts $G4$ et $G5$ avec comme unificateur le plus général ($\theta \equiv \{t_1 \leftarrow t_2;; \Lambda\})^4$ génère le tableau suivant :

Assertion	But	Transaction
	G4. $\neg \underline{C'(s_0;t_1,e)}^- \wedge Cont(s_0;t_1)$	$s_0;t_1;insert_E(e)$
	G5. $\underline{C'(s_0;t_2,e)}^+ \wedge Cont(s_0;t_2;delete_C(e))$	$s_0;t_2;delete(a);insert_E(e)$
	$Cont(s_0;t_2)\wedge$ $Cont(s_0;\ t_2;delete_C(e))$	*if* $C'(s_0;t_2,e)$ *then* $s_0;t_2;delete_C(e);insert_E(e)$ *else* $s_0;t_2;insert_E(e)$

De la même manière, l'auteur propose d'autres règles qui permettent par exemple d'extraire des boucles. L'application de l'ensemble des règles permet d'obtenir la transaction attendue (e représente le nom de l'employé à créer) :

IF $C(e)$ **THEN**
    For each x tel que x.ca_nom=e **do**
        Delete x **FROM** *CANDIDAT*
        Delete (e,...) **FROM** *ENTRETIEN*
    Edo
INSERT $(e,\ldots)$ **INTO** *EMPLOYÉ*

2.5.5 Commentaires

L'approche proposée dans ce travail concerne la dérivation de transactions sûres dans le cadre d'applications base de données. Cette approche est basée sur l'utilisation de la technique de tableaux déductifs permettant l'extraction d'un programme par établissement de la propriété que ce dernier doit satisfaire. Cette approche, appelée déductive par l'auteur, procède par raffinement d'une formule logique représentant la propriété attendue de la transaction. Cette propriété exprime une relation entre les éventuels paramètres d'entrée de la transaction ainsi que les états avant et après de la base de données. Bien que cette approche soit formelle, celle-ci présente quelques limites :

- Il s'agit plus d'une démarche que d'une méthode systématique de développement. En effet, l'approche ne précise pas comment la spécification de la formule logique, représentant la transaction, est obtenue. De plus a priori à chaque étape de raffinement, plusieurs règles peuvent être appliquées. L'approche ne définit aucun critère de décision qui permet d'appliquer une règle plutôt qu'une autre.
- Comme on l'a vu, des simplifications sur les buts générées sont souvent réalisées afin de rendre possible l'application d'une règle. Ces simplifications n'étant pas systématiques, cette approche demande donc un effort conséquent et des connaissances mathématiques pointues de l'utilisateur.
- Seul le développement de transactions est traité dans ce travail. L'approche suppose déjà l'existence d'un schéma logique de données.

4. Λ désigne la transition qui ne fait rien.

2.6 Commentaires généraux

Cette section est consacrée à une étude comparative des travaux présentés et commentés dans les sections précédentes. Cette étude comparative permet une meilleure perception de notre contribution au développement formel d'applications base de données. Notre comparaison porte sur les critères suivants :

1. L'écart d'abstraction entre les niveaux abstrait et concret : bien que l'ensemble des travaux existants aient pour but le développement d'applications base de données, les niveaux d'abstraction de leurs points de départ sont différents. Cet écart d'abstraction entre les niveaux abstrait et concret influence directement sur la technique utilisée et aussi sur la complexité de l'approche. Suivant l'importance de cet écart d'abstraction l'approche de développement varie entre une simple étape de traduction comme dans [5, 6] à plusieurs étapes complexes de raffinement comme dans [19, 20, 66, 29].
2. Technique de développement utilisée : l'étude des travaux existants permet de dégager trois approches de développement dans le contexte des bases de données. L'approche par raffinement [19, 20, 66, 29] permet la génération d'une implémentation par des transformations successives et graduelles de la spécification abstraite initiale. Cette approche est utilisée quand la spécification initiale ne peut pas être directement traduite dans le langage d'implémentation cible choisi. L'approche déductive est une technique par raisonnement sur la spécification abstraite. L'approche par traduction est utilisée principalement dans le cas où les langages de départ et d'arrivée offrent des concepts similaires ou équivalents.
3. Degré de généralité de la technique utilisée : ce critère indique si le processus de développement est désigné par des règles précises. La présence de règles pour un processus permet son automatisation. Dans le cas contraire, le processus est plus une démarche qu'une méthode rigoureuse de développement.
4. Concepts pris en compte : attributs simples (AtS), associations (Asso), héritages (Her), transactions (Trans),
5. Existence d'un outil d'aide au développement : ce critère est significatif pour le degré de généralité et de précision des règles associée à la méthode de développement sous-jacente. La mise en œuvre d'un outil permet le plus souvent de s'assurer que les règles sont suffisamment formalisées et précises pour constituer une base d'automatisation.
6. Fiabilité de la méthode utilisée : ce critère renseigne sur le degré de certitude des résultats produits par une méthode de développement. Seules les méthodes dont les différentes étapes ont fait l'objet de preuves de correction peuvent être qualifiées de fiables. En particulier, les phases de codage faites en dehors de tout système ou méthode formels ne peuvent être validées de manière formelle et sûre.

Travaux / Critères	[19, 20]	[66, 29]	[5, 6]	[57]
L'écart entre les niveaux abstrait et concret	fort	fort	faible	fort
Technique de développement utilisée	Raffinement	Raffinement+Codage	Codage	Déduction
Degré de formalisation de la technique	Faible	Moyen	Moyen	Faible
Concepts pris en compte	AtS,Asso,Trans	AtS,Asso,Trans	-	Trans
Existence d'un outil	Non	Non	Partiel	Non
Fiabilité	Forte	Moyenne	Faible	Forte

Ce tableau montre qu'aucun des travaux étudiés n'offre une méthode qui répondrait pleinement aux besoins de développement d'applications bases de données sûres. Aucune des approches étudiées ne permet la prise en compte de l'ensemble des concepts présents dans les systèmes d'information. Les concepts d'attributs complexes et d'héritage ne sont pas du tout abordés. Excepté les travaux de [66, 29], la correction du processus de codage des différente méthodes ne peut être garantie. La plupart des approches étudiées ne sont pas outillées. L'absence d'outils ne favorise pas la mise en pratique de ces méthodes de développement.

Dans le domaine industriel, plusieurs outils AGL commerciaux ont été développés pour la conception d'applications base de données. La plupart de ces outils traitent plutôt l'aspect données qui est représenté par la structure des tables et des différentes contraintes d'intégrité. Quant aux autres outils qui génèrent du code, ils ne prennent pas en compte les contraintes d'intégrité.

2.7 Conclusion

Dans le cadre de cette thèse, nous proposons un processus de dérivation automatique outillé d'une implémentation relationnelle à partir de spécifications **B** bien définies. La spécification abstraite **B**, point de départ de notre développement, est obtenue par l'application d'un ensemble de règles de traduction formelle sur des diagrammes **UML** bien formés. Nous avons utilisé pour cela la technique de raffinement **B**. L'utilisation de cette technique nous permet de garantir la correction des différentes étapes de notre approche. Nous nous situons au même niveau d'abstraction que celui de [29], et utilisons également le principe de raffinement comme technique de développement. Notre approche a l'avantage d'être plus formalisée. En effet d'une part, la spécification abstraite initiale est bien caractérisée, d'autre part, les règles que nous proposons sont plus précises. De plus, la plupart des concepts des systèmes d'information comme les attributs multivalués et l'héritage sont pris en compte. La phase de traduction, du dernier niveau de raffinement, vers du code exécutable n'est formalisée que pour la définition des tables et des différentes contraintes. Nous nous sommes contentées d'illustrer la traduction des transactions à travers un exemple pratique. Nous avons également étudié la réutilisation de preuves élémentaires au cours de la validation de développements complexes. Un outil permettant la mise en pratique de nos propositions a été développé. Notre processus de développement est décrit en détails dans les chapitres 4 et 5.

Chapitre 3

Une modélisation formelle B et graphique UML pour les systèmes d'information

Le langage **UML** est un langage graphique de modélisation orientée objets des systèmes. Ce langage, actuellement très utilisé en industrie, permet une spécification visuelle des structures et comportements du système étudié. Le langage **UML** est issu de l'unification de trois méthodes : **Booch**, **OOSE**, et **OMT** [10, 64, 34], et a pour but de fournir un standard universel pour la modélisation orientée objets des systèmes. Ce langage a été normalisé par l'OMG (**O**bjet **M**anagement **G**roup) [54].

En systèmes d'information, l'usage des notations graphiques pour la description des applications représente des avantages indéniables. En effet, ces notations constituent un bon support de communication entre les différents acteurs du système. Les modélisations obtenues ont non seulement un caractère intuitif mais sont aussi présentées de manière claire et modulaire. Le système est représenté par plusieurs diagrammes décrivant chacun un de ses aspects spécifiques. Le langage **UML** offre diverses sortes de diagrammes permettant ainsi de couvrir les différentes étapes du processus de développement d'un système de l'analyse des besoins jusqu'à l'implémentation. Dans la modélisation conceptuelle des systèmes d'information, les diagrammes suivants sont suffisants pour la description de l'ensemble des aspects essentiels d'un système :

- le diagramme de classes représentant l'aspect statique ou structurel du système à modéliser,
- les diagrammes d'états/transitions représentant le comportement des objets d'une classe donnée,
- les diagrammes de collaborations représentant les fonctionnalités du système.

La représentation d'un système selon plusieurs vues permet la gestion de sa complexité et facilite la compréhension du fonctionnement pour l'utilisateur. Cependant ces diagrammes ne sont évidemment pas indépendants, ils doivent être cohérents les uns par rapport aux autres. Cette validation peut être réalisée par une suite d'aller/retour entre diagrammes afin d'en assurer la cohérence. On peut être amené, par exemple, à ajouter un attribut dans une classe afin de pouvoir exprimer un état du diagramme d'états/transitions.

Cependant, ce type de représentation graphique utilise souvent des concepts dont la sémantique n'est pas toujours précise. Bien que ce manque de sémantique peut constituer une souplesse d'expression du langage, elle réduit considérablement les possibilités de raisonnement et de preuve sur les modélisations obtenues. Pour pallier ce manque de précision, plusieurs travaux ont proposé de combiner ces méthodes graphiques avec des notations formelles [18, 53, 38]. Le principe de base de ces travaux consiste à définir un ensemble de règles de traduction de ces notations graphiques en notations mathématiques. L'objectif d'une telle traduction est l'obtention d'une modélisation non seulement précise et concise mais surtout analysable par des outils (contrôles de type, preuves, simulations, . . .).

Le point de départ de notre étude est une spécification abstraite **B** obtenue par le processus de traduction proposé dans [53]. Ce processus de traduction, défini pour des notations **OMT**, a été adapté pour des notations

UML [22]. Cette adaptation concerne principalement les diagrammes de séquences utilisés dans **OMT**. Les règles de traduction des diagrammes de séquences ont été adaptées pour la traduction des diagrammes de collaborations **UML**.

La génération de spécifications **B** à partir des diagrammes **UML** sus-cités est réalisée par les étapes suivantes :

1. Élaboration du diagramme de classes permettant la description de l'aspect statique du système. Les contraintes non graphiquement exprimables sont spécifiées en **B**.
2. Génération de spécifications **B** à partir du diagramme de classes établi lors de la première étape.
3. Description graphique des transactions du système à l'aide de diagrammes d'états/transitions et de collaborations. Afin d'assurer une traduction automatique de ces diagrammes, ces derniers sont annotés par des expressions **B** générées lors de la première phase. Cette étape peut nécessiter l'ajout d'attributs au diagramme de classes et par conséquent la réitération des deux premières étapes.
4. Génération de spécifications **B** correspondant aux diagrammes d'états/transitions et de collaborations.
5. Établissement des preuves de correction de l'ensemble des spécifications générées.

Seules les propriétés fonctionnelles du système sont prises en compte par ce processus de traduction. L'utilisation du langage **B**, ne permet pas la spécification des propriétés dynamiques telles que les contraintes temporelles, les propriétés de vivacité, l'absence d'interblocage, etc. La prise en compte de telles propriétés nécessite l'utilisation d'un autre langage de spécification comme la logique temporelle, la logique des traces [26], ou le **B** événementiel [30].

La première partie du chapitre présente les principaux concepts **UML** dédiés à l'étude des systèmes d'information. La deuxième décrit la modélisation formelle **B** de ces concepts. Nous terminons par la présentation d'une méthodologie de conception d'applications sûres et une conclusion.

3.1 Description des diagrammes UML

3.1.1 Aspect statique (Diagramme de classes)

Dans cette section, les descriptions informelle et graphique des principaux concepts constituant un diagramme de classes **UML** sont présentées. Un diagramme de classes décrit l'état statique d'un système en terme de classes, d'attributs, d'associations et de structures d'héritages.

Classe, attribut, clé. Une classe définit un type de données décrit par des attributs et des opérations. On appelle objet une instance de ce type. Un attribut a un type qui restreint et caractérise l'ensemble des valeurs qu'un objet peut prendre pour cet attribut. Le type d'un attribut peut être un type de base (NAT, STRING, BOOL), un ensemble abstrait, un ensemble énuméré, ou un type complexe construit à partir des types de base et des constructeurs "partie-de" ou "séquence-de". Dans ce dernier cas, nous faisons l'hypothèse que ces attributs sont monovalués. En effet, en pratique, un attribut complexe multivalué est plutôt représenté par une association définie entre la classe de l'attribut et une classe représentant le type complexe. Ceci facilite considérablement la manipulation d'un tel attribut. La multiplicité, associée à chaque attribut, indique le nombre de valeurs qu'un objet peut avoir pour cet attribut. Un attribut dont la valeur peut varier est dit mutable. Un tel attribut sera désigné par une étiquette $\{M\}$.

À une classe peut être associée une clé définie comme un ensemble d'attributs permettant de désigner chaque objet de manière unique. Une classe peut avoir plusieurs clés possibles dites candidates. Néanmoins en pratique, une seule clé qualifiée de clé primaire est conservée. Une clé sera désignée par l'étiquette $\{K\}$.

En plus des attributs, une classe **UML** est caractérisée par un ensemble d'opérations. Ces opérations décrivent l'évolution de l'état des objets de cette classe. Dans un système d'information, les classes sont décrites par un ensemble d'opérations dites de base. Ces opérations sont implicitement générées lors de la déclaration de chaque classe. Elles consistent à créer/supprimer des objets et à modifier les valeurs des attributs mutables.

Relations. Les objets d'un système sont en interaction les uns par rapport aux autres. Ces interactions sont modélisées par des liens entre objets. À chaque type de lien correspond une relation entre les classes associées à ces objets. Les liens constituent l'instanciation des relations existantes entre ces classes. **UML** définit deux sortes de relations entre classes : association et héritage.

Association. Une association exprime une liaison sémantique entre objets des différentes classes. En **UML**, la plupart des associations sont binaires (reliant deux classes). Néanmoins, **UML** permet la prise en compte des associations n-aires. Une association n-aire peut être transformée en une association binaire en ajoutant des contraintes supplémentaires [53]. Graphiquement, une association binaire est représentée par un segment reliant les deux classes mises en jeu. Chaque extrémité de ce segment est appelée rôle d'association. Un rôle, éventuellement nommé, est caractérisé par une multiplicité qui indique le nombre minimum et maximum de liens auxquels un objet de la classe opposée peut participer.

Un rôle d'une association est fixe si les éventuels liens auxquels participe un objet donné, de la classe opposée, sont créés à la création de l'objet et sont supprimés à sa disparition. Les liens d'un objet ne peuvent varier durant le cycle de vie de l'objet. Par conséquent, les liens d'une association dont un des rôles est fixe (frozen) n'admettent pas d'opérations indépendantes des classes mises en œuvre.

Plusieurs contraintes peuvent être définies sur une association ou un groupe d'associations. La multiplicité citée ci-dessus est l'une de ces contraintes. Parmi les contraintes pouvant être exprimées sur des associations, figurent l'inclusion, l'exclusion et l'égalité. Plus de détails peuvent être trouvés dans [53]. Comme pour une classe, une association non fixe **UML** est caractérisée par un ensemble d'opérations de base. Ces opérations concernent :

– la création d'un lien d'une association,
– la suppression d'un ensemble de liens d'une association.

Héritage. L'héritage est un concept de modélisation de liens inter-classes. Nous considérons l'héritage dans le sens lien de généralisation/spécialisation. La généralisation d'un ensemble de classes $C_1, \ldots, C_k$ en une classe C consiste à regrouper l'ensemble de leurs caractéristiques communes (attributs et opérations) en une classe de niveau supérieur appelée super-classe. Les classes $C_1, \ldots, C_k$ sont alors dites sous-classes de C. L'héritage modélise l'inclusion des objets de chaque sous-classe dans les objets de la super-classe. Dans le cadre de ce travail, seul l'héritage simple est considéré.

Plusieurs contraintes peuvent être spécifiées sur les classes d'une structure d'héritage donnée. Dans le cadre de ce travail, nous nous intéressons aux contraintes de couverture et de disjonction.

a) **Contrainte de couverture (Complète/Incomplète) :** cette contrainte signifie que chaque objet de la super-classe appartient au moins à l'une des sous-classes. Dans ce cas, la super-classe est dite abstraite ou non instanciable. Graphiquement, le nom d'une classe abstraite est inscrit en italique.
b) **Contrainte de disjonction :** cette contrainte signifie que chaque objet de la super-classe appartient au plus à une des sous-classes. Ceci exprime le fait que les différentes sous-classes sont deux-à-deux disjointes. Graphiquement, cette contrainte est exprimée en attachant l'étiquette {*disjointes*} aux liens d'héritage.

Exemple de diagramme de classes : la figure 3.1 modélise l'aspect statique d'un système de gestion de cassettes. Le lien d'héritage étant disjonctif, une cassette, identifiée par un titre et décrite par le nombre de copies disponibles, peut être soit une série, soit un film. Une série est décrite par un numéro, un film est décrit par un genre. Les acteurs participant à une cassette ne peuvent varier durant la durée de vie de la cassette (car l'association *Participe* est fixe). Un acteur, identifié par un numéro, possède une adresse. Une cassette peut être empruntée par plusieurs clients. Un client, identifié par un numéro et décrit par un attribut indiquant sa suspension ou non, ne peut emprunter plus de cinq cassettes à la fois. Chaque prêt est caractérisé par une date. Afin d'alléger notre diagramme de classes, nous avons choisi de ne pas faire apparaître les opérations de base.

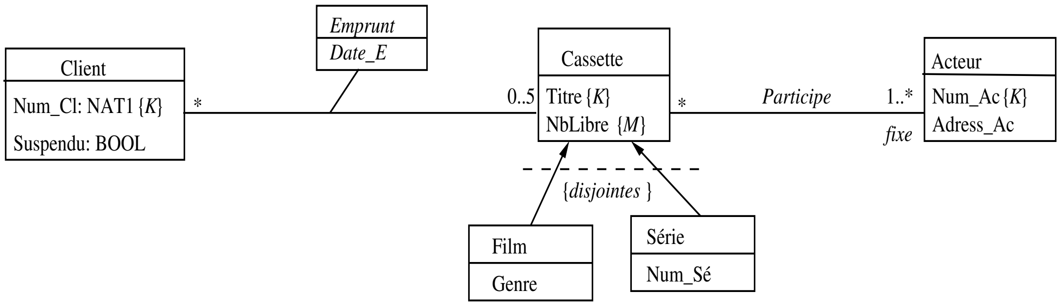


FIG. 3.1 – *Exemple d'un diagramme de classes*

3.1.2 Aspect dynamique

Dans cette section sont présentés deux concepts **UML** permettant la description graphique de l'aspect dynamique d'un système. Il s'agit des diagrammes d'états/transitions et des diagrammes de collaborations permettant la représentation graphique des transactions du système étudié. Dans le cadre de notre travail, nous utilisons les diagrammes états/transitions pour décrire des transactions ne faisant intervenir qu'une seule classe ou association. Si une transaction met en jeu plusieurs classes ou associations, cette dernière est décrite par un diagramme de collaborations.

Diagramme d'états/transitions Un diagramme états/transitions, dérivé des notations de Harel [32], est un graphe orienté formé de nœuds dits états et d'arcs dits transitions. Il permet la description du comportement commun et isolé des objets d'une classe ou d'une association indépendamment des objets des autres classes. Un diagramme états/transitions visualise les différents états dans lesquels un objet de la classe considérée peut se trouver. Le changement d'état, représenté par des transitions, est déclenché par des événements.

Événement. Un événement est un moyen de transmission d'information. Les événements permettent le déclenchement des changements d'états des objets. Un événement est caractérisé par son nom, la liste des paramètres d'entrée et l'objet destinataire ou cible.

État. À un instant donné, chaque objet du système se trouve dans un état bien défini. Un état est une abstraction des valeurs des attributs de l'objet considéré et des liens que peut posséder cet objet avec d'autres objets. Un objet possède un état initial et éventuellement un ou plusieurs états finaux.

Transition. Une transition est le passage pour un objet d'un état source à un état cible pas nécessairement distincts. Une transition est soit déclenchée par un événement, soit automatique, soit temporisée. Le déclenchement d'une transition automatique est lié à la satisfaction d'une condition donnée. Une transition temporisée est déclenchée par l'écoulement d'un laps de temps bien défini. Dans le cadre de ce travail, on s'intéresse aux deux premiers types de transitions. Rappelons que le changement d'état est fait de façon instantanée de manière à ce que l'objet soit toujours dans un état défini.

Garde. Une garde est une condition booléenne qui autorise ou interdit le déclenchement d'une transition à l'occurrence d'un événement donné. Les gardes permettent de réduire l'indéterminisme lorsque a priori plusieurs transitions sont possibles lors de l'arrivée d'un événement. La garde est exprimée en fonction des paramètres de l'événement et des valeurs de l'objet concerné (valeurs des attributs et des liens).

Exemple: l'exemple de la figure 3.2 illustre les différents états dans lesquels un objet de la classe *client* peut se trouver. À la réception de l'événement de création d'un client, l'opération *BasicAjoutClient* est invoquée sans aucune condition supplémentaire. En revanche, à l'apparition de l'événement *SuppressionClient*, une alternative est possible : le client est en règle et s'il n'a pas de prêts en cours, il est supprimé par invocation de l'opération *BasicSuppressionClient* sinon il est suspendu momentanément par invocation de l'opération *BasicChangeSuspendu*.

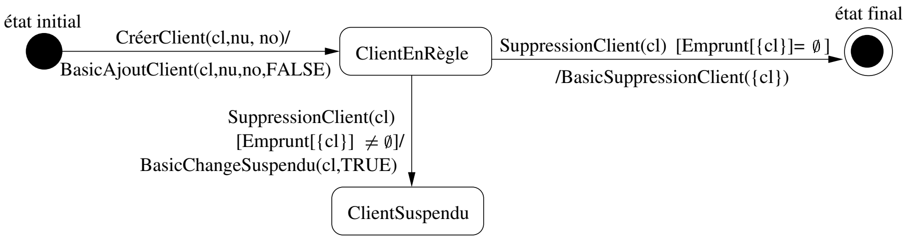


FIG. 3.2 – *Diagramme états/transitions de la classe Client*

Dans le cadre de ce travail, on considère que les gardes des transitions associées à un état et un événement donnés sont exclusives. De même, on considère que la disjonction exclusive des prédicats décrivant les états est toujours vraie. Cette dernière caractéristique assure qu'à un instant donné, chaque objet ne se trouve que dans un état et un seul.

3.1.3 Diagrammes de collaborations

En UML, les diagrammes de collaborations permettent de visualiser les interactions des objets des différentes classes du système. Ces interactions sont matérialisées par des envois de messages entre objets. Le système étant en interaction continue avec l'environnement où il évolue, ce dernier reçoit des messages externes de son environnement. Au niveau conceptuel, nous utilisons les diagrammes de collaborations afin de décrire l'effet du message déclencheur sur les différentes classes et associations indépendamment de tout ordre d'exécution. Cet effet est représenté en termes d'appels aux opérations spécifiées sur ces classes et associations. Pour cela, on s'interdit les envois de messages entre les objets de classes différentes d'un diagramme de collaborations. On fait l'hypothèse de l'existence d'un objet particulier appelé *Système* qui d'une part reçoit les messages externes de l'environnement, et d'autre part, envoie aux différents objets les messages internes nécessaires au traitement du message reçu en entrée (Voir figure 3.3).

La sémantique du diagramme de la figure 3.3 est la suivante : le message *m* ne peut survenir que si la condition C est vérifiée. Dans ce cas, si la garde associée à un message interne est vérifiée alors ce dernier est envoyé à la classe correspondante.

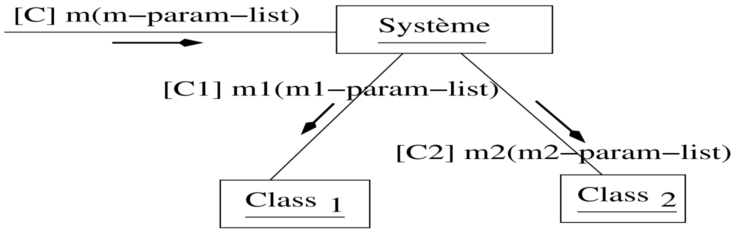


FIG. 3.3 – *Structure des diagrammes de collaborations UML dédiés SI*

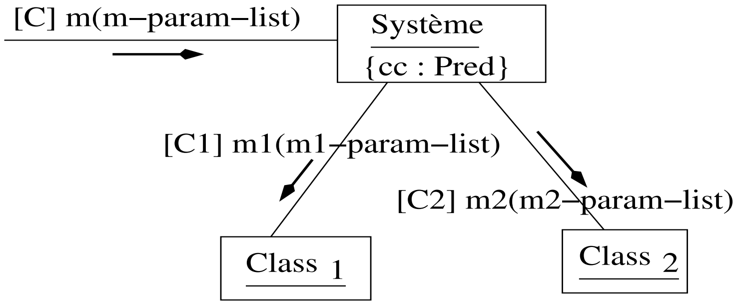


FIG. 3.4 – *Un exemple spécifique de diagrammes de collaborations*

Souvent dans le systèmes d'information, les paramètres des messages internes sont déterminés de manière dynamique. Ceci introduit une notion de séquence sur le traitement. En effet, ces paramètres doivent donc être déterminés avant l'envoi des messages internes. La notion de séquence n'étant pas admise au niveau le plus abstrait du langage **B**, on propose pour cela une notation particulière de ces diagrammes (Voir figure 3.4). La sémantique de ce type de diagrammes est la suivante : à l'occurrence de l'événement *m* (la condition C est donc vérifiée), le système sélectionne un objet *cc* satisfaisant le prédicat *Pred*, et envoie chaque message *mi* dont la garde est vraie.

3.2 Traduction en B des diagrammes UML

Dans cette partie, nous rappelons les règles de traduction en **B** des concepts **UML** présentés précédemment. Une présentation plus détaillée de ces règles est donnée dans [53]. Nous illustrons ces règles de traduction à travers des exemples.

3.2.1 Aspect statique (Diagramme de classes)

Classe, attribut, clé, opération. Une classe C, ne participant pas à une structure d'héritage, est traduite en une machine abstraite *BasicC*. Cette machine définit un ensemble abstrait C représentant l'ensemble des (identités d') objets possibles, et une variable abstraite c associée à la classe (qu'on notera dorénavant $v.a.c$) représentant l'ensemble des objets existants à un instant donné. Un attribut *Att* de type T_{Att} d'une classe C est modélisé par une relation définie de la $v.a.c$ c dans le type T_{Att}. Selon la multiplicité de cet attribut, un ou plusieurs invariants **B** sont ajoutés. En particulier, un attribut obligatoire (resp. optionnel) monovalué est modélisé par une fonction totale (resp. partielle). L'attribut (ou l'ensemble des attributs) clé d'une classe est modélisé par une fonction injective.

L'invariant de la machine traduisant une classe comprend trois types de prédicats :

1. typage de la variable associée à la classe *C*,
2. typage des attributs,
3. typage des éventuelles clés de la classe,
4. typage des éventuelles associations dont le rôle relatif à la classe C est fixe.

Pour chaque classe, sont générées automatiquement des opérations de base d'ajout/suppression d'objets et de modification des attributs mutables.

Ajout d'un objet. Comme toute opération **B**, l'opération d'ajout d'un objet d'une classe *C* doit rétablir l'invariant contenu dans la machine associée à la classe *C*. L'opération d'ajout d'un objet d'une classe *C* a pour paramètres les éléments suivants:

1. l'objet *cc* à ajouter avec la précondition $cc \in C$ - c,
2. la valeur de chaque attribut obligatoire Att. Si l'attribut considéré est monovalué (resp. multivalué) alors cette valeur est atomique (resp. ensembliste) et est du type de l'attribut.
3. les associations fixes définies sur cette classe sont traitées de la même manière qu'un attribut de la classe.

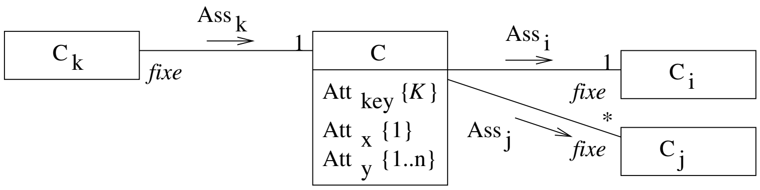


FIG. 3.5 – *Exemple d'une classe et de ses différentes caractéristiques*

Formellement, l'opération d'ajout d'un objet dans la classe C de la figure 3.5 est modélisée par :

BasicAjoutC(cc, val_att$_{key}$,...,val_att$_n$, ElemC$_1$,..., ElemC$_m$)=
PRE
$cc \in C - c \wedge$
$val\_att_{key} \in Type\_Att_{key} - ran(Att_{key}) \wedge$
$val\_att_i \in Type\_Att_i \wedge$
$val\_att_j \subseteq Type\_Att_j \wedge$
$ElemC_i \in C_i \wedge$
$ElemC_j \subseteq C_j \wedge$// $ElemC_j \neq \emptyset$ si la multiplicité minimale est 1
$ElemC_k \in C_k$
THEN
$c := c \cup \{cc\} \parallel$
$Att\_var_{key} := Att\_var_{key} \cup \{cc \mapsto val\_att_{key}\} \parallel$
$Att\_var_x := Att\_var_x \cup \{cc \mapsto val\_att_i\} \parallel$
$Att\_var_y := Att\_var_y \cup \{cc\} \times val\_att_j \parallel$
$Ass_i := Ass_i \cup \{cc \mapsto ElemC_i\} \parallel$
$Ass_j := Ass_j \cup \{cc\} \times ElemC_j \parallel$
$Ass_k := Ass_k \cup \{ElemC_k \mapsto cc\}$
END

C_i, C_j et C_k sont les ensembles abstraits associés aux classes correspondantes.

Suppression d'objets. La suppression d'un ensemble d'objets d'une classe *C* se fait simplement en fournissant l'ensemble de ces objets. La suppression d'un ensemble d'objets cc consiste à retirer les éléments cc de la $v.a.c$ c, et à mettre à jour les différentes variables, associées aux attributs et associations fixes, définies sur les classes associées à ces objets. La suppression d'un ensemble d'objets cc de la classe *C*, de la figure 3.5 est modélisée comme suit :

BasicSuppressionC(cc)=
PRE $cc \subseteq c$
THEN
$c := c - cc \parallel$
$Att\_var_m := cc \lhd Att\_var_m \parallel$ //$m \in \{Key,x,y\}$
$Ass_n := cc \lhd Ass_n \parallel$ //$n \in \{i,j\}$
$Ass_p := Ass_p \rhd cc$ //$p \in \{k\}$
END

Modification de la valeur d'un attribut. Pour chaque attribut *Att* mutable d'une classe *C*, annoté par l'étiquette $\{M\}$ dans le diagramme de classes, une opération de modification de la valeur de cet attribut est générée. Cette opération reçoit en entrée l'objet *cc*, de la classe *C*, pour laquelle on voudrait modifier la valeur ainsi que la nouvelle valeur val. La nouvelle valeur doit être du type *TT* de l'attribut. Cette opération est spécifiée par :

BasicChangeAtt(cc, val)=
PRE $cc \in c \wedge val \in TT$
THEN $Att := Att \lhd\!\!\!- \{cc \mapsto val\}$
END

Association. Une association est modélisée par une variable abstraite définie comme une relation entre les variables abstraites associées aux classes qu'elle met en jeu. Comme pour un attribut, selon les multiplicités des deux rôles, un ou plusieurs invariants **B** sont ajoutés.

Si l'un au moins des rôles d'une association est fixe, la modélisation **B** de cette association est définie dans la machine associée à la classe du rôle en question. Cette machine utilise (lien **USES**) la machine traduisant la classe opposée. Ce lien, offrant le partage des variables, permet la définition de l'association considérée.

À une association *Ass* dont les rôles sont non fixes est associée une machine abstraite *BasicAss*. Comme précédemment, cette machine utilise les machines associées aux classes mises en jeu.

Pour chaque association, des opérations génériques d'ajout/suppression de liens sont automatiquement générées.

Ajout d'un lien. L'ajout d'un lien d'une association *Ass*, définie sur les classes C_1 et C_2, consiste à ajouter à la variable représentant l'association Ass le tuple $cc_1 \mapsto cc_2$ où cc_1 et cc_2 désignent des objets de C_1 et C_2 respectivement. Comme pour l'ajout d'un objet d'une classe, on affecte une valeur à chaque attribut obligatoire de cette association. La précondition de cette opération comprend le typage des objets à relier ainsi que celui des valeurs associées aux attributs de l'association. De manière générale, l'opération d'ajout d'un lien s'exprime comme suit :

BasicAjoutAss(cc_1, cc_2, val_att_1,...,val_att_n)=
PRE
Typage des objets cc_1 *et* cc_2 $\wedge$
$cc_1 \mapsto cc_2 \notin Ass \wedge$
Typage des attributs Att_i *de l'association*

THEN
$Ass := Ass \cup \{cc_1 \mapsto cc_2\} \parallel$
$\parallel_{i=1..n}$ $Att\_var_i := Att\_var_i \cup \{cc_1 \mapsto cc_2 \mapsto val\_att_i\}$
END

Suppression de liens. Il existe différents types d'opérations de suppression de liens. En effet, un lien (ou un ensemble de liens) peut être identifié de trois manières :

- les objets sources des liens,
- les objets cibles des liens,
- les objets source et cible du lien.

La suppression de liens d'une association consiste à soustraire l'ensemble de ces liens de l'association et aussi de l'ensemble des variables correspondant aux attributs de l'association. Soit une association Ass définie de la classe C_1 dans la classe C_2. Les différentes opérations de suppression de liens de cette association sont données ci-après :

a) **Suppression des liens associés à un ensemble d'objets de** C_1

BasicSuppressionAssC$_1$(cc$_1$)=
PRE $cc_1 \subseteq c_1$
THEN
$Ass := cc_1 \lhd\!\!\!- Ass \parallel$
$\parallel_{i=1..n} Att_i := (cc_1 \times c_2) \lhd\!\!\!- Att_i$
END

b) **Suppression des liens associés à un ensemble d'objets de** C_2

BasicSuppressionAssC$_2$(cc$_2$)=
PRE $cc_2 \subseteq c_2$
THEN
$Ass := Ass \,\, -\!\!\!\rhd cc_2 \parallel$
$\parallel_{i=1..n} Att_i := (c_1 \times cc_2) \lhd\!\!\!- Att_i$
END

c) **Suppression d'un lien**

BasicSuppressionAss(cc$_1$, cc$_2$)=
PRE $cc_1 \in c_1 \wedge cc_2 \in c_2$
THEN
$Ass := Ass - \{cc_1 \mapsto cc_2\} \parallel$
$\parallel_{i=1..n} Att_i := \{cc_1 \mapsto cc_2\} \lhd\!\!\!- Att_i$
END

Exemple : la classe *Cassette* et l'association fixe *Participe*, de la figure 3.1, sont représentées par la machine **B** suivante :

MACHINE *BasicCassette*
USES *BasicActeur*
SETS *Cassette*
VARIABLES *cassette, Titre, NbLibre, Participe*
INVARIANT
$cassette \subseteq Cassette \wedge$
$Titre \in cassette \rightarrowtail STRING \wedge$

$NbLibre \in cassette \rightarrow NAT \wedge$
$Participe \in Acteur \leftrightarrow cassette \wedge$
$ran(Participe) = cassette$
INITIALISATION $cassette, Titre, NbLibre, Participe := \emptyset,\emptyset,\emptyset,\emptyset$
OPERATIONS
BasicAjoutCassette(ca, ti, nb,ac)=
PRE $ca \in Cassette\text{-}cassette \wedge ti \in STRING\text{-}ran(Titre) \wedge nb \in NAT \wedge ac \subseteq acteur \wedge ac \neq \emptyset$
THEN
$cassette := cassette \cup \{ca\} \parallel$
$Titre := Titre \cup \{ca \mapsto ti\} \parallel$
$NbLibre := NbLibre \cup \{ca \mapsto nb\} \parallel$
$Participe := Participe \cup \{ca\} \times ac$
END;
BasicSuppressionCassette(Ca)=
PRE $Ca \subseteq cassette$
THEN
$cassette := cassette - Ca \parallel$
$Titre := Ca \lhd Titre \parallel$
$NbLibre := Ca \lhd NbLibre \parallel$
$Participe := Participe \rhd Ca$
END;
BasicChangeNbLibre(ca, nb)=
PRE $ca \in cassette \wedge nb \in NAT$
THEN $NbLibre := NbLibre \Leftarrow \{ca \mapsto nb\}$
END
END

De même, l'association *Emprunt* est représentée par la machine :

MACHINE *BasicEmprunt*
USES *BasicCassette, BasicClient*
VARIABLES *Emprunt, Date_E*
INVARIANT
$Emprunt \in client \leftrightarrow cassette \wedge$
$\forall cl. (cl \in client \Rightarrow card(Emprunt[\{cl\}]) \leq 5 \wedge$
$Date\_E \in Emprunt \rightarrow NAT$
INITIALISATION $Emprunt, Date\_E := \emptyset,\emptyset$
OPERATIONS
BasicAjoutEmprunt(cl, ca, da)=
PRE
$ca \in Cassette \wedge cl \in Client \wedge$
$cl \mapsto ca \notin Emprunt \wedge da \in NAT$
THEN
$Emprunt := Emprunt \cup \{cl \mapsto ca\} \parallel$
$Date\_E := Date\_E \cup \{cl \mapsto ca \mapsto da\}$
END;
BasicSuppressionEmpruntClient(Cl)=
PRE $Cl \subseteq client$
THEN

Emprunt:= $Cl \lhd$ *Emprunt* $\|$
Date_E:= $(Cl \times Cassette) \lhd$ *Date_E*
END;
BasicSuppressionEmpruntCassette(Ca)=
PRE $Ca \subseteq cassette$
THEN
Emprunt:= Emprunt $\rhd Ca \|$
Date_E:= $(Client \times Ca) \lhd$ *Date_E*
END;
BasicSuppressionEmprunt(cl, ca)=
PRE $cl \in client \wedge ca \in cassette$
THEN
Emprunt:= Emprunt - $\{cl \mapsto ca\} \|$
Date_E:= $\{cl \mapsto ca\} \lhd$ *Date_E*
END
END

Héritage. À chaque classe d'une hiérarchie d'héritage est associée une machine abstraite. Une machine supplémentaire permettant de modéliser le lien d'héritage est ajoutée. Cette machine inclut l'ensemble des machines associées aux différentes classes de la hiérarchie. C'est dans la machine incluante que seront spécifiées les opérations réellement exécutées sur les sous-classes. Formellement, la spécification **B** traduisant une structure d'héritage de sous-classes $C_1, \ldots, C_k$ associées à la super-classe C est modularisée comme suit :

– à la super-classe C est associée une machine abstraite comportant la traduction de cette super-classe : un ensemble abstrait C, une variable abstraite associée à cette classe c, les variables abstraites associées aux attributs et un ensemble d'opérations de base,
– chaque sous-classe C_i est traduite par une machine abstraite qui utilise la machine associée à la super-classe afin d'avoir accès à la variable de classe c. Cette machine ne contenant pas d'ensemble abstrait, sa variable abstraite associée à la classe c_i est typée par : $c_i \subseteq c$. Un ensemble d'opérations de base est spécifié,
– la sémantique d'héritage est modélisée par une machine supplémentaire qui inclut les machines précédentes. On définit dans cette machine les invariants associés aux contraintes inter-classes et les opérations qui prennent en compte la sémantique d'héritage. Ces opérations font appel aux opérations de base spécifiées dans les machines précédentes

Exemple. La structure d'héritage de la figure 3.1, est traduite par les machines suivantes (la machine associée à la classe *Cassette* est donnée précédemment) :

• La machine associée à la sous-classe *Série*.

MACHINE *BasicSérie*
VARIABLES *série, Num_Sé*
USES *BasicCassette*
INVARIANT
série $\subseteq$ *cassette* $\wedge$
Num_Sé $\in$ *série* $\rightarrow NAT$
INITIALISATION *série, Num_Sé:=*$\emptyset,\emptyset$
OPERATIONS

BasicAjoutSérie(se,nu)=
PRE *se* $\in$ *Cassette - série* $\wedge$ *nu* $\in$ *NAT*
THEN
série := série $\cup$ *{se}* $\|$
Num_Sé :=Num_Sé $\cup$ *{se* $\mapsto$ *nu}*
END;
BasicSuppressionSérie(Sé)=
PRE *Sé* $\subseteq$ *série*
THEN
série := série - Sé $\|$
Num_Sé := Sé $\lhd$ *Num_Sé*
END
END

• La machine associée à la sous-classe *Film* est similaire à la précédente.

• La machine représentant la hiérarchie d'héritage :

MACHINE *Cassette*
INCLUDES *BasicCassette, BasicSérie, BasicFilm*
//promotion d'opérations jugées d'interface
PROMOTES *BasicAjoutCassette, ...*
INVARIANT
//spécifier par exemple que l'héritage est disjonctif
série $\cap$ *film=* $\emptyset$
...
OPERATIONS
AjoutSérie(se, nu, nb, ti)=
PRE *se* $\in$ *Cassette - cassette* $\wedge$ *nu* $\in$ *NAT* $\wedge$ *nb* $\in$ *NAT* $\wedge$ *ti* $\in$ *STRING-ran(Titre)*
THEN
BasicAjoutCassette(se, ti, nb) $\|$
BasicAjoutSérie(se, nu)
END;
SuppressionSérie(Se)=
PRE *Se* $\subseteq$ *série*
THEN
BasicSuppressionCassette(Se) $\|$
BasicSuppressionSérie(Se)
END;
...
END

3.2.2 Aspect dynamique

Dans cette section, nous donnons la traduction formelle en **B** des diagrammes états/transitions et de collaborations "dédiés" systèmes d'information. Comme pour l'aspect statique, cette traduction est illustrée à travers des exemples.

Traduction d'un diagramme états/transitions. Un diagramme états/transitions associé à une classe *C* est traduit par une machine abstraite *ÉtatC* qui inclut la machine de base *BasicC* afin de pouvoir appeler ses opérations. Cette machine utilise éventuellement les autres machines afin d'avoir accès aux différentes variables utilisées pour l'expression des états et des gardes associées aux transitions.

Chaque évènement du diagramme états/transitions est traduit par une opération **B**. La précondition de cette opération contient le typage des paramètres d'entrée de l'événement et la disjonction d'un ensemble de conjonctions. Chaque conjonction, associée à une transition, est constituée du prédicat décrivant l'état, de l'éventuelle garde de la transition et de la précondition de l'action associée à la transaction. Afin d'obtenir une traduction **B** complète, nous exigeons que :

- les prédicats associés aux états et les gardes des transitions soient exprimés en syntaxe **B**. Par exemple l'état *ClientEnRègle*, de la figure 3.2 est exprimé en **B** par :

$$\textit{ClientEnRègle(cl)} \triangleq \textit{cl} \in \textit{Suspendu}^{-1}\textit{[\{FALSE\}]}$$

- les actions associées aux transitions correspondent aux opérations de base de la classe considérée.

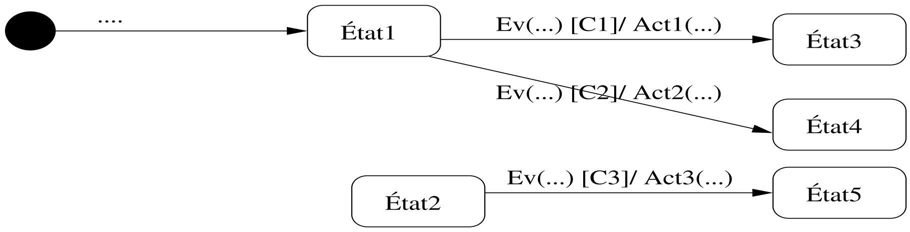


FIG. 3.6 – *Structure d'un diagramme états/transitions*

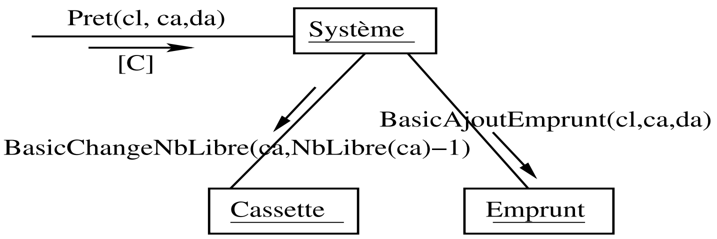
L'événement Ev du diagramme états/transitions de la figure 3.6 est traduit en **B** par :

Ev(. . .)=
PRE
  Typage des paramètres d'entrée $\wedge$
  [(*État1* $\wedge$ *C1* $\wedge$ $Precond_{Act1}$) $\vee$ (*État1* $\wedge$ *C2* $\wedge$ $Precond_{Act2}$) $\vee$ (*État2* $\wedge$ *C3* $\wedge$ $Precond_{Act3}$)]
THEN
  SELECT *État1* $\wedge$ *C1* **THEN** *Act1(. . .)*
  WHEN *État1* $\wedge$ *C2* **THEN** *Act2(. . .)*
  WHEN *État2* $\wedge$ *C3* **THEN** *Act3(. . .)*
  END
END

avec $Precond_{Act_i}$ désignant la précondition de l'opération de base Act_i.

Exemple : l'événement *SupprimerClient*, de la figure 3.2, est traduit par l'opération **B** suivante :

SupprimerClient(cl)
PRE
  cl $\in$ *client* $\wedge$
  [(*ClientEnRègle(cl)* $\wedge$ *Emprunt[{cl}]*= $\emptyset$ $\wedge$ *{cl}* $\subseteq$ *client*) $\vee$
  (*ClientEnRègle(cl)* $\wedge$ *Emprunt[{cl}]* $\neq$ $\emptyset$ $\wedge$ *cl* $\in$ *client*)]
THEN
  SELECT (*ClientEnRègle(cl)* $\wedge$ *Emprunt[{cl}]*= $\emptyset$) **THEN** *BasicSuppressionClient({cl})*



$$C \mathrel{\widehat{=}} (cl \mapsto ca \notin Emprunt \wedge Emprunt[\{cl\}] \leq 4 \wedge NbLibre(ca) > 0)$$

FIG. 3.7 – *Exemple d'un diagramme de collaborations*

WHEN (*ClientEnRègle(cl)* $\wedge$ *Emprunt[{cl}]* $\neq \emptyset$) **THEN** *BasicChangeSuspendu(cl, TRUE)*
END
END

Diagramme de collaborations. La traduction des différents diagrammes de collaborations est contenue dans une machine abstraite *Système*. Cette machine inclut les machines associées aux diagrammes états/transitions des classes du système. Si à une classe ou association n'est associé aucun diagramme d'états/transitions, la machine *Système* inclut dans ce cas les machines de base traduisant ces classes et ces associations. Chaque diagramme de collaborations est traduit par une opération **B** associée au message déclencheur du diagramme. Les diagrammes de collaborations des figures 3.3 et 3.4 sont traduits par les opérations suivantes :

m(m-param-list)
PRE
Typage des paramètres d'entrée $\wedge$ *C*
THEN
IF *C1* **THEN** *m1(m1-param-list)* **END**||
IF *C2* **THEN** *m2(m2-param-list)* **END**
END

m(m-param-list)
PRE
Typage des paramètres d'entrée $\wedge$ *C*
THEN
ANY *cc* **WHERE** *Pred* **THEN**
IF *C1* **THEN** *m1(m1-param-list)* **END** ||
IF *C2* **THEN** *m2(m2-param-list)* **END**
END
END

Exemple: le diagramme de collaborations de la figure 3.7 décrivant le prêt d'une cassette par un client est traduit par l'opération **B** :

Prêt(cl, ca, da)=
PRE $cl \in client \wedge ca \in cassette \wedge da \in NAT \wedge$
$cl \mapsto ca \notin Emprunt \wedge Emprunt[\{cl\}] \leq 4 \wedge NbLibre(ca) > 0$
THEN
BasicAjoutEmprunt(cl, ca, da) ||
BasicChangeNbLibre(ca, NbLibre(ca)-1)
END

3.2.3 Architecture générale de la spécification abstraite

L'application des règles de traduction définies précédemment permet l'obtention d'une architecture générique constituée de trois niveaux. Ces trois niveaux correspondent aux trois niveaux d'abstraction de la description **UML** d'un système : diagramme de classes, diagrammes états/transitions et diagrammes de collaborations. La figure 3.8 représente l'architecture de la spécification correspondant au système de gestion des cassettes.

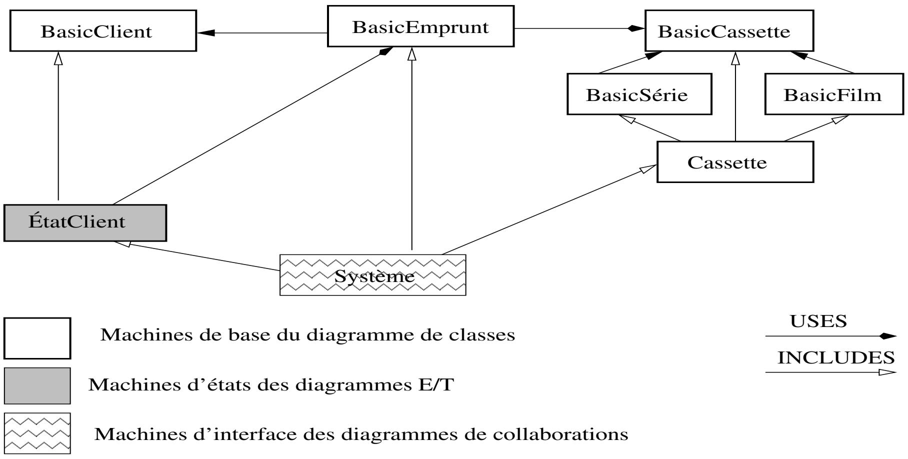


FIG. 3.8 – *Architecture globale de la spécification abstraite*

Cette architecture permet une bonne compréhension des spécifications formelles par rapport aux spécifications **UML** de départ. Elle est bien adaptée aux éventuelles extensions du système : ajout de classes ou d'associations, etc. Elle permet une meilleure localisation des incohérences qui peuvent être soulevées durant la phase de validation des spécifications abstraites générées. En effet, les règles de traduction étant inversibles, l'élément **UML** correspondant à une incohérence détectée lors de la phase de preuve peut être déterminé de manière efficace.

3.2.4 Preuve

L'ensemble des spécifications obtenues par le processus de traduction décrit ci-dessus ont été validées en utilisant l'AtelierB. Les principaux résultats de cette phase de validation sont les suivants :

- La correction des machines abstraites de base est entièrement automatique.
- Les machines correspondant aux diagrammes états/transitions ne contenant pas d'invariants, les seules obligations de preuves générées pour ces machines concernent les appels aux opérations de base. En effet, pour chaque branche du **SELECT** de prédicat $Pred$ faisant appel à l'opération de base op_b, une obligation de preuve, testant la satisfaction de la précondition associée à cette opération de base est générée. Cette preuve est de la forme :

$$(Precond \wedge Pred) \Rightarrow Precond_{op_b} \tag{3.1}$$

avec $Precond$ et $Precond_{op_b}$ désignant les préconditions de l'opération contenant la substitution **SELECT** et de l'opération appelée respectivement. La prédicat $Pred$ est la conjonction du prédicat $Pred_{etat_k}$ de l'état source et de l'éventuelle garde $garde_l$ associés à la transition.

$$Pred \mathrel{\widehat{=}} Pred_{etat_k} \wedge garde_l \tag{3.2}$$

Par construction des opérations issues des diagrammes états/transitions, la précondition $Precond$ contient un terme de la forme :

$$\bigvee_{i,j}(Pred_{etat_i} \wedge garde_j \wedge Precond_{op_{b_j}}) \tag{3.3}$$

L'un des termes $(Pred_{etat_i} \wedge garde_j)$ est évidement $Pred$ associé à l'opération op_b. Prouver la formule (3.1) revient à prouver :

$$Precond_{op_b} \tag{3.4}$$

sous l'hypothèse $(Precond \wedge Pred)$. La formule (3.3) étant vérifiée, l'un au moins des termes de cette disjonction est vérifié. Or nous avons posé la restriction que seul un terme $(Pred_{etat_i} \wedge garde_j)$ de la formule (3.3) peut être vérifié à la fois. La formule $Pred$ étant vérifiée donc $Precond_{op_b}$ l'est aussi. Dans le prouveur l'AtelierB, l'obligation de preuve (3.1) est déchargée automatiquement.

– La correction de la machine d'interface *Système* génère deux types de preuves. Le premier type de preuves, concerne le test de satisfaction des préconditions des opérations appelées. Pour chaque branche du **IF** de prédicat $Pred$ faisant appel à une opération op, une obligation de preuve, testant la validité de la précondition associée à cette opération, est générée. Cette preuve est de la forme :

$$(Precond \wedge Pred) \Rightarrow Precond_{op}$$

avec $Precond$ et$Precond_{op}$ désignant les préconditions de l'opération contenant la substitution **IF** et de l'opération appelée respectivement.
Par construction des opérations issues des diagrammes de collaborations, la précondition $Precond$ contient l'implication $Pred \Rightarrow Precond_{op}$. La preuve précédente devient triviale et est donc déchargée automatiquement par l'AtelierB.
Le deuxième type concerne la preuve de rétablissement des invariants, spécifiés dans les différentes machines ayant des liens **USES**, par chaque opération de la machine système.

3.3 Méthodologie de développement d'applications sûres

Rappelons que les opérations **B** ont été décrites avec une stratégie de programmation offensive : utilisation des préconditions. Pour chaque opération, on définit les conditions sous lesquelles cette opération doit être appelée afin de préserver les contraintes d'intégrité spécifiées dans la clause invariant de la machine. En utilisant le prouveur de l'AtelierB, la spécification ainsi obtenue est prouvée formellement correcte. En effet, pour chaque opération Op de la machine, une obligation de preuve est automatiquement générée. Cette obligation de preuve permet de vérifier que, si la précondition Q de l'opération Op, et l'invariant Inv de la machine sont vérifiés, alors l'exécution de l'opération Op rétablit l'invariant Inv :

$$(Q \wedge Inv) \Rightarrow [Op]Inv$$

Cependant, établir cette preuve n'interdit pas que l'opération puisse être appelée sans que sa précondition soit vérifiée. Le système pourrait donc se retrouver dans un état incohérent. En effet, une précondition ne garantit un résultat (exécution correcte de la substitution associée) que si elle est vraie, dans le cas contraire le résultat devient imprévisible : violation des contraintes, non terminaison du calcul, infaisabilité, etc. Par opposition à la précondition, une substitution conditionnée **IF** a toujours un comportement bien précis : selon la valeur booléenne de la condition, la branche **THEN** ou **ELSE** est exécutée. Un niveau supplémentaire de sécurité s'impose. Nous proposons pour cela d'ajouter une machine (*RobusteSystème*) qui permet d'assurer que chaque opération n'est appelée que si sa précondition est vérifiée. Cette machine contient donc les transactions bases de données sûres. Les opérations, spécifiées dans cette machine, n'ont comme précondition que le typage des paramètres d'entrée. Elles correspondent aux opérations utilisateurs qui appellent (si les conditions sont vérifiées) les opérations spécifiées au niveau de la machine initiale.

La machine *RobusteSystème* peut être construite comme suit :

- La machine *RobusteSystème* inclut la machine *Système*.

- À chaque opération *op* de la machine *Système* :

  op($x_1, \ldots, x_n$)=
  PRE
  $\wedge_{i=1..n}$ *cond_typage(x_i)*$\wedge_{i=1..k}$ *Precond$_i$*
  THEN S
  END

  correspond une opération *Robusteop* dans la machine *RobusteSystème*. L'opération *Robusteop* est obtenue comme suit :

  resultat⟵**Robusteop($x_1, \ldots, x_n$)=**
  PRE
  $\wedge_{i=1..n}$ *cond_typage(x_i)*
  THEN
  IF *Precond$_1$* **THEN**
  IF *Precond$_2$* **THEN**
  ...
  IF *Precond$_k$* **THEN**
  op($x_1, \ldots, x_n$) || *resultat:=OK*
  ELSE *resultat:=NonPrecond$_k$*
  END
  ...
  ELSE *resultat:=NonPrecond$_2$*
  END
  ELSE *resultat:=NonPrecond$_1$*
  END
  END

  avec :

  – *cond_typage(x_i)* correspond au typage le moins contraignant du paramètre x_i. C'est le typage par rapport aux ensembles abstraits ou aux types de base de **B**.
  – $\wedge_{i=1..k}$ *Precond$_i$* est la partie restante de la précondition globale de l'opération *op*.

Exemple : soit la machine d'interface *Système* qui contient l'opération traduisant le diagramme de collaborations de la figure 3.7.

MACHINE *Système*
INCLUDES *ÉtatClient, Cassette, BasicEmprunt*
OPERATIONS
Prêt(cl, ca, date)=
PRE
$ca \in cassette \wedge cl \in client \wedge date \in NAT \wedge cl \mapsto ca \notin Emprunt \wedge$
$card(Emprunt[\{cl\}]) \leq 4 \wedge NbLibre(ca) > 0$
THEN

BasicAjoutEmprunt(cl, ca, date) ||
BasicChangeNbLibre(ca, NbLibre(ca)-1) ||
END
END

La machine *RobusteSystème* correspondant à cette machine est la suivante :

MACHINE *RobusteSystème*
SETS *Résultat={Ok, NbMaxAtteint, CassetteDejaEmpruntéeParClient, CassetteInexistante, CassetteIndisponible, ClientInexistant}*
INCLUDES *Système*
OPERATIONS
resultat $\longleftarrow$ **RobustePrêt(cl, ca, date)=**
PRE
$ca \in Cassette \wedge cl \in Client \wedge date \in NAT$
THEN
IF $ca \in cassette$ **THEN**
IF $NbLibre(ca) > 0)$ **THEN**
IF $cl \in client$ **THEN**
IF $ca \mapsto cl \notin Emprunt$ **THEN**
IF $card(Emprunt^{-1}[\{cl\}]) \leq 4$ **THEN**
AjoutEmp(cl, ca, date)||
resultat := Ok
ELSE *resultat:=NbMaxAtteint*
END
ElSE *resultat := CassetteDejaEmpruntéeParClient*
END
ElSE *resultat := ClientInexistant*
END
ELSE *resultat:= CassetteIndisponible*
END
ELSE *resultat:= CassetteInexistante*
END
END
END

La construction de la machine *RobusteSystème* n'est pas entièrement automatique. En effet, les différents messages d'erreurs ne peuvent être générés automatiquement car ils dépendent de la sémantique de la précondition associée. Le concepteur devra compléter manuellement ces parties de spécification. Il peut aussi définir ses propres stratégies de gestion d'erreurs.

Une fois la machine *Système* prouvée correcte par rapport à son invariant, il est évident que la machine *RobusteSystème* l'est aussi. Les preuves associées à la correction de cette machine sont déchargées par le prouveur de l'AtelierB. En effet, cette machine ne contenant pas d'invariants supplémentaires, seules les préconditions des opérations appelées sont à vérifier. Comme chaque opération n'est appelée que sous ses préconditions (toutes les préconditions sont testées avant l'appel de l'opération), alors la machine *RobusteSystème* est correcte.

Remarque Il est évidemment possible de considérer directement la machine *RobusteSystème* sans passer par la machine *Système*. Une telle solution entraîne le traitement des cas d'erreurs dès les premières étapes de conception. Ceci ne reflète pas la démarche habituelle de développement qui consiste dans un premier temps à décrire la fonction à réaliser sous l'hypothèse que toutes les conditions requises au bon déroulement sont réunies. Le traitement des cas d'exception est souvent effectué à une étape ultérieure.

3.4 Conclusion

Dans ce chapitre ont été présentés les concepts graphiques **UML** permettant la description des systèmes d'information. Ces concepts ont été spécialisés par ajout de propriétés (pour les attributs et les classes) afin de pouvoir exprimer des caractéristiques additionnelles largement utilisées dans la modélisation des systèmes d'information. Nous avons essentiellement ajouté la propriété de clé pour les classes et la propriété de multiplicité pour les attributs. Le modèle dynamique, par contre, a été restreint pour répondre au niveau conceptuel dans lequel on se place. Les diagrammes d'états/transitions et de collaboration ont été utilisés pour l'étude de l'aspect dynamique du système. Ces diagrammes se sont avérés suffisants pour la modélisation des fonctionnalités du système. Une méthodologie de conception d'applications sûres a également été abordée. Un outil permettant la mise en pratique de ce processus de traduction est décrit au chapitre 7.

Chapitre 4

Raffinement B dédié à la génération d'une implémentation relationnelle

4.1 Introduction

Dans le chapitre 3, nous avons présenté la traduction en **B** des principaux concepts **UML** qui permettent la description de la structure et des fonctionnalités d'une application base de données. L'objectif d'une telle traduction est l'obtention d'un modèle mathématique non seulement précis et concis, mais surtout sur lequel on peut raisonner (contrôles de types, preuves, simulations,...). Cette traduction permet une validation de l'application dès le niveau conceptuel. Il s'agit de vérifier que les transactions sont correctes par rapport aux données et aux contraintes exprimées sur ces dernières. De plus, elle offre l'opportunité de génération d'implémentations prouvées. Cependant, les spécifications générées contiennent souvent des structures de données et de programmation qui ne sont pas directement supportées par un langage d'implémentation. Elles correspondent au niveau conceptuel du développement considéré. Une étape de raffinement devient donc indispensable.

Notre recherche s'inscrit dans le cadre de la génération d'une implémentation relationnelle à partir de spécifications abstraites **B**. Cette génération est réalisée par raffinements successifs des spécifications abstraites obtenues par la phase de traduction des notations graphiques **UML**. L'étape de raffinement constitue la phase la plus critique et difficile du développement d'un projet **B**. De manière générale, le processus de raffinement demeure une tâche manuelle, relativement lourde, particulièrement en phase de preuve. Notre but est de concevoir un outil d'aide au raffinement, dédié au domaine des applications base de données, qui réduirait le plus possible le coût élevé de cette phase.

Théoriquement, il n'existe pas de méthode définie qui permettrait de raffiner n'importe quelle spécification abstraite vers un langage cible donné. En effet, une telle méthode devrait être en mesure, pour chaque implémentation possible, de déterminer les variables concrètes et aussi les invariants de "collage" liant ces dernières aux variables abstraites de la spécification. De plus, tout raffinement doit être prouvé. Néanmoins, la définition d'un processus de raffinement spécialisé pour un domaine bien précis est possible[8]. En effet, si les variables abstraites d'une part et concrètes d'autre part sont bien connues et ont toujours la même forme, alors il est possible (si le raffinement est réalisable) de définir, de manière générique, l'invariant de collage qui les relie.

Dans ce chapitre, nous décrivons un ensemble de règles génériques de raffinement **B** permettant le raffinement des spécifications abstraites obtenues par le processus de traduction présenté au chapitre précédent. Ces règles considèrent conjointement l'aspect statique et dynamique de l'application étudiée, permettant ainsi la définition de la structure des tables relationnelles et du code des différentes opérations **SQL** (insertion, suppression, ...). Le caractère générique de ces règles de raffinement permet d'une part l'automatisation du processus ainsi défini et, d'autre part, la réutilisation par instanciation de leurs preuves de correction. En effet, pour chaque règle, un scénario de preuve est défini, et il suffit par la suite de l'instancier avec les données effectives de l'application à raffiner.

Après une présentation rapide des principaux concepts du modèle relationnel, nous décrivons un algorithme de traduction d'un diagramme de classes en un modèle relationnel. Les étapes de cet algorithme nous servent de guide pour la définition des règles de raffinement des données de notre processus. Une idée globale de notre démarche de raffinement est donnée à la section 4.3. Compte tenu des contraintes actuelles de la méthode **B** et du type d'applications considéré, une étape préliminaire à notre processus de raffinement consiste à expanser l'ensemble de la spécification obtenue au chapitre 3. Nous présentons ensuite les différentes règles de raffinement et donnons un compte rendu des différentes étapes de preuve. Nous terminons par l'étude de quelques propriétés de notre processus et une conclusion.

4.2 Algorithme de transformation d'un diagramme de classes en un schéma relationnel

4.2.1 Modèle relationnel

Le modèle relationnel a été introduit par Codd [13]. Il permet de décrire le monde réel en terme de relations que nous appelons ici "tables" pour ne pas confondre avec le concept de relation du langage **B**. La définition d'une table comprend deux parties complémentaires : l'intention et l'extension. L'intention appelée aussi schéma, définit son type i.e les attributs qui la constituent et leurs domaines respectifs. L'extension d'une table désigne l'ensemble des tuples existants à un instant donné. Chaque tuple de cette extension représente une valeur du produit cartésien des domaines des attributs de la table. Dans ce manuscrit, nous utilisons la norme 92 du langage **SQL** [50].

Des contraintes d'intégrité peuvent être définies aussi bien sur les attributs d'une table que sur un ensemble de tables. Une contrainte, introduite par le mot clé **CHECK**, définit un prédicat devant être vérifié par chaque extension possible de(s) la table(s) mise(s) en œuvre. Les contraintes pouvant être définies sur les attributs d'une table sont les suivantes :

- **NOT NULL** : définie sur un attribut, cette contrainte permet de spécifier que l'attribut concerné doit être valué pour chaque tuple de la table.
- **UNIQUE** : définie sur un (ou un ensemble d') attribut(s), cette contrainte permet de spécifier que la valeur de (ou des) l'attribut(s) est unique dans toute extension possible de la table.
- **Clé** : définie sur un (ou un ensemble d') attribut(s), cette contrainte permet de spécifier une clé possible d'une table donnée. Une table possède au moins une clé désignée comme primaire (**PRIMARY KEY**). Les autres clés candidates sont spécifiées par les deux contraintes **NOT NULL** et **UNIQUE**.
- **Contrainte référentielle** : définie sur un (ou un ensemble d') attribut(s) R_1 d'une table T_1 vers un (ou un ensemble d') attribut(s) R_2 clé d'une table T_2, cette contrainte spécifie que l'ensemble des valeurs de R_1 est inclus dans l'ensemble des valeurs de R_2.

Le modèle relationnel est un modèle orienté valeurs : chaque tuple est identifié par la valeur d'un ou plusieurs attributs qui constituent la clé de la table.

4.2.2 Raffinement d'un diagramme de classes vers un schéma relationnel

Les méthodes classiques d'analyse et de conception de systèmes d'information, utilisant des diagrammes de classes (ou aussi des schémas Entités/Associations) pour la description des données, proposent des algorithmes qui permettent la génération d'un schéma relationnel à partir d'un diagramme de classes [9]. De manière générale, les étapes de tels algorithmes sont les suivantes :

(i) élimination des liens d'héritage : trois solutions sont possibles : conservation de la généralisation et élimination des spécialisations, la solution opposée ou conservation de toute la hiérarchie. Dans tous les cas, des contraintes d'intégrités sont ajoutées.

(ii) passage en première forme normale : dans le modèle relationnel, le domaine de chaque attribut doit être atomique et monovalué. Selon la multiplicité de l'attribut et de son type (composé ou atomique), ce dernier est soit transformé en une nouvelle classe ou éclaté en plusieurs attributs atomiques et monovalués.

(iii) passage d'un modèle orienté objets à un modèle orienté valeurs : dans le modèle relationnel, toute table doit posséder une clé. Par opposition au modèle relationnel, le modèle **UML** est un modèle orienté objets : chaque instance est identifiée par un identifiant qui est indépendant des valeurs des différents attributs.

(iv) élimination des associations : une association, définie entre deux classes A et B, ayant au moins un rôle monovalué (associé à la classe A par exemple) est remplacée par un attribut supplémentaire ajouté à la classe A. Cet attribut est lié par une contrainte référentielle à la clé de la classe B. Les autres associations sont transformées en classes, ayant, en plus de leurs propres attributs, deux attributs supplémentaires (un par classe) liés par des contraintes référentielles aux clés des classes A et B. Ces deux attributs constituent une clé de la classe ainsi créée.

(v) chaque classe du diagramme est traduite en une table.

4.2.3 Exemple de traduction d'un diagramme de classes en schéma relationnel

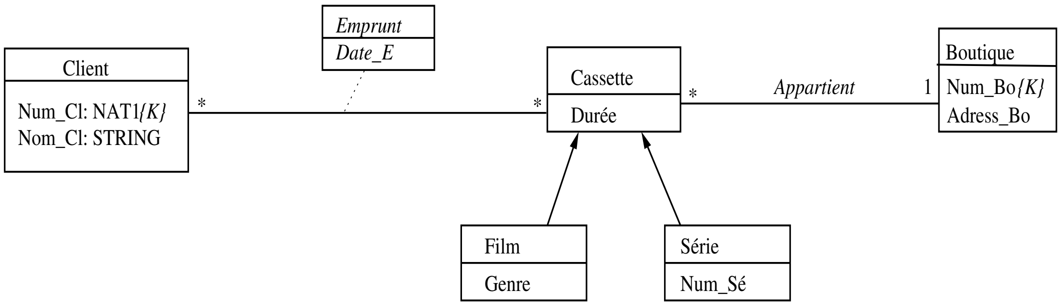


FIG. 4.1 – *Exemple de diagramme de classes UML*

L'application des différentes étapes de l'algorithme précédent sur le diagramme de classes de la figure 4.1, permet la définition des tables suivantes (on suppose avoir choisi la conservation de la super-classe comme solution pour l'élimination de l'héritage) :

CREATE TABLE *Cassette*(
Num_Ca **INT PRIMARY KEY**,
Durée **INT NOT NULL**,
Genre **CHAR(20)**,
Num_Sé **INT**,
Appartient_Mig **INT NOT NULL REFERENCES** *Boutique*,
Est_série **BOOLEAN NOT NULL**,
Est_film **BOOLEAN NOT NULL**,
CONSTRAINT *GenreNul*
CHECK((*Est_film = true* **OR** *Genre* **IS NULL**) **AND** (*Est_film= false* **OR** *Genre* **IS NOT NULL**)),
CONSTRAINT *Num_SéNul*
CHECK((*Est_série = true* **OR** *Num_Sé* **IS NULL**) **AND** (*Est_série = false* **OR** *Num_Sé* **IS NOT NULL**)))

```
CREATE TABLE Client(
  Num_Cl INT PRIMARY KEY,
  Nom_Cl CHAR(20) NOT NULL)

CREATE TABLE Boutique(
  Num_Bo INT PRIMARY KEY,
  Adress_Bo CHAR(20) NOT NULL)

CREATE TABLE Emprunt(
  Emprunt_client INT REFERENCES Client,
  Emprunt_cassette INT REFERENCES Cassette,
  Date_E INT NOT NULL,
  PRIMARY KEY (Emprunt_client, Emprunt_cassette))
```

Les sous-classes *Série* et *Film* sont représentées par deux vues définies par :

```
CREATE VIEW Série AS
SELECT  Num_Ca, Durée, Num_Sé
FROM  Cassette
WHERE  Est_série=true

CREATE VIEW Film AS
SELECT  Num_Ca, Durée, Genre
FROM  Cassette
WHERE  Est_film=true
```

4.3 Démarche générale du processus de raffinement dédié aux applications bases de données

Le processus de raffinement désigne la reformulation d'une spécification abstraite en une spécification concrète plus proche du langage d'implémentation choisi. Cette reformulation concerne aussi bien les données que les opérations (Cf. Annexe A).

Le processus de raffinement que nous proposons est dédié au développement des applications bases de données. Il est défini par un ensemble de règles génériques. Ces règles permettent un raffinement des données et des différentes opérations de base associées (insertion, suppression, mise à jour) obtenues par le processus de traduction des diagrammes **UML** décrit dans le chapitre 3. Ces opérations sont appelées substitutions de base dans la suite. Le raffinement des données que nous proposons, basé sur l'algorithme présenté à la section 4.2.2, a pour buts de :

1. transformer toutes les relations en fonctions totales : ces relations correspondent aux associations et attributs multivaluées des diagrammes de classes **UML**.
2. transformer l'ensemble d'arrivée de chaque fonction totale en un type simple : type de base ou abstrait non associé à une classe **UML**. Cette transformation agit sur les attributs monovalués à types complexes ("Partie-de" et "Séquence de") et les associations monovaluées.
3. associer à chaque variable modélisant une classe ou une association une clé : fonction injective.

Contrairement à notre approche, l'algorithme classique sus-cité ne permet la génération que du modèle des données i.e les tables. L'écriture des programmes est réalisée dans une étape ultérieure indépendamment des données. Il devient donc impossible de prouver de manière sûre la correction de ces programmes vis-à-vis de leurs données. En revanche, notre raffinement conjoint données/opérations permet l'obtention d'une implémentation globale cohérente. Ne considérer que les substitutions de base ne constitue pas une limite à notre processus. En effet, les fonctionnalités d'une application base de données se décrivent toujours par la combinaison de ces trois formes de substitutions. La combinaison est réalisée via les structures de programmation : test, séquence, etc. Le système de substitutions ainsi considéré est donc complet. De plus, le raffinement en **B** étant monotone, le raffinement d'une substitution complexe se ramène toujours au raffinement des substitutions élémentaires qui la composent. Il restera bien évidemment le raffinement des structures de programmation, qui lui par contre ne peut être automatique, car ce dernier dépend du contexte de l'application considérée. Les différentes étapes du processus de raffinement et les interventions éventuelles du concepteur sont représentées par la figure 4.2. Ces interventions ne sont évidemment pas indispensables au processus de raffinement. Des choix par défaut peuvent être adoptés.

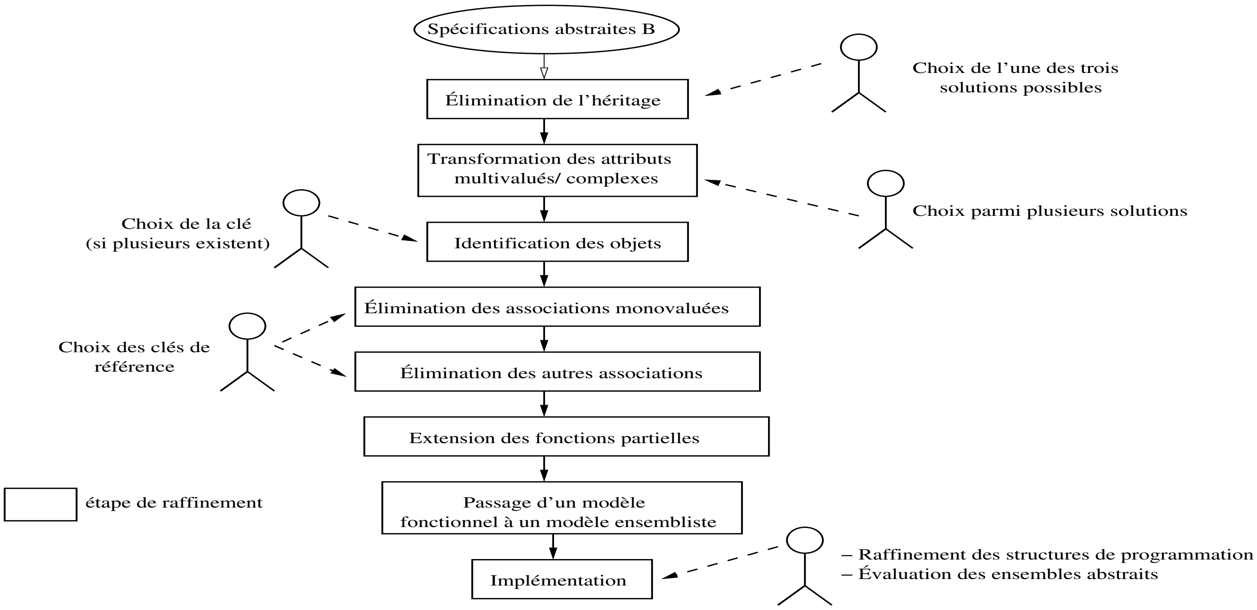


FIG. 4.2 – *Déroulement des étapes du raffinement dédié à la génération d'une implémentation relationnelle*

Notre approche de développement par raffinement peut être divisée en trois phases principales :

- La première phase concerne le raffinement des données et des substitutions de base. Cette étape est complètement automatique. Le raffinement des substitutions de base consiste en la réécriture de ces substitutions en fonction des données concrètes de chaque étape. Cette étape n'agit que sur les machines de base associées aux classes et aux associations. En effet, c'est dans ces dernières que se trouve l'ensemble des variables du système. Une étape préalable est la construction de la machine expansée équivalente à la machine d'interface *Système* (Cf. section 4.4).
- La seconde phase concerne l'implémentation du dernier niveau de raffinement obtenu à la phase précédente. Cette implémentation est basée sur l'importation d'un ensemble de machines abstraites **B**. Ces machines correspondent aux tables relationnelles nécessaires à l'implémentation du système. Chaque machine importée spécifie la structure d'une table donnée et des opérations de base permettant leur gestion. Ces machines automatiquement générées sont traduites automatiquement en **SQL**.
- La dernière phase permet l'implémentation des opérations correspondant aux transactions. Elle a pour but de remplacer les substitutions du dernier niveau de raffinement, non admises en implémentation, par d'autres substitutions équivalentes (au sens de la correction). Sont concernées par cette étape, particulièrement, les substitutions indéterministes et parallèles.

En pratique, les contraintes actuelles de la méthode **B** exigent que les deux dernières étapes soient réalisées simultanément (Voir figure 4.3).

4.4 Expansion de la spécification abstraite initiale : *Système_exp*

Rappelons que la spécification abstraite obtenue lors du processus de traduction, présenté dans le chapitre 3, est constituée de trois niveaux reflétant les niveaux de description **UML**. Rappelons aussi que les systèmes

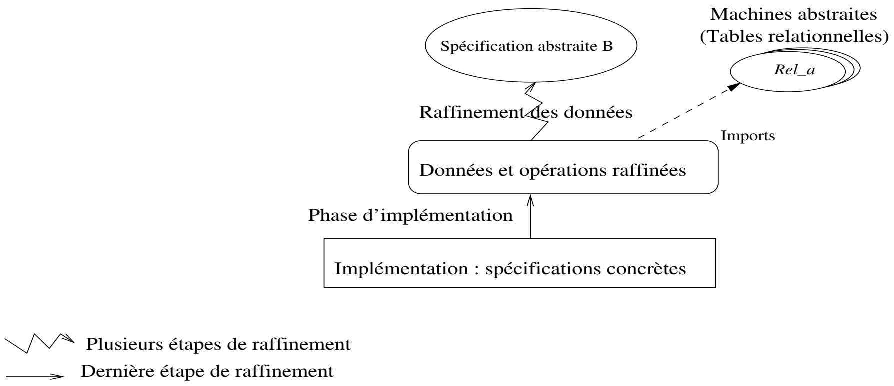


FIG. 4.3 – *Démarche de développement par raffinement d'implémentations relationnelles*

que nous étudions sont caractérisés par des données fortement liées. Cette forte dépendance se retrouve bien évidemment au cours du raffinement : les raffinements des classes et des associations sont interdépendants. Les contraintes actuelles de la méthode **B** exigent que de telles données soient raffinées dans une même machine. Pour cela, une étape préalable au raffinement est de construire la machine équivalente à l'ensemble des machines de l'architecture précédente. La machine expansée obtenue sera par la suite importée par l'implémentation de la machine d'interface *RobusteSystème*.

4.4.1 Construction de la machine expansée

La machine expansée équivalente à l'architecture de la spécification abstraite, obtenue dans le chapitre précédent, est réalisée comme suit :
• Les ensembles abstraits, variables, invariant et initialisation de la machine expansée sont l'union de ceux des différentes machines. Les variables doivent être renommées afin d'éviter un conflit de noms. Chaque variable doit être renommée par une variable fraîche v_1 par exemple. En effet en **B**, les variables abstraites des machines importées par l'implémentation d'une machine *A* doivent avoir des noms différents de ceux des variables abstraites définies dans *A* et disparaissant dans son implémentation (règle d'anti-collision d'identificateurs) [16].

Exemple : la machine expansée, représentant la figure 4.1, contient les ensembles abstraits et variables suivants :

SETS *Client, Cassette, Boutique*
VARIABLES
$client_1$, Num_Cl_1, Nom_Cl_1, $cassette_1$, $Durée_1$, $film_1$, $Genre_1$, $série_1$, $Num\_Sé_1$,
$boutique_1$, Num_Bo_1, $Adress\_Bo_1$, $Emprunt_1$, $Date\_E_1$

• En plus des définitions renommées, présentes dans les différentes machines, les opérations de chaque machine (à l'exception de la machine *Système*) sont transformées en définitions. On gardera les mêmes noms pour ces définitions et on renommera bien évidemment toutes les variables. Les préconditions de ces définitions sont éliminées. Ces dernières seront toujours appelées sous des préconditions plus fortes. Les opérations promues dans les machines états/ transitions sont ignorées. En effet, celles-ci correspondent aux opérations de base des classes et des associations et donc sont déjà incluses dans l'expansion.

Exemple : l'opération de base d'ajout d'un client spécifiée par :

BasicAjoutClient(cl, nu, no)=
PRE $cl \in Client\text{-}client \wedge nu \in NAT\text{-}ran(Num\_Cl) \wedge no \in STRING$

THEN

$client := client \cup \{cl\} \|$
$Num\_Cl := Num\_C \cup \{cl \mapsto nu\} \|$
$Nom\_Cl := Nom\_C \cup \{cl \mapsto no\}$

END

est transformée en la définition suivante :

BasicAjoutClient(cl, nu, no) $\widehat{=}$
BEGIN

$client_1 := client_1 \cup \{cl\} \|$
$Num\_Cl_1 := Num\_C_1 \cup \{cl \mapsto nu\} \|$
$Nom\_Cl_1 := Nom\_C_1 \cup \{cl \mapsto no\}$

END

• Seules les opérations présentes dans la machine représentant la sémantique d'une architecture d'héritage sont transformées en définitions. Ces opérations sont expansées en ligne. Les opérations des sous-classes sont ignorées. Nous avons opté pour une extension en ligne, car lors de l'élimination de l'héritage, nous aurons besoin de savoir sur quelle sous-classe une substitution donnée est définie. Par exemple, l'opération *AjoutSérie* spécifiée par :

AjoutSérie(sé, nu, du)=
PRE $sé \in Cassette\text{-}série \wedge nu \in NAT \wedge du \in NAT$
THEN

BasicAjoutCassette(sé, du) $\|$
BasicAjoutSérie(sé, nu)

END;

est transformée en la définition :

AjoutSérie(sé, nu, du) $\widehat{=}$
BEGIN

$cassette_1 := cassette_1 \cup \{ca\} \|$
$Durée_1 := Durée_1 \cup \{ca \mapsto du\} \|$
$série_1 := série_1 \cup \{sé\} \|$
$Num\_Sé_1 := Num\_Sé_1 \cup \{sé \mapsto nu\}$

END;

• Les opérations de la machine expansée sont les mêmes que celles décrites dans la clause **OPERATIONS** de la machine *Système*. Leurs variables sont renommées. Les opérations promues dans la machine *Système* sont réécrites dans la clause opération de la machine *Système_exp*. Chaque opération promue op(...)=**PRE**... **THEN** ... **END** est réécrite en op_1(...)=**PRE**... **THEN** op(...) **END**.

4.4.2 Correction de la machine *Système_exp*

Par construction, la machine *Système_exp* est correcte par rapport à l'ensemble architecturé de la spécification initiale. En effet, les opérations de la machine *Système_exp* sont les mêmes que celles de la machine *Système* au renommage près des variables. La machine *Système_exp* est une extension purement syntaxique de la machine *Système*. Aucune nouvelle notion sémantique n'est introduite. Par conséquent, si la machine *Système* est correcte alors la machine *Système_exp* l'est aussi.

La machine *Système_exp* peut être construite par extension totale en ligne. Les opérations de cette machine ne contiendront plus des appels à des définitions mais directement les corps de ces dernières. L'avantage de l'utilisation des définitions est de conserver le plus possible la structure des opérations de base et l'architecture de la spécification initiale.

Dans la suite, on continuera d'utiliser le terme d'opération pour désigner la définition qui lui correspond.

4.5 Raffinement des données et des substitutions de base

Dans cette section, sont présentées les étapes du processus de raffinement des données et des substitutions de base. Chaque étape de notre processus de raffinement est spécifiée par deux règles complémentaires de raffinement : la première correspond au raffinement des données, la seconde au raffinement des substitutions de base agissant sur les données raffinées.

- Règle de raffinement des données *Nom_D* : cette règle permet de raffiner l'environnement abstrait *env_abs*, vérifiant la contrainte *C*, par l'environnement concret *env_conc*. La contrainte C désigne une formule qui doit être déduite (ou ne doit pas pouvoir se déduire) de la spécification abstraite. Un environnement est défini par les ensembles et les variables abstraits qu'il contient. Les deux environnements sont reliés par l'invariant de collage *J(env_abs, env_conc)*. Cette règle est modélisée comme suit :

```
RULE Nom_D IS
ABSTRACT_ENV
    env_abs
CONSTRAINT
    C
CONCRETE_ENV
    env_conc
INVARIANT
    J(env_abs,env_conc)
END
```

- Règle de raffinement des substitutions *Nom_S* : cette règle permet le raffinement de la substitution abstraite *sub_abs*, agissant sur l'environnement abstrait env_abs, par la substitution concrète *sub_conc* qui agirait sur l'environnement concret env_conc. Le lien entre la règle de raffinement des données et celle des substitutions est réalisé par l'invariant $J(env\_abs, env\_conc)$ figurant dans les deux règles. Dans la suite pour simplifier, on omettra de noter cet invariant dans la règle de raffinement des substitutions.

  En pratique, une substitution est bien évidemment contenue dans une opération pouvant éventuellement contenir d'autres substitutions. Des contraintes supplémentaires C' sur ces substitutions peuvent être précisées. Elles concernent l'obligation de présence ou d'absence d'une substitution particulière. On adopte pour cela la notation ($\neg S$) qui indique que la substitution S ne doit pas être présente dans l'opération contenant la substitution à raffiner. Les règles de raffinement des substitutions se modélisent comme suit :

```
RULE Nom_S IS
REFINES
        sub_abs
CONSTRAINTS
        J(env_abs, env_conc) ∧ C'
REFINEMENT
        sub_conc
END
```

À chaque règle de raffinement de données correspond une ou plusieurs règles de raffinement des substitutions. En effet, pour chaque forme de substitutions, opérant sur les données raffinées par la règle *Nom_D*, correspond une règle de raffinement permettant la réécriture de cette substitution dans l'environnement concret.

Pour chaque règle de raffinement de substitutions *Nom_S* associée à la règle de raffinement de données *Nom_D*, une obligation de preuve est générée. Cette obligation de preuve vérifie la correction de la substitution concrète sub_conc par rapport à la substitution abstraite sub_abs relativement à l'invariant de collage $J(env\_abs, env\_conc)$. D'après la définition de la correction du raffinement (Cf. définition A.3.1), cette obligation de preuve est de la forme :

$$\begin{gathered} Trm(sub\_abs) \wedge J(env\_abs, env\_conc) \Rightarrow \\ (Trm(sub\_conc) \wedge \forall\, env\_conc''.(Prd_{env\_conc, env\_conc''}(sub\_conc) \Rightarrow \\ \exists\, env\_abs'.(Prd_{env\_abs, env\_abs'}(sub\_abs) \wedge J(env\_abs', env\_conc'')))) \end{gathered}$$

Les contraintes C et C' peuvent être utilisées comme hypothèses supplémentaires au cours de l'établissement de cette preuve. Un exemple de preuve est donné à la page 102.

Dans la suite, on suppose qu'une étape préliminaire à notre processus de raffinement a permis de raffiner les substitutions parallèles, présentes dans les définitions, en substitutions séquentielles. Cette étape de raffinement est correcte car les définitions correspondent aux opérations de base des classes et des associations. Ces substitutions agissent sur des ensembles de variables disjoints. Cette étape est souhaitée (non nécessaire) afin de faciliter l'écriture des substitutions séquentielles introduites lors du raffinement.

4.5.1 Élimination de l'héritage

Rappelons que dans le processus de transformation d'un diagramme de classes en un modèle relationnel, trois alternatives sont possibles pour l'élimination de l'héritage :

- conservation de la généralisation et élimination des spécialisations.
- conservation des spécialisations et élimination de la généralisation.
- conservation de l'ensemble de la hiérarchie.

Le choix entre l'une ou l'autre des solutions dépend de plusieurs critères : le caractère abstrait ou non de la généralisation, le nombre d'attributs et d'associations relatifs aux spécialisations, etc. Dans tous les cas, l'utilisateur peut exprimer son propre choix.

Les deux sous-sections suivantes sont dédiées à la définition formelle des deux premières alternatives. La troisième n'impliquera que des contraintes d'intégrité supplémentaires (Cf. Annexe C).

4.5.1.1 Conservation de la généralisation et élimination des spécialisations

Cette solution est choisie quand le nombre d'attributs et d'associations relatifs aux spécialisations est assez faible. L'idée principale du raffinement est de redéfinir toutes les caractéristiques des spécialisations dans la généralisation.

Raffinement des données. Le raffinement des données correspondant à l'élimination des sous-classes C_1, ..., C_k d'une super-classe C est réalisé comme suit :

$i.$ chaque relation définie de la variable c_i, associée à la sous-classe C_i, est étendue sur la variable c associée à la super-classe C. Une relation conserve sa nature par extension, tout autre type de fonction (totale, partielle, injective, ...) est étendu en une fonction partielle.

ii. la trace des sous-classes auxquelles appartient chaque instance peut être modélisée de deux manières différentes :

- on ajoute un attribut *Est_Élement* défini de c dans l'ensemble énuméré **SETS** *Type*=$\{c_1, \ldots, c_k\}$. Afin d'éviter d'avoir des attributs multivalués, cette solution n'est conseillée que si les sous-classes sont deux à deux disjointes. Dans ce cas, l'attribut *Est_Élement* est une fonction partielle (resp. totale) si l'héritage est incomplet (resp. complet). Cette solution a été exposée dans [42].
- si les sous-classes ne sont pas deux à deux disjointes, chaque classe c_i est remplacée par une fonction booléenne totale *Est*_c_i. Cette fonction indique si une instance donnée appartient ou non à la sous-classe c_i. C'est cette solution que nous présentons dans ce manuscrit.

iii. chaque sous-classe est remplacée par une définition. Cette définition sera traduite par la suite en une vue dans le modèle relationnel (Cf. Annexe C).

Formellement, ce raffinement s'exprime par la règle suivante (seul le cas où la classe c_i est source d'une relation est présenté, les autres cas se traitent de manière similaire) :

RULE *ÉlimSpécia_D* **IS**
ABSTRACT_ENV
  SETS *C*
  VARIABLES
    c,
    $c_1, f_1^{c_1}, \ldots, f_{n_1}^{c_1}$,
    $\ldots$
    $c_k, f_1^{c_k}, \ldots, f_{n_k}^{c_k}$
CONSTRAINT
    $c \subseteq C \wedge \bigwedge_{i=1,k} c_i \subseteq c \wedge \bigwedge_{i=1,k} (\bigwedge_{j=1,n_i} f_j^{c_i} \in c_i \leftrightarrow D_j^{c_i})$
CONCRETE_ENV
  DEFINITIONS $c_i \triangleq Est\_c_i^{-1}[\{TRUE\}]$
  VARIABLES
    c,
    $f_1^{c_1}, \ldots, f_{n_1}^{c_1}$
    $\ldots$
    $f_1^{c_k}, \ldots, f_{n_k}^{c_k}$,
    $Est\_c_1, \ldots, Est\_c_k$
INVARIANT
    $\bigwedge_{i=1,k} (\bigwedge_{j=1,n_i} f_j^{c_i} \in c \leftrightarrow D_j^{c_i}) \wedge$
    $\bigwedge_{i=1,k} Est\_c_i \in c \rightarrow BOOL \wedge \bigwedge_{i=1,k} c_i = Est\_c_i^{-1}[\{TRUE\}]$
END

Exemple : l'aspect statique de la structure d'héritage de la figure 4.1 spécifié en **B** par :

SETS *Cassette*
VARIABLES *cassette, film, série, Durée, Genre, Num_Sé*
INVARIANT
  cassette $\subseteq$ *Cassette* $\wedge$ *film* $\subseteq$ *cassette* $\wedge$ *série* $\subseteq$ *cassette* $\wedge$ *Durée* $\in$ *cassette* $\rightarrow$ *NAT* $\wedge$
  Genre $\in$ *film* $\leftrightarrow$ *STRING* $\wedge$ *Num_Sé* $\in$ *série* $\rightarrow$ *NAT* $\wedge$
  $\ldots$

est raffiné en :

DEFINITIONS

film $\widehat{=}$ *Est_film*$^{-1}$*[{TRUE}]*;
série $\widehat{=}$ *Est_série*$^{-1}$*[{TRUE}]*
VARIABLES *cassette, Durée, Genre, Num_Sé, Est_film, Est_série*
INVARIANT
Genre $\in$ *cassette* $\leftrightarrow$ *STRING* $\wedge$ *Num_Sé* $\in$ *cassette* $\nrightarrow$ *NAT* $\wedge$
Est_film $\in$ *cassette* $\rightarrow$ *BOOL* $\wedge$ *film* $=$ *Est_film*$^{-1}[\{TRUE\}]$
Est_série $\in$ *cassette* $\rightarrow$ *BOOL* $\wedge$ *série=Est_série*$^{-1}[\{TRUE\}]$

Raffinement des substitutions. La règle de raffinement des données précédente introduit un ensemble de données supplémentaires *Est_*c_i et supprime les variables c_i associées aux sous-classes. Le raffinement des substitutions est réalisé de la façon suivante :

1. l'attribut *Est_*c_i est mis à jour à *TRUE* lors de l'ajout d'une instance dans une sous-classe C_i.
2. lors de l'ajout d'une instance dans la super-classe *C*, si cette instance n'est pas ajoutée à la variable c_j, alors l'attribut *Est_*c_j est mis à *FALSE* pour cette instance.
3. les substitutions relatives à l'ajout ou la suppression de la variable c_i sont supprimées car cette dernière n'existe plus : elle est implicitement prise en compte par l'ajout ou la suppression relatives à la variable associée à la super-classe.
4. lors d'une suppression d'instances de la sous-classe C_i deux cas sont à distinguer : si ces instances sont supprimées de la super-classe alors on supprime de chaque attribut *Est_*c_i tous les couples relatifs à ces instances, sinon, on met la valeur de l'attribut *Est_*c_i à *FALSE* pour ces instances.

Ces raffinements sont modélisés par les règles suivantes :

a) Ajout d'une instance dans la sous-classe C_i

RULE *ÉlimSpécia_Ajout_*C_i **IS**
REFINES
c_i:=$c_i \cup$ *{cc}*
REFINEMENT
*Est_*c_i:= *Est_*c_i $\Lleftarrow$ *{cc* $\mapsto$ *TRUE}*
END

L'utilisation de la surcharge permet de traiter les deux cas d'ajout dans une sous-classe. En effet, l'ajout dans une sous-classe peut concerner aussi bien une nouvelle instance (elle est donc ajoutée aussi à la super-classe) qu'une instance qui existe déjà dans la super-classe.

b) Ajout d'une instance dans la super-classe C

RULE *ÉlimSpécia_Ajout_C* **IS**
REFINES
c := *c* $\cup$ *{cc}*
CONSTRAINT
$\neg$ (c_j:=$c_j \cup$ *{cc}*)
REFINEMENT
c := *c* $\cup$ *{cc}* ;
*Est_*c_j := *Est_*c_j $\cup$ *{cc* $\mapsto$ *FALSE}*
END

Exemple : on considère les deux opérations d'ajout d'une nouvelle cassette et d'une nouvelle série. Ces opérations spécifiées en **B** par :

Ajout_Ca (ca, du) $\widehat{=}$
BEGIN
cassette := cassette $\cup$ *{ca}*;
Durée := Durée $\cup$ *{ca* $\mapsto$ *du}*
END

Ajout_Sé(sé, nu, du)$\widehat{=}$
BEGIN
cassette := cassette $\cup$ *{sé}* ;
Durée:=Durée $\cup$ *{sé* $\mapsto$ *du}* ;
série := série $\cup$ *{sé}*;
Num_Sé := Num_Sé $\cup$ *{sé* $\mapsto$ *nu}*
END

sont raffinées par :

Ajout_Ca (ca, du)$\widehat{=}$
BEGIN
cassette := cassette $\cup$ *{ca}* ;
Durée := Durée $\cup$ *{ca* $\mapsto$ *du}*;
Est_série := Est_série $\cup$ *{ca* $\mapsto$ *FALSE}* ;
Est_film := Est_film $\cup$ *{ca* $\mapsto$ *FALSE}*
END

Ajout_Sé (sé, nu, du)$\widehat{=}$
BEGIN
cassette := cassette $\cup$ *{sé}*;
Durée:=Durée $\cup$ *{sé* $\mapsto$ *du}*;
Num_Sé := Num_Sé $\cup$ *{sé* $\mapsto$ *nu}* ;
Est_série := Est_série $\Leftarrow\!\!+$ *{sé* $\mapsto$ *TRUE}* ;
Est_film := Est_film $\cup$ *{sé* $\mapsto$ *FALSE}*
END

c) Suppression d'un ensemble d'instances de la sous-classe C_i

RULE *ÉlimSpécia_Supp_C_i* **IS**
REFINES
c_i:=c_i -cc
CONSTRAINT
$\neg$ (c:=c - cc)
REFINEMENT
Est_c_i:= Est_c_i $\Leftarrow\!\!+$ *cc* $\times$ *{FALSE}*
END

d) Suppression d'un ensemble d'instances de la super-classe C

RULE *ÉlimSpécia_Supp_C* **IS**
REFINES
c_i:=c_i-cc
CONSTRAINT
c:=c-cc ;
REFINEMENT
Est_c_i:= cc $\lhd$ *Est_c_i*
END

Exemple : les opérations de suppression d'un ensemble de cassettes et d'un ensemble de séries spécifiées en **B** par :

Supp_Ca (Ca)$\widehat{=}$
BEGIN
cassette := cassette - Ca ;
Durée := Ca $\lhd$ *Durée* ;
série := série -Ca ;
Num_Sé := Ca $\lhd$ *Num_Sé* ;
film := film -Ca ;
Genre := Ca $\lhd$ *Genre*
END

Supp_Sé(Sé)$\widehat{=}$
BEGIN
série := série - Sé ;
Num_Sé := Sé $\lhd$ *Num_Sé*
END

sont raffinées par :

Supp_Ca (Ca)$\widehat{=}$
BEGIN
cassette := cassette - Ca ;
Durée:=Ca $\lhd$ *Durée* ;
Est_série := Ca $\lhd$ *Est_série* ;
Num_Sé := Ca $\lhd$ *Num_Sé* ;
Est_film := Ca $\lhd$ *Est_film* ;
Genre := Ca $\lhd$ *Genre*
END

Supp_Sé(Sé) $\widehat{=}$
BEGIN
Est_Sé := Est_Sé $\lhd\!\!+$ *Sé* $\times$ *{FALSE}* ;
Num_Sé :=Sé $\lhd$ *Num_Sé*
END

4.5.1.2 Conservation des spécialisations et élimination de la généralisation

Cette solution est choisie quand la super-classe est abstraite et que les spécialisations sont deux à deux disjointes. La super-classe n'ayant pas d'instances propres, celle-ci est instanciée via ses sous-classes. L'idée principale du raffinement est de reporter toutes les caractéristiques de la généralisation dans chacune des spécialisations.

Raffinement des données. Le raffinement des données correspondant à l'élimination de la super-classe C des sous-classes $C_1, \ldots, C_k$ est réalisé comme suit :

i chaque relation f définie de la variable c, associée à la super-classe C, est décomposée en k relation $f_1, \ldots, f_k$ de même nature. Chacune des variables f_i est définie de la variable c_i, associée à la sous-classe C_i, dans le même ensemble d'arrivée que la variable f. Les relations f_i et f sont de même nature.

ii la super-classe est remplacée par une définition. Cette définition sera traduite par la suite en une vue (Cf. Annexe C).

Ce raffinement s'exprime par la règle suivante (seul le cas où la variable c est source d'une relation est présenté, les autres cas se traitent de la même manière) :

RULE *ÉlimGéné_D* **IS**
ABSTRACT_ENV
  SETS C
  VARIABLES
    $c, c_1, \ldots c_k, f_1, \ldots, f_n$
CONSTRAINT
    $c \subseteq C \wedge \bigwedge_{i=1,k} c_i \subseteq c \wedge$
    $\bigwedge_{i=1,n} f_i \in c \leftrightarrow D_i$
    $\bigcup_{i=1..k} c_i = c \wedge \forall\, i, j.\ (i \neq j \Rightarrow c_i \cap c_j = \emptyset)$
CONCRETE_ENV
    DEFINITIONS $c \mathrel{\widehat{=}} \bigcup_{i=1..k} c_i$
    VARIABLES
      $c_1, \ldots, c_k,$
      $f_1^{c_1}, \ldots, f_1^{c_k},$
      $\ldots$
      $f_n^{c_1}, \ldots, f_n^{c_k}$
INVARIANT
    $\bigwedge_{i=1,k}(\bigwedge_{j=1,n} f_j^{c_i} \in c_i \leftrightarrow D_j)$
    $\bigwedge_{i=1..n} (\bigcup_{j=1,k} f_i^{c_j} = f_i)$
END

Exemple : en considérant toujours la structure d'héritage de la figure 4.1, on suppose maintenant que la classe *cassette* est abstraite et que les sous-classes *Film* et *Série* sont disjointes. En plus de la spécification **B** précédente, l'invariant est augmenté par les deux prédicats suivants :

$$cassette = film \cup série$$
$$série \cap film = \emptyset$$

L'application de cette règle de raffinement donne :

DEFINITIONS
  cassette $\mathrel{\widehat{=}}$ *film* $\cup$ *série*
VARIABLES *film, série, Durée_film, Durée_série, Genre, Num_Sé*
INVARIANT
  Durée_film $\in$ *film* $\rightarrow$ *NAT* $\wedge$
  Durée_série $\in$ *série* $\rightarrow$ *NAT* $\wedge$
  Durée_film $\cup$ *Durée_série* = *Durée*

Raffinement des substitutions. La règle de raffinement des données précédente supprime la variable c et duplique (avec renommage) chaque variable f_i sur l'ensemble des sous-classes. Le raffinement des substitutions est tel que :

$i.$ la mise à jour de la variable f_i de la super-classe est remplacée par une mise à jour de l'une de ses duplications définies dans les sous-classes. La variable à mettre à jour correspond à la variable définie dans la classe à laquelle appartient l'instance considérée.

$ii.$ les substitutions relatives à la variable c sont supprimées car cette dernière n'existe plus : elles sont implicitement incluses dans l'une des substitutions relatives aux variables des sous-classes.

Formellement, ce raffinement s'exprime comme suit :

a) **Ajout d'une instance dans** c_i [1]

RULE *ÉlimGéné_Ajout_*C_i **IS**
REFINES
$c := c \cup \{cc\}$;
$c_i := c_i \cup \{cc\}$;
$;_{j=1..n} (f_j := f_j \cup \{cc \mapsto val_j\})$
REFINEMENT
$c_i := c_i \cup \{cc\}$;
$;_{j=1..n} (f_j^{c_i} := f_j^{c_i} \cup \{cc \mapsto val_j\})$
END

Exemple : par ce raffinement, l'opération abstraite d'ajout d'une série de la page 78 est raffinée par :

Ajout_Sé (sé, nu, du)$\widehat{=}$
BEGIN
Durée_série := Durée_série $\cup$ *{sé* $\mapsto$ *du}* ;
série := série $\cup$ *{sé}*;
Num_sé := Num_Sé $\cup$ *{sé* $\mapsto$ *nu}*
END

b) **Suppression d'un ensemble d'instances de** C_i [2]

RULE *ÉlimGéné_Supp_*C_i **IS**
REFINES
$c := c - cc$;
$c_i := c_i - cc$;
$;_{j=1..n} f_j := cc \lhd f_j$
REFINEMENT
$c_i := c_i - cc$;
$(;_{j=1..n} f_j^{c_i} := cc \lhd f_j^{c_i})$
END

Exemple : l'opération abstraite de suppression d'un ensemble de séries de la page 79 est raffinée par :

Supp_Sé(Sé)$\widehat{=}$
BEGIN
Série := Série - Sé ;
Durée_Série := Sé $\lhd$ *Durée_Série* ;
Num_Sé := Sé $\lhd$ *Num_Sé*
END

1. L'ordre des substitutions dans la substitution séquentielle est sans importance.
2. L'ordre des substitutions dans la substitution séquentielle est sans importance.

c) **Modification de la valeur** f_i

RULE *ÉlimGén_Change* **IS**
REFINES
    $f_i(cc) := val$
REFINEMENT
    SELECT $cc \in c_1$ **THEN** $f_i^{c_1}(cc) := val$
    WHEN$_{j=2..k}$ $cc \in c_j$ **THEN** $f_i^{c_j}(cc) := val$
    END
END

4.5.2 Passage en première forme normale

Un schéma relationnel est en première forme normale si et seulement si chaque attribut n'a qu'une seule valeur au plus et que cette dernière est de type atomique. Cette étape de raffinement va permettre de transformer les attributs de types complexes (partie-de, séquence-de) ou multivalués en attributs monovalués de types atomiques (types de base). Rappelons que nous avons contraint les attributs de types complexes à être monovalués.

4.5.2.1 Élimination du constructeur "Partie-de"

Cette étape a pour rôle d'éliminer les attributs dont le type est défini comme des parties des types de base. En **B**, un attribut *Att* monovalué, d'une classe *C*, de type $\mathbb{P}_1(T)$ est modélisé par une fonction définie de la variable de c vers $\mathbb{P}_1(T)$. Afin d'avoir un type atomique, cet attribut est raffiné par la règle suivante.

Raffinement des données. Un attribut *Att* monovalué de type $\mathbb{P}_1(T)$ est raffiné par un attribut multivalué de type T. Intuitivement, cela revient à éclater chaque partie en les valeurs qui la constituent. Formellement, cette règle de raffinement se modélise par :

RULE *ÉlimPartMono_D* **IS**
ABSTRACT_ENV
    VARIABLES *Att, c*
CONSTRAINT
    $Att \in c \rightarrow\!\!\!\!\!\!+ \mathbb{P}_1(T)$
CONCRETE_ENV
    VARIABLES Att_1, c
INVARIANT
    $Att_1 \in c \leftrightarrow T \wedge$
    $Att_1 = rel(Att)$[3]
END

Dans le cas où la fonction Att est totale, une contrainte supplémentaire, traduisant en **SQL** la contrainte ($dom(Att_1) = c$), doit être ajoutée lors de la traduction en schéma relationnel. Cette contrainte est implicitement exprimée par l'invariant de collage ($Att_1 = rel(Att)$). Une expression explicite, dans la clause invariant, est donc inutile.

Exemple : supposons à présent que le genre d'un film soit représenté par un sous-ensemble de chaînes de caractères. En **B**, cet attribut modélisé par :

$$Genre \in film \rightarrow \mathbb{P}_1(STRING)$$

3. $rel(r) = \{(x,y) \mid x \in dom(r) \wedge y \in r(x)\}$ avec $r \in s \rightarrow\!\!\!\!\!\!+ \mathbb{P}(t)$

est donc raffiné en :

$$Genre_1 \in film \leftrightarrow STRING \wedge$$
$$Genre_1 = rel(Genre)$$

Raffinement des substitutions. Le raffinement des données précédent transforme une paire dont la deuxième composante est un ensemble en un ensemble de paires. Cet ensemble de paires est le produit cartésien de l'ensemble formé par la première composante et la deuxième composante. Le raffinement des substitutions de suppression relatives à la variable *Att* est réalisé simplement en remplaçant chaque occurrence de la variable *Att* par la variable Att_1. Le raffinement des substitutions d'ajout et de modification est formalisé comme suit :

a) Mise à jour de la variable Att lors de l'ajout d'un objet cc dans c

RULE *ÉlimPartMono_Ajout* **IS**
REFINES
$Att := Att \cup \{cc \mapsto val\}$
REFINEMENT
$Att_1 := Att_1 \cup \{cc\} \times val$
END

Exemple : on considère l'opération d'ajout d'un film spécifiée en **B** par :

Ajout_Fl(fl, du, ge)$\widehat{=}$
BEGIN
  $film := film \cup \{fl\}$;
  $Durée\_film := Durée\_film \cup \{fl \mapsto du\}$;
  $Genre := Genre \cup \{fl \mapsto ge\}$
END

cette opération est raffinée par :

Ajout_Fl(fl, du, ge) $\widehat{=}$
BEGIN
  $film := film \cup \{fl\}$;
  $Durée\_fl := Durée\_fl \cup \{fl \mapsto du\}$;
  $Genre_1 := Genre_1 \cup \{fl\} \times ge$
END

b) Modification de la variable Att

RULE *ÉlimPartMono_Change* **IS**
REFINES
$Att := Att \lhd\!\!- \{cc \mapsto val\}$
REFINEMENT
$Att_1 := Att_1 \lhd\!\!- \{cc\} \times val$
END

4.5.2.2 Élimination des attributs multivalués à multiplicité bornée

En **B**, un attribut multivalué borné *Att*, d'une classe *C* et de type *T*, est représenté par une relation définie de la variable associée à la classe *c* dans *T*. Une contrainte spécifiant le nombre maximal de valeurs qu'un objet de la classe *C* peut avoir pour cet attribut est ajoutée. Le raffinement d'un attribut multivalué de multiplicité maximale bornée *k* (*k* étant relativement petit et fixé par le concepteur) est réalisé par la décomposition de ce dernier en k attributs monovalués $Att_1, \ldots, Att_k$. Afin de faciliter le raffinement des substitutions relatives à l'ajout et la modification de cet attribut, on considère que les k attributs sont obligatoires. Pour cela, on étendra l'ensemble d'arrivée T par une valeur particulière $Null_T$ de type T. Cette valeur représente la non applicabilité d'un attribut (Cf. section 4.5.7). Formellement, ce raffinement est modélisé comme suit :

RULE *AttMultiBor_D* **IS**
ABSTRACT_ENV
Att, c
CONSTRAINT
$Att \in c \leftrightarrow T \wedge$
$\forall xx. (xx \in c \Rightarrow j \leq card(Att[\{xx\}]) \leq k)$
CONCRETE_ENV
$Att_1, \ldots, Att_k, c$
INVARIANT
$Att_1 \in c \rightarrow T \wedge$
$\ldots$
$Att_j \in c \rightarrow T \wedge$
$Att_{j+1} \in c \rightarrow T \cup \{Null_T\} \wedge$
$\ldots$
$Att_k \in c \rightarrow T \cup \{Null_T\} \wedge$
$(\bigcup_{i=1..k} Att_i) \rhd \{Null_T\} = Att$
END

Exemple: supposons à présent qu'un film ait au moins un genre et au plus trois. En **B**, cet attribut modélisé par :

$$Genre \in film \leftrightarrow STRING \wedge$$
$$\forall xx. (xx \in film \Rightarrow 1 \leq card(Genre[\{xx\}]) \leq 3)$$

est raffiné en :

VARIABLES $Genre_1, Genre_2, Genre_3$
INVARIANT
$Genre_1 \in film \rightarrow STRING \wedge$
$Genre_2 \in film \rightarrow STRING \cup \{Null_{STRING}\} \wedge$
$Genre_3 \in film \rightarrow STRING \cup \{Null_{STRING}\} \wedge$
$Genre = (Genre_1 \cup Genre_2 \cup Genre_3) \rhd \{Null_{STRING}\}$

Raffinement des substitutions. La règle de raffinement des données précédente éclate un attribut multivalué en un ensemble d'attributs monovalués. En appliquant le même processus d'éclatement, les substitutions d'ajout et de suppression dans l'attribut multivalué sont raffinées par les règles suivantes :

a) Mise à jour de la variable Att lors de l'ajout d'un objet cc dans c

RULE *AttMultiBor_Ajout* **IS**
REFINES
$Att := Att \cup \{cc\} \times b$
REFINEMENT
VAR *qq* **IN**
$qq \longleftarrow Cons\_Seq\ (b, Null_T, k)$;
$;_{\ i=1..k}\ Att_i := Att_i \cup \{c \mapsto qq(i)\}$
END
END

L'opération *Cons_Seq* est contenue dans une machine abstraite *CONSEQ*. Cette machine abstraite est paramétrée par un ensemble *TT* et est incluse par le composant raffinement. Cette opération reçoit trois paramètres : un entier rr, un sous-ensemble *tt* de *TT* et de cardinal inférieur ou égal à *rr* et une valeur particulière x représentant la valeur $Null_T$. Elle renvoie une séquence de longueur *rr* dont les premières valeurs sont celles de l'ensemble *tt* complétée par la valeur *x* jusqu'à la longueur *rr*. La spécification de cette machine est la suivante :

MACHINE $CONSEQ(TT)$
OPERATIONS
$val \longleftarrow$ **Cons_Seq(tt,xx,rr)=**
PRE
$tt \subseteq TT \wedge xx \in TT \wedge rr \in NAT$
THEN
$val : (val \in seq(TT) \wedge size(val) = rr \wedge ran(1..card(tt) \lhd val) = tt \wedge ran(val) = tt \cup \{xx\})$
END
END

Exemple : l'opération d'ajout d'un film spécifiée en **B** par :

Ajout_Fl(fl, du, ge) $\widehat{=}$
BEGIN
film := film $\cup$ *{fl}* ;
Durée := Durée $\cup$ *{fl* $\mapsto$ *du}*
Genre := Genre $\cup$ *{fl}* $\times$ *ge*
END

est raffinée par :

Ajout_Fl(fl, du, ge) $\widehat{=}$
BEGIN
film := film $\cup$ *{fl}* ;
Durée := Durée $\cup$ *{fl* $\mapsto$ *du}*
VAR *qq* **IN**
$qq \longleftarrow Cons\_Seq\ (ge, Null_{STRING}, 3)$;
$Genre_1 := Genre_1 \cup \{fl \mapsto qq(1)\}$;
$Genre_2 := Genre_2 \cup \{fl \mapsto qq(2)\}$;
$Genre_3 := Genre_3 \cup \{fl \mapsto qq(3)\}$
END
END

b) Suppression dans la variable *Att*

RULE *AttMultiBor_Supp* **IS**
REFINES
$Att := cc \lhd Att$
REFINEMENT
$;_{\ i=1..k}\ Att_i := cc \lhd Att_i$
END
END

La substitution relative à la modification d'un tel attribut se raffine de manière similaire à celle relative à l'ajout. Il suffit de remplacer l'union par une surcharge.

Remarque. Le raffinement d'un attribut multivalué non borné est reporté à la section 4.5.6. Le raffinement d'un tel attribut nécessite l'existence d'une clé pour la classe ou l'association sur laquelle est défini l'attribut considéré (une fonction injective sur la variable associée à la classe ou l'association).

4.5.2.3 Élimination du constructeur de séquences

Comme pour le constructeur "Partie-de", le constructeur de séquences doit être éliminé de la spécification initiale afin que toutes les valeurs d'un attribut soient atomiques. Le constructeur de séquences permet d'introduire un ordre sur un ensemble de valeurs. L'idée principale de ce raffinement est de considérer un attribut Att de type séquence, défini sur une variable c associée à la classe C, comme une association ($Rang\_Att$) entre la classe et l'ensemble des entiers naturels. Cette association est décrite par un attribut $Valeur\_Att$ qui renseigne sur la valeur de l'objet pour un rang donné. Formellement, l'élimination de la séquence est modélisée de la manière suivante :

RULE *ÉlimSeq*$_1$*_D IS*
ABSTRACT_ENV *c, Att*
CONSTRAINT
$Att \in c \nrightarrow \mathbf{seq}_1(T)$
CONCRETE_ENV
VARIABLES *c, Rang_Att, Valeur_Att*
INVARIANT
$Rang\_Att \in c \leftrightarrow NAT_1\ \wedge$
$Valeur\_Att \in Rang\_Att \rightarrow T\ \wedge$
$\forall\ (xx,yy).\ (xx \mapsto yy \in Att \Rightarrow$
$\forall\ uu.(uu \in dom(yy) \Rightarrow xx \mapsto uu \in Rang\_Att \wedge (xx \mapsto uu) \mapsto yy(uu) \in Valeur\_Att))\ \wedge$
$\forall\ (aa,bb,cc).(\ aa \mapsto bb \in Rang\_Att \wedge (aa \mapsto bb) \mapsto cc \in Valeur\_Att \Rightarrow cc=Att(aa)(bb))$
END

Remarques

1. Les deux derniers invariants expriment la relation existante entre les variables concrètes et les variables abstraites.
2. Comme pour le cas du constructeur "Partie-de", si la fonction *Att* est totale, des contraintes supplémentaires sont spécifiées lors du passage au schéma relationnel.

Exemple : supposons à présent qu'un film ait plusieurs genres et que ces derniers sont ordonnés. En **B**, cet attribut est modélisé par :

$$Genre \in film \rightarrow \mathbf{seq}_1(STRING)$$

et est donc raffiné en :

VARIABLES *Rang_Genre, Valeur_Genre*
INVARIANT
$Rang\_Genre \in film \leftrightarrow NAT \wedge$
$Valeur\_Genre \in Rang\_Genre \rightarrow STRING \wedge$
$\forall (xx,yy).\ (xx \mapsto yy \in Genre \Rightarrow$
$\quad \forall uu.(uu \in dom(yy) \Rightarrow xx \mapsto uu \in Rang\_Genre \wedge (xx \mapsto uu) \mapsto yy(uu) \in Valeur\_Genre)) \wedge$
$\forall (aa,bb,cc).(\ aa \mapsto bb \in Rang\_Genre \wedge (aa \mapsto bb) \mapsto cc \in Valeur\_Genre \Rightarrow cc{=}Genre(aa)(bb))$

Raffinement des substitutions. Les substitutions d'ajout, de suppression et de modification relatives à la variable *Att* se raffinent comme suit :

a) Mise à jour de la variable Att lors de l'ajout d'un objet cc dans c

RULE *ÉlimSeq*$_1$*_Ajout* **IS**
REFINES
$Att{:=}Att \cup \{cc \mapsto val\}$
REFINEMENT
$Rang\_Att := Rang\_Att \cup \{cc\} \times dom(val);$
$Valeur\_Att := Valeur\_Att \cup (prj_2(\{cc\}, dom(val)); val)$[4]
END

Exemple : l'opération d'ajout d'un nouveau film spécifiée en **B** par :

Ajout_Fl(fl, du, ge)$\widehat{=}$
BEGIN
$film := film \cup \{fl\}$;
$Durée := Durée\_fl \cup \{fl \mapsto du\}$
$Genre := Genre \cup \{fl \mapsto ge\}$
END

est raffinée par :

Ajout_Fl(fl, du, ge)$\widehat{=}$
BEGIN
$film := film \cup \{fl\};$
$Durée := Durée \cup \{fl \mapsto du\};$
$Rang\_Genre := Rang\_Genre \cup \{fl\} \times dom(ge);$
$Valeur\_Genre := Valeur\_Genre \cup (prj_2(\{fl\}, dom(ge)); ge)$
END

4. $prj_2(A,B) = \{((a,b),b) \mid (a,b) \in A \times B\}$

b) Suppression dans l'attribut *Att*

RULE *ÉlimSeq$_1$_Supp* **IS**
REFINES
$Att := cc \lhd Att$
REFINEMENT
$Rang\_Att := cc \lhd Rang\_Att$;
$Valeur\_Att := (cc \times NAT) \lhd Valeur\_Att$
END

c) Modification de l'attribut *Att*
Le raffinement de la substitution de modification de l'attribut *Att* pour une instance *cc* s'exprime en une séquence de suppression des tuples relatifs à l'instance *cc* puis d'affectation de la nouvelle valeur à *cc*. Formellement, ce raffinement est modélisé comme suit :

RULE *ÉlimSeq$_1$_Change* **IS**
REFINES
$Att := Att \mathbin{\lhd\!\!\!+} \{cc \mapsto val\}$
REFINEMENT[5]
$Rang\_Att := cc \lhd Rang\_Att$;
$Valeur\_Att := (cc \times NAT) \lhd Valeur\_Att$;
$Rang\_Att := Rang\_Att \cup \{cc\} \times dom(val)$;
$Valeur\_Att := Valeur\_Att \cup (prj_2(\{cc\}, dom(val)); val)$
END

4.5.3 Identification des objets

Le processus de traduction des diagrammes **UML** en **B** décrit au chapitre 3 est orienté objets. En revanche, le modèle relationnel est un modèle par valeurs : l'identification des différents objets se fait sur la base des valeurs d'un ou plusieurs attributs qui constituent la clé. Donc, une étape du processus de raffinement va consister à introduire cette notion de clé dans le système. Cette étape est une préparation au passage d'un modèle par objets vers un modèle par valeurs. La transition du modèle par objets vers le modèle par valeurs consiste en la substitution de chaque objet par la valeur de sa clé. En **B**, cette transition n'est possible qu'à la dernière étape de raffinement (implémentation) car, ceci correspondra au raffinement des ensembles abstraits, i.e à leur valuation.

Raffinement des données. Pour chaque classe, une clé est à exhiber. En **B**, une classe *C* est modélisée par une variable *c* qui est sous-ensemble d'un ensemble abstrait *C*. La clé de cette classe est traduite par une fonction totale injective définie de *c* dans le type de cette clé. Pour des contraintes liées à la méthode **B**, on exige qu'une telle fonction soit une variable. En effet, un ensemble abstrait ne peut être valué que par un type de base. Si une telle fonction n'existe pas dans la spécification abstraite initiale, cette étape permet de définir une clé pour cette classe. Formellement, cette règle s'exprime comme suit : L'ensemble abstrait TT_c désigne le type abstrait de la clé introduite. Ce dernier sera implémenté (valué) par l'un des types de base : NAT, Z,. . . . Bien que cette évaluation soit possible à ce niveau même (remplacement de TT_c par un type de base), une meilleure solution

5. Une écriture possible du raffinement est : $(Rang\_Att := Rang\_Att \mathbin{\lhd\!\!\!+} \{cc\} \times dom(val)\}; \; Valeur\_Att := Valeur\_Att \mathbin{\lhd\!\!\!+} \{cc\} \times dom(val))$. L'intérêt de l'écriture adoptée est de réutiliser les preuves des règles de raffinement déjà établies pour les substitutions d'ajout et de suppression. De plus, cela évite l'écriture de règles supplémentaires de raffinement d'une substitution de surcharge aux étapes suivantes.

6. Cet invariant restreint l'application de la règle aux classes ne possédant pas de clé. Cet invariant assure en outre un nombre limité d'application de cette règle, et par conséquent, la terminaison du processus de raffinement.

RULE *IntroClé_D* **IS**
ABSTRACT_ENV
  SETS *C*
  VARIABLES c
CONSTRAINT
  $c \subseteq C \wedge$
  $\neg(\exists\, c_{clé}.(c_{clé} \in c \rightarrowtail TT))^6$
CONCRETE_ENV
  SETS TT_c
  VARIABLES $c,\ c_{clé}$
INVARIANT $c_{clé} \in c \rightarrowtail TT_c$
END

est de la différer à l'implémentation. En effet, d'une part, une telle information peut être considérée comme un détail de bas niveau et donc ne doit pas figurer à ce niveau d'abstraction, et d'autre part, ceci permettra une meilleure maintenance du code : un éventuel changement du type de cette variable n'affectera que le composant implémentation.

Exemple : on considère le diagramme de classes de la figure 4.1 avec la première alternative comme solution de l'élimination de l'héritage. La classe *Cassette* ne possédant pas de clé, cette étape permet d'ajouter une variable *Num_Ca*. Cette variable est définie comme une fonction injective totale de *cassette* vers l'ensemble abstrait $TT_{Cassette}$. La spécification du raffinement est la suivante :

DEFINITIONS
  $film \mathrel{\widehat=} Est\_film^{-1}[\{TRUE\}];$
  $série \mathrel{\widehat=} Est\_série^{-1}[\{TRUE\}]$
SETS $TT_{cassette}$
VARIABLES *cassette, Durée, Genre, Num_Sé, Est_film, Est_série, Num_Ca*
INVARIANT $Num\_Ca \in cassette \rightarrowtail TT_{cassette}$

Raffinement des substitutions. La règle de raffinement précédente a introduit une nouvelle donnée $c_{clé}$ sur la variable *c*. Afin de préserver l'invariant (c'est l'invariant de collage de la règle) du composant raffinement, il est nécessaire de mettre à jour cette nouvelle variable à chaque mise à jour de la variable *c*. Les mises à jour dans notre cas sont l'ajout d'une instance à *c* et la suppression d'un ensemble d'instances de *c*. Les règles de raffinement correspondant à ces deux substitutions sont les suivantes :

a) Ajout d'une instance à la variable c

RULE *IntroClé_Ajout* **IS**
REFINES $c:=c \cup \{cc\}$
REFINEMENT
  $c:=c \cup \{cc\};$
  VAR x **IN**
    $x \longleftarrow generer(TT_c\text{-}ran(c_{clé}));$
    $c_{clé}:=c_{clé} \cup \{cc \mapsto x\}$
  END
END

L'opération *generer* est contenue dans une machine abstraite *GENERE*. Cette machine abstraite est paramétrée par un ensemble *TT* et est incluse par le composant raffinement. Cette opération reçoit un paramètre tt sous-ensemble de *TT*. Elle renvoie une valeur arbitraire *val* de l'ensemble tt. La spécification de cette machine est la suivante :

MACHINE GENERE(TT)
OPEARTIONS
$val \longleftarrow$ **generer(tt)=**
PRE
  $tt \subseteq TT$
THEN
  ANY rr **WHERE** $rr \in tt$ **THEN**
    val:=rr
  END
END
END

Exemple : l'opération abstraite d'ajout d'une cassette de la page 78 se raffine par :

Ajout_Ca (ca, du) $\widehat{=}$
BEGIN
  cassette :=cassette $\cup$ *{ca}*;
  VAR *nu* **IN**
    nu $\longleftarrow$ *generer(*$TT_{cassette}$*- ran(Num_Ca))*;
    Num_Ca := Num_Ca $\cup$ *{ca* $\mapsto$ *nu}*
  END ;
  Durée := Durée $\cup$ *{ca* $\mapsto$ *du}*
END

b) Suppression d'un ensemble d'instances de la variable c

RULE *IntroClé_Supp* **IS**
REFINES $c := c - cc$
REFINEMENT
  $c := c - cc;$
  $c_{clé} := cc \lhd c_{clé}$
END

4.5.4 Élimination des associations monovaluées : fonctions dont l'ensemble d'arrivée est une variable

Une association monovaluée est une association ayant au moins un rôle dont la borne supérieure de sa multiplicité est 1. Si l'association n'a pas d'attributs, celle-ci est éliminée en ajoutant un attribut supplémentaire à la classe associée à ce rôle monovalué.

Raffinement des données. Chaque fonction *Ass* définie de la variable *a* dans la variable *b* de clé $b_{clé}$ est remplacée par la fonction *Ass_Mig*, de même nature que *Ass*, définie de *a* dans le type de la clé de *b*. Intuitivement, cela revient à considérer cette association comme une caractéristique de la classe A associée à la variable a. Formellement, cette règle s'exprime comme suit :

RULE *ÉlimAssoMono_D* **IS**
ABSTRACT_ENV
  SETS A, B, TT_b
  VARIABLES $a, b, b_{clé}, Ass$
CONSTRAINT
  $a \subseteq A \wedge b \subseteq B \wedge b_{clé} \in b \rightarrowtail TT_b \wedge$
  $Ass \in a \twoheadrightarrow\!\!\!\!\!\!+ \; b \wedge$
  $\neg (\exists f.(f \in Ass \twoheadrightarrow\!\!\!\!\!\!+ \; TT))$[7]
CONCRETE_ENV
  VARIABLES $a, b, b_{clé}, Ass\_Mig$
INVARIANT
  $Ass\_Mig \in a \twoheadrightarrow\!\!\!\!\!\!+ \; TT_b \wedge$[8]
  $Ass\_Mig = (Ass;\, b_{clé})$
END

Exemple : la classe *Cassette*, *Boutique* et l'association *Appartient*, de la figure 4.1 sont spécifiées en **B** par :

SETS *Boutique, Cassette*
VARIABLES *cassette, boutique, Num_Bo, Appartient, . . .*
INVARIANT
  $cassette \subseteq Cassette \wedge boutique \subseteq Boutique \wedge Num\_Bo \in boutique \rightarrowtail NAT \wedge$
  $Appartient \in cassette \rightarrow boutique$
  . . .

cette spécification se raffine en :

VARIABLES *cassette, boutique, Num_Bo, Appartient_Mig, . . .*
INVARIANT
  $Appartient\_Mig \in cassette \rightarrow NAT \wedge$
  $Appartient\_Mig = (Appartient;\, Num\_Bo)$

Raffinement des substitutions. La règle de raffinement précédente a remplacé une variable de la spécification abstraite initiale par une nouvelle variable. Le raffinement des substitutions dans ce cas consiste à remplacer chaque substitution liée à la variable éliminée par une substitution sur la nouvelle variable qui satisfait l'invariant de collage. Intuitivement, ce raffinement consiste à remplacer chaque instance de *b* par la valeur de sa clé. Les substitutions concernées par ce raffinement sont : l'ajout d'un lien et la suppression d'un ensemble de liens. Formellement, le raffinement de ces substitutions est exprimé par les règles suivantes :

a) Ajout d'un lien dans *Ass*

RULE *ÉlimAssMono_Ajout* **IS**
REFINES
  $Ass := Ass \cup \{a_1 \mapsto b_1\}$
REFINEMENT
  $Ass\_Mig := Ass\_Mig \cup \{a_1 \mapsto b_{clé}(b_1)\}$
END

7. Ce prédicat signifie que l'association est sans attributs.

8. Cet invariant est superflu car il peut être déduit du deuxième invariant $Ass\_Mig = (Ass;\ b_{clé})$. Son explicitation est utile pour les phases suivantes.

Exemple : l'opération d'ajout d'un lien à l'association *Appartient* est spécifiée en **B** par :

BasicAjoutAppartient(ca, bb)$\widehat{=}$
BEGIN
Appartient := Appartient $\cup$ *{ca* $\mapsto$ *bb}*
END

cette opération se raffine par :

BasicAjoutAppartient(ca,bb)$\widehat{=}$
BEGIN
Appartient_Mig := Appartient_Mig $\cup$ *{ca* $\mapsto$ *Num_Bo(bb)}*
END

b) Suppression des liens associés à un ensemble d'instances a_1 de la classe source de l'association

RULE *ÉlimRoleMono_Supp_a* **IS**
REFINES
$Ass := a_1 \lhd Ass$
REFINEMENT
$Ass\_Mig := a_1 \lhd Ass\_Mig$
END

4.5.5 Raffinement des autres associations

Une association, dont les deux rôles sont multivalués ou ayant des attributs, est transformée en une classe ayant, en plus des attributs de l'association, deux attributs supplémentaires (un par classe) liés par des contraintes référentielles aux clés des classes concernées par l'association. Ces deux attributs constituent une clé primaire de la classe ainsi créée. Formellement, cette règle s'exprime comme suit :

RULE *ÉlimAsso_D* **IS**
ABSTRACT_ENV
SETS *A, B,* TT_a, TT_b
VARIABLES *a, b,* $a_{clé}$, $b_{clé}$, *Ass*
CONSTRAINT
$a \subseteq A \wedge b \subseteq B \wedge$
$Ass \in a \leftrightarrow b \wedge$
$a_{clé} \in a \rightarrowtail TT_a \wedge$
$b_{clé} \in b \rightarrowtail TT_b$
CONCRETE_ENV
VARIABLES *a, b,* $a_{clé}$, $b_{clé}$, *Ass,* Ass_a, Ass_b
INVARIANT
$Ass_a = Ass \lhd (prj_1(a,b);\ a_{clé}) \wedge$
$Ass_b = Ass \lhd (prj_2(a,b);\ b_{clé}) \wedge$
$(Ass_a \otimes Ass_b) \in Ass \rightarrowtail TT_a \times TT_b$ [9]
END

Exemple : les classes *Client* et *Cassette* et l'association *Emprunt* de la figure 4.1 sont spécifiées en **B** par :

SETS *Client, Cassette*
VARIABLES *client, Num_Cl, cassette, Num_Ca, Emprunt, ...*

9. Comme pour l'élimination des associations monovaluées, cet invariant est superflu car ce il peut être déduit des deux premiers. Son explicitation est utile pour les phases suivantes

INVARIANT
$client \subseteq Client \wedge cassette \subseteq Cassette \wedge$
$Num\_Cl \in client \rightarrowtail NAT \wedge Num\_Ca \in cassette \rightarrowtail TT_{cassette} \wedge$
$Emprunt \in client \leftrightarrow cassette \wedge$
$\ldots$

Cette spécification se raffine en :

VARIABLES *client, Num_Cl, cassette, Num_Ca, Emprunt,* $Emprunt_{client}$, $Emprunt_{cassette}$
INVARIANT
$Emprunt_{client} = Emprunt \lhd (prj_1(client,cassette);\ Num\_Cl) \wedge$
$Emprunt_{cassette} = Emprunt \lhd (prj_2(client,cassette);\ Num\_Ca) \wedge$
$(Emprunt_{client} \otimes Emprunt_{cassette}) \in Emprunt \rightarrowtail NAT \times NAT$

Raffinement des substitutions. La règle de raffinement précédente a introduit deux nouvelles données Ass_a et Ass_b sur la variable *Ass*. Afin de préserver l'invariant (c'est l'invariant de collage de la règle) du composant raffinement, il est nécessaire de mettre à jour cette nouvelle variable à chaque mise à jour de la variable *Ass*. Comme pour le raffinement des rôles monovalués, les mises à jour dans notre cas sont l'ajout d'un lien à *Ass* et la suppression d'un ensemble de liens associés à un ensemble d'instances de a ou de b. Les règles de raffinement correspondant à ces opérations sont les suivantes :

a) Ajout d'un lien dans *Ass*

RULE *ÉlimAss_Ajout* **IS**
REFINES $Ass := Ass \cup \{a_1 \mapsto b_1\}$
REFINEMENT [10]
$Ass := Ass \cup \{a_1 \mapsto b_1\};$
$Ass_a := Ass_a \cup \{a_1 \mapsto b_1 \mapsto a_{clé}(a_1)\};$
$Ass_b := Ass_b \cup \{a_1 \mapsto b_1 \mapsto b_{clé}(b_1)\}$
END

Exemple : l'opération d'ajout d'un emprunt spécifiée en **B** par :

BasicAjoutEmprunt(cl,ca,da)$\widehat{=}$
BEGIN
$Emprunt := Emprunt \cup \{cl \mapsto ca\}$;
$Date\_E := Date\_e \cup \{cl \mapsto ca \mapsto da\}$
END

est raffinée en :

BasicAjoutEmprunt(cl,ca,da)$\widehat{=}$
BEGIN
$Emprunt := Emprunt \cup \{cl \mapsto ca\}$;
$Emprunt_{client} := Emprunt_{client} \cup \{cl \mapsto ca \mapsto Num\_Cl(cl)\}$;
$Emprunt_{cassette} := Emprunt_{cassette} \cup \{cl \mapsto ca \mapsto Num\_Ca(ca)\}$;
$Date\_e := Date\_e \cup \{cl \mapsto ca \mapsto da\}$
END

10. Une écriture possible pour la mise à jour des attributs Ass_a et Ass_b serait : $Ass_a := Ass \lhd (prj_1(a,b);\ a_{clé});\ Att_b := Att \lhd (prj_2(a,b);\ b_{clé})$. Néanmoins, l'écriture adoptée est plus avantageuse. En effet, d'une part son expression est simple, d'autre part, les substitutions obtenues sont implémentables.

b) Suppression d'un lien de Ass

RULE *ÉlimAss_Supp_Ass* **IS**
REFINES $Ass := Ass - \{a_1 \mapsto b_1\}$
REFINEMENT
$Ass := Ass - \{a_1 \mapsto b_1\};$
$Ass_a := \{a_1 \mapsto b_1\} \lhd Ass_a;$
$Ass_b := \{a_1 \mapsto b_1\} \lhd Ass_b$
END

c) Suppression des liens associés à un ensemble d'instances a_1

RULE *ÉlimAss_Supp_a* **IS**
REFINES $Ass := a_1 \lhd Ass$
REFINEMENT
$Ass := a_1 \lhd Ass;$
$Ass_a := (a_1 \times b) \lhd Ass_a;$
$Ass_a := (a_1 \times b) \lhd Ass_b$
END

Exemple : l'opération de suppression des liens de l'association *Emprunt* relatifs à un ensemble de clients *Cl* spécifiée en **B** par :

BasicSuppressionEmpruntClient(Cl) $\widehat{=}$
BEGIN
Emprunt:= Cl $\lhd$ *Emprunt* ;
Date_E:=(Cl $\times$ *cassette)* $\lhd$ *Date_E*
END

est raffinée en :

BasicSuppressionEmpruntClient(Cl)$\widehat{=}$
BEGIN
Emprunt := Cl $\lhd$ *Emprunt* ;
$Emprunt_{client}$:= *(Cl* $\times$ *cassette)* $\lhd$ $Emprunt_{client}$;
$Emprunt_{cassette}$:= *(Cl* $\times$ *cassette)* $\lhd$ $Emprunt_{cassette}$;
Date_E := (Cl $\times$ *cassette)* $\lhd$ *Date_E*
END

d) Suppression des liens associés à un ensemble d'instances b_1

RULE *ÉlimAss_Supp_b* **IS**
REFINES $Ass := Ass \rhd b_1$
REFINEMENT
$Ass := Ass \rhd b_1;$
$Ass_a := (a \times b_1) \rhd Ass_a;$
$Ass_b := (a \times b_1) \rhd Ass_b$
END

4.5.6 Raffinement des attributs multivalués non bornés

Un attribut multivalué non borné, de type *T* défini sur la variable *c*, est transformé en une classe. L'ensemble abstrait de cette nouvelle classe est implicitement égal au produit de l'ensemble abstrait associé à c et du type T. Chaque instance de cette nouvelle classe est caractérisée par une instance c et la valeur qu'a cette l'instance pour l'attribut Att. Ces deux caractéristiques sont représentées par les deux attributs Att_c et Att_{val}. Le premier attribut fait référence à la clé de la classe *C*, le second représente la valeur de l'attribut en question. Le couple d'attributs (Att_c, Att_{val}) forme une clé pour la classe ainsi créée.

Raffinement des données.

RULE *ÉlimAttMuli_D* **IS**
ABSTRACT_ENV
  VARIABLES c, $c_{clé}$, Att
CONSTRAINT
  $c_{clé} \in c \rightarrowtail TT_c \wedge Att \in c \leftrightarrow T$
CONCRETE_ENV
  VARIABLES c, $c_{clé}$, Att_c, Att_{val}, Att
INVARIANT
  $Att_c = Att \lhd (prj_1{}^{11}(c,T);\ c_{clé}) \wedge$
  $Att_{val} = Att \lhd (prj_2(c,T);\ id(T)) \wedge$
  $(Att_c \otimes Att_{val}) \in Att \rightarrowtail TT_c \times T$
END

Exemple : on considère à présent qu'un film a plusieurs genres, et qu'on a choisi de conserver les sous-classes comme solution de l'élimination de l'héritage. La variable *Genre* est donc spécifiée par (on considère avoir rajouté la clé *Num_Fl* pour la classe *Film*) :

VARIABLES *film, Num_Fl, Genre*
INVARIANT
  $Num\_Fl \in film \rightarrowtail TT_{film} \wedge$
  $Genre \in film \leftrightarrow STRING$

qui se raffine en :

VARIABLES *film, Num_Fl, Genre,* $Genre_{film}$, $Genre_{val}$
INVARIANT
  $Genre_{film} = Genre \lhd (prj_1(film,STRING);\ Num\_Fl) \wedge$
  $Genre_{val} = Genre \lhd (prj_2(film,STRING);\ id(STRING)) \wedge$
  $(Genre_{film} \otimes Genre_{val}) \in Genre \rightarrowtail TT_{Film} \times STRING$

Raffinement des substitutions. La règle de raffinement précédente a introduit deux nouvelles données Att_c et Att_{val} sur la variable *Att*. Afin de préserver l'invariant de collage, il est nécessaire de mettre à jour cette nouvelle variable à chaque mise à jour de la variable *Att*. Les mises à jour dans notre cas sont l'ajout d'une instance de *c*, la suppression d'un ensemble d'instances de *c* et la modification de la valeur de l'attribut *Att* pour une instance de *C*. Les règles de raffinement correspondant à ces opérations sont les suivantes :

11. $prj_1(A,B) = \{((a,b),a) \mid (a,b) \in A \times B\}$

a) Ajout d'une instance de c

RULE *ÉlimAtt_Ajout* **IS**
REFINES $Att := Att \cup \{cc\} \times v$
REFINEMENT [12]
$Att := Att \cup \{cc\} \times v;$
$Att_c := Att \lhd (prj_1(c,T);\ c_{clé});$
$Att_{val} := Att \lhd (prj_2(c,T);\ id(T))$
END

Exemple : l'opération d'ajout d'un film spécifiée en **B** par :

Ajout_Film(fl,ge)$\widehat{=}$
BEGIN
film := film $\cup$ *{fl}* ;
Genre := Genre $\cup$ *{fl}* $\times$ *ge*
END

est raffinée en [13] :

Ajout_Film(fl,ge)$\widehat{=}$
BEGIN
VAR nu **IN**
nu $\longleftarrow$ *generer(*TT_{film} *- ran(Num_Fl))*;
film:= film $\cup$ *{fl}* ;
Num_Fl:= Num_Fl $\cup$ *{fl* $\mapsto$ *nu}* ;
Genre := Genre $\cup$ *{fl}* $\times$ *ge* ;
$Genre_{film}$*:= Genre* $\lhd$ *(prj*$_1$*(film,STRING)*; *Num_Fl)*;
$Genre_{val}$*:= Genre* $\lhd$ *(prj*$_2$*(film,STRING)*; *Id(STRING))*
END
END

b) Suppression d'instances de c

RULE *ÉlimAttMult_Sup* **IS**
REFINES $Att := cc \lessdot Att$
REFINEMENT
$Att := cc \lessdot Att;$
$Att_c := (cc \times T) \lessdot Att_c;$
$Att_{val} := (cc \times T) \lessdot Att_{val}$
END

Le raffinement de la substitution de modification d'un attribut multivalué se raffine de la même manière que la substitution d'ajout. Il suffit de remplacer l'union par une surcharge.

4.5.7 Extension des fonctions partielles

Dans le modèle relationnel, chaque attribut doit être valué. La valeur nulle dans les base des données est un artifice de représentation de données non renseignées, inapplicables ou incomplètes. En **B**, l'inapplicabilité

12. Une écriture possible du raffinement est : $(Att_c := Att_c \cup prj_1(\{c_1\},v);\ Att_{val} := Att_{val} \cup prj_2(\{c_1\},v))$. Cette écriture a l'inconvénient d'augmenter de 2 le nombre d'obligations de preuve. En effet, pour chaque partie de l'invariant une preuve est générée. La première écriture évite de prouver les deux premiers invariants.

13. La variable Num_Fl est la clé rajoutée pour la variable *film* lors de la phase d'identification des objets.

d'une information est modélisé par une fonction partielle *f*. Cette étape de raffinement va permettre de prolonger le domaine de toute fonction partielle à son ensemble de départ : la fonction devient donc totale.

Raffinement des données. Intuitivement l'extension d'une fonction partielle *f* en fonction totale f_{totale} consiste à associer une valeur spéciale ($Null_T$) à chaque élément de l'ensemble de départ qui n'appartenait pas au domaine de *f*. Pour cela, l'ensemble d'arrivée de la fonction *f* est étendu par la valeur $Null_T$. Formellement, l'extension d'une fonction partielle est spécifiée par :

RULE *ExtenFonctPart_D* **IS**
ABSTRACT_ENV
  VARIABLES *f, c*
CONSTRAINT
  $f \in c \nrightarrow T \wedge dom(f) \neq c$ [14]
CONCRETE_ENV
  f_{totale}, c
INVARIANT
  $f_{totale} \in c \rightarrow T \cup \{Null_T\} \wedge$
  $f = f_{totale} \rhd \{Null_T\}$
END

En **B**, l'extension d'un ensemble (type de base, ensemble enuméré ou abstrait) est irréalisable. Pour cela, on se réserve une valeur particulière de chaque type de base de **B** qui représentera par la suite la valeur nulle du type considéré. Par exemple, on se réserve 0 ou $MAXINT$ pour le type NAT, la chaîne vide pour le type STRING et la valeur 2 pour le type BOOL qui est simulé dans notre cas (par commodité) par l'ensemble des deux valeurs $\{0,1\}$. La valeur nulle choisie doit avoir une sémantique représentative de l'inapplicabilité. En effet, on ne doit pas choisir une valeur que la fonction peut prendre là où elle serait réellement applicable. Par exemple, la valeur nulle de la fonction *Age_Conjoint* peut être égale à zéro. Par contre la valeur nulle de la fonction *Note_Etud* ne peut être zéro mais par exemple -1, car un étudiant peut avoir effectivement zéro à une évaluation. La valeur nulle dépend donc de la sémantique de la fonction à étendre. Cette dernière doit être définie en concertation avec l'utilisateur.

Exemple : on considère le diagramme de classes de la figure 4.1 avec la première alternative comme solution de l'élimination d'héritage. L'attribut *Num_Sé*, de la page 76, est étendu comme suit :

VARIABLES *cassette*, $Num\_Sé_{totale}, \ldots$
INVARIANT
  $Num\_Sé_{totale} \in cassette \nrightarrow (NAT \cup \{-1\}) \wedge$
  $Num\_Sé = Num\_Sé_{totale} \rhd \{-1\}$

Comme nous pouvons le remarquer, Bien que la fonction *Num_Sé* soit injective, son extension ne l'est plus. En effet, nous avons choisi d'affecter la même valeur nulle à chaque instance initialement non renseignée pour l'attribut étendu.

Raffinement des substitutions. Le raffinement des substitutions consiste à remplacer chaque substitution agissant sur la variable supprimée par une nouvelle substitution sur la variable introduite. Le raffinement des

14. Cet invariant restreint l'application de la règle aux fonctions réellement partielles. L'absence de cet invariant n'exclut pas son application à des fonctions totales. En effet, une fonction totale est un cas particulier de fonctions partielles dont le domaine est égal à son ensemble de départ. Cet invariant assure en outre un nombre limité d'application de cette règle, et par conséquent, la terminaison du processus de raffinement.

substitutions consiste à affecter la valeur *Null* à la variable f_{totale} à chaque création d'une nouvelle instance de la variable a, et à remplacer simplement toute occurrence de la variable f par f_{totale} pour les autres substitutions.

RULE *ExtenFonctionPart_Ajout* **IS**
REFINES
  $c := c \cup \{cc\}$
CONSTRAINT
  $\neg(f := f \cup \{cc \mapsto val\})$[15]
REFINEMENT
  $c := c \cup \{cc\}$;
  $f_{totale} := f_{totale} \cup \{cc \mapsto Null_T\}$
END

Exemple : l'opération d'ajout d'une cassette de la page 78 se raffine en :

Ajout_Ca (ca, du) $\widehat{=}$
BEGIN
  cassette := cassette $\cup$ *{ca}* ;
  $Num\_S\acute{e}_{totale} := Num\_S\acute{e}_{totale} \cup \{ca \mapsto -1\}$
  Durée := Durée $\cup$ *{ca* $\mapsto$ *du}*;
  Est_film :=Est_film $\cup$ *{ca* $\mapsto$ *FALSE}* ;
  Est_série := Est_série $\cup$ *{ca* $\mapsto$ *FALSE}*
END

4.5.8 Transition d'un modèle fonctionnel vers un modèle ensembliste

Le but de cette étape est la définition des structures des différentes tables. L'idée intuitive est de rassembler sous le nom d'une variable (table) les caractéristiques concernant une même entité (classe). Cette variable sera par la suite traduite en une table **SQL**. Afin de faciliter l'écriture des substitutions liées aux opérations de modification de la valeur d'un attribut, on exprimera chaque attribut par une définition. Les noms de ces définitions doivent être différents des attributs disparaissant lors du raffinement afin d'éviter le conflit avec les invariants de collage.

Raffinement des données. À cette étape de raffinement, toutes les données définies sur les variables sont des fonctions totales. Chaque ensemble de fonctions $f_1, \ldots, f_n$ définies sur le même ensemble source a est remplacé par une unique variable T_a définie de a dans le produit cartésien des types d'arrivée des fonctions $f_1, \ldots, f_n$.

15. Cette contrainte est nécessaire pour le raffinement des fonctions partielles issues de l'élimination des spécialisations. En effet, l'opération d'ajout dans une des sous-classes comporte déjà l'ajout dans la fonction considérée.

Cette fonction T_a représente le produit direct des fonctions initiales. Formellement :

RULE *DéfStruct_D* **IS**
ABSTRACT_ENV
  VARIABLES $a, f_1, \ldots, f_n$
CONSTRAINT
  $f_1 \in a \rightarrowtail T_1 \wedge$
  $\ldots$
  $f_n \in a \rightarrow T_n$
CONCRETE_ENV
  DEFINITIONS
    $f'_n \mathrel{\widehat{=}} (T_a; prj_2(T_1 \times \ldots \times T_{n-1}, T_n));$
    $f'_{n-1} \mathrel{\widehat{=}} (T_a; prj_1(T_1 \times \ldots \times T_{n-1}, T_n); prj_2(T_1 \times \ldots \times T_{n-2}, T_{n-1}));$
    $\ldots$
  VARIABLES a, T_a
INVARIANT
  $T_a \in a \rightarrow T_1 \times \ldots \times T_n \wedge$
  $T_a = f_1 \otimes \ldots \otimes f_n$
END

Exemple : récapitulons l'exemple de raffinement de la figure 4.1. On suppose avoir choisi la première solution pour l'élimination de l'héritage. L'application successive des différents raffinements sur la variable *série* permet l'obtention de la spécification suivante (seuls les invariants nécessaires pour le raffinement sont repris) :

VARIABLES *série, Num_Sé, Durée_Série, Appartient_Série_Mig*
INVARIANT
  $Num\_Sé \in série \rightarrowtail NAT \wedge$
  $Durée\_Sé \in série \rightarrow NAT$
  $Appartient\_Série\_Mig \in série \rightarrow NAT$

L'application de la règle de raffinement ci-dessus, définit la variable $T_{série}$ comme suit :

VARIABLES *série*, $T_{série}$
INVARIANT
  $T_{série} \in série \rightarrowtail NAT \times NAT \times NAT \wedge$
  $T_{série} = Num\_Sé \otimes Durée\_Série \otimes Appartient\_Série\_Mig$

Raffinement des substitutions En suivant le principe de regroupement de variables pour le raffinement des données, le raffinement des opérations consiste à regrouper l'ensemble des substitutions relatives à la mise à jour des fonctions $f_1, \ldots, f_n$ en une seule substitution de mise à jour de la fonction T_a. Comme pour les règles de raffinement précédentes, nous distinguons trois types de mise à jour :

a) Ajout d'une instance de a : trois cas sont à distinguer. Dans notre cas, une table est soit l'implémentation d'une classe (une vraie), un attribut multivalué ou une séquence. Bien que la règle de raffinement des données soit la même pour les trois cas, le raffinement des substitutions d'ajout est différent. En effet, la forme des substitutions est différente.

• **Cas d'un attribut multivalué**

RULE *DéfStruct_AttMult* **IS**
REFINES
$Att := Att \cup \{cc\} \times ge;$
$Att_c := Att \lhd (prj_1(c,TT);\ c_{cl\acute{e}});$
$Att_{val} := Att \lhd (prj_2(c,TT);\ id(TT))$
REFINEMENT
$Att := Att \cup \{cc\} \times ge;$
$T_{Att} := T_{Att} \cup ((\{cc\} \times ge \times \{c'_{cl\acute{e}}(cc)\}) \otimes prj_2(\{cc\},ge))$
END

• **Cas de la séquence**

RULE *DéfStruct_Seq* **IS**
REFINES
$Rang\_Att := Rang\_Att \cup \{cc\} \times dom(val);$
$Att_c := Rang\_Att \lhd (prj_1(c,TT);\ c_{cl\acute{e}});$
$Att_{val} := Rang\_Att \lhd (prj_2(c,TT);\ id(TT));$
$Valeur\_Att := Valeur\_Att \cup (prj_2(\{cc\}, dom(val));\ val)$
REFINEMENT
$Rang\_Att := Rang\_Att \cup \{cc\} \times dom(val);$
$T\_Seq := T\_Seq \cup ((\{cc\} \times dom(val) \times \{c'_{cl\acute{e}}(cc)\}) \otimes prj_2(\{cc\},dom(val)) \otimes (prj_2(\{cc\},dom(val));\ val))$
END

• **Cas d'une classe standard**

RULE *DéfStruct_A* **IS**
REFINES
$;_{\ i=1..n}\ f_i := f_i \cup \{a_1 \mapsto val_i\}$
REFINEMENT
$T_a := T_a \cup \{a_1 \mapsto (val_1 \mapsto val_2 \mapsto \ldots \mapsto val_n)\}$
END

b) Suppression d'un ensemble d'instances de a

RULE DéfStruct_Supp IS
REFINES
$;_{\ i=1..n}\ f_i := a_1 \lhd\!\!\!- f_i$
REFINEMENT
$T_a := a_1 \lhd\!\!\!- T_a$
END

c) Modification de la valeur de f_i

RULE DéfStruct_Supp IS
REFINES
$f_i(a_1) := val$
REFINEMENT
$T_a := f'_1 \otimes \ldots \otimes f'_{i-1} \otimes (f'_i \lhd\!\!\!+ \{a_1 \mapsto val\}) \otimes \ldots f'_n$
END

4.6 Correction du processus de raffinement

La correction d'un développement par raffinement repose sur la preuve de correction de chaque étape de raffinement qui permet, de proche en proche, d'assurer la correction du code généré à la dernière étape. Pour cela, la preuve de correction de chacune des règles de raffinement, présentées dans ce chapitre, doit être établie. Il est important de noter que, dans notre cas, la preuve de chaque règle ne dépend que de la règle elle-même, elle est indépendante des spécificités de l'application étudiée. Ce caractère générique de la preuve permet d'une part l'automatisation du processus de raffinement et d'autre part la réutilisation des preuves ainsi établies. En utilisant le prouveur de l'AtelierB (version 3.5), l'ensemble des règles définies dans ce processus a été prouvé (environ 75% des preuves sont prouvées automatiquement sans ajout de règles), donnant lieu à des scénarios de preuves génériques et donc réutilisables par instanciation (un exemple de preuve est donné ci-après). L'échec du prouveur automatique à établir certaines preuves est dû essentiellement au manque de règles et de tactiques dans sa base. Le tableau suivant résume les principaux résultats obtenus lors de cette phase de preuve (Aj : Ajout, Su : Suppression, Maj : Mise à jour, NbPrTriv: nombre de preuves triviales, NbPo: nombre de preuves prouvées automatiquement, NbUn: nombre de preuves restant à faire, %Pr: pourcentage de preuves établies) :

Preuve / RULE	NbPrTriv			NbPo			NbUn			%Pr
	Aj	Su	Maj	Aj	Su	Maj	Aj	Su	Maj	
Conservation de la super-classe	15	12	/	14	15	/	3	4	/	100
Conservation des sous-classes	5	5	12	5	5	6	0	1	2	100
Élimination du constructeur “Partie-de”	5	2	3	2	2	1	1	1	1	100
Élimination des attributs multivalués bornés	5	2	5	6	7	6	2	1	2	100
Élimination du constructeur de séquences	6	2	11	3	9	2	7	1	8	100
Identification des objets	5	2	/	3	5	/	0	0	/	100
Raffinement des autres attributs multivalués	3	2	6	3	3	1	4	5	4	100
Élimination des associations monovaluées	6	4	6	4	4	3	0	0	0	100
Élimination des autres associations	7	19	/	1	5	/	2	4	/	100
Extension des fonctions partielles	4	2	4	3	4	2	1	1	1	100
Transition modèle objets/valeurs	2	2	3	3	2	0	1	2	3	100
TOTAUX	**63**	**54**	**50**	**47**	**61**	**21**	**21**	**20**	**21**	**100%**

Afin de réaliser interactivement les preuves restantes (pour lesquelles le prouveur automatique a échoué), nous avons été amenée à rajouter des règles utilisateur, à la base de règle de l'AtelierB, que nous avons validées en utilisant la commande vr du prouveur interactif. Le tableau suivant donne les règles les plus représentatives :

$f \in s \leftrightarrow t \wedge$ $x \subseteq s$ $\Longrightarrow$ $x ⩤ f = f - x \times t$	$f \in s \leftrightarrow t \wedge$ $x \subseteq t$ $\Longrightarrow$ $f ⩥ x = f - s \times x$
$prj_1(A-a,B) == prj_1(A,B) - prj_1(a, B)$	$prj_2(A-a,B) == prj_2(A,B) - prj_2(a, B)$
$prj_1(A,B-b) == prj_1(A,B) - prj_1(A, b)$	$prj_2(A,B-b) == prj_2(A,B) - prj_2(A, b)$
$prj_1(A \cup a,B) == prj_1(A,B) \cup prj_1(a, B)$	$prj_2(A \cup a,B) == prj_2(A,B) \cup prj_2(a, B)$
$prj_1(A,B \cup b) == prj_1(A,B) \cup prj_1(A, b)$	$prj_2(A,B \cup b) == prj_2(A,B) \cup prj_2(A, b)$
$btrue \Longrightarrow prj_1(A,B) \in A \times B \nrightarrow A$	$btrue \Longrightarrow prj_2(A,B) \in A \times B \nrightarrow B$
$f \in s \rightarrow t \wedge$ $g \in s \rightarrow r$ $\Longrightarrow$ $f \otimes g \in s \rightarrow t \times r$	$f \in s \rightarrow t \wedge$ $g \in s \rightarrow r$ $\Longrightarrow$ $f = ((f \otimes g); prj_1(t,r)) \wedge g = ((f \otimes g); prj_2(t,r))$

$(a \cup b) \otimes c = (a \otimes c) \cup (b \otimes c)$	$a \otimes (b \cup c) = (a \otimes b) \cup (a \otimes c)$
$x \mapsto (y \mapsto z) \in f \otimes g$ $\Longrightarrow$ $x \mapsto y \in f \wedge x \mapsto z \in g$	$x \mapsto y \in f \wedge x \mapsto z \in g$ $\Longrightarrow$ $x \mapsto (y \mapsto z) \in f \otimes g$
$x \subseteq X \wedge$ $f \in X \rightarrow Y$ $\Longrightarrow$ $prj_1(X, Y)[x \lhd f] = x$	$\{a \mapsto b\} \otimes \{a \mapsto c\} == \{a \mapsto (b \mapsto c)\}$

Exemple de preuve En utilisant la définition A.3.1 de la correction du raffinement, la preuve de la règle d'élimination d'association monovaluée pour la substitution d'ajout revient à prouver les conditions suivantes :

P1. $\exists (Ass, Ass\_Mig).Ass\_Mig = (Ass;\ b_{clé})$
P2. $\forall Ass, b_{clé}, a, b, Ass\_Mig.\ (Trm(S) \wedge Ass\_Mig = (Ass;\ b_{clé}) \Rightarrow Trm(T))$
P3. $\forall Ass, b_{clé}, a, b, Ass\_Mig.(Trm(S) \wedge Ass\_Mig = (Ass;\ b_{clé}) \Rightarrow$
$\forall Ass\_Mig'.(Ass\_Mig' = Ass\_Mig \cup \{a \mapsto b_{clé}(b)\} \Rightarrow$
$\exists Ass'.(Ass' = Ass \cup \{a \mapsto b\} \wedge Ass\_Mig' = (Ass';\ b_{clé}))))$

où :

- $S \widehat{=} Ass = Ass \cup \{a \mapsto b\}$
- $T \widehat{=} Ass\_Mig = Ass\_Mig \cup \{a \mapsto b_{clé}(b)\}$

- **Preuve de P1 :** P1 est vrai pour $Ass = \emptyset$ et $Ass\_Mig = \emptyset$.
- **Preuve de P2 :** b étant un élément de B, $b_{clé}(b)$ est toujours défini. Donc T est bien définie et termine : $Trm(T) = True$. P2 est donc vrai.
- **Preuve de P3 :** après élimination du quantificateur universel et l'application de la déduction, le but à prouver devient :

$$\forall Ass\_Mig'.(Ass\_Mig' = Ass\_Mig \cup \{a \mapsto b_{clé}(b)\} \Rightarrow$$
$$\exists Ass'.(Ass' = Ass \cup \{a \mapsto b\} \wedge Ass\_Mig' = (Ass';\ b_{clé})))$$

sous l'hypothèse (car $Trm(S) = True$) :

$$Ass\_Mig = (Ass;\ b_{clé})$$

Une fois encore, par élimination du quantificateur universel et l'application de la déduction, le but à prouver devient :

$$\exists Ass'.(Ass' = Ass \cup \{a \mapsto b\} \wedge Ass\_Mig' = (Ass';\ b_{clé}))$$

Ce but est vrai pour Ass' égale à $Ass \cup \{a \mapsto b\}$. En effet :

$$\begin{aligned}(Ass';\ b_{clé}) &= (Ass \cup \{a \mapsto b\});\ b_{clé} \\ &= (Ass;\ b_{clé}) \cup (\{a \mapsto b\});\ b_{clé}) \text{ (car la composition est distributive sur l'union)} \\ &= Asso\_Mig \cup \{a \mapsto b_{clé}(b)\} \text{ (car } b_{clé} \text{ est injective)} \\ &= Asso\_Mig'. \quad \text{CQFD.}\end{aligned}$$

Les preuves des autres règles sont similaires et sont basées principalement sur l'application des propriétés des différents opérateurs relationnels. D'autres exemples de preuve peuvent être trouvés dans [42].

4.7 Propriétés du processus de raffinement

Dans cette section sont étudiées les propriétés de bonne définition du processus de raffinement. Nous vérifions que l'algorithme termine, qu'il est correct, complet et déterministe.

4.7.1 Terminaison

La propriété de terminaison d'un processus assure que pour toute instance des données reçues en entrée, le processus s'arrête au bout d'un temps fini. Pour prouver la terminaison de notre processus, il suffit de s'assurer que chaque règle de raffinement ne peut s'appliquer qu'un nombre fini de fois. De plus, pour chaque type de données non terminales, il existe une règle de raffinement qui peut lui être appliquée Une spécification terminale désigne, dans notre cas, une fonction totale dont l'ensemble d'arrivée est soit un type de base, un ensemble abstrait, ou un ensemble énuméré.

– *Règles d'élimination de l'héritage :* cette règle ne peut s'appliquer qu'une fois au plus sur tout morceau de spécification. Après sa première application, la variable associée à la super-classe ou celles associées aux sous-classes disparaissent. Sa condition d'application devenant non vérifiée, cette règle devient donc inapplicable. Les données générées par cette règle sont soit des fonctions totales dont les co-domaines désignent des ensembles abstraits, donc des spécifications terminales, ou des relations représentant des attributs multivalués ou des associations. Ces dernières sont raffinées par la suite par les règles correspondantes. De plus aucune règle de raffinement ne génère une spécification définissant une structure d'héritage.
– *Règle d'élimination du constructeur "Partie-de" :* cette règle ne peut s'appliquer qu'une fois au plus sur tout morceau de spécification. L'application de cette règle, n'introduit pas de constructeur "Partie de". De plus, aucune autre règle de raffinement n'introduit un tel constructeur. Par conséquent, cette règle n'est applicable que sur la spécification abstraite initiale. Les données générées par cette règle représentent des attributs multivalués non bornés qui sont raffinés par la suite par la règle correspondante.
– *Règles d'élimination des attributs multivalués bornés :* cette règle ne peut s'appliquer qu'une fois au plus sur tout morceau de spécification. Son application produit des fonctions totales donc des spécifications terminales.
– *Règle d'élimination du constructeur la séquence :* idem constructeur "Partie-de".
– *Règle d'identification des objets :* cette règle n'est applicable qu'une seule fois sur toute variable associée à une classe donnée. En effet, cette règle contient une garde qui interdit son application dès qu'une classe possède une clé.
– *Règle d'élimination des attributs multivalués non bornés :* idem attribut multivalué borné.
– *Règle d'élimination des associations monovaluées :* cette règle n'est applicable qu'une fois au plus sur tout morceau de la spécification. La condition d'application de cette règle n'est plus vérifiée après une première application de cette dernière. De plus, son application produit soit une fonction totale donc une spécification terminale, soit une fonction partielle qui se raffine en une fonction totale par la règle d'extension des fonctions partielles.
– *Règle de raffinement des autres associations :* idem attributs multivalués non bornés.
– *Règle d'extension des fonctions partielles :*cette règle n'est applicable qu'une seule fois sur chaque fonction partielle puisque celle-ci devient totale et donc terminale.
– *Règle de transition d'un modèle fonctionnel à un modèle ensembliste :* cette règle peut être appliquée plusieurs fois. Néanmoins, à chaque application le nombre de variables diminue. En effet, cette règle regroupe un ensemble de variables sous une seule nouvelle variable. Pour k fonctions totales, cette règle s'applique au plus $(k-1)$ fois.

Le graphe de la figure 4.4 visualise l'application des différentes règles de raffinement selon le type de données à raffiner. Un arc entre deux nœuds (rectangles à angles droits) A et B signifie que le raffinement de données de type A produit des données de type B. Le problème de terminaison peut être vu comme un problème d'existence d'un chemin sans cycles entre un nœud quelconque du graphe et le nœud terminal "FIN'.

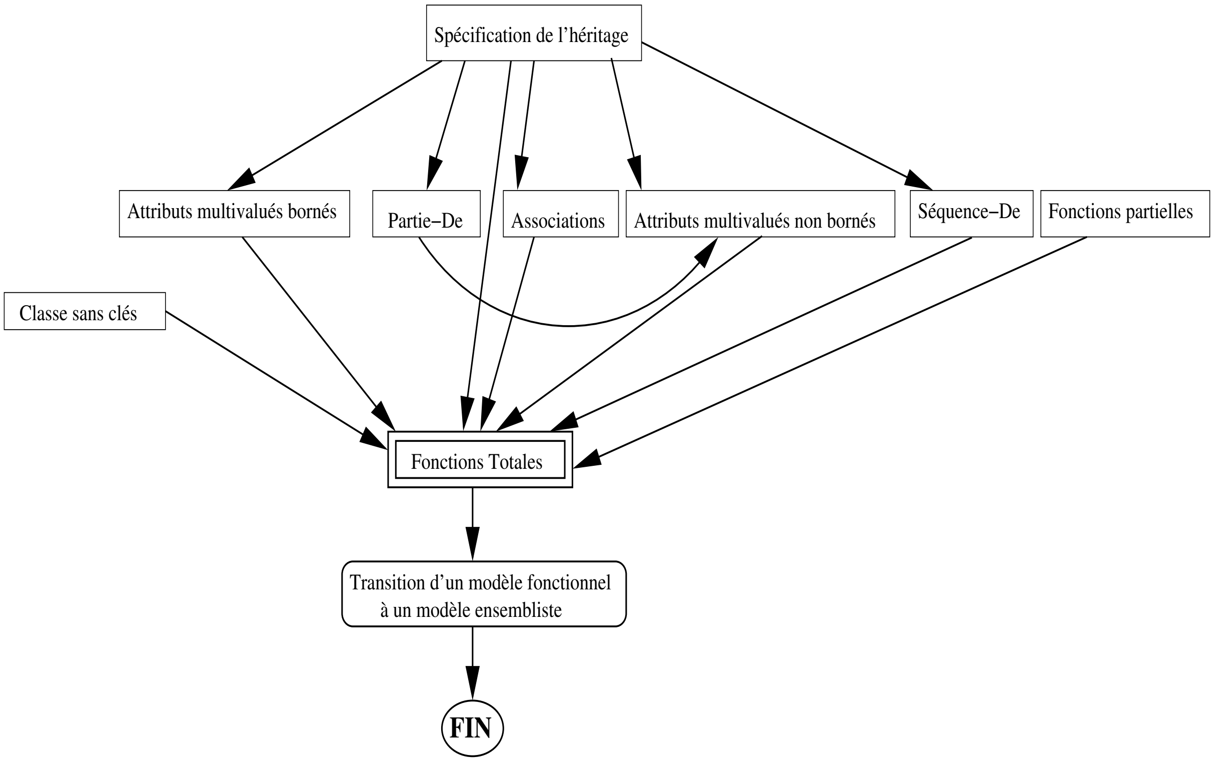


FIG. 4.4 – *Enchaînement des règles de raffinement sur les types de données*

4.7.2 Validation

La validation d'un processus signifie que les résultats obtenus par le processus sont conformes à ceux escomptés. La validation de notre processus repose, d'une part, sur le type de données engendré par la dernière étape, et d'autre part, sur la correction de chaque règle de raffinement. En effet, la structure de données générée par la dernière étape de raffinement est l'une des définitions formelles possibles du type de données utilisé dans le modèle relationnel. De plus, la correction de chaque étape du processus assure la correction de la spécification finale par rapport à la spécification la plus abstraite.

4.7.3 Complétude

Notre processus de raffinement est complet en ce sens qu'il définit une règle de raffinement pour chaque type de données de la spécification initiale.

4.7.4 Déterminisme

Le processus de raffinement que nous avons défini est complètement déterministe. En effet, sur chaque type de données (fonction, association monovalué, ...) une seule règle de raffinement est applicable à un instant donné. De plus, l'algorithme classique de passage d'un diagramme Entités/Associations à un modèle relationnel, sur lequel repose notre processus de raffinement, est complètement déterministe.

4.8 Conclusion

Dans ce chapitre, nous avons présenté un processus de raffinement **B** permettant la génération d'une implémentation relationnelle à partir d'une spécification **B** obtenue par l'application du processus de traduction du chapitre 3. Ce processus de raffinement consiste en un ensemble de règles opérant conjointement sur les données et les substitutions de la spécification initiale. Certaines règles de raffinement de données proviennent des algorithmes classiques de transformation d'un diagramme Entités/Associations en un schéma relationnel. L'apport de notre processus au regard des algorithmes existants est la prise en compte des traitements. De plus, ces algorithmes ne traitent pas les attributs de type “Partie-de” et “Sequence-De” qui, pour notre part, nous ont demandé une réflexion et un travail importants. En effet pour le type séquence par exemple, plusieurs solutions possibles aux premières étapes de raffinement se sont avérées irréalisables à l'implémentation.

L'ensemble des règles proposées dans ce chapitre a été vérifié en utilisant le prouveur de l'AtelierB. La vérification de la correction de certaines de ces règles de raffinement a nécessité l'ajout d'un ensemble de règles utilisateur vérifiées à leur tour en utilisant le vérificateur de règles. Un outil automatisant ce processus de raffinement est décrit dans le chapitre 7.

Le niveau d'abstraction atteint par le dernière étape de raffinement n'est évidement pas implémentable. Afin d'assurer une traduction quasi-automatique et naturelle, une phase supplémentaire de raffinement est nécessaire. Cette phase (implémentation) ainsi qu'un processus de traduction vers du code exécutable sont présentés dans le chapitre suivant.

Chapitre 5

Implémentation et codage SQL/JAVA

La dernière étape du processus de raffinement **B** produit une implémentation. Une implémentation **B** est une spécification directement traduisible vers le langage de programmation cible choisi. Par abus de langage, on utilisera le terme implémentation pour désigner cette dernière étape de raffinement ainsi que le composant produit. Dans le cadre de ce travail, le langage cible est **SQL**. Notre niveau d'implémentation est donc atteint lorsque nos spécifications ne font intervenir que des concepts directement traduisibles en **SQL**, i.e, des enregistrements et les opérations associées.

Nous appelons spécification concrète, une spécification directement implémentable en **SQL**. Cette spécification est contenue dans un composant implémentation **B**. En **B**, l'implémentation des données et des opérations peut être réalisée de deux manières différentes :

- Si les structures de données et de programmation du langage cible choisi sont exprimables en B0 [1], alors les variables et les opérations abstraites sont implémentées par des variables concrètes déclarées dans le composant implémentation lui même.
- Dans le cas contraire, un ensemble de machines abstraites de service sont importées par l'implémentation. Le code associé aux machines importées n'est pas généré automatiquement par l'AtelierB, mais écrit manuellement par l'utilisateur. Notre langage cible étant **SQL**, c'est cette solution qui sera mise en œuvre pour l'implémentation de notre dernier raffinement obtenu dans le chapitre précédent.

L'implémentation du dernier niveau de raffinement est réalisée via l'importation d'un ensemble de machines abstraites. Ces machines spécifient les opérations de manipulation d'enregistrements. Cette importation permet de récupérer non seulement le type de données mais aussi le code des opérations qu'elles spécifient. L'architecture globale de notre raffinement est représentée par la figure 5.1.

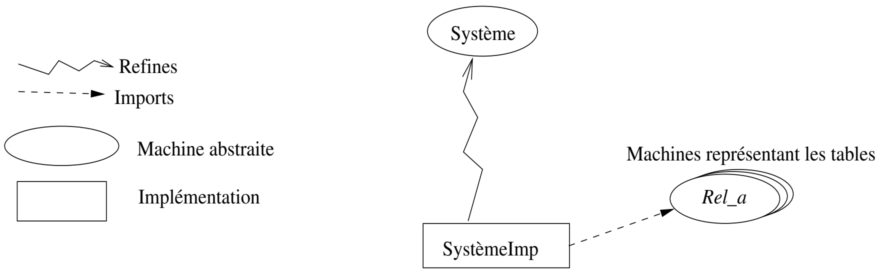


FIG. 5.1 – *Architecture globale du processus de raffinement*

1. $B0$ désigne le sous langage de **B** automatiquement traduisible en C, C++ ou ADA.

La suite de ce chapitre est organisée comme suit. Nous commençons par la présentation de la structure générique des machines définissant les tables. La réalisation des données et des opérations de base par celles des machines importées est décrite dans les sections 5.2 et 5.3 respectivement. Ensuite, nous décrivons l'implémentation des opérations correspondant aux diagrammes états/transitions, de collaborations ainsi que celles de la machine *RobusteSystème*. La traduction des différents composants terminaux vers le langage cible (**SQL**) est décrite dans la section 5.5. Cette traduction concerne aussi bien les machines importées que les composants implémentation. Les opérations de ces derniers comportant souvent des structures de contrôle (boucle, conditionnelle, ...), leur traduction est réalisée par des ordres **SQL** imbriqués dans le langage hôte **JAVA**. Le choix du langage **JAVA** est complètement arbitraire, un autre langage procédural (C, PASCAL, ...) aurait été possible.

5.1 Structure des machines importées

La structure de données utilisée dans le modèle relationnel est le "tuple". Un tuple désigne une valeur du produit cartésien d'un ensemble de types appelés domaines. On choisit de représenter ce tuple sous forme d'un enregistrement **B**. Chaque champ de cet enregistrement représente l'un des attributs du tuple. Pour chaque variable T_a représentant un ensemble de tuples du dernier niveau de raffinement, une machine abstraite *Rel_a* est importée. La machine permet d'une part d'implémenter cette variable et d'autre part de réaliser les opérations définies sur cette dernière.

Les types des attributs correspondant à des ensembles abstraits sont définis dans une machine dite de contexte. Cette machine est vue (lien **SEES**) par l'ensemble des machines importées *Rel_a*, et aussi par l'implémentation. Dans la machine importée *Rel_a*, sont spécifiées en particulier les opérations suivantes :

• L'ajout d'un tuple qui correspond à l'ajout d'un objet dans une classe ou une association. Les paramètres d'entrée de cette opération sont les valeurs des différents champs.

• Contrairement à l'opération d'ajout, plusieurs cas sont à distinguer pour l'opération de suppression.
  - **Suppression de tuples d'une table issue d'une classe :** les tuples à supprimer sont désignés par la valeur de leur attribut clé.
  - **Suppression de tuples d'une table issue d'une association:** comme nous l'avons vu dans le chapitre 3, deux types de suppression sont possibles: suppression d'un lien désigné par les deux objets concernés, ou suppression d'un ensemble de liens désignés par un ensemble d'objets de l'une des deux classes mises en œuvre par l'association.

• Mise à jour de la valeur d'un champ pour un tuple donné. Ce tuple est défini par la valeur de son attribut clé.

En plus de ces opérations, on spécifie dans la machine importée associée à une classe n'ayant pas initialement de clés une opération qui renvoie une valeur non encore attribuée de la clé . Cette opération correspond au terme $generer(TT_c\text{-}ran(v_{cl\acute{e}}))$ de la règle de raffinement relative à l'introduction de la clé.

Dans cette section, seule la machine implémentant une classe est présentée. Dans cette machine, on spécifie exclusivement les opérations d'ajout dans une classe, de mise à jour d'un attribut d'une classe et de la génération d'une valeur pour l'attribut clé. La spécification complète de cette machine ainsi que celle relative à une association sont données à l'annexe C.

MACHINE *Rel_a*
//la machine *Contexte* définit les ensembles abstraits correspondant aux attributs dont le type est un ensemble abstrait
SEES *Contexte*
DEFINITIONS

//f_i représente une des fonctions du produit cartésien associé à T_a.
TABLE $\widehat{=}$ **struct**(f_1: T_1, ..., f_n: T_n)
VARIABLES *Table_a*
INVARIANT
Table_a $\subseteq$ *TABLE*
INITIALISATION *Table_a* :=$\emptyset$
OPERATIONS
//opération d'ajout d'un objet dans une classe ou une association
Insert_a(val$_1$,...,val$_n$)=
PRE $val_1 \in T_1 \wedge \ldots \wedge val_n \in T_n$ **THEN**
$Table\_a := Table\_a \cup \{\mathbf{rec}(val_1, \ldots, val_n)\}$
END;

//opération de mise à jour d'un attribut f_i pour un tuple d'une classe défini par la valeur de sa clé f_1
Update_f$_i$(val$_1$, v)=
PRE $val_1 \in T_1 \wedge v \in T_i$ **THEN**
Table_a : (*Table_a* $\subseteq$ *TABLE* $\wedge$[2]
$\forall$ *xx*. (*xx* $\in$ *Table_a\$0*[3] $\wedge$ *xx'f*[4]$_1$ $\neq val_1 \Rightarrow$ *xx* $\in$ *Table_a*) $\wedge$
$\forall$ *xx*. (*xx* $\in$ *Table_a\$0* $\wedge$ *xx'f_1* = val_1) $\Rightarrow$ *rec*(*xx'f_1*,..., *xx'f_{i-1}*,*v*,...,*xx'f_n*) $\in$ *Table_a*) $\wedge$
$\forall$ *xx*.(*xx* $\in$ *Table_a* $\Rightarrow$ (*xx'f_1* $\neq val_1$ $\wedge$ *xx* $\in$ *Table_a\$0*) $\vee$ (*xx'f_1* = val_1 $\wedge$ *xx'f_i*=*v*)))
END;

//opération renvoyant une nouvelle valeur de l'attribut clé f_1
val $\longleftarrow$ NouvelleValeurClé=
BEGIN
val : (*val* $\in T_1 \wedge \neg(\exists$ *xx*.(*xx* $\in$ *Table_a* $\wedge$ *xx'f_1* =*val*)))
END
END

L'opération de mise à jour d'un champ est exprimée par une substitution “devient tel que”. Bien que cette substitution soit indéterministe par définition, son résultat dans ce cas précis est unique. Le premier invariant de la substitution permet de typer le résultat cherché, les deux suivants spécifient la façon dont sont transformés les éléments “avant” de $Table\_a$. Enfin, le dernier prédicat permet d'avoir, dans $Table\_a$, le même nombre d'éléments avant et après mise à jour. La non présence de ce prédicat n'interdit pas l'introduction de nouveaux éléments.

Remarquons que ces machines importées ont été spécifiées de manière la moins contraignante possible afin de permettre une réutilisation maximale de ces dernières. En effet, par exemple, on ne spécifie pas les contraintes de clé au sein de ces machines. Ces contraintes sont exprimées implicitement par un invariant de collage spécifié dans le composant implémentation (cf 5.2.3).

Exemple : la classe *Cassette* de la figure 5.2 est spécifiée en **B** par :

SETS *Cassette*
VARIABLES *cassette, NbLibre, Durée*
INVARIANT
cassette $\subseteq$ *Cassette* $\wedge$
NbLibre $\in$ *cassette* $\rightarrow$ *NAT* $\wedge$
Durée $\in$ *cassette* $\rightarrow$ *NAT*

2. $(v : P)$ affecte à la variable v une valeur satisfaisant le prédicat P.
3. v\$0 désigne la valeur avant de la variable v.
4. $y'f$ désigne la valeur du champ f pour l'enregistrement y

Le processus de raffinement du chapitre précédent permet de raffiner cette spécification abstraite en[5] :
VARIABLES *cassette,* $T_{cassette}$
INVARIANT
$T_{cassette}$ ∈ *cassette* ↣ $TT_{Cassette}$ × *NAT* × *NAT* ∧
$T_{cassette}$= *Num_Ca* ⊗ *NbLibre* ⊗ *Durée*

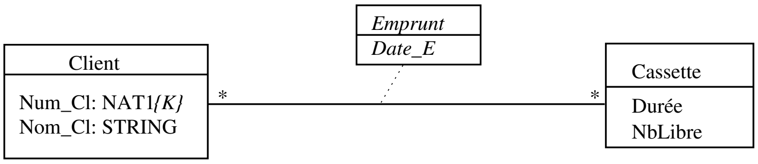


FIG. 5.2 – *Diagramme de classes UML pour la gestion de prêts de cassettes*

Afin d'implémenter la variable $T_{cassette}$, nous importons la machine *Rel_cassette* spécifiée comme suit :
MACHINE *Rel_cassette*
//la machine *Contexte* définit l'ensemble abstrait $TT_{Cassette}$
SEES *Contexte*
DEFINITIONS *TABLE* $\widehat{=}$ **struct**(*Num_Ca:* $TT_{Cassette}$*, NbLibre: NAT, Durée: NAT)*
VARIABLES *Table_cassette*
INVARIANT
Table_cassette ⊆ *TABLE*
INITIALISATION *Table_cassette* :=∅
OPERATIONS
//opération d'ajout d'une cassette
Insert_Cassette(nu, nb, du)=
PRE *nu* ∈ $TT_{Cassette}$ ∧ *nb* ∈ *NAT* ∧ *du* ∈ *NAT* **THEN**
$Table\_cassette$:=$Table\_cassette$ ∪ {**rec**(nu,nb,du)}
END;
...
END

5.2 Implémentation des données

Dans cette section est présentée la réalisation des variables abstraites du dernier raffinement par les structures de données définies dans les machines importées. Cette implémentation est faite par la valuation des ensembles abstraits et l'expression des variables abstraites en fonction des données concrètes des machines importées .

5.2.1 Valuation des ensembles abstraits.

Cette étape permet le passage effectif d'un modèle par objets (modèle **B**) vers un modèle par valeurs (modèle relationnel). Les objets devant être remplacés par les valeurs de leurs clés, chaque ensemble abstrait *C* associé à la variable *c* est donc valué par le type TT_C de la clé primaire définie sur cette variable.
VALUES C=TT_C

5. La variable Num_Ca est la clé ajoutée pour la classe *Cassette*

Suite à cette valuation, chaque typage par rapport à l'ensemble abstrait C est réécrit implicitement et automatiquement en typage par rapport à TT_C. En particulier, l'ensemble des objets c devient un sous ensemble de TT_C: $c \subseteq TT_C$.

Par exemple, l'ensemble abstrait *Cassette* de la spécification abstraite précédente est valué en :

VALUES *Cassette*$= TT_{Cassette}$ (* car le type de la clé *Num_Ca* est $TT_{Cassette}$*)

Cette valuation réécrit implicitement le typage de la variable *cassette* en : *cassette* $\subseteq TT_{Cassette}$.

5.2.2 Réalisation des variables abstraites

Chaque variable *Table_a*, de la machine importée *Rel_a*, réalise la variable T_a du dernier raffinement. La correspondance entre les éléments de *Table_a* et T_a est exprimée par un invariant de collage spécifié dans le composant implémentation. Dans ce qui suit, on suppose que le premier attribut de la variable T_a est la clé :

$$(T_a;\ prj_1(T_1 \times \ldots \times T_{n-1}, T_n);\ \ldots;\ prj_1(T_1, T_2)) = id(a) \tag{5.1}$$

$$\forall (x_1, \ldots, x_n).(x_1 \mapsto (x_1 \mapsto \ldots \mapsto x_n) \in T_a \Rightarrow rec(x_1, \ldots, x_n) \in Table\_a) \tag{5.2}$$

$$\forall (x_1, \ldots, x_n).rec(x_1, \ldots, x_n) \in Table\_a \Rightarrow (x_1 \mapsto (x_1 \mapsto \ldots \mapsto x_n) \in T_a) \tag{5.3}$$

Expliquons chaque partie de cet invariant:

- (5.1): cet invariant est équivalent à $a_{clé} = id(a)$. Ceci exprime la substitution de chaque objet de la variable a par la valeur de sa clé. Cet invariant n'est exprimé explicitement que pour les tables correspondant aux classes. Pour les associations, cet invariant devient implicite. En effet, si chaque objet est remplacé par la valeur de sa clé alors chaque lien entre objets est implicitement remplacé par un lien entre les valeurs correspondant à leurs clés.
- (H_{14}) et (5.3): ces deux invariants expriment le lien entre les éléments de T_a et ceux de *Table_a*. Ces deux invariants sont basés sur le premier. En effet grâce à ce premier invariant, dans chacun de ces deux invariants, seule la connaissance de l'objet ou de sa clé est nécessaire car ces deux quantités sont égales.

Dans la machine importée, on ne spécifie pas le fait que chaque enregistrement ait une clé. Cette propriété est exprimée implicitement par l'invariant de collage. En effet, la conjonction des deux derniers invariants de collage définit une sorte d'équivalence (bijection, isomorphisme) entre les deux variables liées. Donc, toute propriété exprimée sur T_a devient implicitement exprimée sur *Table_a*.

Exemple: l'implémentation de la variable $T_{cassette}$, du raffinement précédent, est exprimée par:

$$(T_{cassette};\ prj_1(TT_{Cassette} \times NAT, NAT);\ prj_1(TT_{Cassette}, NAT)) = id(cassette)$$
$$\forall (x_1, x_2, x_3).(x_1 \mapsto (x_1 \mapsto x_2 \mapsto x_3) \in T_{cassette} \Rightarrow rec(x_1, x_2, x_3) \in Table\_cassette)$$
$$\forall (x_1, x_2, x_3).rec(x_1, x_2, x_3) \in Table\_cassette \Rightarrow (x_1 \mapsto (x_1 \mapsto x_2 \mapsto x_3) \in T_{cassette})$$

5.2.3 Propagation implicite des propriétés

Dans la spécification abstraite, plusieurs contraintes peuvent être exprimées sur les données du système à développer. Ces contraintes, exprimées en **B** dans la clause **INVARIANT**, sont raffinées implicitement : elles sont présentes implicitement à tous les niveaux de raffinement. Ces contraintes exprimées sur les données abstraites n'ont pas besoin d'être explicitement réexprimées sur les données concrètes contenues dans les machines importées. En effet, toute contrainte sur les données abstraites est implicitement propagée sur les données concrètes. Par exemple, on peut facilement prouver que le champ correspondant à l'attribut clé de la variable T_a constitue aussi une clé pour la variable *Table_a*. En effet:

Supposons que l'attribut f_1 soit une clé de la variable T_a. On voudrait montrer que le champ f_1 est aussi une clé pour la variable $Table\_a$. Pour cela, soient x=$rec(a_1,\ldots,a_n)$ et y=$rec(b_1,\ldots,b_n)$ deux tuples de $Table\_a$ avec:

$$x \neq y \wedge a_1 = b_1$$

on a d'après l'invariant 5.3:

$$a_1 \mapsto (a_1,\ldots,a_n) \in T\_a \wedge b_1 \mapsto (b_1,\ldots,b_n\ ) \in T\_a$$

donc:

$$\bigwedge\nolimits_{i=1,n} (a_1 \mapsto a_i \in f_i) \wedge \\ \bigwedge\nolimits_{i=1,n} (b_1 \mapsto b_i \in f_i)$$

en remplaçant b_1 par a_1, on obtient:

$$\bigwedge\nolimits_{i=1,n} (a_1 \mapsto a_i \in f_i) \wedge \\ \bigwedge\nolimits_{i=1,n} (a_1 \mapsto b_i \in f_i)$$

or, chaque f_i est une fonction, donc:

$$\bigwedge\nolimits_{i=1,n} (a_i = b_i)$$

contradiction avec $x \neq y$.

5.3 Implémentation des opérations de base

Dans cette section est décrite l'implémentation des opérations de base, présentes dans la clause **DEFINITIONS** du dernier niveau de raffinement, via les opérations spécifiées dans les machines importées. Dans cette section, nous présentons l'implémentation des opérations les plus significatives. Des règles d'implémentation relatives aux autres opérations sont données à l'annexe C.

5.3.1 Substitution d'ajout

5.3.1.1 Ajout élémentaire

La substitution d'ajout d'un objet dans une classe ou une association a, représentée par la variable de raffinement T_a, est implémentée par l'opération *Insert_a* définie dans la machine importée *Rel_a*.

RULE *Implément_AjoutÉlem* **IS**
REFINES
    $a := a \cup \{a_1\}$;
    $T_a := T_a \cup \{a_1 \mapsto (val_1 \mapsto val_2 \mapsto \ldots \mapsto val_n)\}$
REFINEMENT
    $Insert\_a(val_1, val_2, \ldots, val_n)$
END

Par exemple, l'opération d'ajout d'une cassette spécifiée en **B** par :

BasicAjoutCassette(ca, nu, nb, du) $\widehat{=}$
BEGIN
    cassette := cassette $\cup$ *{ca}* $\|$
    Num_ca := Num_ca $\cup$ *{ca* $\mapsto$ *nu}* $\|$
    NbLibre := NbLibre $\cup$ *{ca* $\mapsto$ *nb}* $\|$
    Durée:= Durée $\cup$ *{ca* $\mapsto$ *du}*
END

et raffinée par :
BasicAjoutCassette(ca, nu, nb, du) $\widehat{=}$
BEGIN
cassette:= *cassette* $\cup$ *{ca}* ;
$T_{cassette}$:= $T_{cassette}$ $\cup$ *{ca* $\mapsto$ *(nu* $\mapsto$ *nb* $\mapsto$ *du)}*
END
est implémentée par :
BasicAjoutCassette(ca, nu, nb, du) $\widehat{=}$
BEGIN
Insert_Cassette(nu, nb, du) //opération de la machine *Rel_cassette* de la page 110
END

Remarque. La règle de raffinement précédente, prise séparément, est bien évidemment incorrecte. En effet, celle-ci est basée sur l'égalité de l'objet a_1 et de sa clé val_1. Or cette hypothèse ne peut exister au niveau le plus abstrait car ces deux quantités appartenant à des ensembles abstraits différents ne sont pas du même type. La preuve de cette règle doit être réalisée à travers la preuve du raffinement de l'opération où est définie la substitution abstraite. En effet, la signature de l'opération de base d'ajout d'un objet dans une classe contient l'objet c_1 à rajouter et éventuellement la valeur de l'attribut clé si la classe possède initialement une clé. Cette opération de base est appelée par une opération op d'interface qui contient une substitution indéterministe **ANY** qui permet de choisir l'objet c_1 parmi l'ensemble des objets pouvant être créés. Pour implémenter l'ensemble de ces deux opérations, deux cas sont à distinguer:

- **Cas d'une classe avec clé**: dans ce cas, les opérations d'interface et de base sont spécifiées comme suit:

BasicAjout(c_1 **,**v_1**, . . .)**$\widehat{=}$	**op(. . . , v**$_1$**)=**
PRE . . .	**PRE** . . . $v_1 \in TT_C$ - *ran(*$c_{clé}$*)*
THEN	**THEN**
. . .	**ANY** c_1 **WHERE** $c_1 \in$ *C* - *c* **THEN**
. . .	*BasicAjout(*c_1 *,*v_1*, . . .)*
. . .	. . .
. . .	**END**
END	**END**

où :

- v_1: désigne la valeur de la clé $c_{clé}$ pour l'objet c_1,
- TT_C: désigne le type de la clé $c_{clé}$,
- C: désigne l'ensemble abstrait associé à la classe.

Comme nous l'avons noté précédemment, l'idée principale de l'implémentation consiste à identifier les objets à leurs clés. Pour cela, la substitution **ANY** est implémentée en choisissant comme valeur pour l'objet c_1 la valeur v_1. Cette implémentation est correcte, car en effet v_1 satisfait bien le prédicat de la substitution **ANY** :

$$v_1 \in TT_C - ran(c_{clé})$$
$$\Rightarrow \quad v_1 \in C - c \qquad \text{(* car l'ensemble abstrait } C \text{ est valué à } TT_C \text{ et } c_{clé} = id(c)\text{*)}$$

L'implémentation de l'opération op est réalisée comme suit :

op(. . . , v_1**)**
BEGIN
BasicAjout(v_1*,* v_1*, . . .)*
. . .
END

- **Cas d'une classe sans clé**: dans ce cas, les opérations d'interface et de base sont spécifiées comme suit:

BasicAjout($c_1, \ldots$)$\widehat{=}$
PRE $\ldots$
THEN
$\ldots$
$\ldots$
$\ldots$
$\ldots$
END

op($\ldots$)=
PRE $\ldots$
THEN
ANY c_1 **WHERE** $c_1 \in C$ - c **THEN**
BasicAjout($c_1, \ldots$)
$\ldots$
END
END

À l'étape d'identification des objets, une clé $c_{clé}$ est rajoutée à la classe c. L'opération de base est donc raffinée par :

BasicAjout($c_1, \ldots$)$\widehat{=}$
PRE $\ldots$
THEN
VAR v_1 **IN**
$v_1 \longleftarrow$ *generer*(TT_C - *ran*($c_{clé}$));
$c_{clé}$:= $c_{clé} \cup \{c_1 \mapsto v_1\}$;
$\ldots$
END
END

On remarque la présence de deux substitutions indéterministes séléctionnant deux valeurs devant être égales. Pour cela, on éliminera le deuxième choix en prenant la même valeur que celle choisie par la première. L'implémentation est donc réalisée comme suit:

BasicAjout($c_1, \ldots$)$\widehat{=}$
BEGIN
$c_{clé}$:= $c_{clé} \cup \{c_1 \mapsto c_1\}$;
$\ldots$
$\ldots$
$\ldots$
$\ldots$
$\ldots$
END

op($\ldots$)=
BEGIN
VAR *rancle*, c_1 **IN**
rancle $\longleftarrow$ *SelectClé_c*;
$c_1 \longleftarrow$ *generer*(TT_C - *rancle*);
BasicAjout($c_1, \ldots$)
$\ldots$
END
END

5.3.1.2 Ajout ensembliste

La substitution d'ajout d'un ensemble de tuples se ramène au cas d'un ajout élémentaire en introduisant une boucle. L'ajout ensembliste apparaît dans le cas du raffinement d'une séquence et aussi d'un attribut multivalué non borné. Nous présentons en détails le cas de l'attribut multivalué, le cas de la séquence est donné à l'annexe C.

L'implémentation d'ajout dans un attribut multivalué, de type TT_{Att}, est réalisée comme suit :

RULE *Implément_AttMulti_Ajout* **IS**
REFINES
$Att := Att \cup \{cc\} \times val;$
$T_{Att} := T_{Att} \cup ((\{cc\} \times val \times \{c_{clé}(cc)\}) \otimes prj_2(\{cc\}, val))$
REFINEMENT
VAR xx **IN**
xx:=val;
WHILE $(xx \neq \emptyset)$ **DO**
VAR uu **IN**
uu $\longleftarrow$ *generer(xx)*;
Insert_Att(cc,uu);
xx:=xx-{uu}
END
INVARIANT
$xx \subseteq val \wedge$
$Table\_Att \subseteq struct(Att_c\!: T_{c_{clé}}, Att_{val}\!:TT_{Att}) \wedge$
$\forall (xx,yy).((xx \mapsto yy) \mapsto (xx \mapsto yy) \in T_{Att} \cup id(\{cc\} \times (val\text{-}xx)) \Rightarrow rec(xx,yy) \in Table\_Att) \wedge$
$\forall (yy,zz).(yy \in T_{c_{clé}} \wedge zz \in TT_{Att} \wedge rec(Att_c\!:yy, Att_{val}\!:zz) \in Table\_Att \Rightarrow$
$(yy \mapsto zz) \mapsto (yy \mapsto zz) \in T_{Att} \cup id(\{cc\} \times (val\text{-}xx))) \wedge$
$\forall yy.(yy \in val\text{-}xx \Rightarrow rec(Att_c\!: cc, Att_{val}\!: yy) \in Table\_Att)$
VARIANT $card(xx)$
END
END

L'idée intuitive de ce raffinement est de parcourir l'ensemble de valeurs *val* ($xx := val$ et $xx := xx - \{uu\}$) jusqu'à son épuisement (condition d'arrêt: $xx = \emptyset$). À chaque pas de ce parcours, on insère un nouveau tuple $Insert\_Att(cc,uu)$. Ceci est exprimé formellement par le dernier prédicat de l'invariant de la boucle. Les deuxième et troisième invariants expriment la relation existant entre la variable abstraite T_{Att} et la variable concrète $Table\_Att$.

5.3.2 Substitution de suppression

Bien que les langages **B** et **SQL** supportent les opérations ensemblistes comme la suppression d'un ensemble de tuples, nous implémentons ces opérations par une suite de suppressions élémentaires. En effet, la plupart des langages hôtes d'implémentation ne supportent pas la notion d'ensembles. Comme pour la substitution d'ajout, la substitution de suppression est implémentée par l'opération *Delete* de la machine importée correspondant à la variable. Nous distinguons quatre règles de suppression: suppression d'un ensemble d'objets d'une classe, suppression d'un lien d'une association, suppression d'un ensemble de liens d'une association désignés par des objets de la classe source et enfin suppression d'un ensemble de liens d'une association désignés par des objets de la classe cible. Nous commençons par donner la règle de suppression d'un lien (la plus simple: suppression élémentaire), puis celle correspondant à la suppression d'un ensemble d'objets d'une classe. Par la suite, nous montrons comment les opérations de suppression d'un ensemble de liens sont obtenues à partir de cette dernière.

• **Suppression d'un lien d'une association**

RULE *Implément_Supp_Ass* **IS**
REFINES
$Ass := Ass - \{a_1 \mapsto a_2\}$;
$T_{Ass} := \{a_1 \mapsto a_2\} \lhd T_{Ass}$
REFINEMENT
Delete_Ass(a_1, a_2)
END

• **Suppression d'un ensemble d'objets d'une classe:** l'idée d'implémentation de cette suppression ensembliste est similaire à celle de l'ajout ensembliste pour le cas de l'attribut multivalué. Il s'agit de parcourir, élément par élément, l'ensemble des tuples à supprimer jusqu'à son épuisement. À chaque pas, on supprime un tuple particulier.

RULE *Implément_Supp* **IS**
REFINES
$a := a - a_1$;
$\underline{T_a := a_1 \lhd T_a}$
REFINEMENT
VAR *xx* **IN**
$xx := a_1$;
WHILE $(x \neq \emptyset)$ **DO**
VAR *yy* **IN**
yy $\longleftarrow$ *generer(xx)*;
xx:=xx-yy;
<u>*Delete_a (yy)*</u>
END
INVARIANT
$xx \subseteq a_1$
Table_a $\subseteq$ *struct*(f_1:T_1,...,f_n:T_n)
$\forall(yy_1,\ldots,yy_n).(yy_1 \mapsto (yy_1 \mapsto \ldots \mapsto yy_n) \in \underline{(a_1 - xx) \lhd T_a} \Rightarrow rec(yy_1,\ldots,y_n) \in Table\_a) \wedge$
$\forall(yy_1,\ldots,yy_n).(yy_1 \in T_1 \wedge \ldots \wedge yy_n \in T_n \wedge rec(f_1 : yy_1,\ldots,f_n : yy_n) \in Table\_a \Rightarrow$
$yy_1 \mapsto (yy_1 \mapsto yy_n) \in \underline{(a_1 - nn) \lhd T_a}) \wedge$
$\forall(yy_1,\ldots,yy_n).(yy_1 \in T_1 \wedge \ldots \wedge yy_n \in T_n \wedge rec(f_1 : yy_1,\ldots,f_n : yy_n) \in Table\_a \Rightarrow yy \notin (a_1 - xx))$
VARIANT *card(xx)*
END
END
END

• **Suppression d'un ensemble de liens:** la règle de raffinement de la substitution de suppression d'un ensemble de liens d'une association désignés par l'ensemble des objets de la classe source (resp. cible) peut être obtenue à partir de la règle de suppression d'un ensemble d'objets d'une classe. En effet, il suffit de remplacer les parties soulignées comme suit:

- la substitution $a := a - a_1$ est remplacée par $ass := a_1 \lhd ass$ (resp. $ass := ass \rhd b_1$).
- la substitution $T_a := a_1 \lhd T_a$ est remplacée par $T_{ass} := (a_1 \times b) \lhd T_{ass}$ (resp. $T_{ass} := (a \times b_1) \lhd T_{ass}$).
- la substitution $xx := a_1$ reste inchangée (resp. est remplacée par $xx := b_1$).
- la substitution *Delete_a* est remplacée par *Delete_ass_a* (resp. *Delete_ass_b*).
- le terme $((a_1 - xx) \lhd T_a)$ est remplacé par $((a_1 - xx) \times b) \lhd T_a)$ (resp. $(a \times (b_1 - xx) \lhd T_a)$).
- le terme $((a_1 - nn) \lhd T_a)$ est remplacé par $((a_1 - nn) \times b) \lhd T_a)$ (resp. $(a \times (b_1 - nn) \lhd T_a)$).

5.4 Implémentation des opérations complexes

Dans cette section est décrite l'implémentation des opérations des diagrammes états/transitions, de collaborations et enfin celles de la machine *RobusteSystème* . Contrairement aux opérations de base, l'implémentation de ces opérations n'est pas complètement automatique. Seules des squelettes d'opérations, devant être complétés manuellement par l'utilisateur, peuvent être générés.

5.4.1 Implémentation des opérations des diagrammes états/transitions

Dans cette partie, on donne des indications sur la façon dont les opérations générées à partir des diagrammes états/transitions sont implémentées. On rappelle que la structure générale d'une telle opération est la suivante :

$$
\begin{array}{l}
\textbf{op(...)} \,\widehat{=} \\
\textbf{BEGIN} \\
\quad \textbf{SELECT}\ cond_1\ \textbf{THEN}\ Act_1(\ldots) \\
\quad \textbf{WHEN}\ cond_2\ \textbf{THEN}\ Act_2(\ldots) \\
\qquad\qquad \ldots \\
\quad \textbf{WHEN}\ cond_n\ \textbf{THEN}\ Act_n(\ldots) \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

où les $cond_i$ et Act_i désignent des prédicats et des opérations de base respectivement. Implémenter une telle opération revient à raffiner la substitution **SELECT**, mais aussi les différents prédicats $cond_i$. En effet, ces prédicats dépendent des variables du dernier niveau de raffinement qui ne sont plus visibles du fait qu'elles soient raffinées par les variables concrètes des machines importées. Nous devons donc les exprimer en fonction de ces variables concrètes. Cependant, les variables concrètes, définies dans les machines importées, ne sont pas visibles directement à partir de l'implémentation. Pour cela, une opération de test de chaque condition $cond_i$ doit être spécifiée. Par manque de temps, nous n'avons pas étudié l'automatisation de cette spécification qui nous semble demander une réflexion importante sur une éventuelle décomposition de chaque condition $cond_i$. En effet, une condition $cond_i$ peut faire référence à plusieurs variables n'appartenant nécessairement pas à la même machine importée. De ce fait, le test d'une condition peut nécessiter l'utilisation (appel) d'opérations de machines différentes. Néanmoins, les applications base de données étant génériques, on génère automatiquement, pour chaque machine Rel_a un ensemble d'opérations qui constitueront une aide pour le concepteur. Par exemple, on génère:

- une opération qui teste l'existence d'un objet (enregistrement) désigné par sa clé,
- pour chaque machine Rel_a, représentant une association, une opération qui renvoie l'ensemble des liens (enregistrements) associés à un objet désigné par sa clé.
- ....

Comme nous l'avons noté au chapitre 3, nous avons exigé que les prédicats $cond_i$ des substitutions **SELECT** soient deux à deux disjoints. Par conséquent, la substitution **SELECT** est raffinée en une substitution conditionnée **IF**. Chaque branche de la substitution **SELECT** :

$$\textbf{SELECT}\ cond_i\ \textbf{THEN}\ Act_i(\ldots)\ \textbf{WHEN}\ \ldots\textbf{END}$$

est implémentée par :

$$
\begin{array}{l}
\textbf{VAR}\ x_i,\ x_i^1,\ldots,\ x_i^m\ \textbf{IN} \\
\quad x_i^1 \longleftarrow op_i^1(\ldots); \\
\qquad \ldots \\
\quad x_i^m \longleftarrow op_i^m(\ldots); \\
\quad x_i \longleftarrow cond_i(x_i^1,\ \ldots,\ x_i^m); \\
\quad \textbf{IF}\ x_i\ \textbf{THEN}\quad Act_i \\
\quad \textbf{ELSE}\ \ldots \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

où:

- op_j^i: désignent des opérations spécifiées dans les machines importées.

– $cond_i(y_1, \ldots y_n)$: désigne une formule logique dépendant des variables y_i de même valeur booléenne que $cond_i$.

5.4.2 Implémentation des opérations du système: Transactions utilisateur

Dans cette partie, on décrit l'implémentation des opérations générées à partir des diagrammes de collaborations. On rappelle que ces opérations sont les opérations de la machine expansée. Comme pour les opérations des diagrammes états/transitions, l'implémentation de ces opérations n'est pas complètement automatique. En effet, ces opérations contiennent aussi des prédicats dépendant de variables qui ne sont plus visibles au niveau de l'implémentation. Deux formes d'opérations sont à distinguer :

m(m-param-list)
PRE
  Typage des paramètres d'entrée ∧ *C*
THEN
  IF *C1* **THEN** *m1(m1-param-list)* **END** ||
  IF *C2* **THEN** *m2(m2-param-list)* **END**
END

m(m-param-list)
PRE
  Typage des paramètres d'entrée ∧ *C*
THEN
  ANY *cc* **WHERE** *Pred* **THEN**
    IF *C1* **THEN** *m1(m1-param-list)* **END** ||
    IF *C2* **THEN** *m2(m2-param-list)* **END**
  END

L'implémentation de ces opérations consiste à supprimer les préconditions et à raffiner les substitutions **ANY** et parallèles (||). Le raffinement de la substitution **ANY** revient à spécifier une opération qui renvoie un objet cc vérifiant le prédicat $Pred$. À la page 113, nous avons présenté un cas particulier d'implémentation d'une telle substitution : cc désigne un nouvel objet d'une classe sans conditions supplémentaires. Cependant dans le cas général, l'implémentation d'une telle substitution nécessite l'intervention de l'utilisateur. De même, la substitution parallèle ne peut être implémentée automatiquement. Le seul cas où cela serait possible (passage automatique à la séquence) est quand les ensembles de variables modifiées par la première substitution et lues par la deuxième sont disjoints. Dans le cas contraire, l'implémentation nécessite l'intervention de l'utilisateur.

5.4.3 Implémentation des opérations de la machine *RobusteSystème*

Comme pour les opérations de la machine expansée, l'implémentation des opérations de la machine *RobusteSystème* n'est pas entièrement automatique. Seuls des squelettes d'opérations peuvent être générés. L'implémentation de cette machine importe la machine abstraite *Système_exp*. Un invariant de collage portant sur l'égalité de chaque variable v, de la spécification abstraite, et sa variable fraîche correspondante v_1, de la machine expansée *Système_exp*, est inclus dans la clause **INVARIANT**. On rappelle qu'une opération de la machine *RobusteSystème* est de structure :

$resultat \longleftarrow$ **Robusteop($\mathbf{x}_1, \ldots, \mathbf{x}_n$)=**
PRE $\wedge_{i=1..n}$ *cond_typage(x_i)*
THEN
  IF $Precond_1$ **THEN**
    IF $Precond_2$ **THEN**
        …
      IF $Precond_k$ **THEN**
          $op(x_1, \ldots, x_n)$ || *resultat:=OK*
      ELSE *resultat:=NonPrecond$_k$*
      END
        …
    ELSE *resultat:=NonPrecond$_2$*
    END
  ELSE *resultat:=NonPrecond$_1$*
  END
END

Pour implémenter cette opération, on remplacera la substitution parallèle par une substitution séquentielle. Chaque substitution **IF** de la forme :

IF $Precond_i$ **THEN** … **END**

est implémentée par :

VAR x_i **IN**
  $x_i \longleftarrow op_i(Precond_i)$;
  IF x_i **THEN**
    $\dots$
  END
END

Les termes $op_i(Precond_i)$ désignent des appels à des opérations de la machine *Système_exp*. L'opération $op(Precond_i)$ renvoie la valeur booléenne de l'évaluation de la condition $Precond_i$. Un exemple de l'implémentation de cette machine est donnée à l'annexe D.

La figure 5.3 illustre l'architecture générale du développement d'applications base de données sûres.

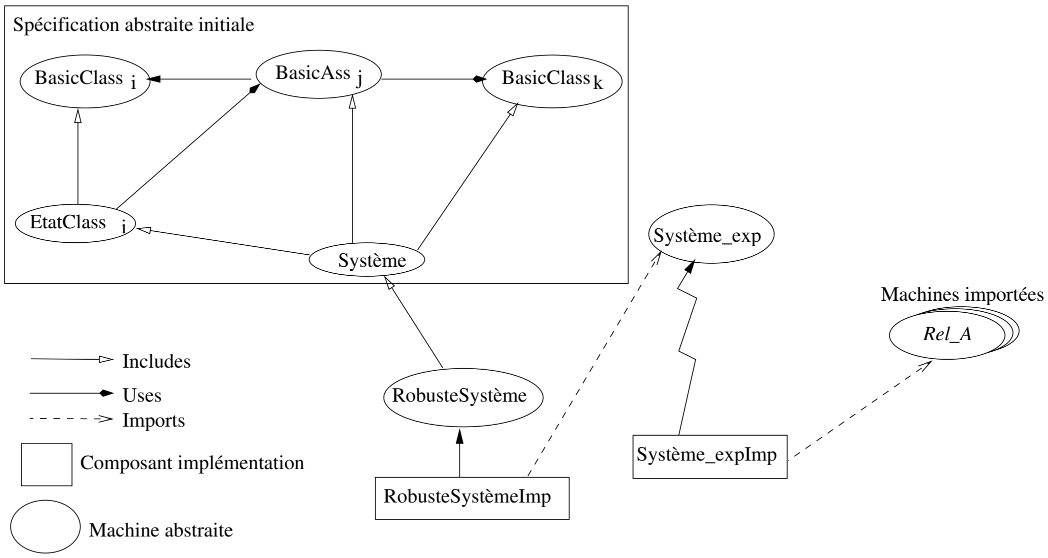


FIG. 5.3 – *Architecture B de développement d'applications sûres*

La machine abstraite *Système_exp* désigne la machine expansée équivalente à la spécification abstraite initiale.

5.5 Traduction en SQL des machines B importées

Le codage du composant implémentation repose sur la traduction des machines importées et celle des différents opérateurs de substitutions. Dans cette partie, on décrit la traduction des machines importées. Cette phase de codage dépend fortement du système de gestion de base de données (SGBD) choisi. Un même concept peut être traduit différemment selon la version du langage **SQL** choisie mais aussi selon le SGBD utilisé. Dans ce qui suit, nous utilisons le SGBD *Postgres* version 7.1.3 implémentant le standard SQL92.

5.5.1 Aspect statique

L'aspect statique comporte la définition des structures des tables ainsi que les différentes contraintes. Les schémas des tables sont obtenus par traduction de la partie statique des machines importées. Certaines contraintes sont déduites par traduction des invariants de la spécification la plus abstraite, d'autres sont générées à partir des différentes étapes du processus de raffinement.

Schéma d'une table. Chaque machine importée *Rel_a* définit une variable abstraite *Table_a* de type:

$$Table\_a \subseteq \mathbf{struct}(f_1{:}T_1,\ldots,f_n{:}\ T_n)$$

où les T_i représentent des types de base, abstraits ou des ensembles énumérés. Cette variable est traduite en **SQL** comme suit :

– Si T_i correspond à un type de base alors ce dernier doit être supporté par **SQL**. Par exemple, les types *NAT* et *STRING* sont traduits respectivement par les types *INT* et *CHARACTER VARYING(n)* de **SQL**. le nombre n désigne la longueur maximale de la chaîne. Cette valeur peut être choisie par concertation avec l'utilisateur. Cette contrainte ajoutée par l'utilisateur lors de la traduction vers **SQL** nécessite l'ajout de conditions de test supplémentaires lors de la mise à jour de la valeur de cet attribut. Actuellement, ces contraintes ne sont pas exprimables en **B**. Les types énumérés sont traduits par des domaines énumérés **SQL**. Si T_i désigne un ensemble abstrait alors ce dernier est traduit par l'utilisateur en un type de base de **SQL**.

– La variable $Table\_a$ est traduite par une table (relation) A. Les colonnes de cette table sont les champs $f_1,\ldots,f_n$. Le domaine de l'attribut f_i est la traduction **SQL** de T_i.

Il est important de noter que la traduction des machines importées représentant les différentes tables ne permet l'obtention que des structures des tables: les attributs et leurs domaines. Aucune contrainte d'intégrité n'est générée. En effet, ces dernières ne sont pas spécifiées explicitement dans ces machines. La génération de ces contraintes est faite à partir des invariants introduits au cours des étapes de raffinement. Dans ce chapitre, on présente les contraintes les plus fréquentes. Une description complète de la génération de l'ensemble des contraintes est donnée à l'annexe C.

Contraintes sur les attributs. À la phase précédente, seule la contrainte de domaine a été définie. Cette phase permet de définir l'ensemble des contraintes qui peuvent être exprimées sur un (ou un ensemble d') attribut(s).

Clé. Chaque attribut[6] défini comme une fonction totale injective sur une variable a constitue une clé candidate pour la table A. La clé primaire (**PRIMARY KEY**) d'une table correspondant à une classe est la fonction injective retenue dans la phase d'identification des objets. Les attributs correspondant à des fonctions injectives sont contraints par **UNIQUE**. Les attributs correspondant à des fonctions totales (avant l'étape d'extension des fonctions partielles) sont contraints par **NOT NULL**.

Par exemple, la variable concrète *Table_Cassette* est traduite en la table **SQL** *Cassette* définie par :

CREATE TABLE *Cassette*(
Num_Ca **INT PRIMARY KEY**,//on suppose avoir choisi **INT** comme traduction **SQL** de $TT_{Cassette}$
NbLibre **INT NOT NULL**,
Durée **INT NOT NULL**)

Contraintes référentielles. Les contraintes référentielles traduisent les associations en UML. Ces contraintes sont définies lors du raffinement des associations (mono ou muti valuées) et aussi lors du raffinement des attributs multi-valués.

- **contrainte issue d'une association monovaluée:** on rappelle qu'une association monovaluée Ass définie sur les classes C et D est raffinée par ($T^D_{clé}$ est le type de la clé primaire sur D):

$$Ass\_Mig \in c \nrightarrow T^D_{clé} \wedge$$
$$Ass\_Mig = (Ass;\ d_{clé})$$

Le deuxième invariant de collage définit une contrainte référentielle de Ass_Mig, attribut de la table correspondant à c, vers la clé $d_{clé}$ de la table correspondant à d.

- **contrainte issue d'une association multivaluée:** on rappelle qu'une association multivaluée Ass définie sur les classes C et D est raffinée par ($T^D_{clé}$ et $T^D_{clé}$ sont respectivement les types des clés primaires de C et D):

$$Ass_c = Ass \lhd (prj_1(c,d);\ c_{clé}) \wedge$$
$$Ass_d = Ass \lhd (prj_2(c,d);\ d_{clé}) \wedge$$
$$(Ass_c \otimes Ass_d) \in Ass \rightarrowtail T^C_{clé} \times T^D_{clé}$$

6. C'est la fonction correspondant à l'attribut qui est une fonction.

La variable Ass est traduite par une table ayant, entre autres, deux attributs Ass_c et Ass_d. Les deux premiers invariants de collage définissent deux contraintes référentielles de Ass_c et Ass_d vers les clés $c_{clé}$ et $d_{clé}$ des tables correspondant à c et d respectivement. Le dernier invariant spécifie que le couple d'attributs (Ass_c, Ass_d) forme une clé pour cette table Ass.

L'association *Emprunt* de la page 110 spécifiée en **B** par:

$$Emprunt \in client \leftrightarrow cassette$$

est raffinée par:

$$\begin{array}{c} Emprunt_{client} = Emprunt \lhd (prj_1(client, cassette); Num\_Cl) \wedge \\ Emprunt_{cassette} = Emprunt \lhd (prj_2(client, cassette); Num\_Ca) \wedge \\ (Emprunt_{client} \otimes Emprunt_{cassette}) \in Emprunt \rightarrowtail NAT \times NAT \end{array}$$

La variable *Emprunt* sera implémentée par une variable *Table_Emprunt* contenant trois champs: $Emprunt_{cassette}$, $Emprunt_{client}$ et *Durée_E*. La variable *client* est aussi implémentée par une variable *Table_Client* contenant entre autres un champ *Num_Cl*. Ces deux variables sont traduites en deux tables *Emprunt* et *Client*. Les attributs $Emprunt_{cassette}$ et $Emprunt_{client}$, constituant une clé primaire pour la table *Emprunt*, sont liés par deux contraintes référentielles aux clés *Num_Ca* et *Num_Cl* respectivement.

CREATE TABLE *Client*(
Num_Cl **INT PRIMARY KEY**,
Nom_Cl **CHARACTER VARYING(20) NOT NULL**)

CREATE TABLE *emprunt*(
$Emprunt_{cassette}$ **INT REFERENCES** *cassette*,
$Emprunt_{client}$, **INT REFERENCES** *client*,
Durée_E **INT NOT NULL**,
PRIMARY KEY
($Emprunt_{cassette}$, $Emprunt_{client}$))

- **contrainte issue d'un attribut multivalué:** ce cas est similaire au cas précédent. Dans ce cas, une seule contrainte est générée car au deuxième attribut ne correspond pas de table.

5.5.2 Aspect dynamique

Les opérations spécifiées dans les machines importées correspondent aux opérations de base. Elles concernent l'ajout et la suppression d'enregistrements (ou tuples), la mise à jour d'un champ d'un enregistrement désigné par la valeur de sa clé. La traduction de l'opération de génération pour un nouvel enregistrement à créer d'une valeur non encore attribuée de la clé est donnée à l'annexe C.

5.5.2.1 Insertion d'un tuple

L'insertion d'un nouvel enregistrement dans une variable *Table_a* (*Table_a* $\subseteq$ **struct**(f_1:T_1,..., f_n: T_n)) consiste à affecter une valeur, du domaine de l'attribut, à chaque champ de l'enregistrement. En **B**, cette opération est spécifiée comme suit:

$$Table\_a := Table\_a \cup \{rec(v_1, \ldots, v_n)\}$$

et est traduite par l'ordre d'insertion **SQL** :

INSERT INTO *Table_a* **VALUES** $v_1, \ldots, v_n$

Parmi les valeurs v_i, certaines sont égales à la valeur nulle de leurs types. Celles-ci sont donc traduites par la valeur *NULL* de **SQL**. Ces valeurs correspondent aux attributs dont la valuation est optionnelle.

5.5.2.2 Suppression d'un tuple d'une classe

La suppression d'un enregistrement d'une variable *Table_a* représentant une classe (*Table_a* $\subseteq$ **struct**(f_1:T_1,..., f_n: T_n)) est réalisée en fournissant la valeur v_1 de l'attribut clé f_1 du tuple à supprimer. En **B**, cette suppression est exprimée par :

LET *ens_tuples* **BE**
$ens\_tuples = \bigcup xx.(xx \in Table\_a \wedge f_1 = v_1 \mid \{xx\})$
IN
Table_a :=*Table_a* -*ens_tuples*
END

et est traduite par l'ordre de suppression **SQL** :

DELETE FROM *Table_a* **WHERE** $f_1 = v_1$

5.5.2.3 Mise à jour d'un attribut d'un tuple

La mise à jour d'un attribut f_j pour un tuple d'une variable *Table_a* (*Table_a* $\subseteq$ **struct**$(f_1:T_1,\ldots,f_n:T_n)$) consiste à affecter une nouvelle valeur v à l'attribut f_j de ce tuple. Comme pour l'opération de suppression, le tuple est désigné par la valeur de l'attribut clé . En **B**, cette opération est spécifiée comme suit[7] :

LET *r* **BE**

$r = \bigcup xx.(xx \in Table\_a \wedge f_1 \neq v_1 \mid \{xx\}) \cup$
$\bigcup xx.(xx \in Table\_a \wedge f_1 = v_1 \mid$
$\{rec(xx'f_1,\ldots,xx'f_{j-1},v,xx'f_{j+1},\ldots,xx'f_n)\})$

IN

$Table\_a := r$

END

et est traduite par l'ordre de mise à jours **SQL** :

UPDATE *Table_a* **SET** $f_j = v$ **WHERE** $f_1 = v_1$

5.6 Codage de l'application en SQL

Dans cette section, nous décrivons la traduction (Codage) des différents composants terminaux **B** : implémentation et machines définissant les structures. Cette traduction est réalisée en **SQL** imbriqué dans le langage hôte **JAVA**. Il ne s'agit pas de proposer des règles formelles de traduction, mais de montrer la faisabilité de ce processus. Une proposition plus formelle nécessite une étude comparative approfondie des langages **B** et **JAVA**. On donnera l'architecture globale de cette traduction. Un exemple détaillé illustrant nos propositions est donné à l'annexe D.

L'architecture du logiciel, obtenu lors de la phase de codage, est composée de deux niveaux :

• Niveau table: ce niveau est constitué des tables définies précédemment ainsi que des contraintes générées lors de la traduction vers **SQL**.

• Niveau traitement: ce niveau contient le code associé à chaque opération de base d'une table donnée, ainsi que les traitements définis dans notre système. Il se divise en quatre couches:

1. **Couche table**: on associe à chaque machine *Rel_a* une classe **JAVA** *A* permettant l'accès à la table *A* créée précédemment. Les opérations de cette machine sont traduites par des méthodes.
2. **Couche intermédiaire** : ce niveau contient un ensemble de classes **JAVA** *ÉtatA*. Chaque classe correspond à une des machines **B** traduisant les diagrammes états/transitions. Les méthodes de cette classe sont les opérations **B** générées du diagramme états/transitions considéré.
3. **Couche application**(*Système_expImp*) : ce niveau contient la classe **JAVA** *Système_expImp* représentant le codage du composant *Système_expImp*. Chaque opération de ce composant est traduite par une méthode faisant appel aux méthodes décrites dans les classes précédentes.
4. **Couche de sécurité** (*RobusteSystèmeImp*) : ce niveau contient la classe **JAVA** *RobusteSystèmeImp* représentant le codage du composant *RobusteSystèmeImp*. Chaque opération de ce composant est traduite par une méthode faisant appel aux méthodes décrites dans la classe *Système_expImp*.

5.7 Conclusion

Dans ce chapitre, nous avons présenté l'implémentation **B** et la traduction **SQL** du dernier niveau de raffinement obtenu par l'application successive des différentes règles de raffinement proposées au chapitre précédent. L'implémentation a été réalisée via l'importation d'un ensemble de machines abstraites dites de service. Ces machines décrivent des

7. La substitution avec le **LET** est équivalente à celle utilisée à la page 109

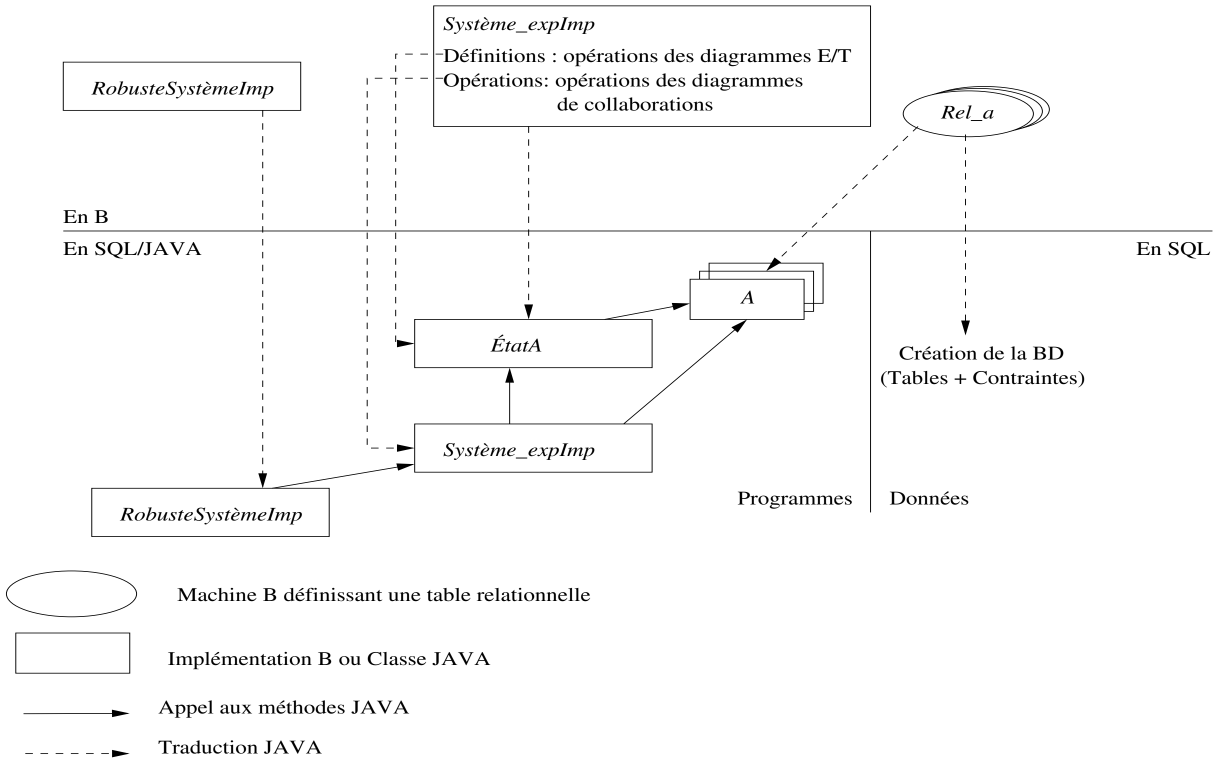


FIG. 5.4 – *Architecture du code généré*

opérations de manipulation d'enregistrements représentant une des formalisations **B** possibles du modèle relationnel. Les différentes règles d'implémentation présentées ont été prouvées en utilisant l'AtelierB. Un processus de traduction de l'implémentation ainsi obtenue et des machines importées a été également décrit. La traduction a été réalisée en langage **SQL** imbriqué dans le langage hôte **JAVA**.

Lors du processus de traduction, seules les contraintes de clés et référentielles, ajoutées au cours du processus de raffinement, sont traduites automatiquement. En effet, ces contraintes sont facilement identifiées puisqu'elles correspondent aux invariants de collage. Les autres contraintes d'intégrité définies dans le système (dans la machine *Système_exp*) existent implicitement dans le composant "Implémentation". Leur raffinement est automatique et invisible à l'utilisateur. Pour pouvoir les traduire en **SQL**, il suffit de les réécrire explicitement, lors de chaque phase de raffinement, en fonction des variables du composant raffinant. Bien que cette réécriture ne puisse pas être automatisée, néanmoins, l'établissement de sa correction est possible. En effet, pour chaque réécriture $R(CI)$ de la contrainte CI, il suffit de prouver l'équivalence des deux expressions en ajoutant la conjonction ci-dessous à l'invariant du composant raffinant.

$$(R(CI) \Rightarrow CI) \wedge (CI \Rightarrow R(CI))$$

L'inclusion des contraintes dans la définition du schéma de la base de données n'est pas indispensable. En effet, les programmes générés prennent déjà en compte ces contraintes. Ces dernières dont la vérification par le SGBD consomme un temps non négligeable peuvent donc être désactivées.

Il est important de noter que la phase de codage vers **JAVA** est faite en dehors de l'AtelierB. Ceci ne garantit donc pas la correction du code obtenu. En effet, une règle de traduction fausse engendrerait un code incorrect. Pour cela, l'outil de traduction devrait être validé à son tour. Cette validation ne rentre pas dans le cadre de cette thèse.

Chapitre 6

Réutilisation des Preuves

Dans le chapitre précédent, un ensemble de règles de raffinement a été proposé pour les substitutions de base. Pour chaque règle, nous avons établi une preuve permettant d'affirmer la correction des transformations opérées par la règle. En pratique, les fonctionnalités d'un système sont décrites par des substitutions plus complexes (élaborées). Une substitution complexe désigne la combinaison des substitutions de base via les divers opérateurs de substitutions (parallèle, séquence, conditionnelle,...). Le processus de raffinement **B** étant monotone, le raffinement de substitutions complexes se ramène toujours au raffinement des substitutions qui les composent. De ce fait, une question se pose naturellement: le raffinement étant monotone, en est-il de même pour le processus de preuve? Plus précisément, peut-on réutiliser les preuves de raffinement des substitutions de base, déjà établies, pour prouver la correction du raffinement global. Le présent chapitre définit des conditions nécessaires et suffisantes permettant la réutilisation des preuves des raffinements, préalablement établies, lors de l'établissement de la correction d'un raffinement plus complexe.

Le problème de la réutilisation des preuves est actuellement intensivement étudié. Les techniques utilisées sont différentes les unes des autres et se situent dans des contextes très variés. Après une présentation de quelques unes de ces techniques, nous illustrons à travers un exemple notre approche de réutilisation de preuves pour l'établissement de la correction de raffinements **B**. Ensuite, nous étudions, de manière générale (indépendamment de tout domaine d'application), cette technique de réutilisation de preuves de raffinements **B**. Dans un premier temps nous définissons, de manière formelle, les conditions nécessaires et suffisantes à la réutilisation des preuves élémentaires lors de l'établissement de la correction d'un raffinement mettant en jeu deux raffinements élémentaires au plus. Les résultats obtenus sont ensuite généralisés au cas de raffinements plus complexes résultant de l'application simultanée de plusieurs raffinements élémentaires. Enfin, une application de cette stratégie de réutilisation pour notre domaine d'étude (le raffinement des bases de données) est présentée.

6.1 Quelques exemples de techniques de réutilisation de preuves

Dans la littérature, la réutilisation de preuves a été largement abordée, particulièrement dans le domaine de l'intelligence artificielle. Deux classes de méthodologies ont été distinguées: *l'apprentissage par la justification* (*explanation-based learning*)[21, 52, 56, 75] et le *raisonnement par analogie* (*analogical reasoning*) [14, 31, 45, 49, 55]. Dans le premier type de méthodes, ayant établi la preuve (ou solution) d'un problème donné, cette dernière est analysée afin d'en extraire les caractéristiques essentielles qui ont permis de mener la preuve avec succès. Ces caractéristiques, constituant la justification de la preuve, sont ensuite généralisées pour être réutilisées lors de la résolution de nouveaux problèmes similaires. Dans le second type de méthodes, la résolution d'un problème permet de guider l'utilisateur pour la résolution de nouveaux problèmes similaires. Dans ce type de méthodologies, l'intervention de l'utilisateur peut être nécessaire. En effet, cette méthodologie est largement basée sur l'intuition et le savoir-faire.

Dans le domaine des preuves formelles, de nombreuses techniques de réutilisation de preuves, sous-jacentes aux méthodologies sus-citées précédemment, ont été développées. Parmi ces techniques nous citons les suivantes:

– *réutilisation par transformation.*
– *réutilisation par isomorphisme.*
– *réutilisation par analogie.*

– *réutilisation par généralisation.*
– *réutilisation par sous-typage.*

Les sous sections suivantes sont consacrées à l'étude des principaux travaux existants sur les quatre premières techniques de réutilisation. La dernière technique permet de rejouer sur un type *B* la preuve *p* d'une propriété établie sur le super-type *A* de *B*. Cette technique est très utilisée dans le prouveur Coq [4] car ce dernier est fortement typé. Actuellement, il n'existe aucun travail concret relatif à cette technique.

6.1.1 Réutilisation de preuves par transformation

Cette technique est liée au changement du type de données utilisées dans un programme ou une preuve déjà existante. Ce changement du type ou de structure de données peut être motivé par le besoin d'améliorer l'efficacité de calcul d'un programme. Dans [43], il a été montré que la correspondance preuve/programme permet de reformuler une transformation d'un programme en terme de transformation sur la preuve qui lui est associée. En effet, une preuve de l'énoncé:

$$\forall\, x.P(x) \Rightarrow \exists\, y \mid Q(x,y)$$

peut être vue comme un programme fonctionnel calculant y à partir de x. Dans [2], l'auteur propose, dans le système *ELF*, une stratégie d'optimisation d'un programme par transformation de sa preuve. Une technique similaire est présentée dans [60]. Nicolas Magaud [45] utilise cette technique de transformation de preuves afin de réutiliser le principe d'induction, généré pour une représentation à la *Peano* des entiers naturels, pour une représentation binaire de ces même entiers. En effet, à partir des définitions inductives à la *Peano* et binaire des entiers naturels suivantes:

```
Inductive nat:Set:=
   0:nat |
   S: nat->nat

Inductive positif : Set :=
   one:positif |                  (*1*)
   pO:positif ->positif |         (*2x,x > 0*)
   pI:positif ->positif           (*2x + 1,x > 0*)
Inductive bin:Set:=zero:bin| pos:positif->bin
```

deux principes d'induction sont automatiquement générés par le compilateur Coq:

$$P(0) \wedge (\forall\, n \in nat.\ P(n) \Rightarrow P(S\, n)) \Rightarrow \forall\, n \in nat.\ P(n) \qquad (*Peano*)$$
$$P(zero) \wedge P(one) \wedge (\forall\, p \in positif.\ P(p) \Rightarrow P(pO\, p) \wedge P(pI\, p)) \Rightarrow \forall\, n \in bin.\ P(n) \qquad (*\text{binaires}*)$$

Les deux définitions inductives ayant un nombre de constructeurs différents, leurs principes d'induction respectifs sont évidement différents. Le but étant de réutiliser les preuves, procédant par induction, des théorèmes établis dans l'arithmétique de *Peano* pour la preuve de ces mêmes théorèmes dans l'arithmétique binaire. Un principe d'induction (non structurel) similaire à celui des entiers à la *Peano* doit donc être défini:

$$P(zero) \wedge (\forall\, n \in bin.\ P(n) \Rightarrow P(S'\, n)) \Rightarrow \forall\, n \in bin.\ P(n)$$

Ce principe d'induction est basé sur la définition du constructeur S' désignant le successeur d'un entier binaire et de deux fonctions injectives de conversion permettant le passage d'une représentation à l'autre:

$$Nconvert : bin \rightarrow nat \qquad anti\_Nconvert : nat \rightarrow bin$$

La propriété d'injection permet la substitution, dans le nouveau principe d'induction, de l'entier binaire n par $anti\_Nconvert$ $(Nconvert(n))$. Il est alors possible de ré-appliquer le même principe d'induction défini pour les entiers de *Peano* avec comme prédicat non pas P mais $Q = P \circ anti\_Nconvert$.

La définition de ce principe d'induction n'est évidement pas suffisant pour réutiliser les preuves établies sur les entiers de *Peano*. Tout les symboles de fonctions définis sur les entiers de *Peano* doivent être redéfinis sur les entiers binaires. Une fois cette définition faite, une correspondance (isomorphisme) entres les différents symboles des entiers binaires et de *Peano* est établie. Cette correspondance permet de traduire syntaxiquement les termes de preuves obtenus sur les entiers de *Peano* afin de générer d'autres termes de preuves qui sont cette fois-ci des preuves des mêmes propriétés sur les entiers binaires. La définition de la fonction S', de l'isomorphisme, ainsi qu'un exemple de cette traduction syntaxique sont donnés dans [44].

Cette technique de réutilisation de preuves a été aussi exploitée par Mathieu Jaume [36] pour établir la propriété d'unification[1] sur les termes. Une preuve de la même propriété ayant été établie par J. Rouyer [63] sur un ensemble plus général, les quasi-termes, il suffit donc de définir des mécanismes (fonctions de passage des quasi-termes unifiables aux termes correspondant et vice-versa) permettant de transposer la preuve existante de l'ensemble des quasi-termes vers l'ensemble des termes. Dans ce cadre, deux approches ont été étudiées, l'une basée sur la structure inductive (en utilisant les constructeurs) des quasi-termes, l'autre sur l'objet preuve lui même.

6.1.2 Réutilisation par isomorphisme

Cette technique a pour but de récupérer les preuves établies sur un type *A* afin de pouvoir être rejouées sur un type *B* isomorphe[2] au type *A*. L'isomorphisme de types a été utilisé principalement pour la réutilisation de programmes [15, 61, 62]: un programme réalisé pour un type *A* peut être réutilisé pour un type *B* isomorphe à *A*. Actuellement, l'utilisation de l'isomorphisme comme technique de réutilisation de preuves demeure un sujet peu abordé. Cette technique de réutilisation a été introduite par [7]. Elle traduit le fait que les deux types *A* et *B* sont convertibles: on peut utiliser indifféremment *A* ou *B*. Elle consiste à modifier le système de typage par isomorphisme. En reprenant l'exemple des entiers naturels binaires et de *Peano* représentés par:

Inductive nat:Set:= 0:nat| S: nat->nat

Inductive bin:Set:= B_H:bin| B_O:bin ->bin| B_I:bin ->bin

L'isomorphisme est exprimé par des fonctions de conversion d'un type vers l'autre:

1. la fonction successeur S^{bin} sur les binaires est définie par:

   $$\begin{array}{lll} S^{bin}[b\text{:bin}]\text{:bin=case } b \text{ of} & & B_H \Rightarrow B_O\ B_H \\ & | & B_O\ c \Rightarrow B_I\ c \\ & | & B_I\ c \Rightarrow B_O\ (S^{bin}\ c) \end{array}$$

2. la fonction de conversion d'un entier de *Peano* en un entier binaire est définie par:

   $$\begin{array}{lll} nat\_bin\ [n\text{:nat}]\text{:bin=case } n \text{ of} & & 0 \Rightarrow B_H \\ & | & S\ m \Rightarrow S^{bin}\ (nat\_bin\ m) \end{array}$$

3. la fonction B_O^{nat} sur les entiers de *Pean* est définie par: B_O^{nat} [n:nat]:nat=S(double n)
4. la fonction B_I^{nat} sur les entiers de *Pean* est définie par: B_I^{nat} [n:nat]:nat=double(S n)
5. la fonction de conversion d'un entier binaire en un entier de *Peano* est définie par:

   $$\begin{array}{lll} bin\_nat\ [n\text{:bin}]\text{:nat=case } b \text{ of} & & B_H \Rightarrow 0 \\ & | & B_O\ c \Rightarrow B_0^{nat}(bin\_nat\ c) \\ & | & B_I\ c \Rightarrow B_I^{nat}(bin\_nat\ c) \end{array}$$

Ces fonctions de conversion étant définies, on spécifie au système le fait qu'il doit considérer les types nat et bin comme convertibles:

$$nat \quad =_{\sum} \quad bin$$

À présent, l'utilisation d'un binaire à la place d'un entier et vice-versa ne doit pas poser de problèmes de typage. Il reste bien évidement à spécifier au système un ensemble de règles de réécriture qui lui permettront de traiter (d'un point de vue calculatoire) les binaires là où des entiers de *Peano* étaient attendus et vice-versa.

case F of $\{E_N\}$ $\rightarrow_\sigma$ case (bin_nat F) of $\{E_N\}$
case G of $\{E_B\}$ $\rightarrow_\sigma$ case (nat_bin F) of $\{E_B\}$

où:
$F \in \{B_H, B_O\ M, B_I\ M\}$,
$G \in \{0, S\ M\}$.

1. Deux termes t_1 et t_2 sont unifiables s'il existe une substitution σ tel que $\sigma(t_1) = \sigma(t_2)$
2. Deux types *A* et *B* sont isomorphes s'il existe une bijection de *A* vers *B*.

E_N et E_B désignent des fonctions de filtrage sur les constructeurs de nat et bin respectivement.

Cette technique de réutilisation est différente de celle proposée dans [45]. En effet, cette dernière opère directement sur le terme de preuve afin de pouvoir le réadapter au nouveau contexte, le système de typage reste inchangé.

6.1.3 Réutilisation par généralisation

L'idée de base de cette technique est de mettre en évidence pour chaque preuve établie l'ensemble des propriétés (hypothèses) qui justifient sa correction. Une fois ces propriétés généralisées, ces dernières seront ultérieurement instantiées en vue de réutilisation sur un cas pratique. Cette technique a été abordée par plusieurs auteurs [40, 41, 24, 56]. La technique de réutilisation de preuves proposée dans [40, 41] a été définie pour le cas de raisonnement par induction. Deux variantes de cette stratégie ont été étudiées. La première (démarche simplifiée et naïve) ne permettant pas une réutilisation maximale, a été améliorée par une deuxième démarche plus élaborée. Cette technique de réutilisation a été implémentée dans le système PLAGIATOR[39]. Comme toute technique basée sur la généralisation, le principe de base de cette approche est d'étudier le script de la preuve à réutiliser afin d'en extraire les propriétés nécessaires et suffisantes à la correction de la preuve. Une écriture générique symbolique de ces propriétés est alors établie. Les principales étapes de cette approche sont les suivantes:

1. **Établissement d'un ensemble de preuves**: cette phase permet de collecter un ensemble de preuves ϕ qui constitueront par la suite des modèles pour l'établissement de preuves similaires. Ces preuves sont obtenues à partir d'un ensemble d'axiomes en appliquant les techniques de preuve classiques. Par exemple, la preuve de la propriété ϕ suivante:

$$\phi[x,y] :\equiv plus(sum(x),sum(y)) = sum(app(x,y))$$

est obtenue par induction sous les axiomes:

$$\begin{array}{ll}
\text{(plus-1)} & plus(0,y) = y \\
\text{(plus-2)} & plus(s(x),y) = s(plus(x,y)) \\
\text{(sum-1)} & sum(empty) = 0 \\
\text{(sum-2)} & sum(add(n,y)) = plus(n,sum(y)) \\
\text{(app-1)} & app(empty,y) = y \\
\text{(app-2)} & app(add(n,x),y) = add(n,app(x,y)) \\
\text{(assoc-plus)} & plus(plus(x,y),z) = plus(x,plus(y,z))
\end{array}$$

où:
$plus$: désigne l'addition sur les entiers.
s: désigne le successeur d'un entier.
sum: désigne la somme des éléments d'une liste d'entiers.
app: désigne la concaténation de deux listes.
add: désigne l'insertion d'un élément en tête de liste.

Les deux cas de base et d'induction de ϕ sont:

$$\phi_b :\equiv \forall\, y.\phi[empty,y]$$
$$\phi_i :\equiv \forall\, n,x,y.(\phi[x,y] \rightarrow \phi[add(n,x),y])$$

Le cas de base étant trivial, le cas d'induction se prouve de la manière suivante:

$$\begin{array}{rlr}
plus(sum(add(n,x)),sum(y)) = & sum(app(add(n,x),y)) & \\
plus(\underline{plus(n,sum(x))},sum(y)) = & \ldots & \text{(sum-2)} \\
\ldots = & sum(\underline{add(n,app(x,y))}) & \text{(app-2)} \\
\ldots = & \underline{plus(n,sum(app(x,y)))} & \text{(sum-2)} \\
\ldots = & plus(n,\underline{plus(sum(x),sum(y))}) & \text{(Hypothèse d'induction)} \\
\underline{plus(n,plus(sum(x),sum(y)))} = & \ldots & \text{(assoc-plus)} \\
\text{True} & & \text{(X=X)}
\end{array}$$

2. **Analyse:** cette phase permet la définition des caractéristiques essentielles d'une preuve déjà établie. Elle est basée sur le script de la preuve à réutiliser. L'ensemble des caractéristiques c est dit *justifications de la preuve* (*catch proof*). Par

exemple, les justifications du script de preuve de ϕ sont: (**sum-2**), (**app-2**) et (**assoc-plus**). Les règles du type $X = X$ ne font pas partie des justifications car celles-ci sont toujours vraies: elle sont vérifiées quelque soit le contexte de la preuve.

3. **Généralisation:** le but de cette étape est d'exprimer de manière générique la formule ϕ et ses justifications c. Chaque symbole de fonction spécifique est remplacé par un symbole de fonction. La généralisation concerne la formule, ses justifications et aussi sa preuve. La généralisation de la preuve de la propriété ϕ précédente est réalisée comme suit (F=plus, G=sum, H=app, C=empty, D=add):

- **Généralisation de ϕ et de ses cas de base et d'induction**
  $\Phi[F,G,H][x,y] :\equiv F(G(x),G(y)) = G(H(x,y))$
  $\Phi_b[F,G,H][C] :\equiv \forall\, y.\Phi[F,G,H][C,y]$
  $\Phi_i[Fx,G,H][D] :\equiv \forall\, n,x,y,(\varphi[F,G,H][x,y] \to \varphi[F,G,H][D(n,x),y])$

- **Généralisation des preuves des cas de base et d'induction**
  $\Phi'_b[F,G,H][C] :\equiv \quad \ldots$
  $\Phi'_i[F,G,H][D] :\equiv \quad G(D(n,x)) = F(n,G(x))$
  $H(D(n,x),y) = D(n,H(x,y))$
  $F(F(x,y),z) = F(x,F(y,z))$

4. **Réutilisation:** pour établir une preuve d'une nouvelle formule ψ, on recherche dans les preuves déjà établies une preuve dont la formule Φ est de même structure que ψ. Si une telle formule est trouvée, une instanciation des justifications est établie. Si les différentes justifications sont vérifiées, le schéma de preuve de ψ est rien d'autre que l'instantiation de celui de Φ. Par exemple, si l'on considère avoir à prouver la formule ψ définie par:

$$\psi[x,y] :\equiv times(prod(x),prod(y)) = prod(app(x,y))$$

sous les axiomes:

(times-1)	$times(0,y) = 0$
(times-2)	$times(s(x),y) = plus(y,times,y))$
(prod-1)	$prod(empty) = 1$
(prod-2)	$prod(add(n,y)) = times(n,prod(y))$

où:
$times$: désigne le produit sur les entiers.
$prod$: désigne le produit des éléments d'une liste d'entiers.
Comme on peut le remarquer ψ a la même structure que Φ. En effet, il suffit d'instancier F par $times$, G par $prod$ et H par app. On peut donc envisager de réutiliser la preuve de Φ. Pour cela, nous devons tester si les justifications de Φ sont aussi vraies dans le contexte (axiomes) de ψ. On doit donc prouver que:

$\psi'_{1s}[times,prod,app] :\equiv$
$prod(add(n,x)) = times(n,prod(x))$
$app(add(n,x),y) = add(n,app(x,y))$
$times(times(x,y),z) = times(x,times(y),z)$

Les deux premières conditions étant vérifiées (**prod-2** et **app-2**), pour réutiliser la preuve précédemment établie, il suffira de prouver la troisième condition.

Dans [75], les auteurs montrent que cette démarche simplifiée est insuffisante pour assurer une réutilisation maximale des preuves déjà établies. Ils proposent pour cela, une méthode raffinée de cette dernière. Elle consiste à affecter un indice supplémentaire pour chaque occurrence particulière d'une fonction donnée. Cette approche permet une interprétation multiple d'une même fonction. Pour plus de détails, le lecteur est envoyé à [75]. Une méthode similaire, basée sur le script de preuve Coq, a été proposée par Olivier Pons [56].

6.1.4 Réutilisation par analogie

Dans la littérature, le terme analogie est utilisé dans divers domaines: mathématiques, philosophie, etc. La réutilisation de preuves par analogie consiste à prendre une preuve déjà établie, dite *preuve source*, comme modèle (guide, patron) pour l'établissement d'une nouvelle preuve similaire dite *preuve cible*. La difficulté de cette technique est de retrouver la preuve source qui serait analogue à la preuve cible qu'on voudrait établir: *quelle est la définition de deux preuves analogues ?*

Sur quels critères doit-on se baser pour affirmer que deux preuves sont analogues? La réponse à ces questions dépend fortement du domaine d'application considéré. Comme pour les autres techniques, la plupart des travaux existants traitent exclusivement le cas des théorèmes mathématiques prouvables par induction. Le travail de [41], présenté précédemment, peut être vu aussi comme une sorte de réutilisation de preuves par analogie. En effet, on exige que les preuves source et destination soient de même structure. [74] ont implémenté un système, appelé *ABSFOL*, qui permet d'utiliser l'abstraction d'une preuve source pour établir une preuve cible. Par exemple, pour établir le théorème:

$$s(x) \times y = y + (x \times y)$$

ABSFOL réutilise l'abstraction (généralisation):

$$(?\#x)\#y = ?\#(x\#y)$$

du théorème source:

$$s(x) + y = s(x + y)$$

[75] distingue deux types d'analogie: *analogie interne* et *analogie externe*. Contrairement à l'analogie externe, l'analogie interne concerne une même preuve: la preuve d'un sous but est réutilisée afin d'établir la preuve d'un autre sous but au sein de la même preuve. Melis et Whittle ont développé une technique de réutilisation par analogie externe appelé ABALONE. Cette technique a été implémentée en utilisant le système *Clam*. Elle consiste en les étapes suivantes:

1. Établissement du plan de preuve d'un théorème source: soit, par exemple, l'ensemble des axiomes indexés suivant (on affecte aux différentes occurrences des constructeurs des indices différents[3]):

app2:	$(X::_1 Y)$ @ $Z \equiv X::_2 (Y$ @ $Z)$
len2:	$\text{length}(X::_3 Y) \equiv s_1(\text{length}(Y))$
lenapp2:	$\text{length}(X$ @ $(Y::_4 Z)) \equiv s_2(\text{length}(Y$ @ $Z))$

Le plan de preuve du théorème source:

length(x@y)=length(y@x)

est comme suit (seul le cas d'induction est présenté, le cas de base ne pose pas de difficultés):

$$\begin{array}{rcl}
\text{length}(h ::_5 a \text{ @ } b) & = & \text{length}(b \text{ @ } h ::_5 a) \\
::_5 = ::_1 & \downarrow & \text{(app2)} \\
\text{length}(h::_2(a@b)) & = & \text{length}(b@h::_5 a) \\
::_2 = ::_3 & \downarrow & \text{(len2)} \\
s_1(\text{length}(a@b)) & = & \text{length}(b@h::_5 a) \\
::_5 = ::_4 & \downarrow & \text{(lenapp2)} \\
s_1(\text{length}(a@b)) & = & s_2(\text{length}(b@a)) \\
 & \downarrow & \text{Hyp_Ind} \\
s_1(\text{length}(b@a)) & = & s_2(\text{length}(b@a)) \\
s_1 = s_2 & \downarrow & \text{Tautologie} \\
 & \text{True} &
\end{array}$$

Pour chaque étape de ce plan de preuve, on spécifie, d'une part, l'axiome appliqué et d'autre part, les contraintes nécessaires à son application.

2. Filtrage de base (*basic mapping* m_b): cette étape consiste à faire la correspondance entre les symboles de fonctions apparaissant dans les théorèmes source et destination: détecter l'analogie existante entre les deux théorèmes. S'il'on considère à présent, le théorème cible:

$$\mathit{half}(x + y) = \mathit{half}(y + x)$$

sous les axiomes:

plus2:	$s(Y)+Z \equiv s(Y+Z)$
half3:	$\text{half}(s(s(Y))) \equiv s(\text{half}(Y))$

L'identification des deux théorèmes source et cible permet de définir les filtrages suivants:

$m_b(\text{length})=\text{half}$ $\quad$ $m_b(@)=+$

3. Cette indexation est similaire à celle proposée dans [75]. Elle permet une interprétation multiple du même constructeur selon le besoin.

3. Filtrage étendu (*extended mapping* m_e): lors de la phase précédente, seuls les symboles des fonctions apparaissant dans les théorèmes source et cible ont été filtrés. Cette étape permet l'extension du filtrage précédent à l'ensemble des symboles des règles d'inférence source et destination. Elle est bien évidement basée sur les filtrages de base établis à l'étape précédente:

– (app2) filtre (plus2) avec: $m_e(::_1) = \lambda\, x\, \lambda\, y.s(y)$ et $m_e(::_2) = \lambda\, x\, \lambda\, y.s(y)$
– (len2) filtre (half3) avec: $m_e(s_1) = \lambda\, x.s(x)$ et $m_e(::_3) = \lambda\, x\, \lambda\, y.s(s(y))$

Comme on peut le remarquer, ce filtrage ne vérifie pas la contrainte du plan de preuve source: $m_e(::_2)$=$m_e(::_3)$. Une correction (*reformulation*) est donc nécessaire. Pour ce cas assez trivial, l'égalité peut être obtenue en appliquant deux fois la règle (plus2) pour chaque application de la règle (app2). Un ensemble de règles de corrections similaires ont été définies dans ABALONE [78].

4. Établissent, par analogie, du plan cible: une fois le filtrage étendu établi, les reformulations définies, le plan de la preuve cible est obtenu en appliquant les différentes règles de filtrage sur le plan de la preuve source:

Plan source			**Plan cible**		
length(h$::_5$ a @ b)	=	length(b @ h$::_5$ a)	half(s(s(a))+b)	=	half(b+s(s(a)))
	↓			↓	(plus2)
	(app2)		half(s(s(a)+b))	=	half(b+s(s(a)))
	↓			↓	(plus2)
length(h$::_2$(a@b))	=	length(b@h$::_5$ a)	half(s(s(a+b)))	=	half(b+s(s(a)))
	↓	(len2)		↓	(half3)
s_1(length(a@b))	=	length(b@h$::_5$ a)	s(half(a+b))	=	half(b+s(s(a)))
	↓	(lenapp2)		↓	*lemme*
s_1(length(a@b))	=	s_2(length(b@a))	s(half(a+b))	=	s(half(b+a))
	↓	Hyp_Ind		↓	Hyp_Ind
s_1(length(b@a))	=	s_2(length(b@a))	s(half(b+a))	=	s(half(b+a))
	↓	Tautologie		↓	Tautologie
	True			True	

L'application de l'axiome (lenapp2) dans le plan source, est remplacée (par analogie) par l'introduction et l'application d'un lemme dans le plan cible. En effet, lors des deux phases de filtrage, à l'axiome source (lenapp2) ne correspondait aucun axiome cible. À ce dernier, *ABALONE* propose un nouveau lemme qui devrait le filtrer. Dans ce cas précis, le lemme proposé est:

$$half(X + s(s(Z))) = s(half(X + Z))$$

Ce lemme est obtenu par le filtrage: $m_e(::_5) = \lambda\, x,y.s(s(y))$ car $::_4$=$::_5$. $m_e(::_5) = \lambda\, x,y.s(s(y))$ est établi par la première ligne de l'analogie des deux plans.

Dans [78], les auteurs montrent, à travers des expérimentations, que la technique de preuve par analogie permet un gain de temps considérable. De plus, certains plans de preuve ne peuvent être établis qu'en utilisant l'analogie. [35] a étudié l'application de cette technique au calcul des limites de fonctions et aussi à la théorie des groupes.

6.1.5 Commentaires

La plupart des techniques existantes ont été développées pour le cas de théorèmes prouvables par induction. L'extension de ces techniques à d'autres domaines d'étude (autres que les mathématiques) n'a été abordée par aucun des auteurs. Les travaux existants sur la technique d'analogie sont basés sur une analogie de structures des théorèmes source et destination. Dans ce cas l'analogie est presque immédiate. De plus, les théorèmes source et cible appartiennent au même domaine. L'ensemble des auteurs étudiés illustrent leurs techniques à travers des exemples concrets et sur des domaines restreints, aucune des techniques n'est définie de manière formelle et générale. Il n'existe pas de critères formels qui permettraient de choisir de manière rigoureuse la technique la plus adaptée au problème traité, d'autant plus que les frontières entre les différentes techniques ne sont pas bien définies. En effet par exemple, le terme analogie étant un terme général pas précisément défini, on serait tenté de dire que toutes les techniques sont des cas particuliers de la technique par analogie.

Dans le cadre de cette thèse, nous proposons une technique de réutilisation de preuves basée sur les propriétés du domaine. Cette technique étudie la possibilité et la correction de la décomposition d'une preuve complexe en un ensemble de preuves plus élémentaires déjà établies. On s'intéresse plus précisément à la preuve de correction de raffinements complexes **B** en réutilisant les preuves de correction de raffinements plus élémentaires. L'idée de base est donc de pouvoir exhiber les raffinements élémentaires par transformation du raffinement à prouver. Notre technique peut être vue comme un cas particulier de réutilisation par généralisation. Elle est basée non pas sur l'analogie des structures des preuves source et cible, mais sur l'inclusion de l'environnement de la preuve à réutiliser dans l'environnement de la preuve à établir. En effet, pour réutiliser la preuve d'un raffinement déjà établi, on doit être en mesure, dans l'environnement de la preuve à effectuer, d'exhiber les éléments (conditions) qui ont permis l'établissement de cette preuve, c.à.d: substitutions abstraite et concrète, l'invariant de collage, les variables fixes et modifiées. En **B**, le raffinement n'étant pas toujours fonctionnel (l'invariant de collage liant une substitution abstraite à son raffinement n'est pas unique), l'objectif de ce chapitre est de définir de manière formelle les conditions nécessaires et suffisantes, devant être satisfaites dans l'environnement de la preuve à établir, qui permettent la réutilisation de preuves élémentaires déjà établies.

6.2 Réutilisation des preuves dans le contexte du raffinement B

Dans cette section, nous illustrons, à travers deux exemples simplifiés, ce que signifie la réutilisation de preuves lors de l'établissement de la correction d'un raffinement **B**. Pour cela, on considère les deux raffinements de substitutions indépendants suivants:

$$(x : (x \geq 0 \wedge x \leq 5)) \sqsubseteq_{x+y<0} (y := -3) \tag{6.1}$$

$$(z := -6) \sqsubseteq_{z+t\geq 0} (t : (t \geq 6)) \tag{6.2}$$

En utilisant la définition A.3.1 de la correction d'un raffinement, les preuves de ces raffinements peuvent être établies de manière triviale. Le premier raffinement est correct pour toute valeur de x' appartenant à l'ensemble $\{0, 1, 2\}$. En effet, l'invariant de collage $(x' + y' < 0)$ est vérifié pour toute valeur de x' qui appartient à cet ensemble. Le deuxième raffinement est également correct pour la valeur (-6) de z', car pour toute valeur de t' vérifiant $(t' \geq 6)$, $(z' = -6)$ satisfait l'invariant de collage $(z' + t' \geq 0)$.

On considère à présent avoir à établir le raffinement :

$$(x : (x \geq 0 \wedge x \leq 5) \| z := -6) \sqsubseteq_{(x+y<-2 \wedge z+t \geq 0)} (y := -3 \| t : (t \geq 6)) \tag{6.3}$$

Établir la correction de ce raffinement revient à déterminer un couple de valeurs (x',z') associé au couple de valeurs (y',t'), vérifiant $(y' = -3 \wedge t' \geq 6)$, tel que:

$$x' \geq 0 \wedge x' \leq 5 \wedge z' = -6 \wedge x' + y' < -2 \wedge z' + t' \geq 0 \tag{6.4}$$

Comme on peut le remarquer, les substitutions abstraite et concrète de ce raffinement sont celles des raffinements déjà établis (6.1) et (6.2). De plus, les deux invariants de collage $(x + y < 0)$ et $(z + t \geq 0)$ peuvent être déduits de l'invariant de collage courant $(x + y < -2 \wedge z + t \geq 0)$. La stratégie de preuve du raffinement (6.4) par réutilisation des preuves des raffinements (6.1) et (6.2) est donc justifiée. Réutiliser les preuves des raffinements (6.1) et (6.2) revient à s'assurer que leurs solutions vérifient le but (6.4):

- **cas 1.** $(x',z') = (0, -6)$: la substitution de ces valeurs dans (6.4) donne:
  $$0 \geq 0 \wedge 0 \leq 5 \wedge -6 = -6 \wedge 0 + y' < -2 \wedge -6 + t' \geq 0$$
  qui est vrai: $(x',z') = (0, -6)$ permet d'établir le raffinement.
- **cas 2**. $(x',z') = (1, -6)$: la substitution de ces valeurs dans (6.4) donne:
  $$1 \geq 0 \wedge 1 \leq 5 \wedge -6 = -6 \wedge 1 + y' < -2 \wedge -6 + t' \geq 0$$
  qui est faux car $y' = -3$.
- **cas 3.** $(x',z') = (2, -6)$: la substitution de ces valeurs dans (6.4) donne:
  $$2 \geq 0 \wedge 2 \leq 5 \wedge -6 = -6 \wedge 2 + y' < -2 \wedge -6 + t' \geq 0$$

qui est faux car $y' = -3$.

En résumé, le raffinement (6.3) a été prouvé par réutilisation d'une *seule* des solutions déjà établies. L'établissement de raffinements par réutilisation de preuves de raffinements déjà établis, a permis de réduire le coût de recherche de la valeur de x'. Ce gain peut être plus conséquent particulièrement dans le cas où l'invariant de collage est complexe. Il est plus simple et moins coûteux de vérifier si une valeur donnée satisfait un prédicat que de déterminer la valeur elle-même.

Il est important de noter que seules (une dans ce cas précis) les solutions vérifiant l'invariant de collage peuvent être réutilisées. En effet, ce dernier peut être plus fort.

Considérons à présent, les deux raffinements:

$$(y := -6 + x) \sqsubseteq_{2y=z} (z := 2(-6 + x)) \tag{6.5}$$

et

$$(x := 1) \sqsubseteq_{Id} (x := 1)^5 \tag{6.6}$$

prouvables pour $(y',x') = (-6 + x,x)$ et $x' = 1$ respectivement. Soit le nouveau raffinement à prouver:

$$(y := -6 + x \| x := 1) \sqsubseteq_{(2y=z \wedge x=x)} (z := 2(-6 + x) \| x := 1) \tag{6.7}$$

Comme on peut le remarquer, bien que les substitutions abstraite et concrète de ce raffinement soient celles de (6.5) et (6.6), aucune des valeurs $(y',x') = (-6 + x,x)$ ou $(x' = 1)$ ne peut être réutilisée pour prouver le raffinement (6.7). En effet, la variable x' possède deux valeurs distinctes. Ceci est dû au fait que la variable x est fixe dans $(y := -6 + x)$ et modifiée dans $(x := 1)$.

Dans cette section, nous avons montré, à travers deux exemples simples, que la réutilisation de preuves (solutions), pour l'établissement de la correction d'un raffinement **B**, n'est pas toujours possible. Cette réutilisation est sujette à conditions. Les sections suivantes sont dédiées à la définition formelle de ces conditions de réutilisation. Pour cela, soient S_x et T_y deux substitutions telles qu'il existe une preuve formelle *Pr* du raffinement de S_x par T_y ayant pour invariant de collage le prédicat *J(x,y)*: $S_x \sqsubseteq_{J(x,y)} T_y$. Notre but est de définir les conditions et la façon dont cette preuve *Pr* peut être réutilisée pour prouver le raffinement d'une substitution plus complexe construite à partir de la substitution S_x. En **B**, les substitutions étant construites de manière inductive, nous procédons donc par induction sur la forme de la substitution à raffiner. Nous distinguons pour cela six cas d'opérateurs de substitutions:

- substitution préconditionnée: $P \mid S$.
- substitution parallèle $S_1 \| S_2$: deux cas sont à distinguer :
  - le raffinement de $S_1 \| S_2$ par une substitution parallèle.
  - le raffinement de $S_1 \| S_2$ par une substitution séquentielle.
- substitution choix borné: $S_1 [] S_2$.
- substitution conditionnée: **IF** P **THEN** S_1 **ELSE** S_2 **END**.
- substitution séquence: $S_1 ; S_2$.
- raffinement d'une substitution **ANY**.

Le reste de ce chapitre présente en détails le cas de précondition et le premier cas de la substitution parallèle. Les autres cas sont présentés en annexe (Voir l'annexe B). Un tableau récapitulatif des résultats obtenus pour l'ensemble des cas est donné à la page 141. Dans ce qui suit, chaque substitution ou prédicat est indexé par l'ensemble de ses variables libres.

6.3 Preuve par réutilisation du raffinement de la substitution préconditionnée

Soient S_{a_1} et T_{a_2} deux substitutions, $J(a_1,a_2)$ un prédicat tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2} \tag{H$_1$}$$

5. L'invariant de collage *Id* désigne le raffinement de chaque variable par elle-même

Soient P_{b_1}, Q_{b_2} et $J(a_1 \cup b_1, a_2 \cup b_2)$ trois prédicats tels que:

$$(P_{b_1} \mid S_{a_1}) \sqsubseteq_{J(a_1 \cup b_1, a_2 \cup b_2)} (Q_{b_2} \mid T_{a_2}) \tag{G$_1$}$$

On voudrait réutiliser la preuve du raffinement (H_1) lors de la preuve du raffinement (G_1). Cela revient à définir les conditions nécessaires et suffisantes pour que cette réutilisation soit possible.

Le problème qu'on se pose donc est le suivant (on pose $c_1 = a_1 \cup b_1, c_2 = a_2 \cup b_2$):

Hypothèses: par définition du raffinement et de la condition nécessaire à la correction du raffinement d'une précondition (affaiblissement de la précondition), H_1 et G_1 donnent:

$$[Trm(S_{a_1}) \land J(a_1, a_2)] \to [Trm(T_{a_2}) \land \forall\, a_2''.(Prd(T_{a_2}) \to \exists\, a_1'.(Prd(S_{a_1}) \land J(a_1', a_2'')))] \tag{H$_2$}$$

$$(P_{b_1} \land J(c_1, c_2) \to Q_{b_2}) \tag{H$_3$}$$

But: établir une preuve, par réutilisation de (H_1), de:

$$\begin{aligned}[Trm(P_{b_1} \mid S_{a_1}) \land J(c_1, c_2)] \to [Trm(Q_{b_2} \mid T_{a_2}) \land \forall\, c_2''.(Prd(Q_{b_2} \mid T_{a_2}) \to \\ \exists\, c_1'.(Prd(P_{b_1} \mid S_{a_1}) \land J(c_1', c_2'')))]\end{aligned} \tag{G$_2$}$$

Soient d_1 (resp. d_2) l'ensemble des variables libres de P_{b_1} (resp. Q_{b_2}) qui n'apparaissent pas dans S_{a_1} (resp. T_{a_2}):

$$d_1 = b_1 - a_1$$
$$d_2 = b_2 - a_2$$

on a (par définition):

$$Trm(P_{b_1} \mid S_{a_1}) = P_{b_1} \land Trm(S_{a_1}) \qquad Prd(P_{b_1} \mid S_{a_1}) = P_{b_1} \to (Prd(S_{a_1}) \land d_1' = d_1)$$
$$Trm(Q_{b_2} \mid T_{a_2}) = Q_{b_2} \land Trm(T_{a_2}) \qquad Prd(Q_{b_2} \mid T_{a_2}) = Q_{b_2} \to (Prd(T_{a_2}) \land d_2' = d_2)$$

Le but (G_2) se réécrit en:

$$\begin{aligned}[P_{b_1} \land Trm(S_{a_1}) \land J(c_1, c_2)] \to [Q_{b_2} \land Trm(T_{a_2}) \land \forall\, a_2'', d_2''.((Q_{b_2} \to (Prd(T_{a_2}) \land d_2'' = d_2)) \to \\ \exists\, a_1', d_1'.((P_{b_1} \to (Prd(S_{a_1}) \land d_1' = d_1)) \land J(a_1' \cup d_1', a_2'' \cup d_2'')))]\end{aligned} \tag{G$_3$}$$

Par les règles de déduction et de l'élimination du quantificateur universel, on obtient les trois sous buts:

$$Q_{b_2} \tag{G$_4$}$$

$$Trm(T_{a_2}) \tag{G$_5$}$$

$$(Q_{b_2} \to (Prd(T_{a_2}) \land d_2'' = d_2)) \to \exists\, a_1', d_1'.((P_{b_1} \to (Prd(S_{a_1}) \land d_1' = d_1)) \land J(a_1' \cup d_1', a_2'' \cup d_2'')) \tag{G$_6$}$$

sous les hypothèses supplémentaires:

$$P_{b_1} \tag{H$_4$}$$

$$Trm(S_{a_1}) \tag{H$_5$}$$

$$J(c_1, c_2) \tag{H$_6$}$$

Preuve de (G_4): (H_4), (H_6) et (H_3) prouvent (G_4).

Preuve de (G_5): pour prouver (G_5) par réutilisation de (H_2), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(S_{a_1})$ et $J(a_1, a_2)$. $Trm(S_{a_1})$ étant l'hypothèse (H_5), il reste à déduire de notre environnement d'hypothèses, constitué de (H_4), (H_5), et (H_6), l'hypothèse:

$$J(a_1, a_2) \tag{H$_7$}$$

L'une des conditions nécessaires et suffisantes à la réutilisation est donc :

$$\boxed{(P_{b_1} \land Trm(S_{a_1}) \land J(c_1, c_2)) \to J(a_1, a_2)}$$

Cette hypothèse signifie que pour réutiliser la preuve de raffinement de H_1, on doit pouvoir exhiber son invariant de collage dans l'environnement courant de preuve.

(H_7), (H_5) et (H_2) prouvent alors (G_5) et génèrent une hypothèse supplémentaire:

$$\forall\, a_2''.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1', a_2''))) \tag{H_8}$$

Remarque: La condition (H_7) n'est nécessaire que pour la réutilisation de la preuve de correction du raffinement $S_{a_1} \sqsubseteq T_{a_2}$: T_{a_2} peut éventuellement terminer sans que cette condition soit vérifiée: $Trm(T_{a_2}) = True$.

Preuve de (G_6): par l'application de la règle de déduction, le but à prouver devient:

$$\exists\, a_1', d_1'.((P_{b_1} \rightarrow (Prd(S_{a_1}) \wedge d_1' = d_1)) \wedge J(a_1' \cup d_1', a_2'' \cup d_2'')) \tag{$G_{6.1}$}$$

sous l'hypothèse supplémentaire:

$$Q_{b_2} \rightarrow (Prd(T_{a_2}) \wedge d_2'' = d_2) \tag{H_9}$$

(G_4) et (H_9) donnent:

$$Prd(T_{a_2}) \tag{H_{10}}$$

$$d_2'' = d_2 \tag{H_{11}}$$

(H_{10}) et l'instanciation de (H_8) sur a_2''donnent:

$$\exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1', a_2'')) \tag{H_{12}}$$

Soit $sol_{a_1'}$ une des valeurs de a_1' qui satisfait cette dernière hypothèse, on a donc:

$$[a_1' := sol_{a_1'}]Prd(S_{a_1}) \tag{H_{13}}$$

$$[a_1' := sol_{a_1'}]J(a_1', a_2'') \tag{H_{14}}$$

Réutiliser la preuve du raffinement ($S_{a_1} \sqsubseteq T_{a_2}$) revient à vérifier que la solution $sol_{a_1'}$ est aussi une solution du raffinement préconditionné. Pour cela, on doit vérifier que $sol_{a_1'}$ satisfait le but ($G_{6.1}$), c.à.d:

$$\exists\, d_1'.((P_{b_1} \rightarrow ([a_1' := sol_{a_1'}]Prd(S_{a_1}) \wedge d_1' = d_1)) \wedge [a_1' := sol_{a_1'}]J(a_1' \cup d_1', a_2'' \cup d_2'')) \tag{$G_{6.2}$}$$

Pour que $P_{b_1} \rightarrow ([a_1' =: sol_{a_1'}]Prd(S_{a_1}) \wedge d_1' = d_1)$ soit vrai, il faut que $([a_1' := sol_{a_1'}]Prd(S_{a_1}) \wedge d_1' = d_1)$ soit vrai car P_{b_1} est vrai (d'après H_4): d_1' est donc égal à d_1. Comme $[a_1' := sol_{a_1'}]Prd(S_{a_1})$ est vrai (d'après H_{13}), en remplaçant d_2'' par d_2 (d'après H_{11}) et d_1' par d_1, le but à prouver devient donc:

$$[a_1' := sol_{a_1'}]J(a_1' \cup d_1, a_2'' \cup d_2) \tag{$G_{6.3}$}$$

Pour réutiliser la preuve de raffinement (H_2), on doit pouvoir déduire de l'ensemble des hypothèses, constituant l'environnement de preuve le but ($G_{6.3}$). Une deuxième condition suffisante et nécessaire à la réutilisation est:

$$\boxed{\begin{array}{r}(P_{b_1} \wedge Trm(S_{a_1}) \wedge J(c_1, c_2) \wedge [a_1' := sol_{a_1'}]Prd(S_{a_1}) \wedge [a_1' := sol_{a_1'}]J(a_1', a_2'') \wedge Prd(T_{a_2})) \rightarrow \\ [a_1' := sol_{a_1'}]J(a_1' \cup d_1, a_2'' \cup d_2)\end{array}}$$

Résumé

Soient S_{a_1}, T_{a_2} deux substitutions et $J(a_1, a_2)$ un prédicat tels que:

$$S_{a_1} \sqsubseteq_{J(a_1, a_2)} T_{a_2}$$

Soient P_{a_1}, Q_{b_2} et $J(a_1 \cup b_1, a_2 \cup b_2)$ trois prédicats tels que ($c_i = a_i \cup b_i$ et $d_i = b_i - a_i$):

$i.$ $(P_{b_1} \wedge J(c_1, c_2)) \rightarrow Q_{b_2}$

$ii.$ $(P_{b_1} \wedge Trm(S_{a_1}) \wedge J(c_1, c_2)) \rightarrow J(a_1, a_2)$

iii. Soit a_2'' une valeur de a_2 résultant de l'application de T_{a_2}. Une valeur a_1' de a_1 résultant de l'application de S_{a_1} vérifiant:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$$

et

$$(P_{b_1} \wedge Trm(S_{a_1}) \wedge J(c_1,c_2) \wedge Prd(S_{a_1}) \wedge J(a_1',a_2'') \wedge Prd(T_{a_2})) \rightarrow J(a_1' \cup d_1, a_2'' \cup d_2)$$

peut être réutilisée pour prouver le raffinement $(P_{b_1} \mid S_{a_1} \sqsubseteq_{J(c_1,c_2)} Q_{b_2} \mid T_{a_2})$.

Remarques

1. La condition (i) est une condition nécessaire pour la correction du raffinement:
$$P_{b_1} \mid S_{a_1} \sqsubseteq_{J(c_1,c_2)} Q_{b_2} \mid T_{a_2}$$
2. La condition (ii) est une condition nécessaire de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $S_{a_1} \sqsubseteq T_{a_2}$, alors il faut être capable d'exhiber dans l'environnement de preuve de $(P_{b_1} \mid S_{a_1} \sqsubseteq Q_{b_2} \mid T_{a_2})$ l'invariant de collage de $S_{a_1} \sqsubseteq T_{a_2}$. Dans le cas contraire la stratégie de réutilisation n'est pas justifiée.
3. La condition (iii) est une condition nécessaire uniquement pour la réutilisation de la solution du raffinement de $S_{a_1} \sqsubseteq T_{a_2}$. Une autre solution du raffinement sans réutilisation peut éventuellement exister.
4. La réutilisation de la preuve de raffinement de $S_{a_1} \sqsubseteq T_{a_2}$ est réalisable pour toute valeur de a_1' qui vérifie la condition (iii): on n'exige pas que toutes les valeurs a_1' vérifient cette condition.
5. Si la substitution S_{a_1} est déterministe alors la condition (iii) est vérifiée. En effet, il existe une seule valeur de a_1' qui satisfait le raffinement $(S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2})$, celle-ci est déterminée par $Prd(S_{a_1})$. De plus, une solution possible du raffinement $(P_{b_1} \mid S_{a_1} \sqsubseteq_{J(c_1,c_2)} Q_{b_2} \mid T_{a_2})$ doit satisfaire le $Prd(S_{a_1})$ (à moins que le raffinement soit une tautologie) qui lui-même n'admet qu'une seule solution: les deux raffinements admettent donc le même espace de solutions.
6. Si le prédicat $J(a_1,a_2)$[4] est fonctionnel en a_1 alors la condition (iii) est vérifiée. En effet, une solution possible du raffinement $(P_{b_1} \mid S_{a_1} \sqsubseteq_{J(c_1,c_2)} Q_{b_2} \mid T_{a_2})$ doit satisfaire le prédicat $J(c_1,c_2)$. D'après (ii) cette solution satisfait le prédicat $J(a_1,a_2)$. Or $J(a_1,a_2)$ n'admet qu'une solution unique a_1' pour toute valeur de a_2: cette solution unique est donc solution de $(P_{b_1} \mid S_{a_1} \sqsubseteq_{J(c_1,c_2)} Q_{b_2} \mid T_{a_2})$.

6.4 Preuve par réutilisation du raffinement de la substitution parallèle

Soient $S_{a_1}, T_{a_2}, U_{b_1}$ et V_{b_2} quatre substitutions et $J(a_1,a_2), J(b_1,b_2)$ deux prédicats tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2} \qquad (H_1)$$
$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2} \qquad (H_2)$$

Soit $J(a_1 \cup b_1, a_2 \cup b_2)$ un prédicat tels que:

$$(S_{a_1} \| U_{b_1} \sqsubseteq_{J(a_1 \cup b_1, a_2 \cup b_2)} T_{a_2} \| V_{b_2}) \qquad (G_1)$$

On voudrait réutiliser les preuves de raffinement (H_1) et (H_2) lors de la preuve du raffinement (G_1). Cela revient à définir les conditions nécessaires et suffisantes pour que cette réutilisation soit possible.

Le problème qu'on se pose est donc (on pose $c_1 = a_1 \cup b_1$ et $c_2 = a_2 \cup b_2$):

Hypothèses: par définition du raffinement, les hypothèses H_1 et H_2 donnent:

$$(Trm(S_{a_1}) \wedge J(a_1,a_2)) \rightarrow [Trm(T_{a_2}) \wedge \forall a_2''.(Prd(T_{a_2}) \rightarrow \exists a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2'')))] \qquad (H_3)$$
$$(Trm(U_{b_1}) \wedge J(b_1,b_2)) \rightarrow [Trm(V_{b_2}) \wedge \forall b_2''.(Prd(V_{b_2}) \rightarrow \exists b_1'.(Prd(U_{b_1}) \wedge J(b_1',b_2'')))] \qquad (H_4)$$

But: établir une preuve, par réutilisation de (H_1) et (H_2), de:

$$(Trm(S_{a_1} \| U_{b_1}) \wedge J(c_1,c_2)) \rightarrow [Trm(T_{a_2} \| V_{b_2}) \wedge \forall c_2''.(Prd(T_{a_2} \| V_{b_2}) \rightarrow \exists c_1'.(Prd(S_{a_1} \| U_{b_1}) \wedge J(c_1',c_2'')))] \qquad (G_2)$$

4. $J(a_1,a_2)$ est fonctionnel en a_1 si pour toute valeur de a_2 il existe une seule valeur de a_1 qui vérifie ce prédicat.

Soient:

- a_1^m (resp. b_1^m): les variables de a_1 (resp. b_1) modifiées par S_{a_1} (resp. U_{b_1}): $MV(S_{a_1})$ (resp. $MV(U_{b_1})$),
- a_1^f (resp. b_1^f): les variables de a_1 (resp. b_1) non modifiées par S_{a_1} (resp. U_{b_1}): $FV(S_{a_1}) - MV(S_{a_1})$ (resp. $FV(U_{b_1}) - MV(U_{b_1})$),
- a_2^m (resp. b_2^m): les variables de a_2 (resp. b_2) modifiées par T_{a_2} (resp. V_{b_2}): $MV(T_{a_2})$ (resp. $MV(V_{b_2})$),
- a_2^f (resp. b_2^f): les variables de a_2 (resp. b_2) non modifiées par T_{a_2} (resp. V_{b_2}): $FV(T_{a_2}) - MV(T_{a_2})$ (resp. $FV(V_{b_2}) - MV(V_{b_2})$),
- d_1 (resp. d_2): les variables de $a_1 \cup b_1$ (resp. $a_2 \cup b_2$) non modifiées par $S_{a_1} \| U_{b_1}$ (resp. $T_{a_2} \| V_{b_2}$).

On a donc (d'après les définitions A.2.1 et A.2.2):

$$\begin{array}{lll} a_1 = a_1^m \cup a_1^f & b_1 = b_1^m \cup b_1^f & d_1 = (a_1^f \cup b_1^f) - (a_1^m \cup b_1^m) \\ a_2 = a_2^m \cup a_2^f & b_2 = b_2^m \cup b_2^f & d_2 = (a_2^f \cup b_2^f) - (a_2^m \cup b_2^m) \end{array}$$

on a (par définition):

$$\begin{array}{c} Trm(S_{a_1} \| U_{b_1}) = Trm(S_{a_1}) \wedge Trm(U_{b_1}) \\ Trm(T_{a_2} \| V_{b_2}) = Trm(T_{a_2}) \wedge Trm(V_{b_2}) \end{array}$$

et

$$\begin{array}{c} Prd(S_{a_1} \| U_{b_1}) = Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \\ Prd(T_{a_2} \| V_{b_2}) = Prd_{a_2^m, a_2^{m\prime}}(T_{a_2}) \wedge Prd_{b_2^m, b_2^{m\prime}}(V_{b_2}) \wedge d_2' = d_2 \end{array}$$

Le but à prouver devient donc:

$$\begin{array}{r} (Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1, c_2)) \rightarrow \\ [Trm(T_{a_2}) \wedge Trm(V_{b_2}) \wedge \\ \forall\, c_2''.(Prd_{a_2^m, a_2^{m\prime\prime}}(T_{a_2}) \wedge Prd_{b_2^m, b_2^{m\prime\prime}}(V_{b_2}) \wedge d_2'' = d_2 \rightarrow \\ \exists\, c_1'.(Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \wedge J(c_1', c_2'')))] \end{array} \tag{G$_3$}$$

Par les règles de déduction et de l'élimination du quantificateur universel, on obtient les trois sous buts suivants:

$$Trm(T_{a_2}) \tag{G$_4$}$$

$$Trm(V_{b_2}) \tag{G$_5$}$$

$$\begin{array}{r} (Prd_{a_2^m, a_2^{m\prime\prime}}(T_{a_2}) \wedge Prd_{b_2^m, b_2^{m\prime\prime}}(V_{b_2}) \wedge d_2'' = d_2) \\ \rightarrow \exists\, c_1'.(Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \wedge J(c_1', c_2'')) \end{array} \tag{G$_6$}$$

sous les hypothèses supplémentaires:

$$Trm(S_{a_1}) \tag{H$_5$}$$

$$Trm(U_{b_1}) \tag{H$_6$}$$

$$J(c_1, c_2) \tag{H$_7$}$$

Preuve de (G$_4$): pour prouver (G$_4$) par réutilisation de (H$_3$), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(S_{a_1})$ et $J(a_1, a_2)$. $Trm(S_{a_1})$ étant l'hypothèse (H$_5$), il reste à déduire de notre environnement d'hypothèses, constitué de (H$_5$), (H$_6$) et (H$_7$), l'hypothèse:

$$J(a_1, a_2) \tag{H$_8$}$$

L'une des conditions nécessaires et suffisantes à la réutilisation est donc:

$$\boxed{(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1, c_2)) \rightarrow J(a_1, a_2)}$$

(H$_5$), (H$_8$) et (H$_3$) prouvent alors (G$_4$) et génèrent une hypothèse supplémentaire:

$$\forall\, a_2''.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1', a_2''))) \tag{H$_9$}$$

en décomposant les variables a_2 en a_2^m et a_2^f, cette dernière hypothèse se réécrit en:

$$\forall\, a_2^{m\prime\prime}, a_2^{f\prime\prime}.((Prd_{a_2^m, a_2^{m\prime\prime}}(T_{a_2}) \wedge a_2^{f\prime\prime} = a_2^f) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1', a_2^{m\prime\prime} \cup a_2^{f\prime\prime}))) \tag{H$_{10}$}$$

Remarque: La condition (H$_8$) n'est nécessaire que pour la réutilisation de la preuve de correction du raffinement $S_{a_1} \sqsubseteq T_{a_2}$: T_{a_2} peut éventuellement terminer sans que cette condition soit vérifiée: $Trm(T_{a_2}) = True$.

Preuve de (G$_5$): pour prouver (G$_5$) par réutilisation de (H$_4$), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(U_{b_1})$ et $J(b_1,b_2)$. $Trm(U_{b_1})$ étant l'hypothèse (H$_6$), il reste à déduire de notre environnement d'hypothèses, constitué de (H$_5$), (H$_6$) et (H$_7$), l'hypothèse:

$$J(b_1,b_2) \tag{H$_{11}$}$$

Une deuxième condition nécessaire et suffisante à la réutilisation est d'avoir:

$$\boxed{(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2)) \rightarrow J(b_1,b_2)}$$

(H$_6$), (H$_{11}$) et (H$_4$) prouvent alors (G$_5$) et génèrent une hypothèse supplémentaire:

$$\forall\, b_2''.(Prd(V_{b_2}) \rightarrow \exists\, b_1'.(Prd(U_{b_1}) \wedge J(b_1', b_2''))) \tag{H$_{12}$}$$

en décomposant les variables b_2 en b_2^m et b_2^f, cette dernière hypothèse se réécrit en:

$$\forall\, b_2^{m\prime\prime}, b_2^{f\prime\prime}.((Prd_{b_2^m, b_2^{m\prime\prime}}(V_{b_2}) \wedge b_2^{f\prime\prime} = b_2^f) \rightarrow \exists\, b_1'.(Prd(U_{b_1}) \wedge J(b_1', b_2^{m\prime\prime} \cup b_2^{f\prime\prime}))) \tag{H$_{13}$}$$

Remarque: La condition (H$_{11}$) n'est nécessaire que pour la réutilisation de la preuve de correction du raffinement $U_{b_1} \sqsubseteq V_{b_2}$: V_{b_2} peut éventuellement terminer sans que cette condition soit vérifiée: $Trm(V_{b_2}) = True$.

Preuve de (G$_6$): par l'application de la règle de déduction, le but (G$_6$) devient:

$$\exists\, c_1'.(Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \wedge J(c_1', c_2'')) \tag{G$_{6.1}$}$$

sous les hypothèses supplémentaires:

$$Prd_{a_2^m, a_2^{m\prime\prime}}(T_{a_2}) \tag{H$_{14}$}$$
$$Prd_{b_2^m, b_2^{m\prime\prime}}(V_{b_2}) \tag{H$_{15}$}$$
$$d_2'' = d_2 \tag{H$_{16}$}$$

Afin de réutiliser les hypothèses (H$_{10}$) et (H$_{13}$), il faudrait que les deux hypothèses:

$$a_2^{f\prime\prime} = a_2^f \tag{H$_{17}$}$$
$$b_2^{f\prime\prime} = b_2^f \tag{H$_{18}$}$$

apparaissent dans ($d_2'' = d_2$). Ces hypothèses indiquent que chaque variable a_2 (resp. b_2) fixe pour la substitution T_{a_2} (resp. V_{b_2}) est aussi fixe pour la substitution ($T_{a_2} \| V_{b_2}$). Pour cela, on doit avoir:

$$\boxed{a_2^f \cap b_2^m = \emptyset \qquad et \qquad b_2^f \cap a_2^m = \emptyset}$$

(H$_{14}$), (H$_{17}$) et l'instanciation de (H$_{10}$) sur $a_2^{f\prime\prime}$ et $a_2^{m\prime\prime}$ donnent:

$$\exists\, a_1'.(Prd_{a_1, a_1'}(S_{a_1}) \wedge J(a_1', a_2^{m\prime\prime} \cup a_2^f)) \tag{H$_{19}$}$$

(H$_{15}$), (H$_{18}$) et l'instanciation de (H$_{13}$) sur $b_2^{f\prime\prime}$ et $b_2^{m\prime\prime}$ donnent:

$$\exists\, b_1'.(Prd_{b_1, b_1'}(U_{b_1}) \wedge J(b_1', b_2^{m\prime\prime} \cup b_2^f)) \tag{H$_{20}$}$$

Soient $sol_{a_1'}$ et $sol_{b_1'}$ des valeurs de a_1' et b_1' qui satisfont les hypothèses (H_{19}) et (H_{20}) respectivement. On a donc:

$$[a_1' := sol_{a_1'}]Prd_{a_1,a_1'}(S_{a_1}) \wedge [a_1' := sol_{a_1'}]J(a_1', a_2^{m''} \cup a_2^f) \tag{H$_{21}$}$$

$$[b_1' := sol_{b_1'}]Prd_{b_1,b_1'}(U_{b_1}) \wedge [b_1' := sol_{b_1'}]J(b_1', b_2^{m''} \cup b_2^f) \tag{H$_{22}$}$$

en décomposant les variables a_1' et b_1' en variables modifiées et fixes, ces hypothèses se réécrivent en:

$$[a_1^{m\prime} := sol_{a_1^{m\prime}}]Prd_{a_1^m,a_1^{m\prime}}(S_{a_1}) \wedge a_1^{f\prime} = a_1^f \wedge [a_1' := sol_{a_1'}]J(a_1', a_2^{m''} \cup a_2^f) \tag{H$_{23}$}$$

$$[b_1^{m\prime} := sol_{b_1^{m\prime}}]Prd_{b_1^m,b_1^{m\prime}}(U_{b_1}) \wedge b_1^{f\prime} = b_1^f \wedge [b_1' := sol_{b_1'}]J(b_1', b_2^{m''} \cup b_2^f) \tag{H$_{24}$}$$

Réutiliser la preuve des deux raffinements $S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$ et $U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$ revient à vérifier que les solutions $sol_{a_1'}$ et $sol_{b_1'}$ sont aussi une solution du raffinement parallèle. Pour cela, on doit vérifier que $(sol_{a_1'}, sol_{b_1'})$ satisfait le but ($G_{6.1}$), c.à.d:

$$[a_1' := sol_{a_1'} \| b_1' := sol_{b_1'}](Prd_{a_1^m,a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m,b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \wedge J(a_1^{m\prime} \cup b_1^{m\prime} \cup d_1', c_2'')) \tag{G$_{6.2}$}$$

Les substitutions S_{a_1} et U_{b_1} étant en parallèle, les deux ensembles de variables a_1^m et b_1^m ne peuvent être que disjoints: ($a_1^m \cap b_1^m = \emptyset$). Le but ($G_{6.2}$) se décompose en les buts suivants:

$$[a_1^{m\prime} := sol_{a_1^{m\prime}} \| b_1^{f\prime} := sol_{b_1^{f\prime}}]Prd_{a_1^m,a_1^{m\prime}}(S_{a_1}) \tag{G$_{6.3}$}$$

$$[a_1^{f\prime} := sol_{a_1^{f\prime}} \| b_1^{m\prime} := sol_{b_1^{m\prime}}]Prd_{b_1^m,b_1^{m\prime}}(U_{b_1}) \tag{G$_{6.4}$}$$

$$[a_1^{f\prime} := sol_{a_1^{f\prime}} \| b_1^{f\prime} := sol_{b_1^{f\prime}}](d_1' = d_1) \tag{G$_{6.5}$}$$

$$[a_1' := sol_{a_1'} \| b_1' := sol_{b_1'}]J(a_1^{m\prime} \cup b_1^{m\prime} \cup d_1', a_2^{m''} \cup b_2^{m''} \cup d_2) \tag{G$_{6.6}$}$$

Pour prouver ($G_{6.3}$) (resp. ($G_{6.4}$)) par réutilisation de (H_{23}) (resp. (H_{24})), les deux ensembles de variables a_1^m et b_1^f (resp. a_1^f et b_1^m) doivent être disjoints. On doit donc avoir:

$$\boxed{a_1^f \cap b_1^m = \emptyset \qquad et \qquad b_1^f \cap a_1^m = \emptyset}$$

Le but ($G_{6.5}$) est trivial car les variables d_1 désignent les variables communes fixes de a_1 et b_1. Enfin, le but ($G_{6.6}$) doit pouvoir être déduit de l'ensemble des hypothèses constituant l'environnement de preuve. On doit donc prouver:

$$\boxed{\begin{array}{r} (Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge Prd_{a_2^m,a_2^{m''}}(T_{a_2}) \wedge Prd_{b_2^m,b_2^{m''}}(V_{b_2}) \\ [a_1^{m\prime} := sol_{a_1^{m\prime}}]Prd_{a_1^m,a_1^{m\prime}}(S_{a_1}) \wedge [b_1^{m\prime} := sol_{b_1^{m\prime}}]Prd_{b_1^m,b_1^{m\prime}}(U_{b_1}) \wedge [a_1' := sol_{a_1'}]J(a_1', a_2^{m''} \cup a_2^f) \wedge \\ [b_1' := sol_{b_1'}]J(b_1', b_2^{m''} \cup b_2^f)) \rightarrow [a_1^{m\prime} := sol_{a_1'} \| b_1^{m\prime} := sol_{b_1'}]J(a_1^{m\prime} \cup b_1^{m\prime} \cup a_1^f \cup b_1^f, a_2^{m''} \cup b_2^{m''} \cup a_2^f \cup b_2^f) \end{array}}$$

Résumé

Soient S_{a_1}, T_{a_2}, U_{b_1}, V_{b_2} quatre substitutions et $J(a_1,a_2)$, $J(b_1,b_2)$ deux prédicats tels que :

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$$
$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$$

Soit $J(a_1 \cup b_1, a_2 \cup b_2)$ un prédicat vérifiant ($c_i = a_i \cup b_i$):

$i.$ $a_1^f \cap b_1^m = \emptyset$

$ii.$ $b_1^f \cap a_1^m = \emptyset$

$iii.$ $a_2^f \cap b_2^m = \emptyset$

$iv.$ $b_2^f \cap a_2^m = \emptyset$

$v.$ $(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2)) \rightarrow J(a_1,a_2)$

$vi.$ $(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2)) \rightarrow J(b_1,b_2)$

$vii.$ Soient a_2'' et b_2'' les valeurs respectivement de a_2 et b_2 résultant de l'application de $(T_{a_2} \| V_{b_2})$. Les valeurs a_1' et b_1' respectivement de a_1 et b_1 résultant de l'application de S_{a_1} et U_{b_1} respectivement, et vérifiant:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$$
$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$$

et

$$\begin{array}{c}(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge Prd_{a_2^m,a_2^{m\prime\prime}}(T_{a_2}) \wedge Prd_{b_2^m,b_2^{m\prime\prime}}(V_{b_2}) \\ Prd_{a_1^m,a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m,b_1^{m\prime}}(U_{b_1}) \wedge J(a_1',a_2^{m\prime\prime} \cup a_2^f) \wedge J(b_1',b_2^{m\prime\prime} \cup b_2^f)) \\ \rightarrow J(a_1^{m\prime} \cup b_1^{m\prime} \cup a_1^f \cup b_1^f, a_2^{m\prime\prime} \cup b_2^{m\prime\prime} \cup a_2^f \cup b_2^f)\end{array}$$

peuvent être réutilisées pour prouver: $(S_{a_1} \| U_{b_1}) \sqsubseteq_{J(c_1,c_2)} (T_{a_2} \| V_{b_2})$.

Remarques

1. Les quatre premières conditions sont des conditions nécessaires de réutilisation. En effet, si une preuve de raffinement de $S_{a_1} \sqsubseteq T_{a_2}$ (resp. $U_{b_1} \sqsubseteq V_{b_2}$) est établie pour une variable x qui est non modifiée, alors cette preuve ne peut être réutilisée que dans un environnement où cette variable n'est pas modifiée. Dans le cas contraire la preuve ne serait plus valide.
2. La condition (v) (resp. (vi)) est une condition nécessaire de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $S_{a_1} \sqsubseteq T_{a_2}$ (resp. $U_{b_1} \sqsubseteq V_{b_2}$), alors il faut être capable d'exhiber dans l'environnement de preuve de $(S_{a_1} \| U_{b_1} \sqsubseteq T_{a_2} \| V_{b_2})$ l'invariant de collage de $S_{a_1} \sqsubseteq T_{a_2}$ (resp. $U_{b_1} \sqsubseteq V_{b_2}$). Dans le cas contraire la stratégie de réutilisation n'est pas justifiée.
3. La condition (vii) est une condition suffisante de correction du raffinement mais pas nécessaire. En effet, cette condition n'est nécessaire que pour la réutilisation des solutions des raffinements préétablis. Une autre solution du raffinement sans réutilisation peut éventuellement exister.
4. Si les deux substitutions S_{a_1} et U_{b_1} sont déterministes alors la condition (vii) est vérifiée. En effet, il existe une seule valeur du couple (a_1',b_1') qui satisfait les raffinements préétablis, celle-ci est déterminée par $(Prd(S_{a_1}),Prd(U_{b_1}))$. De plus, une solution possible du raffinement $(S_{a_1} \| U_{b_1} \sqsubseteq_{J(c_1,c_2)} T_{a_2} \| V_{b_2})$ doit satisfaire $Prd(S_{a_1})$ et $Prd(U_{b_1})$ qui n'admettent qu'une seule solution, donc celle-ci est réutilisable.
5. Si les deux prédicats $J(a_1,a_2)$ et $J(b_1,b_2)$ sont fonctionnels en a_1 et b_1 respectivement, alors la condition (vii) est vérifiée. En effet, une solution possible du raffinement parallèle doit satisfaire le prédicat $J(c_1,c_2)$. D'après (v) et (vi) cette solution satisfait les prédicats $J(a_1,a_2)$ et $J(b_1,b_2)$ qui n'admettent qu'une solution unique (a_1',b_1'). La réutilisation est donc réalisable.

Le tableau de la page 141 résume les conditions de réutilisation des preuves de raffinement pour les différents opérateurs B.

Substitution	Conditions nécessaires et suffisantes de réutilisation
$P_{c_1} \mid S_{a_1} \sqsubseteq_{J(a_1\cup c_1,a_2\cup c_2)} Q_{c_2} \mid T_{a_2}$	$((P_{c_1} \wedge J(a_1 \cup c_1,a_2 \cup c_2)) \rightarrow Q_{c_2}$ $J(a_1 \cup c_1,a_2 \cup c_2) \rightarrow J(a_1,a_2)$ la valeur $sol_{a_1{}'}$ vérifiant $J(a_1{}' \cup (c_1 - a_1),a_2{}' \cup (c_2 - a_2))$ peut être réutilisée pour l'établissement de la preuve.
$S_{a_1} \parallel U_{b_1} \sqsubseteq_{J(a_1\cup b_1,a_2\cup b_2)} T_{a_2} \parallel V_{b_2}$	$a_1{}^m \cap b_1 = \emptyset$, $a_1{}^f \cap b_1{}^m = \emptyset$ $a_2{}^m \cap b_2 = \emptyset$, $a_2{}^f \cap b_2{}^m = \emptyset$ $J(a_1 \cup b_1,a_2 \cup b_2) \rightarrow J(a_1,a_2)$, $J(a_1 \cup b_1,a_2 \cup b_2) \rightarrow J(b_1,b_2)$ le couple de valeurs $(sol_{a_1{}'},sol_{b_1{}'})$ vérifiant $J(a_1{}' \cup b_1',a_2{}' \cup b_2')$ peut être réutilisée pour l'établissement de la preuve.
IF P_{c_1} **THEN** S_{a_1} **ELSE** T_{a_2} **END** $\sqsubseteq_{J(a_1\cup b_1\cup c_1,a_2\cup b_2\cup c_2)}$ **IF** Q_{c_2} **THEN** U_{b_1} **ELSE** V_{b_2} **END**	$J(a_1 \cup b_1 \cup c_1,a_2 \cup b_2 \cup c_2) \rightarrow (P_{c_1} \leftrightarrow Q_{c_2})$ $(J(a_1 \cup b_1 \cup c_1,a_2 \cup b_2 \cup c_2) \wedge P_{c_1}) \rightarrow J(a_1,a_2)$ $(J(a_1 \cup b_1 \cup c_1,a_2 \cup b_2 \cup c_2) \wedge \neg P_{c_1}) \rightarrow J(b_1,b_2)$ si P_{c_1} est vérifié, la valeur $sol_{a_1{}'}$ vérifiant $J(a_1{}' \cup ((b_1 \cup c_1) - a_1),a_2{}' \cup ((b_2 \cup c_2) - a_2))$ peut être réutilisée pour l'établissement de la preuve. si $\neg P_{c_1}$ est vérifié, la valeur $sol_{b_1{}'}$ vérifiant $J(((a_1 \cup c_1) - b_1) \cup b_1{}',((a_2 \cup c_2) - b_2) \cup b_2{}')$ peut être réutilisée pour l'établissement de la preuve.
$S_{a_1} [] U_{b_1} \sqsubseteq_{J(a_1\cup b_1,a_2\cup b_2)} T_{a_2} [] V_{b_2}$	$J(a_1 \cup b_1,a_2 \cup b_2) \rightarrow J(a_1,a_2)$ $J(a_1 \cup b_1,a_2 \cup b_2) \rightarrow J(b_1,b_2)$ si $Prd(T_{a_2})$ est vérifié, la valeur $sol_{a_1{}'}$ vérifiant $J(a_1{}' \cup (b_1 - a_1),a_2{}' \cup (b_2 \cup -a_2))$ peut être réutilisée pour l'établissement de la preuve. si $Prd(V_{b_2})$ est vérifié, la valeur $sol_{b_1{}'}$ vérifiant $J((a_1 - b_1) \cup b_1{}',(a_2 - b_2) \cup b_2{}')$ peut être réutilisée pour l'établissement de la preuve. si $(Prd(T_{a_2}) \wedge Prd(V_{b_2}) \wedge (b_2 - a_2) = \emptyset)$ est vérifié, $sol_{b_1{}'}$ vérifiant $J(a_1{}' \cup b_1{}',a_2{}')$ peut être réutilisée pour l'établissement de la preuve. si $(Prd(T_{a_2}) \wedge Prd(V_{b_2}) \wedge (a_2 - b_2) = \emptyset)$ est vérifié, $sol_{a_1{}'}$ vérifiant $J(a_1{}' \cup b_1{}',b_2{}')$ peut être réutilisée pour l'établissement de la preuve.
$S_{a_1};\ U_{b_1} \sqsubseteq_{J(a_1\cup b_1,a_2\cup b_2)} T_{a_2};\ V_{b_2}$	$J(a_1 \cup b_1,a_2 \cup b_2) \rightarrow J(a_1,a_2)$ pour chaque valeur (x,y) de $(a_2{}',a_2{}')$: $([a_1{}' := x]Prd(S_{a_1}) \wedge [a_2{}' := y]Prd(T_{a_2}) \wedge [a_1{}' := x;\ a_2{}' := y]J(a_1{}',a_2{}')) \rightarrow [a_1 := x;\ a_2 := y]J(b_1,b_2)$ pour chaque valeur $(a_2 \cup b_2)''$ vérifiant $Prd(T_{a_2};\ V_{b_2})$ associée à la valeur t de $a_2{}'$ tel que $([a_2{}' := t]Prd(T_{a_2}) \wedge [a_2 := t]Prd(V_{b_2}))$, les valeurs $(sol_{a_1{}'},sol_{b_1{}'})$ vérifiant: – $sol_{a_1{}'}$ prouve le raffinement $S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$ pour $a_2{}'$ égal à t. – $sol_{b_1{}'}$ prouve de raffinement $U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$ pour a_1 et a_2 égaux à $sol_{a_1{}'}$ et t respectivement. $-J((a_1 - b_1)'' \cup b_1{}'',a_2{}'' \cup b_2{}'')$ peuvent être réutilisées pour l'établissement de la preuve.
$S_{a_1} \parallel U_{b_1} \sqsubseteq_{J(a_1\cup b_1,a_2\cup b_2)} T_{a_2};\ V_{b_2}$	$J(a_1 \cup b_1,a_2 \cup b_2) \rightarrow J(a_1,a_2)$ $a_1{}^m \cap b_1 = \emptyset$, $a_1 \cap b_1{}^m = \emptyset$ pour chaque valeur y de $a_2{}'$, on a: $[a_2{}' := y]Prd(T_{a_2}) \rightarrow [a_2 := y]J(b_1,b_2)$ pour chaque valeur de $(a_2 \cup b_2)''$ qui satisfait $Prd(T_{a_2};\ V_{b_2})$ associée à la valeur t de $a_2{}'$ tel que $([a_2{}' := t]Prd(T_{a_2}) \wedge [a_2 := t]Prd(V_{b_2}))$, les valeurs $(sol_{a_1{}'},sol_{b_1{}'})$ vérifiant: – $sol_{a_1{}'}$ prouve le raffinement $S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$ pour $a_2{}'$ égale à t. – $sol_{b_1{}'}$ prouve le raffinement $U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$ pour a_2 égal à t. $-J(a_1{}' \cup b_1{}',a_2{}'' \cup b_2{}'')$ peuvent être réutilisées pour l'établissement de la preuve.
$@(z.(P_{b_1} \Rightarrow S_{a_1}))$ $\sqsubseteq_{J(a_1\cup b_1,a_2\cup b_2)}$ $@(z.(Q_{b_2} \Rightarrow T_{a_2}))$	$(Q_{b_2} \wedge J(a_1 \cup b_1,a_2 \cup b_2)) \rightarrow P_{b_1}$ $(Q_{b_2} \wedge Trm(S_{a_1}) \wedge J(a_1 \cup b_1,a_2 \cup b_2)) \rightarrow J(a_1,a_2)$ Soit z un sous-ensemble des variables non raffinées de $(a_1{}^f \cap b_1)$. Pour toute valeur z_0 de z vérifiant: $(Q_{b_2} \wedge Prd(T_{a_2}))$ si l'une des valeurs a_1', associée à z_0, vérifiant: $S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$ vérifie: $J(a_1{}' \cup (b_1 - a_1),a_2{}'' \cup (b_2 - a_2)'')$ alors celle-ci peut être réutilisée pour prouver $(@\ z.(P_{b_1} \Rightarrow S_{a_1})) \sqsubseteq_{J(a_1\cup b_1,a_2\cup b_2)} (@\ z.(Q_{b_2} \Rightarrow T_{a_2}))$ pour la même valeur z_0 de z.

La section suivante présentera une procédure permettant de généraliser les résultats obtenus aux cas de raffinement mettant en jeu plusieurs règles de raffinement. Plus précisément, nous décrivons la façon dont la preuve du raffinement d'une substitution complexe est obtenue à partir des preuves de raffinement des substitutions qui la composent. L'approche que nous développons est similaire à celle décrite dans [25].

6.5 Cas général: substitutions complexes

Dans les sections précédentes, nous avons étudié la réutilisation des preuves de raffinement de substitution déjà établies pour la validation du raffinement d'une nouvelle substitution introduisant un seul opérateur. Nous montrons dans la présente section comment généraliser les résultats obtenus au cas de substitutions comportant plusieurs opérateurs. L'idée de base de notre approche est de se ramener aux cas préalablement étudiés. Le raffinement à prouver doit pouvoir être décomposé en plusieurs sous-raffinements plus élémentaires. Chaque nouveau raffinement ne doit contenir qu'un seul opérateur. Pour cela, la substitution à raffiner et celle raffinant doivent être décomposées en plusieurs sous-parties. Chaque partie de la substitution abstraite raffine une partie de la substitution concrète par une des règles de la base que nous avons définies aux deux chapitres précédents.

La majorité des opérateurs B (à l'exception du **IF**[5]) étant binaires, nous représentons graphiquement les substitutions en utilisant les arbres binaires. Les sous-raffinements sont obtenus par mise en correspondance des deux arbres abstrait et concret. Une condition nécessaire à cette procédure de mise en correspondance est que les deux arbres aient des structures comparables (structure similaire à l'opérateur près). L'approche que nous proposons peut être résumée par les étapes suivantes:

– représentation graphique, par des arbres binaires, des substitutions abstraite et concrète.
– filtrage des raffinements préétablis puis réduction des arbres concret et abstrait.
– restructuration de l'arbre concret.
– filtrage total des arbres abstrait et concret.
– construction de la solution du raffinement.

6.5.1 Représentation graphique d'une substitution

Une substitution B peut être représentée graphiquement par un arbre binaire. Les nœuds de cet arbre sont les opérateurs B qui la composent. Les feuilles correspondent aux substitutions sur lesquelles est construite la substitution considérée. On considère que les opérateurs sont associatifs à gauche. De même, on considère que le nœud de l'opérateur conditionnel **IF** est indicé par son prédicat, le sous-arbre gauche (resp. droit) correspond au cas où la condition est vraie (resp. fausse). Par exemple, les substitutions (**IF** P **THEN** $Q \mid ((S \| T);\ U)$ **END**) et $(S \| T \| U)$ sont représentées par les arbres de la figure 6.1.

6.5.2 Filtrage des raffinements préétablis et réduction des arbres

Cette étape consiste à identifier les parties des deux arbres qui sont liées par une des règles de raffinement dont nous disposons. Quatre types de filtrage sont à distinguer (chaque partie pouvant être soit une feuille, ou un arbre):

– filtrage feuille/feuille.
– filtrage feuille/arbre.
– filtrage arbre/feuille.

5. La substitution **SELECT** peut être exprimée par des choix sur des substitutions gardées: **SELECT** P **THEN** S **WHEN** Q **THEN** T **ELSE** U **END** $\equiv (P \Rightarrow S)\ []\ (Q \Rightarrow T)\ []\ ((\neg P \wedge \neg Q) \Rightarrow U)$

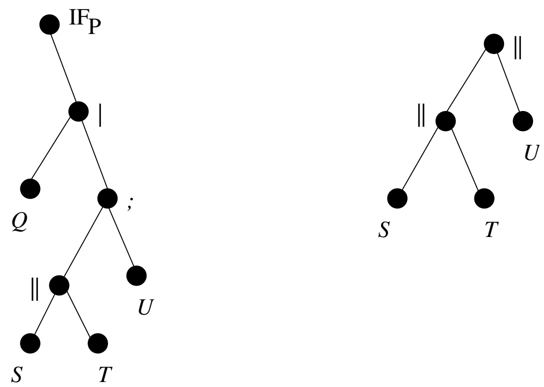


FIG. 6.1 – *Représentation graphique des substitutions (IF P THEN Q* $\mid ((S\|T);\ U)$ *END) et (*$S\|T\|U$*)*

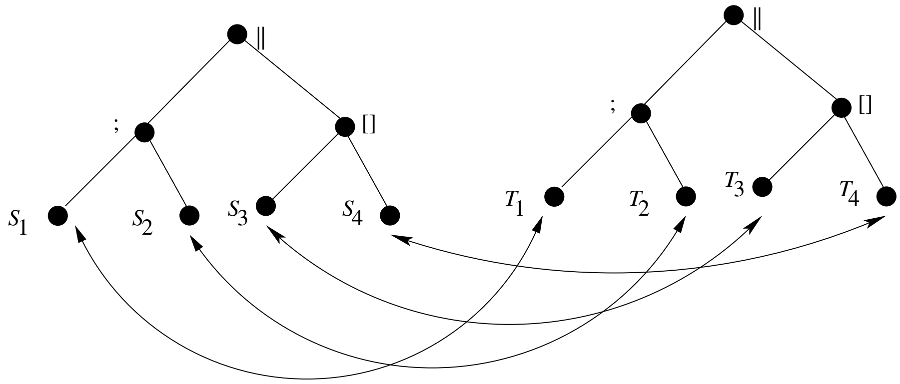


FIG. 6.2 – *Un exemple de filtrage feuille/feuille*

– filtrage arbre/arbre.

Par exemple, si nous avons préalablement établi $(S_i \sqsubseteq T_i)$ pour chaque i $(i = 1..4)$, le filtrage des arbres correspondant aux deux substitutions abstraite et concrète $(SA \equiv (S_1;\ S_2)\|(S_3[]S4))$ et $(SC \equiv (T_1;\ T_2)\|(T_3[]T4))$ respectivement, est représentée par la figure 6.2.

Supposons à présent que notre base de règles contienne les raffinements suivants:

- $(S_1;\ S_2) \sqsubseteq T_1$.
- $S_3 \sqsubseteq T_3$, $S_4 \sqsubseteq T_4$,

Le filtrage des arbres associés à SA et SC est représenté par la figure 6.3.

Une fois les raffinements préalablement établis identifiés et filtrés, les deux arbres sont réduits. On remplace chaque sous-arbre participant à un filtrage par une feuille qui permet de faire abstraction de la structure interne de ces sous-arbres. Les substitutions de ces sous-arbres n'interviennent plus directement dans l'établissement de la preuve de raffinement des autres parties des arbres abstrait et concret. Seules les solutions de raffinement de ces parties sont significatives. La figure 6.4 représente la réduction de l'arbre abstrait de la figure 6.3.

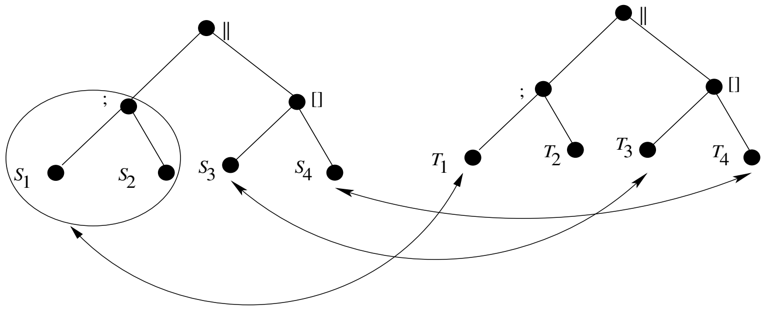


FIG. 6.3 – *Un exemple de filtrage arbre/feuille*

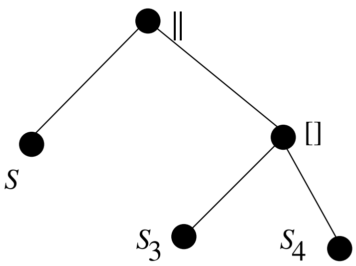


FIG. 6.4 – *L'arbre réduit de l'arbre abstrait de la figure 6.3*

6.5.3 Restructuration de l'arbre concret

Le but de cette étape est l'obtention d'arbres à structures comparables. La restructuration de deux arbres abstrait AA et concret AC est effectuée en les parcourant simultanément de la racine aux feuilles. On définit inductivement la relation de comparaison entre deux arbres de la manière suivante (on note (R,Ar_g,Ar_d) l'arbre de racine R (c'est l'un des opérateurs B), de sous-arbre gauche Ar_g et de sous-arbre droit Ar_d):

- deux feuilles sont comparables.
- $(|\ ,P,Ar_d^1)$ est comparable à $(|\ ,Q,Ar_d^2)$ ssi Ar_d^1 et Ar_d^2 sont comparables.
- $(\|,Ar_g^1,Ar_d^1)$ est comparable à $(\|,Ar_g^2,Ar_d^2)$ ssi Ar_g^1 est comparable à Ar_g^2 <u>et</u> Ar_d^1 est comparable àAr_d^2.
- $(\|,Ar_g^1,Ar_d^1)$ est comparable à $(;\ ,Ar_g^2,Ar_d^2)$ ssi Ar_g^1 est comparable à Ar_g^2 <u>et</u> Ar_d^1 est comparable àAr_d^2.
- (IF_P,Ar_g^1,Ar_d^1) est comparable à (IF_Q,Ar_g^2,Ar_d^2) ssi Ar_g^1 est comparable à Ar_g^2 <u>et</u> Ar_d^1 est comparable àAr_d^2.
- $(;\ ,Ar_g^1,Ar_d^1)$ est comparable à $(;\ ,Ar_g^2,Ar_d^2)$ ssi Ar_g^1 est comparable à Ar_g^2 <u>et</u> Ar_d^1 est comparable àAr_d^2.
- $([],Ar_g^1,Ar_d^1)$ est comparable à $([],Ar_g^2,Ar_d^2)$ ssi Ar_g^1 est comparable à Ar_g^2 <u>et</u> Ar_d^1 est comparable àAr_d^2.

Lors de l'application du processus de raffinement B, l'ordre des substitutions concrètes peut différer de celui des substitutions abstraites qui leurs sont associées. Ceci est particulièrement vrai pour le raffinement des substitutions parallèles et choix. En effet, ces opérateurs sont commutatifs. De plus, le raffinement de la précondition et du choix peut modifier la structure de la substitution abstraite. D'une part, la précondition peut être éliminée, et d'autre part, une structure de choix peut être raffinée par le raffinement de l'une des substitutions abstraites.

Le premier problème peut être résolu en interchangeant les arbres droit et gauche de l'arbre abstrait ou concret. Cette interchangeabilité est décidée par l'identification des raffinements réalisée à la section 6.5.2. Le but de cette interchangeabilité est de rendre les arbres abstrait et concret comparables. À priori lors de la phase d'identification des raffinements, un sous-arbre abstrait droit (resp. gauche) peut filtrer un sous-arbre concret gauche (resp. droit). Néanmoins, un sous-arbre abstrait ne peut filtrer qu'un sous-arbre concret au plus. En effet, sur chaque substitution abstraite, une seule règle de raffinement est applicable. Par conséquent, chaque sous-arbre abstrait ne peut filtrer au plus qu'un sous-arbre concret. Interchanger les deux arbres droit et gauche de l'arbre abstrait ou concret est donc toujours décidable.

Pour le second problème, nous représentons l'élimination de la précondition par un affaiblissement avec précondition égale à *True*. De même, l'élimination du choix est représentée par un choix sur la même substitution. La restructuration de deux arbres abstrait AA et concret AC est effectuée en les parcourant simultanément de la racine aux feuilles tout en appliquant l'algorithme suivant :

Restructurer(AA, AC)
    DEBUT
        $r_a =$ racine de l'arbre AA
        $r_c =$ racine de l'arbre AC
        AA_d,AC_d= sous-arbres droits des arbres AA et AC respectivement
        AA_g,AC_g= sous-arbres gauches de arbres AA et AC respectivement
        SI $(r_a = r_c)$ **ALORS**
            SI $r_a =|$ **ALORS**
                Restructurer(AA_d,AC_d*)*
            SINON
                Normaliser(AA, AC);
                Restructurer(AA_d,AC_d*)*
                Restructurer(AA_d,AC_d*)*
            FSI
        SINON
            Cas où $r_a = ||$:
                        Normaliser(AA, AC);
                        Restructurer(AA_d,AC_d*)*
                        Restructurer(AA_d,AC_d*)*
                    $r_a =|$:
                        $AC = TRUE \mid AC$
                        Restructurer(AA_d,AC_d*)*
                    $r_a = []$
                        $AC = AC[]AC$;
                        Restructurer(AA_d,AC_d*)*;
                        Restructurer(AA_g,AC_g*)*;
                    autres cas : "raffinement incorrect"
            FCas
        FSI
FIN

Normaliser(AA, AC)
    DEBUT
        AA_d,AC_d= sous-arbres droits des arbres AA et AC respectivement
        AA_g,AC_g= sous-arbres gauches des arbres AA et AC respectivement
        SI *une partie de* AA_d *filtre une partie de* AC_g **ALORS** Interchanger les sous-arbres de AC
        FSI
    FIN

Exemple: soient les substitutions abstraite et concrète $SA \equiv (P \mid ((S_1[]S_2) \mid\mid S_3))$ et $SC \equiv (T_3 \| T_1)$ respectivement. On suppose que : $S_i \sqsubseteq T_i$ pour chaque i ($i = 1..3$). La figure 6.5 illustre la restructuration de l'arbre concret associé à la substitution concrète SC.

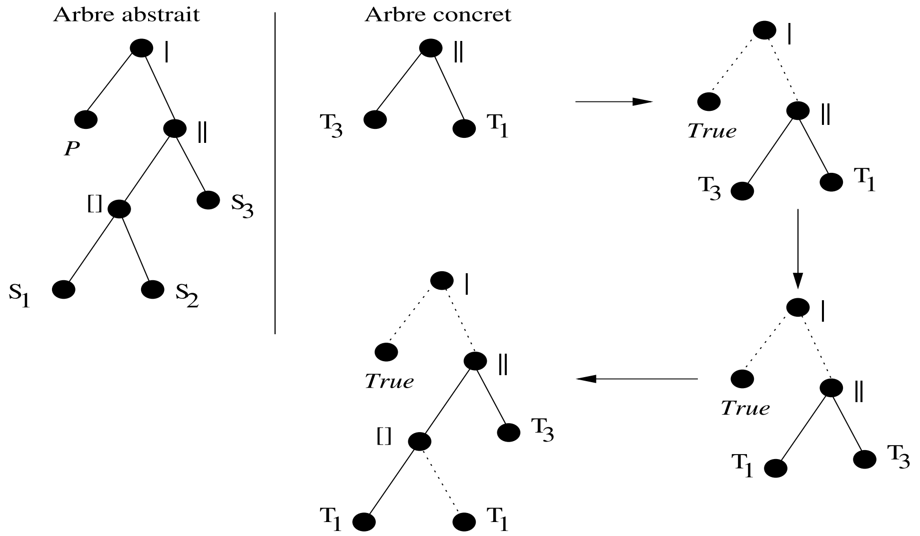


FIG. 6.5 – *Exemple de restructuration de l'arbre concret*

6.5.4 Filtrage total des arbres abstrait et concret

Dans la section 6.5.2, seules les parties auxquelles correspond une règle de raffinement ont été filtrées. Cette étape consiste à étendre cette procédure de filtrage sur la totalité des arbres abstrait et concret. Pour cela, nous utilisons les règles de monotonie du raffinement définie dans [1] et que nous rappelons ci-dessous:

1. Précondition:

$$\frac{P \Rightarrow Q \qquad S \sqsubseteq_J T}{P \mid S \sqsubseteq_J Q \mid T}$$

2. Parallélisme : pour prouver que ($S \| U \sqsubseteq_J T \| V$), il suffit de prouver l'un des séquents suivants :

$$\frac{S \sqsubseteq_J T \qquad U \sqsubseteq_J V}{S \| U \sqsubseteq_J T \| V} \quad \frac{S \sqsubseteq_J V \qquad U \sqsubseteq_J T}{S \| U \sqsubseteq_J T \| V}$$

3. Choix : pour prouver que ($S [] U \sqsubseteq_J T [] V$), il suffit de prouver l'un des séquents suivants :

$$\frac{S \sqsubseteq_J T \qquad U \sqsubseteq_J V}{S [] U \sqsubseteq_J T [] V} \quad \frac{S \sqsubseteq_J T \qquad S \sqsubseteq_J V}{S [] U \sqsubseteq_J T [] V} \quad \frac{S \sqsubseteq_J V \qquad U \sqsubseteq_J T}{S [] U \sqsubseteq_J T [] V} \quad \frac{U \sqsubseteq_J T \qquad U \sqsubseteq_J V}{S [] U \sqsubseteq_J T [] V}$$

4. IF :

$$\frac{P \leftrightarrow Q \qquad S_1 \sqsubseteq_J S_2 \qquad T_1 \sqsubseteq_J T_2}{\textbf{IF } P \textbf{ THEN } S_1 \textbf{ ELSE } T_1 \textbf{ END} \sqsubseteq_J \textbf{IF } Q \textbf{ THEN } S_2 \textbf{ ELSE } T_2 \textbf{ END}}$$

5. Séquencement :

$$\frac{S \sqsubseteq_J T \qquad U \sqsubseteq_J V}{S;\ U \sqsubseteq_J T;V}$$

6. Raffinement d'une parallèle par une séquence : pour prouver que ($S \| U \sqsubseteq_J T;V$), il suffit de prouver l'un des séquents suivants :
$$\frac{S \sqsubseteq_J T \qquad U \sqsubseteq_J V}{S \| U \sqsubseteq_J T;V}$$
sous la condition que les ensembles des variables lues et modifiées de U et S respectivement soient disjoints, ou :
$$\frac{S \sqsubseteq_J V \qquad U \sqsubseteq_J T}{S \| U \sqsubseteq_J T;V}$$
sous la condition que les ensembles des variables lues et modifiées de S et U respectivement soient disjoints.

Comme pour la restructuration de l'arbre concret, le filtrage total des deux arbres est réalisé par le parcours simultané de ces derniers de la racine aux feuilles. À chaque paire de nœuds des deux arbres abstrait et concret, et selon leurs types, l'une des règles sus-citées est appliquée. Bien que certaines règles comportent plusieurs cas, à chaque étape d'identification une seule est valide. En effet, celle-ci est définie de manière unique grâce au filtrage effectuée à la section 6.5.2.

Exemple: on veut prouver par réutilisation que la substitution abstraite $SA \equiv (P \mid ((S_1;S_2)[](S_3 \| S_4)))$ est raffinée par la substitution concrète $SC \equiv (T_3;T_4)$. On suppose l'existence dans notre base de raffinement des règles suivantes:
$$(S_1;S_2) \sqsubseteq T_1$$
$$S_3 \sqsubseteq T_3$$
$$S_4 \sqsubseteq T_4$$
Pour cela, on procède comme suit :

- Représentation graphique des substitutions : les substitutions abstraite et concrète sont représentées par les arbres de la figure 6.6.

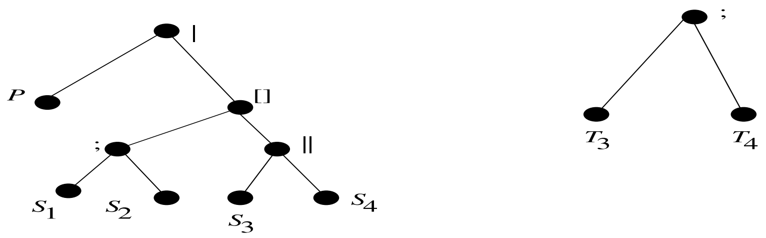


FIG. 6.6 – *Représentation graphique de SA et SC*

- Filtrage des raffinements préétablis et réduction des arbres : la figure 6.7 représente l'identification et le filtrage des raffinements préétablis.
- Restructuration de l'arbre concret : la restructuration de l'arbre concret est représentée par la figure 6.8.
- Filtrage total des deux arbres abstrait et concret: cette étape entraîne la génération des preuves suivantes :
$$P \Rightarrow True \qquad (G_1)$$
$$((S_1;S_2)[](S_3 \| S_4)) \sqsubseteq ((T_3;T_4)[](T_3;T_4)) \qquad (G_2)$$
En appliquant toujours le même principe de filtrage, les arbres correspondant à G_2 génèrent les trois cas suivants :
  – $(S_1;S_2) \sqsubseteq (T_3;T_4)$ et $(S_3 \| S_4) \sqsubseteq (T_3;T_4))$

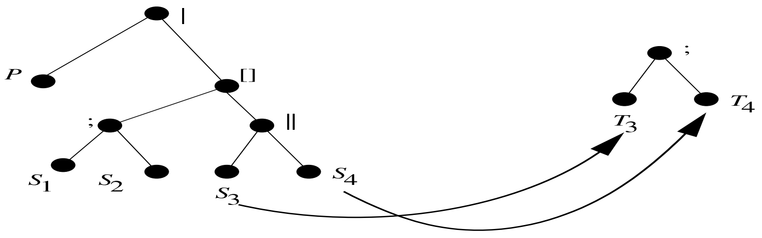


FIG. 6.7 – *Identification et filtrage des raffinements préétablis*

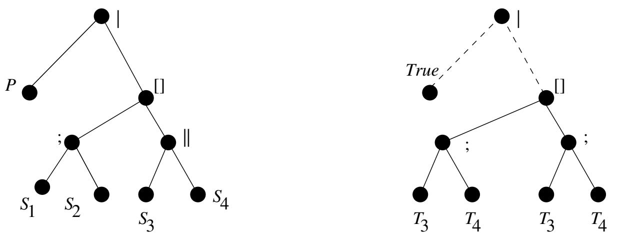


FIG. 6.8 – *Restructuration de l'arbre concret correspondant à SC*

– $(S_3 \| S_4) \sqsubseteq (T_3;T_4)$
– $(S_1;S_2) \sqsubseteq (T_3;T_4)$

Parmi ces cas, seul le deuxième est retenu. Le premier et le troisième cas ne correspondent à aucune règle de raffinement de notre base.

$$(S_3 \| S_4) \sqsubseteq (T_3;\ T_4) \qquad (\mathrm{G}_{2.1})$$

De la même manière, le filtrage des arbres abstrait et concret associés au sous-but $(\mathrm{G}_{2.1})$ génère les sous buts suivants :

$$S_3 \sqsubseteq T_3 \qquad (\mathrm{G}_{2.1.1})$$
$$S_4 \sqsubseteq T_4 \qquad (\mathrm{G}_{2.1.2})$$

En résumé, pour prouver le raffinement $(P \mid ((S_1;S_2)[](S_3\|S_4)) \sqsubseteq (T_3;T_4))$, il suffit d'établir les preuves de (G_1), $(\mathrm{G}_{2.1.1})$, et $(\mathrm{G}_{2.1.2})$. Or les deux derniers buts sont automatiquement déchargés car ils correspondent aux règles de raffinement de notre base et donc ont été déjà prouvés.

6.5.5 Construction de la solution du raffinement

Cette étape permet de construire progressivement la preuve du raffinement à établir. Les sous-buts générés dans l'étape précédente doivent être établis dans l'ordre inverse de leur génération. Pour établir la preuve de $(P \mid ((S_1;\ S_2)[](S_3\|S_4)) \sqsubseteq (T_3;\ T_4))$, on établit successivement les preuves de : $(\mathrm{G}_{2.1.1})$ et $(\mathrm{G}_{2.1.2})$, puis$(\mathrm{G}_{2.1})$, ensuite (G_1) et (G_2), puis enfin la preuve globale du raffinement.

Lors de la preuve d'un sous-but, deux cas sont à distinguer:

- le but à prouver correspond à un raffinement préétabli. Dans ce cas, il suffit de récupérer sa preuve. Aucune contrainte supplémentaire n'est à vérifier.

- le but à prouver correspond à la combinaison de deux raffinements préétablis. Dans ce cas, si les conditions de réutilisation des preuves sont vérifiées, la preuve de raffinement est obtenue par réutilisation des preuves de ces raffinements. Dans le cas contraire, la preuve du but doit être établie indépendamment des preuves déjà établies en utilisant les méthodes classiques de preuve. La preuve ainsi obtenue doit être sauvegardée dans la base pour une éventuelle réutilisation ultérieure.

6.6 Réutilisation de preuves dans le contexte du raffinement base de données

Dans cette section, une application de la stratégie de preuve par réutilisation au domaine du raffinement dédié à la génération d'applications base de données est présentée.

6.6.1 Satisfaction des conditions de réutilisation

Dans les différentes sections précédentes, nous avons étudié, de manière générale (indépendamment de tout domaine d'application), la stratégie de réutilisation de preuves lors de l'établissement de la correction d'un raffinement **B**. Pour chaque opérateur **B**, nous avons défini les conditions nécessaires et suffisantes qui permettent la réutilisation de preuves déjà établies. Pour qu'une preuve de raffinement $(S_{a_1} \sqsubseteq_{J(a_1,b_1)} T_{b_1})$ soit réutilisable, trois conditions sont requises:

1. l'invariant de collage $J(a_1,b_1)$ doit pouvoir être déduit de l'environnement de preuve du raffinement à prouver.
2. la valeur a_1' de la variable a_1 permettant la preuve du raffinement $(S_{a_1} \sqsubseteq_{J(a_1,b_1)} T_{b_1})$ doit vérifier l'invariant de collage du raffinement à prouver.
3. pour le raffinement d'une substitution parallèle par une autre substitution parallèle, les ensembles des variables libres et modifiées des deux substitutions parallèles doivent être disjoints.

Comme nous pouvons le constater, les conditions requises à la réutilisation de preuves sont étroitement liées aux substitutions et aux invariants de collage. Dans cette section, nous montrons que dans le domaine du raffinement dédié au développement d'applications base de données, les substitutions et les invariants de collage de la base de règles que nous avons définie vérifient bien les conditions requises pour la réutilisation. De plus le processus de raffinement défini est fermé. En effet, seules les règles existantes dans la base peuvent être utilisées. Il restera la troisième condition qui elle dépend entièrement de la façon dont l'utilisateur aura spécifié sa substitution abstraite. En particulier, cette condition est non satisfaite pour une transaction qui simultanément modifie la valeur d'un attribut et utilise sa valeur "avant" pour la mise à jour d'un autre attribut. Dans ce cas, la preuve est nécessairement établie sans réutilisation. Néanmoins en pratique, cette catégorie de transactions est peu fréquent.

Vérifions la satisfaction des deux premières conditions de réutilisation sus-citées dans le domaine des applications base de données:

- La première condition relative à l'exhibition de l'invariant de collage de la règle de raffinement est vérifiée car le raffinement étant automatique, le processus génère automatiquement l'invariant de collage de la règle appliquée. En d'autres termes, l'invariant de collage de chaque étape est la conjonction des invariants de collage des différentes règles appliquées. La première condition est donc vérifiée pour chaque étape de raffinement.
- La deuxième condition est aussi vérifiée. En effet, tous les invariants de collage de nos règles de raffinement sont fonctionnels. Or nous avons montré dans chaque cas d'opérateurs **B**, que cette caractéristique était suffisante pour satisfaire la deuxième condition.

6.6.2 Exemple de réutilisation de preuves

Cette section illustre, à travers un exemple, la réutilisation de preuves de raffinement lors du développement d'applications base de données. Pour cela, considérons un système simplifié de gestion de prêts d'un ensemble de cassettes. Une cassette est soit dans une boutique, soit prêtée à un client identifié par un numéro (*NumCl*) (Voir figure 6.9).

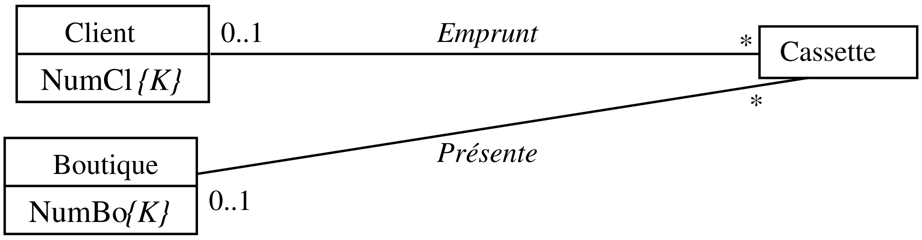


FIG. 6.9 – *Exemple simplifié d'un système de gestion de cassettes*

On se propose de spécifier la transaction de prêt d'une cassette (*PrêtCas*). L'écriture de cette transaction est basée sur la spécification des deux opérations de base *BasicSuppressionPrésente* et *BasicAjoutEmprunt*. Cette transaction est spécifiée par (seules les parties significatives sont reprises, les préconditions sont ignorées):

VARIABLES *cassette, client, boutique, NumCl, NumBo, Emprunt, Présente*
INVARIANT
NumCl $\in$ *client* $\rightarrowtail$ *NAT* $\wedge$
NumBo $\in$ *boutique* $\rightarrowtail$ *NAT* $\wedge$
Emprunt $\in$ *cassette* $\nrightarrow$ *client* $\wedge$
Présente $\in$ *cassette* $\nrightarrow$ *boutique* $\wedge$
...
OPERATIONS
PrêtCas(ca, cl)=
BEGIN
Présente:= {ca} $\lhd$ *Présente* $\|$// substitution de l'opération *BasicSuppressionPrésente*
Emprunt:= Emprunt $\cup$ *{ca* $\mapsto$ *cl}*//substitution de l'opération *BasicAjoutEmprunt*
END

En utilisant la règle de raffinement des associations monovaluées, cette transaction est raffinée en:

PrêtCas(ca, cl)=
BEGIN
PrésenteMig:= {ca} $\lhd$ *PrésenteMig* $\|$
EmpruntMig:= EmpruntMig $\cup$ *{ca* $\mapsto$ *NumCl(cl)}*
END
sous l'invariant de collage:

$$J(a,c) \mathrel{\widehat{=}} (EmpruntMig = (Emprunt;\ NumCl) \wedge PrésenteMig = (Présente;\ NumBo))$$

Nous avons déjà prouvé lors du raffinement des opérations de base que les deux raffinements:

$$Présente := Ca \lhd Présente \sqsubseteq_{J_1} PrésenteMig := Ca \lhd Présente \qquad (H_1)$$

$$Emprunt := Emprunt \cup \{ca \mapsto cl\} \sqsubseteq_{J_2} EmpruntMig := EmpruntMig \cup \{ca \mapsto NumCl(cl)\} \qquad (H_2)$$

sont vérifiés pour:

$$(Présente',NumCl') = (\{ca\} \lhd Présente,NumCl)$$

et

$$(Emprunt',NumCl') = (Emprunt \cup \{ca \mapsto cl\},NumCl)$$

respectivement.

Pour prouver la correction du raffinement de l'opération *PrêtCas*, la preuve suivante doit être établie:

$$[Trm(S_1 \| S_2) \wedge J(a,c)] \rightarrow [Trm(T_1 \| T_2) \wedge \forall c''.(Prd(T_1 \| T_2) \rightarrow \exists a'.(Prd(S_1 \| S_2) \wedge J(a',c'')))] \quad (G_1)$$

avec:

- $S_1 \mathrel{\widehat{=}} (Présente := \{ca\} \lhd Présente)$ et $S_2 \mathrel{\widehat{=}} (Emprunt := Emprunt \cup \{ca \mapsto cl\})$
- $T_1 \mathrel{\widehat{=}} (PrésenteMig := \{ca\} \lhd PrésenteMig)$ et $T_2 \mathrel{\widehat{=}} (EmpruntMig := EmpruntMig \cup \{ca \mapsto NumCl(cl)\})$
- $a \mathrel{\widehat{=}} \{Présente,Emprunt,NumCl,NumBo\}$ et $c \mathrel{\widehat{=}} \{PrésenteMig,EmpruntMig,NumCl,NumBo\}$

Le but (G_1) est équivalent à:

$$[Trm(S_1) \wedge Trm(S_2) \wedge J(a,c)] \rightarrow [Trm(T_1) \wedge Trm(T_2) \wedge \forall c''.(Prd(T_1) \wedge Prd(T_2) \rightarrow \exists a'.(Prd(S_1) \wedge Prd(S_2) \wedge J(a',c'')))] \quad (G_2)$$

En effet, les ensembles de variables modifiées par la substitution S_1 (resp. S_2, T_1, T_2) et fixes par la substitution S_2 (resp. S_1, T_2, T_1) étant disjoints, le prédicat $Prd(S_1 \| S_2)$ (resp. $Prd(T_1 \| T_2)$) peut se réécrire en $(Prd(S_1) \wedge Prd(S_2))$ (resp. $(Prd(T_1) \wedge Prd(T_2))$). Le but G_2 est équivalent aux sous-buts suivants:

$$Trm(T_1) \quad (G_{2.1})$$
$$Trm(T_2) \quad (G_{2.2})$$
$$\exists a'.(Prd(S_1) \wedge Prd(S_2) \wedge J(a',c'')) \quad (G_{2.3})$$

sous les hypothèses:

$$Trm(S_1) \quad (H_3)$$
$$Trm(S_2) \quad (H_4)$$
$$J(a,c) \quad (H_5)$$
$$Prd(T_1) \quad (H_6)$$
$$Prd(T_2) \quad (H_7)$$

La preuve des buts ($G_{2.1}$) et ($G_{2.2}$) ne pose aucun problème. En effet, les substitutions T_1 et T_2 étant des affectations bien typées, celles-ci terminent toujours. Prouver le but ($G_{2.3}$) revient à exhiber des valeurs de $Présente'$, $Emprunt'$ et $NumCl'$ qui le satisfont.

Le sous-but ($G_{2.3}$) se réécrit en

$$\exists Emprunt',Présente',NumCl',NumBo'.(Prd(S_1) \wedge Prd(S_2) \wedge J_1(a_1',c_1'') \wedge J_2(a_2',c_2'')) \quad (G_{2.4})$$

avec:

- $a_1 \mathrel{\widehat{=}} \{Présente, NumBo\}$, $c_1 \mathrel{\widehat{=}} \{PrésenteMig, NumBo\}$
- $a_2 \mathrel{\widehat{=}} \{Emprunt, NumCl\}$, $c_2 \mathrel{\widehat{=}} \{EmpruntMig, NumCl\}$

- $J_1(a_1,c_1) \mathrel{\widehat{=}} (PrésenteMig = (Présente;\ NumBo))$
- $J_2(a_2',c_2) \mathrel{\widehat{=}} (EmpruntMig = (Emprunt;\ NumCl))$

La transaction *PrêtCas* est une combinaison des deux substitutions S_1 et S_2. De même, son raffinement est une combinaison des substitutions qui raffinent les substitutions S_1 et S_2. On peut donc envisager de réutiliser les preuves de raffinement des substitutions S_1 et S_2 pour l'établissement de la correction du raffinement de l'opération *PrêtCas*.

Prouver le but ($G_{2.4}$) par réutilisation de (H_1) et (H_2) se réduit donc à vérifier si leurs solutions le satisfont. Il faut donc prouver que:

$$[Présente' := \{ca\} \lhd Présente \| NumBo' := NumBo](Prd(S_1) \wedge J_1(a_1', c_1''))$$
$$\wedge$$
$$[Emprunt' := Emprunt \cup \{ca \mapsto cl\} \| NumCl' := NumCl)](Prd(S_2) \wedge J_2(a_2', c_2''))$$

Ces mêmes preuves ont été réalisées lors de la preuve des raffinements des deux substitutions de base S_1 et S_2. Elle sont donc déchargées automatiquement.

À travers cet exemple, nous avons montré que les preuves (solutions) de raffinement des substitutions de base peuvent être réutilisées pour la preuve de raffinement d'une substitution combinaison de deux substitutions de base. En effet, la preuve de raffinement de la substitution composée se ramène à la preuve de raffinement des substitutions qui la composent. De plus, nous avons montré que les conditions de réutilisation de preuves de raffinements sont toujours satisfaites dans le domaine des applications base de données.

6.7 Conclusion

Dans les deux chapitres précédents, a été présenté un ensemble de règles de raffinement **B** dédiées à la génération d'une implémentation relationnelle. Pour chaque règle ainsi définie, une preuve de correction a été établie. Chaque règle a été prouvée de manière indépendante des autres règles. Les règles de raffinement opèrent sur un ensemble sélectionné de substitutions dites de base. Ces dernières forment un système complet pour la description d'applications base de données: toute application base de données peut être exprimée par combinaison de ces substitutions via les différents opérateurs de substitutions. Le raffinement B étant monotone, le développement de telles applications par raffinement se ramène au raffinement des substitutions de base. Pour valider un tel développement, un ensemble de preuves est à établir. Le processus de preuve étant long et fastidieux, nous avons étudié l'opportunité de réutilisation des preuves élémentaires déjà établies. Pour chaque opérateur B, nous avons défini, de manière formelle, les conditions nécessaires et suffisantes qui rendent possible la réutilisation de ces preuves élémentaires. De manière générale, ces conditions se résument à vérifier qu'une valeur d'une variable donnée satisfait une formule logique. Or, il est plus facile de vérifier qu'une valeur satisfait une formule que de générer l'une des valeurs pour lesquelles cette formule est vérifiée.

Après une étude générale (indépendante de tout domaine d'application) de ces conditions, on a montré que les conditions requises à la réutilisation sont vérifiées par la base de raffinement que nous avons définie.

Chapitre 7

Un prototype d'outil

Dans les chapitres 3, 4 et 5, nous avons présenté une approche formelle de développement d'applications bases de données par raffinement. Cette approche comporte deux étapes complémentaires de développement. La première étape concerne la traduction d'un ensemble de concepts **UML**, adaptés à la description des applications bases de données, en spécifications formelles **B**. La seconde, basée sur la première, consiste à raffiner les spécifications formelles B obtenues afin de générer une implémentation relationnelle de l'application considérée.

L'utilisation des méthodes formelles en industrie engendre un coût temps/prix non négligeable. En effet, les concepteurs étant habitués à l'utilisation des approches classiques basées essentiellement sur des notations graphiques et l'expérience acquise, leur adhésion à l'utilisation des méthodes formelles demanderait des formations longues et approfondies sur des concepts mathématiques même assez simples. De ce fait actuellement, l'utilisation des méthodes formelles est plutôt restreinte aux domaines critiques. Néanmoins, la mise en œuvre d'outils d'aide au développement permet d'atténuer cette difficulté d'adhésion des concepteurs à l'utilisation de ce type de méthodes. La disposition de tels outils permet de cacher au concepteur toute la théorie sous-jacente à la méthode formelle utilisée. Le concepteur n'est plus contraint de posséder des connaissances accrues en mathématiques.

Dans l'optique de faciliter la mise en pratique de l'approche que nous avons développée, un outil supportant les deux étapes de développement a été réalisé. Cet outil permet l'automatisation des parties du développement ne demandant pas l'intervention de l'utilisateur.

Les étapes de la dérivation outillée d'une implémentation relationnelle à partir d'une spécification **UML** sont représentées par la figure 7.1 :

1. Les diagrammes **UML** sont saisis en utilisant l'environnement **ROSE**[58] de construction de modèles **UML**. Cet environnement est largement utilisé en industrie. Les informations non graphiquement exprimables sont saisies (en notations **B**) par l'utilisateur dans une étape préalable à la phase de génération de spécifications **B**. Ces informations sont enregistrées dans le champ information de l'élément considéré. Par exemple, les informations relatives à la multiplicité, mutabilité et l'appartenance à la clé d'un attribut sont stockées dans le champs documentation de l'attribut considéré (Voir figure 7.3).
2. Les différentes informations véhiculées par les diagrammes **UML** établis à l'étape précédente sont récupérées en utilisant un programme script **ROSE**. Le langage de script **ROSE**, basé sur la syntaxe du langage BASIC, contient un ensemble de fonctions permettant l'extraction des différents éléments constituant un diagramme: les classes, les associations, les attributs d'une classe donnée, etc. Les informations des diagrammes sont récupérées dans un fichier texte ayant une structure bien définie. Ce fichier est transmis au programme de génération de spécifications abstraites **B**.

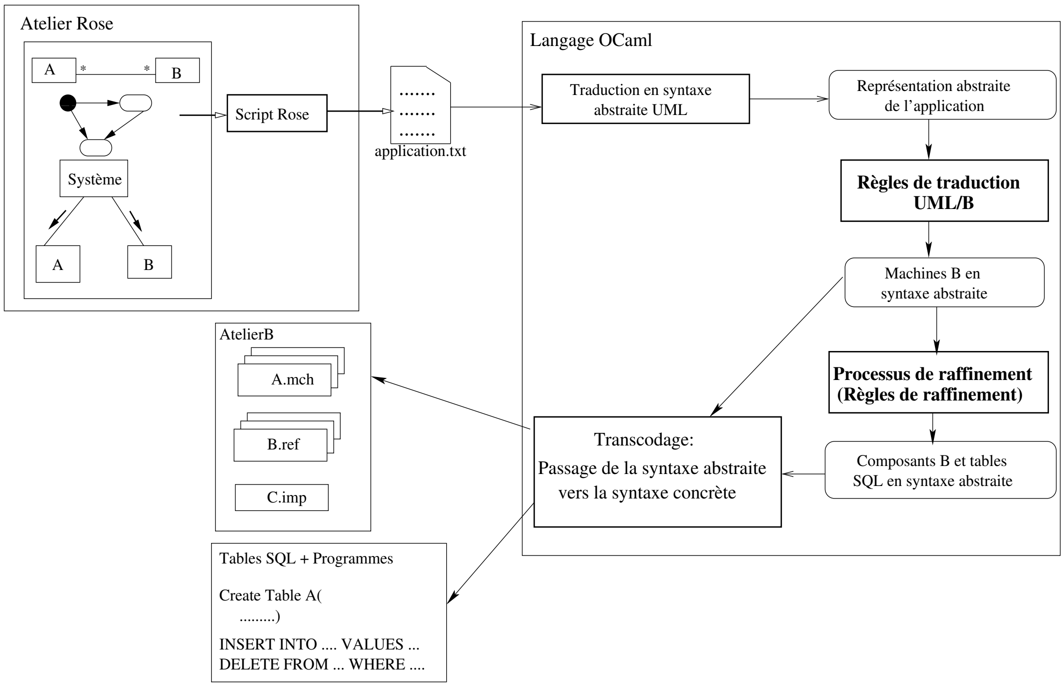


FIG. 7.1 – *Une approche outillée pour la dérivation d'une implémentation relationnelle à partir de diagrammes UML*

3. Une représentation en syntaxe abstraite de l'application étudiée est générée à partir du fichier texte obtenu à l'étape précédente. Elle est basée sur le méta-modèle défini dans [23].
4. Des spécifications abstraites modularisées **B** sont directement générées à partir de la représentation en syntaxe abstraite de l'application étudiée. Une étape préalable à celle traduction est d'effectuer une analyse syntaxique des diagrammes. Cette analyse syntaxique permet de s'assurer d'un minimum de cohérence entre les différents diagrammes. En cas d'erreur un message est envoyé à l'utilisateur. La validation syntaxique concernent les points suivants :
   – le qualifieur d'un rôle sont des attributs de la classe associée au rôle opposé.
   – les actions spécifiées dans les diagrammes états/transitions correspondent aux opérations de base de la classe considérée.
   – les variables apparaissant dans les différents prédicats associés aux états et aux gardes sont des données (variables ou given sets) définies soit dans la machine **B** associée à la classe considérée ou à l'une des associations auxquelles la classe participe. Cette vérification est basée sur une anticipation de l'étape de génération de spécifications **B**. Cette vérification est également réalisée pour les gardes associées aux messages des diagrammes de collaborations.
   – les messages internes spécifiés dans les diagrammes de collaborations correspondent soit aux évènements des diagrammes états/transitions, soit aux opérations de base.
5. Raffinement des spécifications abstraites obtenues en vue de la génération d'une implémentation relationnelle (SQL). Dans cette étape sont générés automatiquement les éléments suivants :
   – les différents composants créés lors du processus de raffinement ainsi que les machines incluses par ces derniers.
   – les différentes machines correspondant aux tables SQL : données et opérations.
   – l'implémentation des données (clause invariant du composant "implémentation") ainsi que l'im-

plémentation des opérations de base. Le concepteur doit implémenter manuellement les opérations (transactions) pour lesquelles seuls des squelettes peuvent être générés. Les preuves de correction sont réalisées avec l'AtelierB.

Notons que pour la partie raffinement, les concepts d'héritage et d'attributs à types complexes ("Partie-de" ou "Séquence") ne sont pas pris en compte. Le but de la réalisation d'un tel outil est avant tout de montrer la faisabilité de l'automatisation de nos propositions.

7.1 Génération outillée d'une implémentation relationnelle à partir de spécifications UML

Cette section illustre, à travers l'exemple de gestion de prêts de cassettes, les principales fonctionnalités du prototype d'outil d'aide à la génération d'une implémentation relationnelle à partir de spécifications **UML**. Nous montrons comment les différents diagrammes **UML**, permettant la description du système, sont saisis dans l'atelier **ROSE**. Ensuite, nous montrons la génération des spécifications **B** et leur raffinement.

7.1.1 Construction du diagramme de classes UML

Comme nous l'avons noté précédemment, le diagramme de classes est le premier diagramme à saisir. En effet, les autres diagrammes sont basés non seulement sur les informations contenues dans le diagramme de classes mais aussi sur les spécifications **B** générées à partir du diagramme de classes. Le lancement de l'atelier **ROSE** donne accès directement à la fenêtre de saisie d'un diagramme de classes "Class Diagram". La figure 7.2 illustre la saisie du diagramme de classes associé à la gestion des prêts d'un ensemble de cassettes.

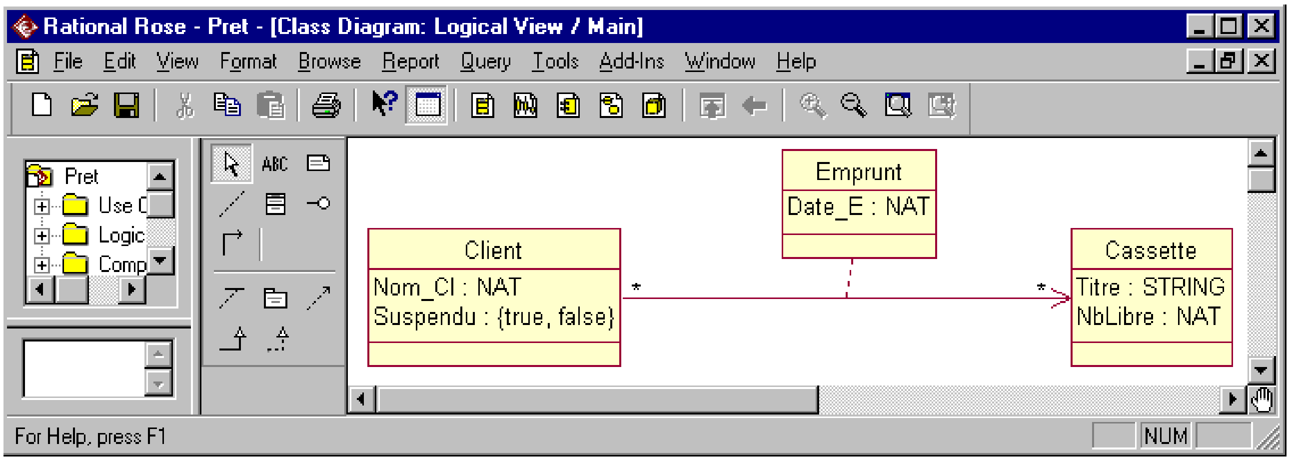


FIG. 7.2 – *Exemple simplifié de diagrammes UML*

Pour chaque élément d'un diagramme, l'atelier **ROSE** fournit un formulaire standard pour la saisie d'informations relatives à l'élément (multiplicités d'une association, qualifieur d'un rôle, etc). Néanmoins, ces formulaires ne permettent pas l'expression de toutes les propriétés qu'on voudrait exprimer sur les différents éléments d'un diagramme. Ces propriétés peuvent être de natures très diverses. Ne pouvant pas être spécifiées graphiquement dans **ROSE**, nous les exprimons textuellement dans les champs documentations associés aux différents éléments du diagramme de classes: attribut, classe, associations, le diagramme lui-même, etc. Une question naturelle donc se pose: "pour chaque propriété, quel est le champ qui convient le mieux ?". Un bon placement des contraintes permet, d'une part, une génération automatique des spécifications **B**, et d'autre part, une meilleure modularisation de ces dernières. En effet, une propriété sur une association doit être exprimée dans la machine où est définie cette association. Une solution naïve et simpliste à ce problème de placement est d'exprimer l'ensemble de ces propriétés dans le formulaire associé au diagramme de classes lui-même, et donc dans la machine d'interface *Système*. L'inconvénient de cette solution est la longueur du champ et par conséquent l'invariant de la machine *Système* devient très important. D'autre part, exprimer par exemple chaque propriété d'attribut dans la classe correspondante réduit considérablement le degré d'automatisation de la génération des opérations de base. Ces opérations doivent alors contenir des préconditions supplémentaires permettant le respect de ces propriétés. Pour cela, nous proposons la solution suivante :

- seules les contraintes liées à une association sont exprimées dans le champs documentation associé à cette association.
- les contraintes associées aux attributs sont décrites dans le champ documentaire associé à l'ensemble du diagramme.

Cette solution peut être considérée comme satisfaisante car en pratique les contraintes sur les attributs sont souvent peu nombreuses.

7.1.2 Génération des machines de base

Une fois le diagramme de classes saisi, nous récupérons à l'aide d'un script **ROSE** les différents éléments constituant ce diagramme. Pour chaque attribut, on demandera (si l'information n'est pas déjà présente dans le diagramme) à l'utilisateur de préciser les informations suivantes (Voir figure 7.3) :

- sa multiplicité,
- l'attribut est-il modifiable ou non,
- l'attribut fait-il partie de la clé ou pas.

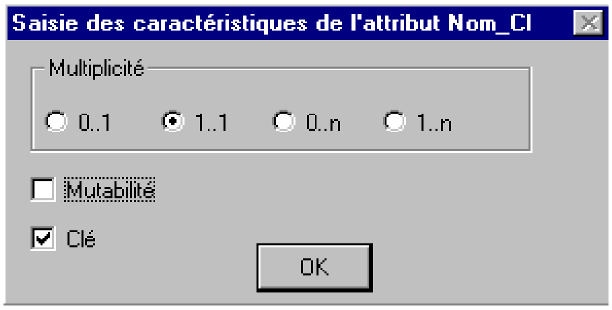


FIG. 7.3 – *Saisie des caractéristiques d'un attribut*

Les informations du diagramme de classes étant récupérées, un programme OCaml permet la génération de spécifications **B** à partir de ces dernières. Comme nous l'avons vu au chapitre 3, les spécifications **B** générées concernent:

- l'aspect statique du diagramme représenté par les ensembles abstraits, les variables et l'invariant (Voir figure 7.4). La partie statique de chaque machine de base est stockée dans le champs documentation de la classe ou de l'association considérée.
- un ensemble d'opérations standards relatives à l'ajout, a la suppression d'instances d'une classe ou d'une association, et la mise à jour des valeurs des attributs mutables (Voir figure 7.5).

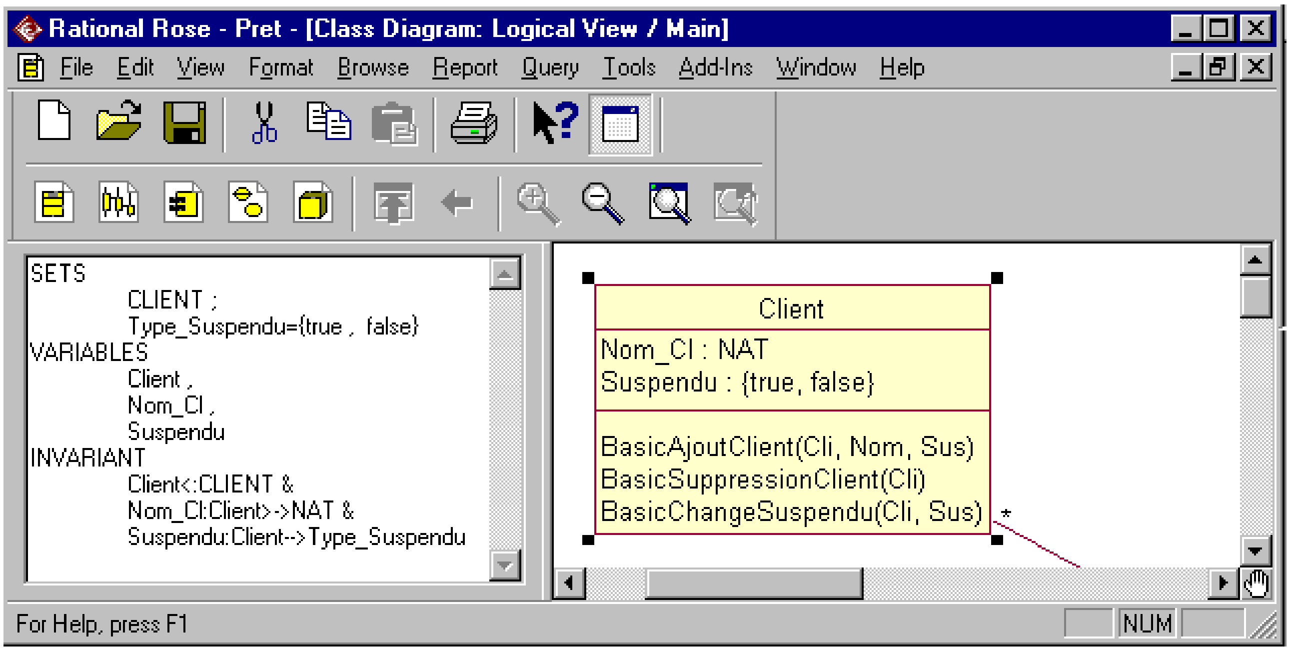


FIG. 7.4 – *Partie statique de la spécification B de la classe Client*

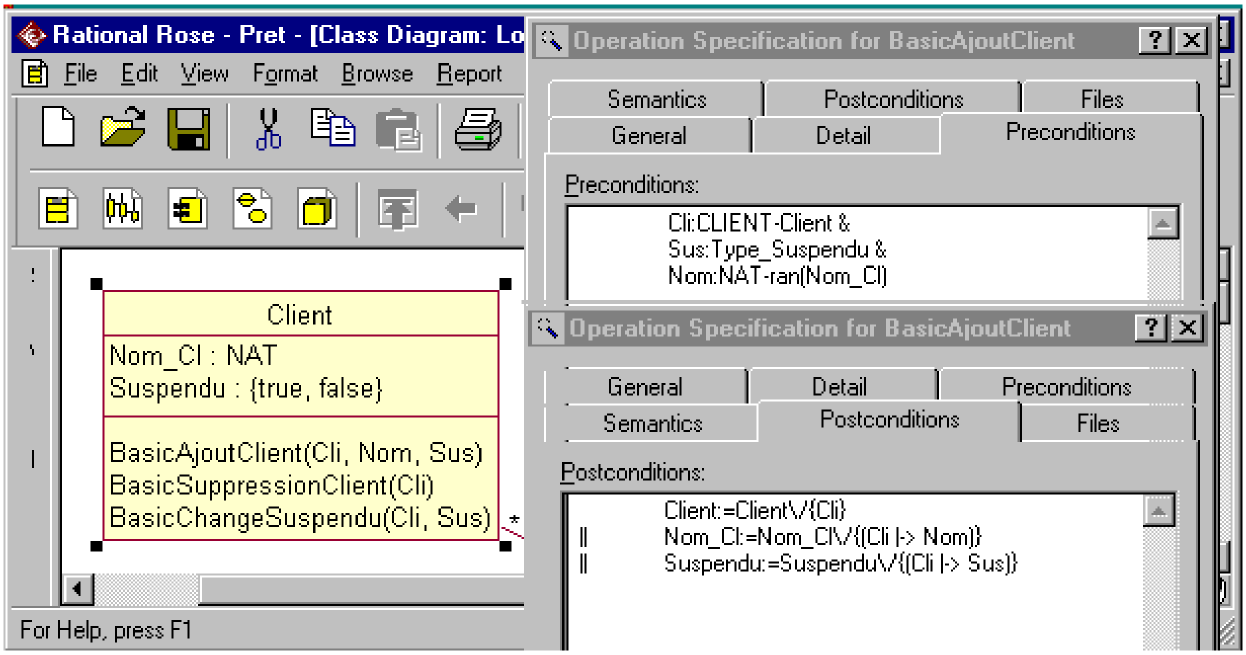


FIG. 7.5 – *Précondition et postcondition de l'opération BasicAjoutClient*

7.1.3 Saisie des diagrammes dynamiques

Le diagramme de classes étant établi, les spécifications **B** correspondantes étant générées, l'étape suivante concerne la saisie des diagrammes décrivant l'aspect dynamique du système.

Diagramme états/transitions. Un diagramme états/transitions décrit le comportement d'un objet d'une classe ou d'une association en termes d'états et de transitions. L'option d'ouverture d'une fenêtre de saisie du diagramme états/transitions est activable dès qu'une classe ou une association est pré-séléctionnée. La figure 7.6 représente le diagramme états/transitions de la classe *Client*.

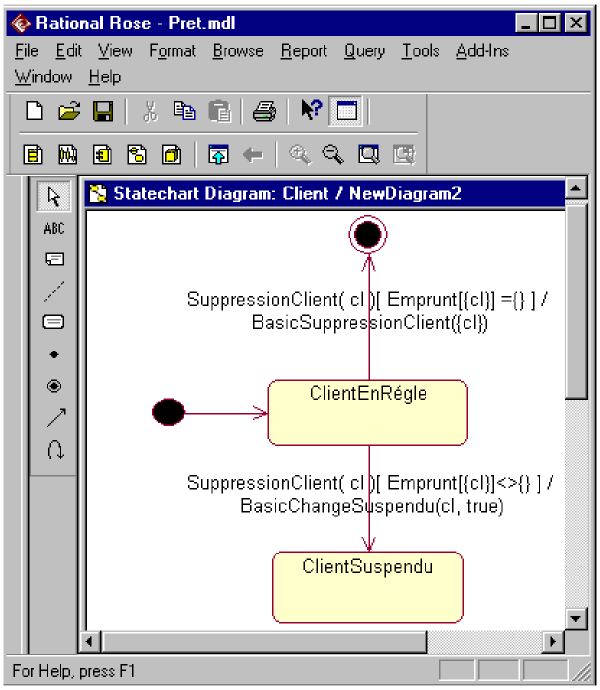


FIG. 7.6 – *Un exemple de saisie de diagrammes états/transitions dans ROSE*

Comme pour le diagramme de classes, certaines informations du diagramme états/transitions sont saisies en **B** et sont stockées dans les champs documentations. La figure 7.7 montre la saisie du typage des paramètres d'entrée de l'événement *SuppressionClient* ainsi que le prédicat associé à l'état *ClientEnRégle*.

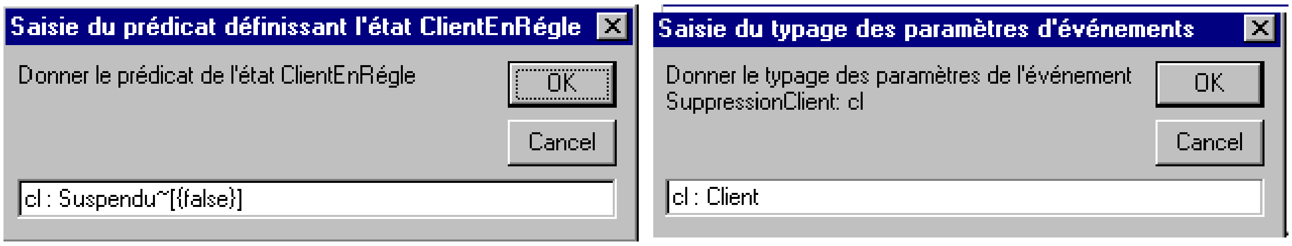


FIG. 7.7 – *Saisie du prédicat d'état et du typage des paramètres d'un événement*

Une fois les spécifications **B** associées aux diagrammes d'états/transitions générées, l'utilisateur a la possibilité de visualiser la traduction de chaque événement en cliquant sur l'une des transitions qui lui est associée et en choisissant l'option "Spécification B" que nous avons ajoutée au menu de l'atelier ROSE. L'opération générée pour l'événement *SuppressionClient* est représentée dans la figure 7.8.

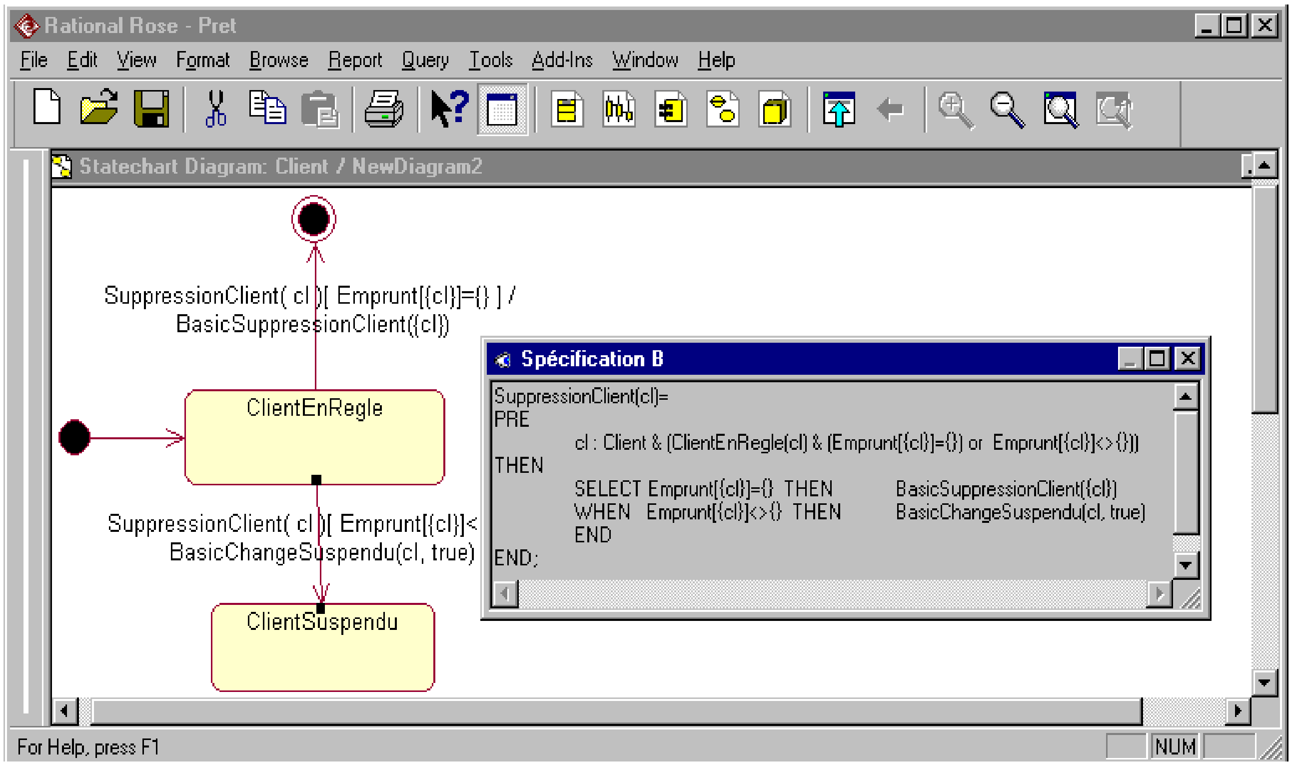


FIG. 7.8 – *Opération B associée à l'événement SuppressionClient*

Diagrammes de collaborations. Un diagramme de collaborations décrit les transactions utilisateur. La figure 7.9 montre la saisie du diagramme de collaborations représentant la transaction de prêt d'une cassette.

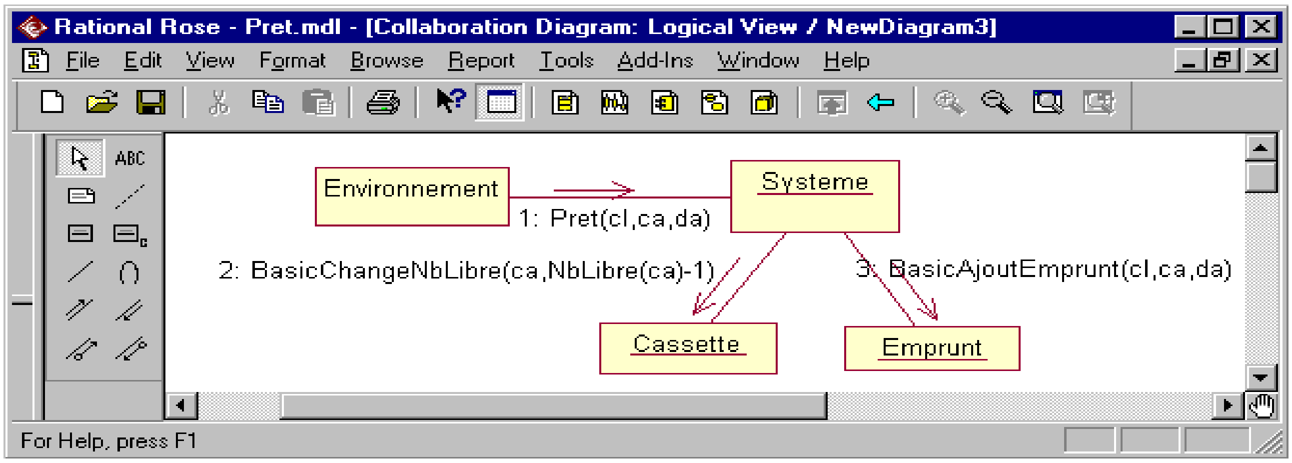


FIG. 7.9 – *Modélisation de la transaction de Prêt d'une cassette*

Comme pour le diagramme états/transitions, l'utilisateur sera invité à saisir interactivement le typage des paramètres d'entrée du message *Pret* déclencheur de la transaction ainsi que les éventuelles gardes des différents messages (Voir figure 7.10)

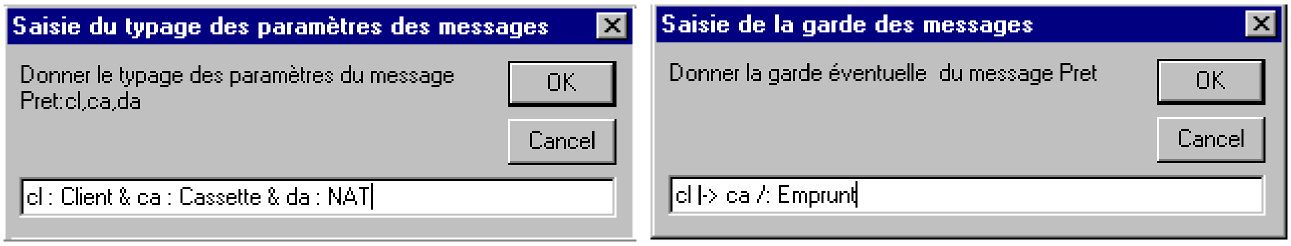


FIG. 7.10 – *Saisie du typage des paramètres d'un message et de son éventuelle garde*

Une fois les diagrammes de collaborations saisis, nous générons pour chacun d'eux l'opération **B** correspondante. L'opération générée pour le diagramme de collaborations de la figure 7.9 est représentée par la figure 7.11.

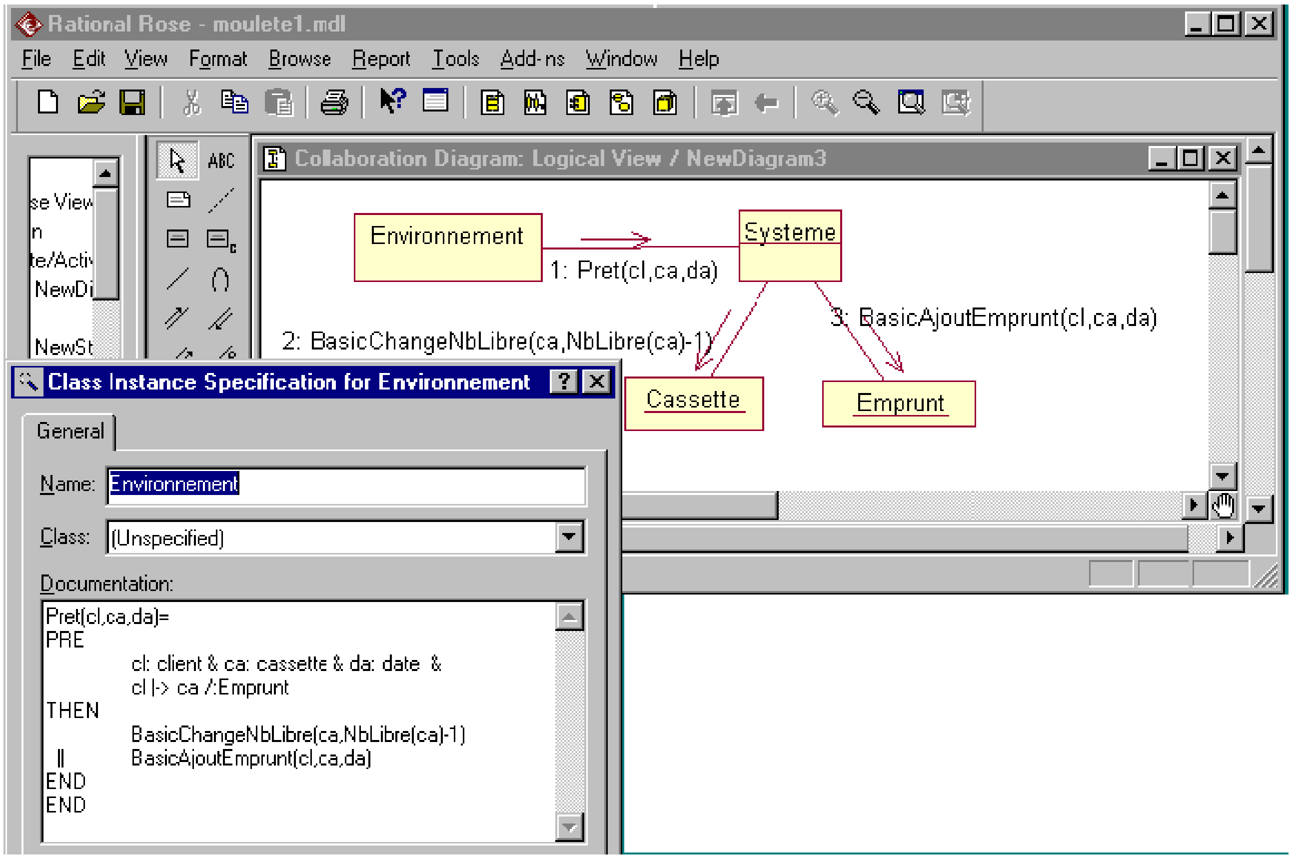


FIG. 7.11 – *Saisie du typage des paramètres d'un message et de son éventuelle garde*

7.1.4 Génération des tables relationnelles

Les spécifications **B** générées, à l'étape précédente, sont raffinées pour la définition des différentes tables relationnelles implémentant le diagramme de classes. Pour chaque classe ou association multivaluée, une table relationnelle est générée. La figure 7.12 visualise la table correspondant à l'association *Emprunt*. L'attribut

Cle_Cassette représente l'attribut clé ajouté à la classe *Cassette* lors de l'étape d'identification des objets.

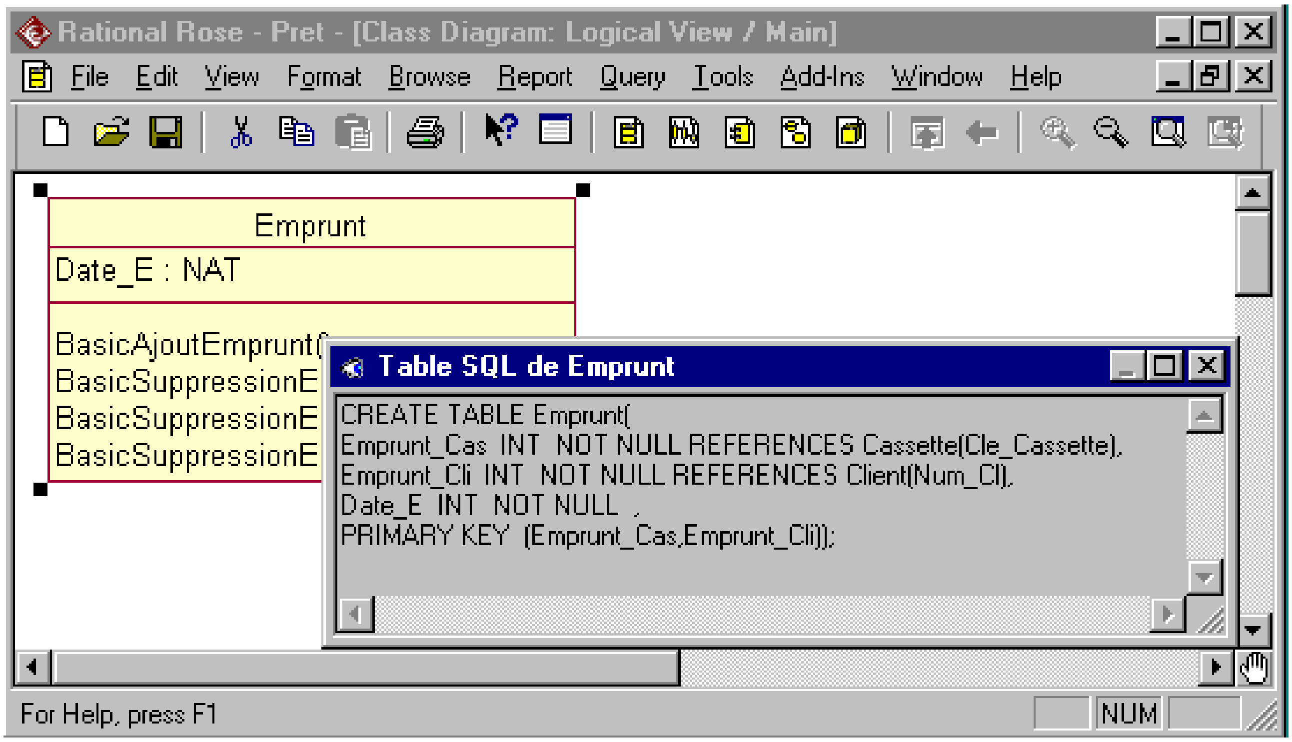


FIG. 7.12 – *Table relationnelle associée à l'association Emprunt*

Un message signalétique est envoyé à l'utilisateur s'il demande d'afficher la table correspondant à une association monovaluée. On précisera la table où est traduite cette association.

7.2 Discussion

7.2.1 Point forts et limites de l'outil

Comme tout outil automatique, l'avantage immédiat du développement d'un tel outil est de décharger l'utilisateur de tâches longues et fastidieuses. En effet, l'écriture (en ASCII) manuelle de spécifications **B** pour un simple diagramme de classes constitué de deux classes et d'une association demanderait au moins une demie journée de travail auquel s'ajoute tous les risques d'erreurs que l'utilisateur peut commettre. Les spécifications générées par notre outil sont assez complètes. Les machines traduisant les classes et les diagrammes états/transitions sont validées automatiquement par le prouveur de l'AtelierB (100% de preuves automatiques). Les opérations générées pour les associations prennent en compte toutes les contraintes de multiplicités. Au moins 80% des preuves générées pour ces opérations sont automatiques. Les preuves interactives restantes peuvent être déchargées automatiquement par l'ajout de quelques règles élémentaires de la théorie des ensembles. À présent, le développeur peut concentrer tout ses efforts sur la preuve des opérations de la machine *Système* et bien évidement les différentes étapes de raffinement. Rappelons que seuls les corps des opérations du composant implémentantion doivent être complétés par l'utilisateur. L'invariant de ce même composant ainsi que les différents composants raffinement sont générés automatiquement.

L'utilisation de l'environnement **ROSE** pour la saisie des différents diagrammes facilite l'adhésion des utilisateurs habitués aux représentations graphiques à l'utilisation des méthodes formelles. De cette façon, la transition des représentations graphiques aux notations mathématiques peut se faire de manière assez naturelle et progressive. Cette transition est aussi facilitée par la possibilité de visualiser les notations graphiques et les spécifications **B** associées dans un même environnement de développement. Enfin, l'utilisation d'un langage de programmation pour l'implémentation des différentes règles de traduction et de raffinement permet une meilleure portabilité de notre outil. Notre outil peut être adapté sans modifications majeures à n'importe quel environnement de saisie des diagrammes UML.

Néanmoins, l'outil développé présente quelques limites. Aucune analyse sémantique n'est effectuée sur les diagrammes UML. On souhaiterait, par exemple, vérifier que le prédicat associé à un état du diagramme états/transitions n'est pas toujours faux ou contradictoire avec la garde de l'une des transitions dont cet état est source. En effet dans le premier cas, l'état est donc inaccessible. Dans le second cas, la transition est infranchissable. Un autre problème auquel on pourrait s'intéresser est l'optimisation des spécifications générées. Les préconditions associées aux gardes des événement peuvent être simplifiées dans des cas précis. Enfin comme nous l'avons noté, certaines informations des diagrammes dynamiques doivent être fournies en **B** (caractères ASCII). Le degré d'automatisation de la génération des spécifications **B** est proportionnel au degré de formalisation de ces diagrammes. Pour des diagrammes partiellement annotés, seuls des squelettes de spécification devant être complétés par l'utilisateur peuvent être générés. L'annotation de ces diagrammes peut éventuellement être facilitée en utilisant le langage d'expression de contraintes **OCL** [76] qui fait partie intégrante des dernières versions de **UML**. Ces annotations **OCL** seront, par la suite, traduite en **B** en utilisant les règles proposées dans [48]

7.2.2 Comparaison à d'autres outils similaires

Dans la littérature, plusieurs outils portant sur le couplage des notations graphiques et des notations formelles ont été développés. L'outil développé par [18] permet, entre autres, la génération de spécifications **Z** à partir de diagrammes de classes **UML** développés dans l'environnement **ROSE**. La génération des opérations ne concerne que les classes. La génération des opérations pour les associations n'est pas prise en compte. De même, les diagrammes dynamiques ne sont pas considérés. Un autre outil similaire à ce dernier a été développé par [12]. Cet outil utilise l'environnement **Paradigm Plus** pour la génération de spécifications **Z** à partir de diagrammes de classes **Fusion**. [70] ont mis en œuvre un outil similaire au notre qui permet la génération de fichiers de spécifications **B** associés à un diagramme de classes. Seuls les diagrammes de classes et d'états/transitions sont pris en compte. Les opérations générées sur les associations ne concernent que l'ajout d'une instance. Les opérations de suppression et de mise à jours de attributs ne sont pas considérées. Notons également que le domaine d'applications considéré est les systèmes réactifs. La sémantique attribuée aux différents diagrammes est totalement différente de la notre.

7.3 Conclusion

Dans ce chapitre a été présenté un outil automatique pour le développement d'applications base de données. Il permet la génération des différentes tables relationnelles et des opérations de base à partir d'un ensemble de diagrammes **UML** à sémantique précise. Cet outil implémente deux fonctions principales. La première est la traduction des diagrammes **UML** en spécifications formelles B. La seconde concerne le raffinement des spécifications générées jusqu'à l'obtention d'une implémentation relationnelle.

Chapitre 8

Conclusion

Nous dressons dans cette conclusion une synthèse générale des propositions présentées tout au long de ce manuscrit. Nous exposons l'apport de notre approche par rapport aux différents travaux antérieurs et donnons quelques perspectives envisageables de recherche.

Dans le cadre de cette thèse, nous nous sommes intéressée à la réalisation d'un environnement formel pour le développement d'applications base de données sûres. Plus précisément, nous avons travaillé sur la définition d'une approche formelle permettant la génération d'une implémentation relationnelle à partir de la spécification **B** d'une application des systèmes d'information. Pour cela, nous avons opté pour une démarche par raffinement.

Dans un premier temps, nous avons caractérisé la forme des spécifications abstraites **B** traitées par notre démarche de développement. Les spécifications abstraites que nous considérons sont générées à partir de diagrammes **UML** (diagrammes de classes, diagrammes états/transitions, diagrammes de collaborations) spécialisés pour la description d'applications du domaine des systèmes d'information. Cette génération est réalisée suivant un ensemble de règles de traduction initialement définies pour des notations **OMT** [53] et que nous avons adaptées pour les notations **UML** [22]. Les spécifications, générées par ce processus de traduction, sont caractérisées par une forme générique et standard. De manière générale, nous distinguons trois composantes. Un ensemble de données décrivant l'aspect statique de l'application, des opérations standards, que nous avons qualifiées de base, permettant la manipulation séparée de chaque entité indépendamment des autres et puis enfin des opérations plus complexes qui décrivent les fonctionnalités du système étudié. Ces opérations complexes correspondent aux transactions utilisateur. Le caractère générique et standard de ces spécifications nous a permis la définition d'une approche formelle de dérivation d'une implémentation relationnelle. Ceci a constitué la deuxième étape de notre travail.

Dans les chapitres 4 et 5, nous avons développé un ensemble de règles de raffinement. Ces règles raffinent conjointement les données et les opérations de base. Le raffinement des données est inspiré des algorithmes classiques utilisés dans le domaines des systèmes d'information pour la traduction d'un diagramme Entités/Associations en un schéma relationnel. Nous avons également donné quelques indications sur la façon dont les opérations complexes pouvaient être implémentées. Enfin, nous avons décrit la traduction des différents composants **B**, obtenus lors de la dernière phase de raffinement, vers le langage relationnel **SQL** imbriqué dans le langage hôte **JAVA**. Un exemple illustratif de nos propositions est donné en Annexe. Il est important de noter que seul le raffinement des données et des opérations de base est automatique, le raffinement des opérations correspondant aux transactions requièrent l'intervention de l'utilisateur.

Comme toute démarche de développement formel, nous avons validé notre approche par l'établissement d'un ensemble de preuves. Prouver chaque étape (ou règle) de raffinement assure de proche en proche la correction du code généré vis-à-vis de la spécification abstraite initiale. La preuve des règles de raffinement a été réalisée en utilisant le prouveur de l'AtelierB. Certaines règles ont nécessité l'ajout d'axiomes que nous avons validés à leur tour.

Les différentes preuves, établies à la phase précédente, n'assurent la correction du raffinement que des opérations de base. Il reste bien évidement la validation des opérations utilisateur dont le raffinement repose sur le raffinement des opérations de base. Nous avons donc naturellement étudié la réutilisation des preuves du raffinement des opérations de base pour la preuve de ces opérations. Les substitutions **B** étant définies de manière inductive, le raisonnement s'est fait par induction sur la forme de la substitution à raffiner. Nous avons procédé en deux étapes. La première étape a consisté à définir pour chaque opérateur **B**, des conditions nécessaires et suffisantes de réutilisation des preuves de raffinement des substitutions qu'il relie pour l'élaboration de la preuve de la substitution résultante. Les résultats obtenus dans cette étape ont été généralisés, dans une deuxième étape, au cas de substitutions faisant intervenir plusieurs opérateurs.

Dans le but de rendre notre approche opérationnelle et d'atténuer la réticence des utilisateurs à l'utilisation des méthode formelles, nous avons développé un outil supportant les différentes phases automatiques de notre approche. Cet outil, réalisé en OCaml, a permis de nous assurer que les différentes règles de traduction et de raffinement sont suffisamment formalisées pour constituer une base d'automatisation. Nous avons utilisé l'environnement de modélisation graphique **ROSE** pour la saisie des différents diagrammes **UML**.

L'apport de notre travail en regard des travaux antérieurs peut être résumé par les points suivants. Les travaux de [77, 66] ne proposent des règles de raffinement que pour l'aspect statique, le raffinement des opérations est réalisé de manière informelle. Contrairement à notre approche, ils ne considèrent qu'un ensemble restreint de concepts. Les concepts élaborés tels que les attributs multivalués et l'héritage ne sont pas du tout abordés. De même, les travaux de [57] ne considèrent que l'aspect transaction. Ils font l'hypothèse de l'existence d'un schéma de la base déjà établi. De plus, leur approche est loin d'être accessible à n'importe quel développeur. Elle demande des connaissances accrues en mathématiques. Enfin, aucun de ces travaux ne propose d'outil d'aide à la la mise en œuvre de leurs approches.

Le travail présenté dans cette thèse ouvre diverses perspectives potentielles de recherche. Nous envisageons de réaliser un assistant spécialisé à la preuve mettant en œuvre la procédure de réutilisation de preuves proposée au chapitre 6. Deux solutions sont possibles pour la réalisation de prouveurs. La première est de concevoir un prouveur autonome qui implémente, en plus de la procédure de réutilisation, une des techniques de preuve classiques. La seconde est d'étendre un prouveur déjà existant par la technique de réutilisation que nous avons définie. Ce travail demande une réflexion conséquente sur le choix du prouveur qui servira de base à ce prouveur spécialisé. Nous pensons que prouveur **B** est le mieux adapté à cette fonction. Un travail préalable à la réalisation de ce prouveur est de vérifier de manière formelle, en **Coq** par exemple, les conditions de réutilisation que nous avons définies. Nous envisageons également d'étendre l'outil actuel pour supporter des concepts additionnels tels que la composition et l'agrégation. Enfin une perspective à long terme est d'étudier de manière plus approfondie la traduction des composants implémentation **B** en langage **SQL** imbriqué dans le langage **JAVA**.

Annexe A

La Méthode B

Conçue par Abrial [1], la méthode **B** est une méthode formelle de développement d'applications sûres. Elle couvre les principales phases de conception d'un projet : de la spécification abstraite jusqu'à la génération du code associé. En **B**, chaque étape de développement est validée par des preuves.

Actuellement, la méthode **B** est utilisée en industrie principalement pour le développement de logiciels critiques (ex : projet METEOR : ligne 14 du métro parisien [8]). Deux ateliers de génie logiciel lui sont associés :

- L'AtelierB développé par DIGILOG (France) [16],
- B-Toolkit développé par B-CORE (Royaume-Uni) [3].

Ces deux outils disposent principalement :

- d'un analyseur syntaxique d'un composant (machine, raffinement ou implémentation),
- d'un contrôleur de types des composants,
- d'un générateur automatique des obligations de preuves,
- d'un prouveur automatique (une base de règles, un ensemble de théories et techniques),
- d'un prouveur interactif,
- d'un traducteur automatique (génération du code C, C++ ou ADA).

L'utilisation de la méthode **B** permet une détection précoce des éventuelles incohérences et erreurs que le concepteur peut commettre. Une telle démarche de développement permet à la méthode **B** d'être un guide pour la conception et la maintenance.

A.1 Le langage de spécification B

Nous présentons succinctement les concepts du langage **B** utilisés dans le cadre de ce travail. Une présentation complète est donnée dans [1]. Le langage **B** est composé de trois sous-langages :

- le langage de la logique du premier ordre enrichi par le prédicat d'égalité et le concept de paire ordonnée. Ce langage permet l'expression des propriétés du système qu'on spécifie,
- le langage des substitutions généralisées qui permet de décrire le comportement dynamique du système,
- la notion de composant, analogue à la notion des modules, qui permet de structurer la spécification.

Dans la suite de ce chapitre, seuls les deux derniers concepts sont détaillés. Ce chapitre introduit également la notion de raffinement des substitutions et des composants.

Nom	Notation B	Conditions de définition	Signification
Skip	*skip*		Ne rien modifier
Substitution élémentaire	x :=E	x est une variable E est une expression	Substituer E à x
Précondition	**PRE** *P* **THEN** *S* **END** (notée $P \mid S$)	P est un prédicat S est une substitution	S'assurer de P et exécuter S
IF	**IF** *P* **THEN** *S* **ELSE** *T* **END** [1]	P est un prédicat S une substitution	Si P est vrai, exécuter S sinon exécuter T
Garde	**SELECT** *P* **THEN** S **END** [2]	P est un prédicat S est une substitution	exécuter S si P est vrai.
ANY	**ANY** *X* **WHERE** *P* **THEN** *S* **END**	X est une liste de variables P est un prédicat S est une substitution	Sélectionner une valeur de X qui vérifie le prédicat P et exécuter S
Appel d'opération	$\llbracket u \leftarrow \rrbracket$ Op $\llbracket (v) \rrbracket$		appeler l'opération *Op* avec les paramètres v et affecter son résultat à u
Simultanée	$S \parallel T$	S et T sont des substitutions	exécuter S et T en même temps
Choix	S [] T	S et T sont des substitutions	exécuter S ou T
Séquence	S;T	S et T sont des substitutions	exécuter S puis T
Boucle	$(B \circlearrowleft S)_{(I,v)}$ [3]	B un prédicat, S une substitution, V une expression entière strictement décroissante et positive I un invariant.	exécuter S tant que B est vrai.

FIG. A.1 – *Quelques substitutions du langage B*

A.2 Substitutions généralisées

Une substitution généralisée est un transformateur de prédicats. Cette notion est analogue à la notion de la plus faible précondition de Hoare [33]. Notre présentation des substitutions se limite aux substitutions nécessaires à la compréhension de notre problématique. Ces substitutions sont données dans la figure A.1.

À une substitution peut être associées deux sémantiques équivalentes. Une substitution peut être perçue soit à travers son effet transformationnel sur un prédicat P, soit comme une formule relationnelle liant les valeurs "après" et "avant" de ses variables. La définition de ces deux sémantiques, nécessite l'introduction des notions de variables libres et de variables modifiées.

Définition A.2.1 *L'ensemble des variables libres d'une substitution S, noté FV(S), est défini inductivement par (z désigne un ensemble de variables) :*

1. **IF** P **THEN** S **END** = **IF** P **THEN** S **ELSE** *Skip* **END**.
2. La syntaxe complète de la substitution **SELECT** est (**SELECT** P **THEN** S **WHEN** Q **THEN** T **ELSE** U **END**) dont la sémantique est : exécuter une des substitutions dont le prédicat est vrai. Si tous les prédicats sont faux, on exécute la substitution associée à la partie **ELSE**. Une écriture possible de cette substitution est: (**SELECT** P **THEN** S **END**)[](**SELECT** Q **THEN** T **END**)[](**SELECT** $(\neg P \wedge \neg Q)$ **THEN** U **END**).
3. $(B \circlearrowleft S)_{I,v}$ est une abréviation de (WHILE B DO S INVARIANT I VARIANT V END)

Substitution	FV(Substitution)
$skip$	$\emptyset$
$x := E$	$\{x\} \cup FV(E)$
$P \mid S$	$FV(P) \cup FV(S)$
IF P **THEN** S **ELSE** T **END**	$FV(P) \cup FV(S) \cup FV(T)$
SELECT P **THEN** S **END**	$FV(P) \cup FV(S)$
ANY z **WHERE** P **THEN** S	$FV(S) \cup FV(P) - z$
$S \mid\mid T$	$FV(S) \cup FV(T)$
$S[]T$	$FV(S) \cup FV(T)$
$S;T$	$FV(S) \cup FV(T)$
$(B \circlearrowleft S)_{(I,v)}$	$FV(B) \cup FV(S) \cup FV(I) \cup FV(V)$

Définition A.2.2 *L'ensemble des variables modifiées d'une substitution S, noté MV(S), est défini inductivement par :*

Substitution	MV(Substitution)
$skip$	$\emptyset$
$x := E$	$\{x\}$
$P \mid S$	$MV(S)$
IF P **THEN** S **ELSE** T **END**	$MV(S) \cup MV(T)$
SELECT P **THEN** S **END**	$MV(S)$
ANY z **WHERE** P **THEN** S	$MV(S) - z$
$S \mid\mid T$	$MV(S) \cup MV(T)$
$S[]T$	$MV(S) \cup MV(T)$
$S;T$	$MV(S) \cup MV(T)$
$(B \circlearrowleft S)_{(I,v)}$	$MV(S)$

A.2.1 Sémantique transformationnelle.

Soient *S* une substitution et *P* un prédicat. La notation [*S*]*P* (lue "*S* établit *P*") signifie que le prédicat *P'*, obtenu à partir de P en remplaçant ses variables libres par leurs nouvelles valeurs assignées par *S*, est vérifié. Le tableau A.2 définit inductivement la sémantique d'une substitution en terme de transformateur de prédicats.

A.2.2 Sémantique relationnelle.

La sémantique relationnelle d'une substitution est définie à l'aide des prédicats de terminaison Trm et avant-après Prd associés à chaque substitution S.

Définition A.2.3 *Pour toute substitution S, le prédicat Trm_x est défini par :*

$$Trm_x = [S]true$$

Définition A.2.4 *Pour toute substitution S, le prédicat $Prd_{x,x'}$ est défini par :*

$$Prd_{x,x'} = \neg[S](x' \neq x)$$

Notons que x représente l'ensemble des variables libres de la substitution S. Les symboles x et x' représentent respectivement l'état des variables avant et après l'exécution de la substitution S. Intuitivement, $Trm_x(S)$ définit les conditions assurant l'exécution correcte de S (S termine), et $Prd_{x,x'}(S)$ établit le lien entre les valeurs des variables avant et après l'exécution de *S*, i.e. entre x et x'.
Les tableaux des figures A.3 et A.4 définissent inductivement ces deux prédicats (les prédicats non indicés sont implicitement indicés par l'ensemble de leurs variables libres).

Substitution	[Substitution]P
Skip	P
$[x := E]P$	dans P, on remplace les occurrences libres de x par E
$Q \mid S$	$Q \wedge [S]P$
IF Q **THEN** S **ELSE** T **END**	$(Q \Rightarrow [S]P) \wedge (\neg Q \Rightarrow [T]P)$
SELECT Q **THEN** S **END**	$Q \Rightarrow [S]P$
ANY X **WHERE** Q **THEN** S **END**	$\forall\ X.(Q \Rightarrow [S]P)$
$R \leftarrow op(E)$	$[X := E][S][R := Y]P$ S désigne le corps de l'opération op X désigne les paramètres d'entrée formels de op Y désigne les paramètres de sortie formels de op
$[x := E \ \|\| \ y := F]P$	dans P, on remplace simultanément les occurrences libres de x et y par E et F respectivement
$[S[]T]P$	$[S]P \wedge [T]P$
$[S;T]P$	$[S][T]P$
$[(B \circlearrowleft S)_{I,v}]P$	$I\wedge$ (l'invariant I est vrai à l'entrée de la boucle) $\forall X.(I \Rightarrow V \in NAT)\wedge$ (la valeur de V est toujours entière) $\forall X.(I \wedge B \Rightarrow [S]I)\wedge$ (chaque exécution de la boucle établit l'invariant I) $\forall X.((I \wedge B) \Rightarrow [n := V][S](V < n))$ (la valeur de V est strictement décroissante) $\forall X.((I \wedge \neg B) \Rightarrow P)$ (À la sortie de la boucle, P est vérifié)

FIG. A.2 – *Sémantique transformationnelle d'une substitution*

Substitution	Trm(Substitution)
skip	*True*
$x := E$	*True*
$P \mid S$	$P \wedge Trm(S)$
IF Q **THEN** S **ELSE** T **END**	$(Q \Rightarrow Trm(S)) \wedge (\neg Q \Rightarrow Trm(T))$
SELECT Q **THEN** S **END**	$Q \Rightarrow Trm_x(S)$
ANY z **WHERE** P **THEN** S **END**	$\forall z.(P \Rightarrow Trm(S))$
$S \| T$	$Trm(S) \wedge Trm(T)$
$S;T$	$Trm(S) \wedge \forall a.([x' := a]Prd(S) \Rightarrow [x := a]Trm(T))$
$S[]T$	$Trm(S) \wedge Trm(T)$
$(B \circlearrowleft S)_{I,v}$	$I(x) \wedge \forall t.(I(t) \Rightarrow \exists n.(V(t) = n \wedge n \in \mathbb{N})\wedge$ $\forall t.(I(t) \wedge B(t) \Rightarrow Trm(S)\wedge$ $(\forall t'(Prd(S) \Rightarrow (I(t) \wedge V(t') < V(t))))))$

FIG. A.3 – *Définition inductive du prédicat de terminaison d'une substitution*

Substitution	Prd(Substitution)	Conditions
$skip$	$x' = x$	
$x := E$	$x' = E \wedge y' = y$	$y = FV(E) - \{x\}$
$P \mid S_1$	$P \Rightarrow (Prd(S_1) \wedge y' = y)$	y=FV(P)-FV(S_1)
IF Q **THEN** S_1 **END**	$(Q \wedge Prd(S_1) \wedge y' = y) \vee (\neg Q \wedge z' = z)$	y=FV(Q)-FV(S_1), z=FV(S)
SELECT Q **THEN** S_1 **END**	$Q \wedge Prd(S_1) \wedge y' = y$	y=FV(Q)-FV(S_1)
ANY z **WHERE** P **THEN** S_1	$\exists\, z.(P \wedge Prd_{x,x'}(S_1) \wedge y' = y)$	x=FV(S_1)-z y=FV(P)-FV(S_1)-z
$S_1 \| T_1$	$Prd_{x_0,x'_0}(S_1) \wedge Prd_{x_1,x'_1}(T_1) \wedge y' = y$	$x_0 = MV(S_1)$ $x_1 = MV(T_1)$ $y = FV(S) - x_0 - x_1$
$S_1;T_1$	$Trm_x(S_1) \Rightarrow$ $\exists\, a.([x' := a]Prd(S_1) \wedge [x := a]Prd(T_1))$	$x = FV(S_1)$
$S_1 [] T_1$	$Prd(S_1) \vee Prd(T_1))$	
$(B \circlearrowleft S)_{I,v}$	$I(x') \wedge \neg B(x')$	

FIG. A.4 – *Définition inductive du prédicat avant-après d'une substitution*

A.3 Composant B

Dans cette section sont présentés les différents concepts de structuration et de modularisation d'une spécification **B**. Selon le niveau d'abstraction où l'on se situe, la méthode **B** offre des mécanismes appropriés permettant un développement incrémental de la spécification. Elle permet de gérer la complexité de conception de logiciels de tailles importantes en les décomposant en plusieurs sous-parties. Dans ce qui suit, seules les concepts utilisés dans notre développement sont présentés. Une présentation plus détaillée et complète de l'ensemble des concepts peut être trouvée dans [1].

A.3.1 Spécification abstraite

Machine abstraite. La machine abstraite est le point de départ de tout développement **B**. Elle spécifie les propriétés et comportements d'un système indépendamment de toute considération algorithmique. Une machine abstraite comporte deux aspects complémentaires : statique et dynamique. L'aspect statique est représenté par des variables et des ensembles pouvant être abstraits ou énumérés. Un ensemble abstrait désigne des objets dont on ne souhaite pas préciser la structure ou le type dès le niveau de la spécification. Ces précisions sont différées aux étapes ultérieures du développement. Les propriétés que doit satisfaire le système sont exprimées dans la partie invariant de la machine. L'invariant est exprimé, entre autres, à l'aide des différents opérateurs relationnels **B** donnés dans la figure A.5.

Nom	Syntaxe	Définition	Précondition
Produit cartésien	$s \times t$	$\{x \mapsto y \mid x \in s \wedge y \in t\}$	
Relation	$s \leftrightarrow t$	P($s \times t$) avec P(S): ensemble des parties de S	
Inverse	r^{-1}	$\{x \mapsto y \mid y \mapsto x \in r\}$	$r \in s \leftrightarrow t$
Domaine	$dom(r)$	$\{x \mid x \in s \wedge \exists\, y.\, (y \in t \wedge x \mapsto y \in r)\}$	$r \in s \leftrightarrow t$
Codomaine	$ran(r)$	$dom(r^{-1})$	$r \in s \leftrightarrow t$
Composition	$(q;\, r)$	$\{x \mapsto y \mid (x \mapsto y) \in s \times u \wedge \exists\, z.(z \in t \wedge x \mapsto z \in q \wedge z \mapsto y \in r)\}$	$q \in s \leftrightarrow t \wedge r \in t \leftrightarrow u$
Restriction	$u \lhd r$	$\{x \mapsto y \mid (x, y) \in r \wedge x \in u\}$	$r \in s \leftrightarrow t \wedge u \subseteq s$
Corestriction	$r \rhd u$	$\{x \mapsto y \mid (x, y) \in r \wedge y \in u\}$	$r \in s \leftrightarrow t \wedge u \subseteq t$
Anti-Restriction	$u \ntriangleleft r$	$\{x \mapsto y \mid (x, y) \in r \wedge x \notin u\}$	$r \in s \leftrightarrow t \wedge u \subseteq s$
Anti-Corestriction	$r \ntriangleright u$	$\{x \mapsto y \mid (x, y) \in r \wedge y \notin u\}$	$r \in s \leftrightarrow t \wedge u \subseteq t$
Image	$r[w]$	$\{y \mid y \in t \wedge \exists\, x.\, (x \in w \wedge x \mapsto y \in r)\}$	$r \in s \leftrightarrow t \wedge w \subseteq s$
Produit direct	$f \otimes g$	$\{x \mapsto (y \mapsto z) \mid ((x, y, z) \in s \times u \times v \wedge x \mapsto y \in f \wedge x \mapsto z \in g)\}$	$f \in s \leftrightarrow u \wedge g \in s \leftrightarrow v$
Fonction partielle	$s \nrightarrow t$	$\{f \mid f \in s \leftrightarrow t \wedge \forall x, y, z.\, (x \mapsto y \in f \wedge x \mapsto z \in f \Rightarrow y = z)\}$	
Fonction totale	$s \rightarrow t$	$\{f \mid f \in s \nrightarrow t \wedge dom(f) = s\}$	
Injection partielle	$s \rightarrowtail\!\!\!\!\!\!+\; t$	$\{f \mid f \in s \nrightarrow t \wedge f^{-1} \in t \nrightarrow s\}$	
Injection totale	$s \rightarrowtail t$	$s \rightarrowtail\!\!\!\!\!\!+\; t \cap s \rightarrow t$	
Surjection partielle	$s \twoheadrightarrow\!\!\!\!\!\!+\; t$	$\{f \mid f \in s \nrightarrow t \wedge ran(f) = t\}$	
Surjection totale	$s \twoheadrightarrow t$	$s \twoheadrightarrow\!\!\!\!\!\!+\; t \cap s \rightarrow t$	
Bijection	$s \rightarrowtail\!\!\!\!\twoheadrightarrow t$	$s \rightarrowtail t \cap s \twoheadrightarrow t$	
Transformée en relation	$rel(r)$	$\{(x,y) \mid x \in dom(r) \wedge y \in r(x)\}$	$r \in s \nrightarrow \mathbb{P}(t)$
Transformée en fonction	$fnc(r)$	$\{(x,y) \mid x \in dom(r) \wedge y = r[\{x\}]\}$	$r \in s \leftrightarrow t$

FIG. A.5 – *Opérateurs relationnels du langage B*

À chaque variable, une valeur initiale est attribuée. Cette initialisation permet, en outre, de vérifier que l'invariant n'est pas contradictoire. L'aspect dynamique est représenté par un ensemble d'opérations exprimées sous forme de pré-post condition. La partie post-condition est représentée par une substitution généralisée. Au niveau le plus abstrait, les substitutions séquentielles et les boucles ne sont pas admises.

Correction d'une machine abstraite. Prouver la correction d'une machine *M* d'invariant *INV* revient à prouver que :

– son initialisation *INIT* vérifie (établit) l'invariant *INV* :

$$[INIT]\ INV$$

– chaque opération *Op* appelée sous sa précondition *PRE* et l'invariant *INV* réétablit cet invariant (l'invariant est toujours vrai après exécution de *Op*) :

$$(INV \wedge PRE) \Rightarrow [Op]\ INV$$

Exemple de machine abstraite. Soit la machine abstraite $Exemple$ contenant la variable ensembliste Res représentant un ensemble de ressources. On suppose que ces ressources sont numérotées de 1 à m. Cette machine spécifie l'opération *allouer(n)* qui alloue n ressources supplémentaires.

MACHINE *Exemple*

VARIABLES Res
INVARIANT $Res \subseteq NAT_1$
INITIALISATION $Res := \emptyset$
OPERATIONS
allouer(n)=
PRE $n \leq card(Res)$
THEN
  ANY res **WHERE** $res \subseteq Res \wedge card(res) = n$
  THEN $Res := Res - res$
  END
END
END

Cette machine est correcte car en effet :

1. $[Res := \emptyset](Res \subseteq NAT_1)$ c'est-à-dire $\emptyset \subseteq NAT_1$
2. $(Res \subseteq NAT_1 \wedge n \leq card(Res)) \Rightarrow \forall\ res.(res \subseteq Res \Rightarrow [Res := Res - res](Res \subseteq NAT_1))$

sont des formules vraies.

Modularisation d'une spécification abstraite. Au niveau le plus abstrait, quatre clauses de modularisation sont possibles :

1. **INCLUDES** : permet de définir une liste de machines qui seront incorporées dans la machine incluante. Conceptuellement, les ensembles et les variables des machines incluses sont ajoutés à ceux de la machine incluante. Le tableau suivant précise la visibilité des ensembles, variables et opérations d'une machine *B* incluse dans une machine *A* :

Clauses de A / Constituants de B	INVARIANT	OPERATIONS
Ensembles	visible	visible
Variables	visible	en lecture
Opérations		visible

Remarque A.3.1

- Bien que les opérations d'une machine incluse soient visibles dans la machine incluante, une opération de la machine incluante ne peut faire appel qu'à une seule opération de la machine incluse. Cette restriction est due à l'exécution simultanée des spécifications abstraites. En effet, si les deux opérations appelées modifient une même variable, alors le résultat devient imprévisible et donc l'invariant de la machine peut être violé. Notons que cette restriction est assez forte. En effet, il suffirait que les deux opérations appelées n'aient pas de variables modifiées communes.
- La clause **INCLUDES** est transitive. Si une machine M_1 inclut une machine M_2 qui à son tour inclut une machine M_3, alors les ensembles et les variables de M_3 sont visibles à partir de M_1.

2. **USES** : permet de définir une liste des machines qui seront utilisées par la machine utilisatrice. La clause **USES** est exclusivement employée pour le partage de données entre machines. L'ensemble des machines utilisatrices et la machine utilisée doivent être incluses dans une seule et même machine afin de pouvoir contrôler que les invariants des machines utilisatrices sont cohérents (non contradictoires).

Le tableau suivant précise la visibilité des ensembles, variables et opérations d'une machine *B* utilisée dans une machine *A* :

Constituants de B \ Clauses de A	INVARIANT	OPERATIONS
Ensembles	visible	visible
Variables	visible	en lecture
Opérations		

La clause **USES** n'est pas transitive. Si une machine M_1 utilise une machine M_2 qui à son tour utilise une machine M_3, alors les ensembles et les variables de M_3 ne sont pas visibles à partir de M_1.

3. **SEES** : cette clause est similaire à la clause **USES**. La différence entre ces deux clauses concerne la visibilité des variables abstraites et des opérations. Les variables d'une machine A vue par une machine B ne sont visibles que dans les opérations. Contrairement à la clause **USES**, les opérations de lecture de la machine A peuvent être utilisées dans celles de B.
4. **PROMOTES** : cette clause n'existe que si la clause **INCLUDES** est déclarée. Elle permet de définir une liste de noms d'opérations des machines incluses qui deviennent des opérations propres à la machine incluante. Ainsi, ces opérations sont visibles de toute machine incluant de nouveau la machine incluante.

A.3.2 Raffinement

Le raffinement est le processus de transformation de spécifications abstraites en spécifications plus concrètes, c'est-à-dire supportées par le langage d'implémentation cible choisi. Un raffinement peut être opéré sur une machine abstraite ou un autre raffinement. Le raffinement d'une machine abstraite vers une implémentation peut être réalisé en plusieurs étapes. Ce raffinement progressif permet, d'une part, de faciliter l'expression de la relation de raffinement existante entre le niveau abstrait et le niveau concret et, d'autre part, d'alléger la preuve globale du raffinement à travers une succession de preuves plus élémentaires et donc plus simples. Le dernier niveau de raffinement est appelé implémentation. Le composant implémentation contient des variables et des substitutions directement traduisibles dans le langage de programmation cible choisi. Le principe général du raffinement est illustré par le schéma de la figure A.6.

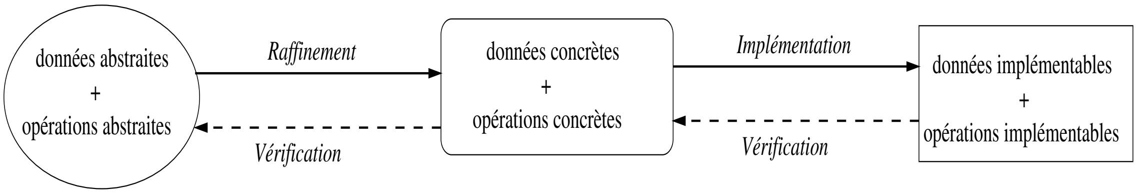


FIG. A.6 – *Principe Général du Raffinement B*

Raffinement des données et des substitutions. Un raffinement **B** agît sur les données et/ou les substitutions. Le raffinement des données consiste à introduire de nouvelles variables ou à remplacer les variables du composant raffiné par des variables plus proches de celles utilisées dans le langage d'implémentation choisi (ex : remplacer une variable du type ensemble par une variable du type tableau,...). Dans ce type de raffinement, un prédicat *J*, dit invariant de collage, est à préciser dans la clause invariant du composant qui raffine. Ce prédicat permet de typer les nouvelles variables introduites et d'exprimer le relation existante entre ces variables et les variables du composant raffiné. Le raffinement algorithmique (cas particulier du raffinement des données associé à un invariant de collage identité sur l'ensemble des variables) consiste à remplacer une substitution abstraite *S* par une substitution concrète *S'* (ex : affaiblissement (élimination) des préconditions, remplacement d'une substitution parallèle par une séquence, ...). Tout raffinement de données implique le raffinement des substitutions en fonction des variables concrètes. En plus des substitutions admises au niveau le plus abstrait, la substitution séquentielle peut être utilisée pour l'expression des raffinements de l'initialisation et des opérations.

Correction du raffinement Dans cette partie est donnée la définition d'un raffinement correct. Nous utilisons la sémantique relationnelle des substitutions qui permet une meilleure traduction de l'idée intuitive qu'on a du raffinement et du changement d'état.

Définition A.3.1 *Pour toutes substitutions S et T, S est raffinée par T par l'invariant de collage J, que l'on notera $S \sqsubseteq_J T$, si et seulement si :*

(i) $\exists (u,v).J(u,v)$
(ii) $Trm_x(S) \wedge J(x,y) \Rightarrow (Trm_y(T) \wedge \forall\, y''.(Prd_{y,y''}(T) \Rightarrow \exists\, x'.(Prd_{x,x'}(S) \wedge J(x',y''))))$

Les ensembles de variables x et y désignent les variables libres de S et T respectivement. Intuitivement, cette définition signifie que :

- il existe au moins une interprétation (u,v) du couple de variables (x,y) qui satisfait l'invariant de collage J : *J* n'est pas une contradiction,
- si l'exécution de la substitution abstraite *S* termine, alors la substitution concrète *T* termine aussi, et pour tout résultat possible y'' de l'exécution de *T* alors, il existe un résultat x' de l'exécution de *S* tel que le couple (x′, y″) vérifie l'invariant de collage J.

Raffinement d'un composant Un composant M_1 est raffiné par un composant M_2 si et seulement si les deux conditions suivantes sont vérifiées :

1. l'initialisation de M_2 raffine l'initialisation de M_1,
2. chaque opération de M_2 raffine sa correspondante dans M_1. Une opération op et son raffinement op' ont la même signature.

Raffinement de la machine *Exemple***.** On désire raffiner la machine *Exemple* en s'intéressant au nombre d'éléments que contient l'ensemble *Res*. Pour cela, on définit le raffinement suivant :

REFINEMENT *Exemple*$_1$
REFINES *Exemple*
VARIABLES Res_1
INVARIANT $Res_1 = card(Res)$
INITIALISATION $Res_1 := 0$
OPERATIONS
allouer(n)=
BEGIN
$Res_1 := Res_1 - n$
END
END

Ce raffinement est correct par rapport à la spécification abstraite, car en effet :

1. $\exists\, u,v.(v = card(u))$ (ceci est vrai pour $u = \emptyset$ et $v = 0$). // condition i de la définition A.3.1

2. $(true \wedge Res_1 = card(Res)) \rightarrow (true \wedge \forall\, Res_1''.(Res_1'' = 0 \rightarrow \exists\, Res'.(Res' = \emptyset \wedge Res_1'' = card(Res'))))$
(ceci est vrai pour $Res' = \emptyset$) // condition ii de la définition A.3.1 pour l'initialisation
3. $(n \leq card(Res) \wedge \exists\ res.(res \subseteq Res \wedge card(res) = n) \wedge Res_1 = card(Res)) \rightarrow true \wedge \forall\ Res_1''.((Res_1'' = Res_1 - n) \rightarrow \exists\, Res'.(Res' = Res - res \wedge Res_1'' = card(Res')))$ // condition ii de la définition A.3.1 pour l'opération $allouer$

ceci est vrai pour $Res' = Res - res$. En effet :

Res'' = *card(Res-res)*
= $card(Res) - card(res)$ (car $res \subseteq Res$)
= $Res_1 - n$

Implémentation L'implémentation désigne le dernier niveau de raffinement. Elle contient des spécifications les plus concrètes possibles, c-à-d, les plus proches du langage d'implantation cible. Les spécification concrètes sont directement traduisibles dans le langage cible choisi. Contrairement à un raffinement, une implémentation ne possède que des variables concrètes (directement implémentables). Chaque ensemble abstrait doit être valué en implémentation. Les substitutions préconditionnées, simultanées, choix et **ANY** ne sont pas admises dans une implémentation.

Modularisation d'une implémentation Une implémentation peut importer (clause **IMPORTS**) une à plusieurs machines abstraites afin de réaliser ses propres données et opérations. L'utilisation de ce lien permet aussi de spécifier l'architecture du logiciel final généré à partir de cette implémentation. Le tableau suivant précise la visibilité des ensembles, variables et opérations d'une machine *B* importée par une implémentation *A*.

Clauses de A / Constituants de B	INVARIANT	OPERATIONS
Ensembles	visible	visible
Variables	visible	
Opération		visible

Annexe B

Condition de réutilisation de preuves

B.1 Preuve par réutilisation du raffinement de la substitution parallèle par une substitution séquentielle

Soient $S_{a_1}, T_{a_2}, U_{b_1}$ et V_{b_2} quatre substitutions et $J(a_1, a_2), J(b_1, b_2)$ deux prédicats tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2} \tag{H$_1$}$$

$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2} \tag{H$_2$}$$

Soit $J(a_1 \cup b_1, a_2 \cup b_2)$ un prédicat tel que:

$$(S_{a_1} \| U_{b_1}) \sqsubseteq_{J(c_1,c_2)} (T_{a_2};\ V_{b_2}) \tag{G$_1$}$$

où: $c_1 = a_1 \cup b_1$ et $c_2 = a_2 \cup b_2$.

On voudrait réutiliser les preuves des raffinements (H_1) et (H_2) lors de la preuve du raffinement (G_1). Cela revient à définir les conditions nécessaires et suffisantes pour que cette réutilisation soit possible.

Le problème qu'on se pose est donc le suivant:

Hypothèses: par définition du raffinement, on a les hypothèses suivantes:

$$(\mathrm{H}_1) \equiv (Trm(S_{a_1}) \wedge J(a_1,a_2)) \rightarrow [Trm(T_{a_2}) \wedge \forall\, a_2'.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2')))] \tag{H$_3$}$$

$$(\mathrm{H}_2) \equiv (Trm(U_{b_1}) \wedge J(b_1,b_2)) \rightarrow [Trm(V_{b_2}) \wedge \forall\, b_2''.(Prd(V_{b_2}) \rightarrow \exists\, b_1'.(Prd(U_{b_1}) \wedge J(b_1',b_2'')))] \tag{H$_4$}$$

But: établir une preuve, par réutilisation de (H_1) et (H_2), de:

$$\begin{aligned}(Trm(S_{a_1} \| U_{b_1}) \wedge J(c_1,c_2)) \rightarrow [Trm(T_{a_2};\ V_{b_2}) \wedge \forall\, c_2''.(Prd(T_{a_2};\ V_{b_2}) \rightarrow \\ \exists\, c_1'.(Prd(S_{a_1} \| U_{b_1}) \wedge J(c_1',c_2'')))]\end{aligned} \tag{G$_2$}$$

Soient:

- a_1^m (resp. b_1^m): les variables de a_1 (resp. b_1) modifiées par S_{a_1} (resp. U_{b_1}): $MV(S_{a_1})$ (resp. $MV(U_{b_1})$),
- a_1^f (resp. b_1^f): les variables de a_1 (resp. b_1) non modifiées par S_{a_1} (resp. U_{b_1}): $FV(S_{a_1}) - MV(S_{a_1})$ (resp. $FV(U_{b_1}) - MV(U_{b_1})$),
- d_1: les variables de $a_1 \cup b_1$ non modifiées par $S_{a_1} \| U_{b_1}$.

On a donc:

$$a_1 = a_1^m \cup a_1^f \qquad b_1 = b_1^m \cup b_1^f \qquad d_1 = (a_1^f \cup b_1^f) - (a_1^m \cup b_1^m)$$

On a (par définition):

$$\begin{aligned} Trm(S_{a_1} \| U_{b_1}) &= Trm(S_{a_1}) \wedge Trm(U_{b_1}) \\ Trm(T_{a_2};\ V_{b_2}) &= Trm(T_{a_2}) \wedge \forall\, y.([a_2' := y]Prd_{a_2,a_2'}(T_{a_2}) \rightarrow [a_2 := y]\, Trm(V_{b_2})) \end{aligned}$$

et

$$\begin{aligned} Prd(S_{a_1} \| U_{b_1}) &= Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \\ Prd(T_{a_2};\ V_{b_2}) &= Trm(T_{a_2}) \rightarrow \exists\, t.([a_2' := t]Prd_{a_2,a_2'}(T_{a_2}) \wedge [a_2 := t]Prd_{c_2,c_2''}(V_{b_2}))) \end{aligned}$$

En rapportant ces définitions dans (G_2), puis par application de la règle de déduction, on obtient les sous buts suivants:

$$Trm(T_{a_2}) \tag{G_3}$$

$$\forall\, y.([a_2' := y]Prd_{a_2,a_2'}(T_{a_2}) \rightarrow [a_2 := y]\, Trm(V_{b_2})) \tag{G_4}$$

$$\forall\, c_2''.(Prd(T_{a_2};\ V_{b_2}) \rightarrow \exists\, c_1'.(Prd(S_{a_1} \| U_{b_1}) \wedge J(c_1', c_2''))) \tag{G_5}$$

sous les hypothèses supplémentaires:

$$Trm(S_{a_1}) \tag{H_5}$$

$$Trm(U_{b_1}) \tag{H_6}$$

$$J(c_1, c_2) \tag{H_7}$$

Preuve de G_3: pour prouver (G_3) par réutilisation de (H_3), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(S_{a_1})$ et $J(a_1, a_2)$. $Trm(S_{a_1})$ étant l'hypothèse (H_5), il reste à générer de notre environnement d'hypothèses, constitué de (H_5), (H_6) et (H_7), l'hypothèse:

$$J(a_1, a_2) \tag{H_8}$$

Une condition nécessaire et suffisante à la réutilisation est:

$$\boxed{(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1, c_2)) \rightarrow J(a_1, a_2)}$$

(H_5), (H_8) et (H_3) prouvent alors (G_3) et génèrent une hypothèse supplémentaire:

$$\forall\, a_2{}'.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1', a_2'))) \tag{H_9}$$

Preuve de G_4: par l'élimination du quantificateur universel, et l'application de la règle de déduction, (G_4) est équivalent à:

$$[a_2 := y]\, Trm(V_{b_2}) \tag{$G_{4.1}$}$$

sous l'hypothèse supplémentaire:

$$[a_2' := y]Prd_{a_2,a_2'}(T_{a_2}) \tag{H_{10}}$$

Pour prouver ($G_{4.1}$) par réutilisation de (H_4), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(U_{b_1})$ et $[a_2 := y]J(b_1,b_2)$. $Trm(U_{b_1})$ étant l'hypothèse (H_6), il reste à générer de notre environnement d'hypothèses, constitué de (H_5), (H_6), (H_7) et (H_{10}), l'hypothèse:

$$[a_2 := y]J(b_1,b_2) \tag{H$_{11}$}$$

Une deuxième condition nécessaire et suffisante à la réutilisation est:

$$\boxed{(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge [a_2' := y]Prd_{a_2,a_2'}(T_{a_2})) \rightarrow [a_2 := y]J(b_1,b_2)}$$

(H_6), (H_{11}) et (H_4) prouvent alors ($G_{4.1}$) et génèrent une hypothèse supplémentaire:

$$\forall\, b_2{}''.([a_2 := y]Prd(V_{b_2}) \rightarrow \exists\, b_1'.(Prd(U_{b_1}) \wedge J(b_1',b_2''))) \tag{H$_{12}$}$$

Preuve de (G_5): par l'élimination du quantificateur universel et l'application de la règle de déduction, le but à prouver devient:

$$\exists\, c_1'.(Prd_{a_1^m,a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m,b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \wedge J(c_1',c_2'')) \tag{G$_{5.1}$}$$

sous l'hypothèse supplémentaire:

$$Trm(T_{a_2}) \rightarrow \exists\, t.([a_2' := t]Prd_{a_2,a_2'}(T_{a_2}) \wedge [a_2 := t]Prd_{c_2,c_2''}(V_{b_2}))) \tag{H$_{13}$}$$

(G_3) et (H_{13}) donnent:

$$\exists\, t.([a_2' := t]Prd_{a_2,a_2'}(T_{a_2}) \wedge [a_2 := t]Prd_{c_2,c_2''}(V_{b_2})) \tag{H$_{14}$}$$

Soit sol_t la valeur de t qui satisfait cette dernière hypothèse. On a donc:

$$[a_2' := sol_t]Prd_{a_2,a_2'}(T_{a_2}) \tag{H$_{15}$}$$
$$[a_2 := sol_t]Prd_{c_2,c_2''}(V_{b_2}) \tag{H$_{16}$}$$

(H_{15}) et l'instanciation des variables $a_2{}'$ par sol_t dans (H_9) donnent:

$$\exists\, a_1'.(Prd(S_{a_1}) \wedge [a_2' := sol_t]J(a_1',a_2')) \tag{H$_{17}$}$$

Comme $c_2 = a_2 \cup b_2$, en posant $d_2 = a_2 - b_2$, l'hypothèse (H_{16}) se réécrit en:

$$[a_2 := sol_t]Prd_{b_2,b_2''}(V_{b_2}) \tag{H$_{18}$}$$
$$d_2{}'' = d_2{}' \tag{H$_{19}$}$$

(H_{18}) et l'instanciation des variables $b_2{}''$ par elles-mêmes dans (H_{12}) donnent:

$$\exists\, b_1'.(Prd(U_{b_1}) \wedge J(b_1',b_2'')) \tag{H$_{20}$}$$

Soient $sol_{a_1'}$ et $sol_{b_1'}$ des valeurs de a_1' et b_1'qui vérifient les hypothèses (H_{17}) et (H_{20}) respectivement, on a donc:

$$[a_1' := sol_{a_1'}]Prd(S_{a_1}) \tag{H$_{21}$}$$
$$[a_1' := sol_{a_1'};\ a_2' := sol_t]J(a_1',a_2') \tag{H$_{22}$}$$
$$[b_1' := sol_{b_1'}]Prd(U_{b_1}) \tag{H$_{23}$}$$
$$[b_1' := sol_{b_1'}]J(b_1',b_2'') \tag{H$_{24}$}$$

Réutiliser la preuve des deux raffinements $S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$ et $U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$ revient à vérifier que les solutions $sol_{a'_1}$ et $sol_{b'_1}$ sont aussi une solution de ce raffinement. Pour cela, on doit vérifier que $(sol_{a'_1}, sol_{b'_1})$ satisfait le but ($G_{5.1}$), c.à.d:

$$[a'_1 := sol_{a'_1} \| b'_1 := sol_{b'_1}](Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge d'_1 = d_1 \wedge J(c'_1, c''_2)) \tag{G5.2}$$

Par définition de la substitution parallèle, ${a_1}^m \cap {b_1}^m = \emptyset$. Le but ($G_{5.2}$) se décompose en:

$$[a_1^{m\prime} := sol_{a_1^{m\prime}} \| b_1^{f\prime} := sol_{b_1^{f\prime}}]Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \tag{G5.3}$$

$$[a_1^{f\prime} := sol_{a_1^{f\prime}} \| b_1^{m\prime} := sol_{b_1^{m\prime}}]Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \tag{G5.4}$$

$$[a'_1 := sol_{a'_1} \| b'_1 := sol_{b'_1}](d'_1 = d_1) \tag{G5.5}$$

$$[a'_1 := sol_{a'_1} \| b'_1 := sol_{b'_1}]J(c'_1, c''_2) \tag{G5.6}$$

L'hypothèse (H_{21}) se réécrit en:

$$[a_1^{m\prime} := sol_{a_1^{m\prime}}]Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \tag{H25}$$

$$a_1^{f\prime} = a_1^f \tag{H26}$$

Le but ($G_{5.3}$) se prouve par (H_{25}) sous la condition:

$$\boxed{a_1^m \cap b_1^f = \emptyset}$$

de même, l'hypothèse (H_{23}) se réécrit en:

$$[b_1^{m\prime} := sol_{b_1^{m\prime}}]Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \tag{H27}$$

$$b_1^{f\prime} = b_1^f \tag{H28}$$

Le but ($G_{5.4}$) se prouve par (H_{28}) sous la condition:

$$\boxed{b_1^m \cap a_1^f = \emptyset}$$

Le but ($G_{5.5}$) est trivial, car d_1 représente l'ensemble des variables communes à ${a_1}^f$ et ${b_1}^f$. Enfin, le but ($G_{5.6}$) doit pouvoir se déduire de notre environnement de preuve. On doit donc avoir:

$$\boxed{\begin{array}{l}(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1, c_2) \wedge [a'_1 := sol_{a'_1}]Prd(S_{a_1}) \wedge [a'_1 := sol_{a'_1};\ a'_2 := sol_t]J(a'_1, a'_2) \wedge \\ [b'_1 := sol_{b'_1}]Prd(U_{b_1}) \wedge [b'_1 := sol_{b'_1}]J(b'_1, b''_2)) \rightarrow [a'_1 := sol_{a'_1};\ b'_1 := sol_{b'_1}]J(a_1^{m\prime} \cup b_1^{m\prime} \cup d_1, c''_2)\end{array}}$$

Résumé
Soient S_{a_1}, T_{a_2}, U_{b_1}, V_{b_2} quatre substitutions et $J(a_1, a_2)$, $J(b_1, b_2)$ deux prédicats tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$$
$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$$

Soit $J(a_1 \cup b_1, a_2 \cup b_2)$ un prédicat vérifiant ($a_1 \cup b_1 = c_1$ et $a_2 \cup b_2 = c_2$):

$i.$ $(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1, c_2)) \rightarrow J(a_1, a_2)$

$ii.$ pour toute valeur y de a'_2, on a:

$$(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1, c_2) \wedge [a'_2 = y]Prd_{a_2, a'_2}(T_{a_2})) \rightarrow [a_2 = y]J(b_1, b_2)$$

$iii.$ $a_1^m \cap b_1 = \emptyset$

iv $a_1 \cap b_1^m = \emptyset$

alors:
pour chaque valeur c_2'' vérifiant:

$$Prd_{c_2,c_2''}(T_{a_2};\ V_{b_2})$$

associée à la valeur t de a_2' vérifiant:

$$[a_2' = t]Prd_{a_2,a_2'}(T_{a_2}) \wedge [a_2 = t]Prd_{c_2,c_2''}(V_{b_2})$$

le couple de valeur $(sol_{a_1'}, sol_{b_1'})$ vérifiant:

– $sol_{a_1'}$ vérifie le raffinement $S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$ pour a_2' égale à t,
– $sol_{b_1'}$ vérifie le raffinement $U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$ pour a_2 égale à t,

permet de prouver le raffinement $(S_{a_1} \| U_{b_1}) \sqsubseteq_{J(a_1 \cup b_1, a_2 \cup b_2)} (T_{a_2};\ V_{b_2})$ si et seulement si:

$$(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge [a_1' = sol_{a_1'}]Prd(S_{a_1}) \wedge [a_1' = sol_{a_1'};\ a_2' = sol_t]J(a_1',a_2') \wedge$$
$$[b_1' = sol_{b_1'}]Prd(U_{b_1}) \wedge [b_1' = sol_{b_1'}]J(b_1',b_2'')) \rightarrow [a_1' = sol_{a_1'};\ b_1' = sol_{b_1'}]J(a_1^{m\prime} \cup b_1^{m\prime} \cup d_1, c_2'')$$

Remarques

1. La condition (i) est une condition nécessaire de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $S_{a_1} \sqsubseteq T_{a_2}$, alors il faut être capable d'exhiber dans l'environnement de preuve de $(S_{a_1} \| U_{b_1} \sqsubseteq T_{a_2};\ V_{b_2})$ l'invariant de collage de $S_{a_1} \sqsubseteq T_{a_2}$. Dans le cas contraire la stratégie de réutilisation n'est pas justifiée.
2. La condition (ii) est une condition nécessaire de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $U_{b_1} \sqsubseteq V_{b_2}$, alors il faut être capable d'exhiber dans l'environnement de preuve après exécution de T_{a_2} l'invariant de collage de $U_{b_1} \sqsubseteq V_{b_2}$. Dans le cas contraire la stratégie de réutilisation n'est pas justifiée.
3. les conditions (iii) et (iv) sont des conditions nécessaires de correction du raffinement.
4. La dernière condition n'est une condition nécessaire que pour la réutilisation des solutions des raffinements de $S_{a_1} \sqsubseteq T_{a_2}$ et $U_{b_1} \sqsubseteq V_{b_2}$. Une autre solution du raffinement sans réutilisation peut éventuellement exister.
5. La réutilisation des preuves de raffinement de $S_{a_1} \sqsubseteq T_{a_2}$ et et $U_{b_1} \sqsubseteq V_{b_2}$ sont réalisables pour tout couple de valeurs (a_1',b_1') qui vérifie la dernière condition: on n'exige pas que tous les couples vérifient cette condition.
6. Si les substitutions S_{a_1} et U_{b_1} sont déterministes alors la dernière condition est vérifiée. En effet, il existe une seule valeur du couple (a_1',b_1') qui satisfait les raffinements $(S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2})$ et $(U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2})$, celle-ci est déterminée par $(Prd(S_{a_1}, Prd(U_{b_1}))$. De plus, une solution possible du raffinement $(S_{a_1} \| U_{b_1} \sqsubseteq_{J(c_1,c_2)} T_{a_2};\ V_{b_2})$ doit satisfaire les prédicats $Prd(S_{a_1})$ et $Prd(U_{b_1})$ qui n'admettent qu'une seule solution: les deux raffinements ont donc le même espace de solutions.
7. Si les prédicats $J(a_1,a_2)$ et $J(b_1,b_2)$ sont fonctionnels en a_1 et b_1 respectivement alors la dernière condition est vérifiée. En effet, une solution possible du raffinement $(S_{a_1} \| U_{b_1} \sqsubseteq_{J(c_1,c_2)} T_{a_2};\ V_{b_2})$ doit satisfaire le prédicat $J(c_1,c_2)$. D'après (i) et (ii) cette solution satisfait les prédicats $J(a_1,a_2)$ et $J(b_1,b_2)$. Or $J(a_1,a_2)$ et $J(b_1,b_2)$ n'admettent qu'une solution unique (a_1',b_1'): cette solution unique est donc solution de $(S_{a_1} \| U_{b_1} \sqsubseteq_{J(c_1,c_2)} T_{a_2};\ V_{b_2})$.

B.2 Preuve par réutilisation du raffinement de la substitution du choix

Soient $S_{a_1}, T_{a_2}, U_{b_1}$ et V_{b_2} quatre substitutions et $J(a_1,a_2), J(b_1,b_2)$ deux prédicats tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2} \qquad (\mathrm{H}_1)$$

$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2} \qquad (\mathrm{H}_2)$$

Soit $J(a_1 \cup b_1, a_2 \cup b_2)$ un prédicat tel que:

$$S_{a_1} [\!]\, U_{b_1} \sqsubseteq_{J(a_1 \cup b_1, a_2 \cup b_2)} T_{a_2} [\!]\, V_{b_2} \qquad (\mathrm{G}_1)$$

On voudrait réutiliser les preuves des raffinements (H_1) et (H_2) lors de la preuve du raffinement (G_1). Cela revient à définir les conditions nécessaires et suffisantes pour que cette réutilisation soit possible.

Le problème qu'on se pose est donc (on pose $c_1 = a_1 \cup b_1$ et $c_2 = a_2 \cup b_2$):

Hypothèses: par définition du raffinement, on a les hypothèses suivantes:

$$(\mathrm{H}_1) \equiv (Trm(S_{a_1}) \wedge J(a_1,a_2)) \rightarrow [Trm(T_{a_2}) \wedge \forall\, a_2''.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2'')))] \qquad (\mathrm{H}_3)$$

$$(\mathrm{H}_2) \equiv (Trm(U_{b_1}) \wedge J(b_1,b_2)) \rightarrow [Trm(V_{b_2}) \wedge \forall\, b_2''.(Prd(V_{b_2}) \rightarrow \exists\, b_1'.(Prd(U_{b_1}) \wedge J(b_1',b_2'')))] \qquad (\mathrm{H}_4)$$

But: établir une preuve, par réutilisation de (H_1) et (H_2), de:

$$\begin{aligned}(Trm(S_{a_1} [\!]\, U_{b_1}) \wedge J(c_1,c_2)) \rightarrow [Trm(T_{a_2} [\!]\, V_{b_2}) \wedge \forall\, c_2''.(Prd(T_{a_2} [\!]\, V_{b_2}) \rightarrow \\ \exists\, c_1'.(Prd(S_{a_1} [\!]\, U_{b_1}) \wedge J(c_1',c_2'')))]\end{aligned} \qquad (\mathrm{G}_2)$$

Soient:
- d_1 (resp. d_2): les variables libres de a_1 (resp. a_2) n'apparaissant pas dans b_1 (resp. b_2),
- e_1 (resp. e_2): les variables libres de b_1 (resp. b_2) n'apparaissant pas dans a_1 (resp. a_2).

On pose:

$$\begin{aligned} d_1 &= a_1 - b_1 \qquad & e_1 &= b_1 - a_1 \\ d_2 &= a_2 - b_2 \qquad & e_2 &= b_2 - a_2 \end{aligned}$$

on a (par définition)

$$\begin{aligned} Trm(S_{a_1} [\!]\, U_{b_1}) &= Trm(S_{a_1}) \wedge Trm(U_{b_1}) \\ Trm(T_{a_2} [\!]\, V_{b_2}) &= Trm(T_{a_2}) \wedge Trm(V_{b_2}) \end{aligned}$$

et

$$\begin{aligned} Prd(S_{a_1} [\!]\, U_{b_1}) &= (Prd(S_{a_1}) \wedge e_1' = e_1) \vee (Prd(U_{b_1}) \wedge d_1' = d_1) \\ Prd(T_{a_2} [\!]\, V_{b_2}) &= (Prd(T_{a_2}) \wedge e_2' = e_2) \vee (Prd(V_{b_2}) \wedge d_2' = d_2) \end{aligned}$$

En rapportant ces définitions dans (G_2), le but à prouver est donc:

$$\begin{aligned}(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2)) \rightarrow [Trm(T_{a_2}) \wedge Trm(V_{b_2}) \wedge \forall\, c_2''.(Prd(T_{a_2} [\!]\, V_{b_2}) \rightarrow \\ \exists\, c_1'.(Prd(S_{a_1} [\!]\, U_{b_1}) \wedge J(c_1',c_2'')))]\end{aligned} \qquad (\mathrm{G}_3)$$

Par la règle de déduction, on obtient les trois sous buts suivants:

$$Trm(T_{a_2}) \qquad (\mathrm{G}_4)$$

$$Trm(V_{b_2}) \qquad (\mathrm{G}_5)$$

$$\forall\, c_2''.(Prd(T_{a_2} [\!]\, V_{b_2}) \rightarrow \exists\, c_1'.(Prd(S_{a_1} [\!]\, U_{b_1}) \wedge J(c_1',c_2''))) \qquad (\mathrm{G}_6)$$

sous les hypothèses supplémentaires:

$$Trm(S_{a_1}) \tag{H$_5$}$$
$$Trm(U_{b_1}) \tag{H$_6$}$$
$$J(c_1,c_2) \tag{H$_7$}$$

Preuve de (G_4): pour prouver (G_4) par réutilisation de (H_3), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(S_{a_1})$ et $J(a_1,a_2)$. $Trm(S_{a_1})$ étant l'hypothèse (H_5), il reste à générer de notre environnement d'hypothèses, constitué de (H_5), (H_6) et (H_7), l'hypothèse:

$$J(a_1,a_2) \tag{H$_8$}$$

L'une des conditions nécessaires et suffisantes à la réutilisation est:

$$\boxed{(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2)) \rightarrow J(a_1,a_2)}$$

(H_5), (H_8) et (H_3) prouvent (G_4) et génèrent une hypothèse supplémentaire:

$$\forall\, a_2''.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2''))) \tag{H$_9$}$$

Remarque: La condition (H_8) n'est nécessaire que pour la réutilisation de la preuve de correction du raffinement $S_{a_1} \sqsubseteq T_{a_2}$: T_{a_2} peut éventuellement terminer sans que cette condition soit vérifiée: $Trm(T_{a_2}) = True$.

Preuve de (G_5): pour prouver (G_5) par réutilisation de (H_4), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(U_{b_1})$ et $J(b_1,b_2)$. $Trm(U_{b_1})$ étant l'hypothèse (H_6), il reste à générer de notre environnement d'hypothèses, constitué de (H_5), (H_6) et (H_7), l'hypothèse:

$$J(b_1,b_2) \tag{H$_{10}$}$$

Une deuxième condition nécessaire et suffisante à la réutilisation est:

$$\boxed{(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2)) \rightarrow J(b_1,b_2)}$$

(H_6), (H_{10}) et (H_4) prouvent alors (G_5) et génèrent une hypothèse supplémentaire:

$$\forall\, b_2''.(Prd(V_{b_2}) \rightarrow \exists\, b_1'.(Prd(U_{b_1}) \wedge J(b_1',b_2''))) \tag{H$_{11}$}$$

Remarque: La condition (H_{10}) n'est nécessaire que pour la réutilisation de la preuve de correction du raffinement $U_{b_1} \sqsubseteq V_{b_2}$: V_{b_2} peut éventuellement terminer sans que cette condition soit vérifiée: $Trm(V_{b_2}) = True$.

Preuve de (G_6) : soient $f_1 = a_1 \cap b_1$ et $f_2 = a_2 \cap b_2$, on a

$$\begin{aligned} a_1 &= f_1 \cup d_1 \\ a_2 &= f_2 \cup d_2 \\ b_1 &= f_1 \cup e_1 \\ b_2 &= f_2 \cup e_2 \end{aligned}$$

En rapportant ces définitions dans (G_6), puis par élimination du quantificateur universel, le but à prouver devient donc:

$$\exists\, f_1',d_1',e_1'.(((Prd(S_{a_1}) \wedge e_1' = e_1) \vee (Prd(U_{b_1}) \wedge d_1' = d_1)) \wedge J(f_1' \cup d_1' \cup e_1', f_2'' \cup d_2'' \cup e_2'')) \tag{G$_{6.1}$}$$

sous l'hypothèse supplémentaire:

$$(Prd(T_{a_2}) \wedge e_2'' = e_2) \vee (Prd(V_{b_2}) \wedge d_2'' = d_2) \tag{H$_{12}$}$$

Compte tenu de cette dernière hypothèse, deux cas sont à distinguer:

- Si $Prd(T_{a_2}) \wedge e_2'' = e_2$

  L'instanciation des variables a_2'' par elles-mêmes dans (H$_9$) et l'application de *Modus Ponens* donnent:

$$\exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2'')) \tag{H$_{13}$}$$

Soit $(sol_{f_1'},sol_{d_1'})$ la valeur de a_1' qui satisfait (H$_{13}$). On a donc:

$$[f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]Prd(S_{a_1}) \tag{H$_{14}$}$$

$$[f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]J(f_1' \cup d_1',a_2'') \tag{H$_{15}$}$$

Deux cas de réutilisation sont à distinguer:

1. Réutilisation de la preuve (H$_4$):
Cette preuve peut être réutilisée si $Prd(V_{b_2})$ est vrai. Dans ce cas l'ensemble e_2 ne peut être que vide. En effet, l'ensemble des variables libres de V_{b_2} doit être inclus dans l'ensemble des variables libres de T_{a_2}. On aura donc:

$$Prd(T_{a_2}) \rightarrow Prd(V_{b_2})$$

L'instanciation des variables b_2'' par elles-mêmes dans (H$_{11}$) et *Modus Ponens* donnent:

$$\exists\, b_1'.(Prd(U_{b_1}) \wedge J(b_1',b_2'')) \tag{H$_{16}$}$$

Soit $(sol_{f_1'},sol_{e_1'})$ la valeur de b_1' qui satisfait (H$_{16}$). On a donc:

$$[f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]Prd(U_{b_1}) \tag{H$_{17}$}$$

$$[f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]J(b_1',b_2'') \tag{H$_{18}$}$$

Réutiliser $(sol_{f_1'},sol_{e_1'})$, signifie que ce couple vérifie le but (G$_{6.1}$), c.à.d:

$$\begin{aligned}\exists\, d_1'.((([f_1' := sol_{f_1'}]Prd(S_{a_1}) \wedge e_1' = e_1) \vee ([f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]Prd(U_{b_1}) \wedge d_1' = d_1)) \wedge \\ [f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]J(f_1' \cup d_1' \cup e_1',f_2'' \cup d_2''))\end{aligned} \tag{G$_{6.2}$}$$

En prenant $d_1' = d_1$, on aura à prouver les deux sous buts suivants:

$$[f_1' := solf_1';\ e_1' := sol_{e_1'}]Prd(U_{b_1}) \tag{G$_{6.3}$}$$

$$[f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]J(f_1' \cup d_1 \cup e_1',f_2'' \cup d_2'') \tag{G$_{6.4}$}$$

(H$_{17}$) prouve (G$_{6.3}$). Le but (G$_{6.4}$) doit pouvoir être déduit de l'ensemble des hypothèses constituant l'environnement de preuves. On doit donc prouver:

$$\boxed{\begin{aligned}(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge [f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]Prd(U_{b_1}) \wedge \\ [f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]J(b_1',b_2'') \wedge Prd(T_{a_2})) \rightarrow [f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]J(f_1' \cup d_1 \cup e_1',f_2'' \cup d_2'')\end{aligned}}$$

2- Réutilisation de la preuve (H_3):
Réutiliser $(sol_{f_1'}, sol_{d_1'})$ signifie que ce couple vérifie le but ($G_{6.1}$), c.à.d:

$$\exists\, e_1'.(([f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]Prd(S_{a_1}) \wedge e_1' = e_1) \vee ([f_1' := sol_{f_1'}]Prd(U_{b_1}) \wedge d_1' = d_1)) \wedge$$
$$[f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]J(f_1' \cup d_1' \cup e_1', f_2'' \cup d_2'' \cup e_2)) \quad (G_{6.2})$$

En prenant $e_1' = e_1$, on aura à prouver les deux sous buts suivants:

$$[f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]Prd(S_{a_1}) \quad (G_{6.3})$$
$$[f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]J(f_1' \cup d_1' \cup e_1, f_2'' \cup d_2'' \cup e_2) \quad (G_{6.4})$$

(H_{14}) prouve ($G_{6.3}$). Le ($G_{6.4}$) doit pouvoir être déduit de l'ensemble des hypothèses constituant l'environnement de preuve. On doit donc prouver:

$$\boxed{\begin{array}{r}(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge [f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]Prd(S_{a_1}) \wedge \\ [f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]J(a_1',a_2'') \wedge Prd(T_{a_2})) \rightarrow [f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]J(f_1' \cup d_1' \cup e_1, f_2'' \cup d_2'' \cup e_2)\end{array}}$$

- <u>Si $Prd(V_{b_2}) \wedge d_2'' = d_2$</u>: idem que pour le cas précédent

Résumé
Soient S_{a_1}, T_{a_2}, U_{b_1}, V_{b_2} quatre substitutions et $J(a_1,a_2)$, $J(b_1,b_2)$ deux prédicats tels que :

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2} \text{ et } U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$$

Soit $J(a_1 \cup b_1, a_2 \cup b_2)$ un prédicat vérifiant ($c_i = a_i \cup b_i$, $d_i = a_i - b_i$, $e_i = b_i - a_i$ et $f_i = a_i \cap b_i$):
 $i.$ $(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2)) \rightarrow J(a_1,a_2)$
 $ii.$ $(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \rightarrow J(b_1,b_2)$
alors:

1. Si $(Prd(T_{a_2}) \wedge e_2'' = e_2)$ est vrai alors la solution $(sol_{f_1'}, sol_{d_1'})$ de $S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$ vérifiant:
$$(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge [f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]Prd(S_{a_1}) \wedge$$
$$[f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]J(a_1',a_2'') \wedge Prd(T_{a_2})) \rightarrow$$
$$[f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]J(f_1' \cup d_1' \cup e_1, f_2'' \cup d_2'' \cup e_2)$$
permet de prouver le raffinement: $(S_{a_1} [] U_{b_1}) \sqsubseteq_{J(c_1,c_2)} (T_{a_2} [] V_{b_2})$
2. Si $Prd(T_{a_2})$, $Prd(V_{b_2})$ et $e_2 = \emptyset$ sont vrais alors la solution $(sol_{f_1'}, sol_{e_1'})$ de $U_{b_1} \sqsubseteq_{J(b_1,b_2)} T_{b_2}$ vérifiant:
$$(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge [f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]Prd(U_{b_1}) \wedge$$
$$[f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]J(b_1',b_2'') \wedge Prd(T_{a_2})) \rightarrow [f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]J(f_1' \cup d_1 \cup e_1', f_2'' \cup d_2'')$$
permet de prouver le raffinement: $(S_{a_1} [] U_{b_1}) \sqsubseteq_{J(c_1,c_2)} (T_{a_2} [] V_{b_2})$
3. Si $(Prd(V_{b_2}) \wedge d_2'' = d_2)$ est vrai alors la solution $(sol_{f_1'}, sol_{e_1'})$ de $U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$ vérifiant:
$$(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge [f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]Prd(U_{b_1}) \wedge$$
$$[f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]J(b_1',b_2'') \wedge Prd(V_{b_2})) \rightarrow$$
$$[f_1' := sol_{f_1'};\ e_1' := sol_{e_1'}]J(f_1' \cup d_1 \cup e_1', f_2'' \cup d_2 \cup e_2'')$$
permet de prouver le raffinement: $(S_{a_1} [] U_{b_1}) \sqsubseteq_{J(c_1,c_2)} (T_{a_2} [] V_{b_2})$
4. Si $Prd(T_{a_2})$, $Prd(V_{b_2})$ et $d_2'' = \emptyset$ sont vrais alors la solution $(sol_{f_1'}, sol_{d_1'})$ de $S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$ vérifiant:
$$(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1,c_2) \wedge [f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]Prd(S_{a_1}) \wedge$$
$$[f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]J(a_1',a_2'') \wedge Prd(T_{a_2}) \wedge Prd(V_{b_2})) \rightarrow$$
$$[f_1' := sol_{f_1'};\ d_1' := sol_{d_1'}]J(f_1' \cup d_1' \cup e_1, f_2'' \cup e_2'')$$

permet de prouver le raffinement: $(S_{a_1} [] U_{b_1}) \sqsubseteq_{J(c_1,c_2)} (T_{a_2} [] V_{b_2})$

Remarques

1. La condition (i) (resp. (ii)) sont des conditions nécessaires de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $S_{a_1} \sqsubseteq T_{a_2}$ (resp. $U_{b_1} \sqsubseteq V_{b_2}$), alors il faut être capable d'exhiber dans l'environnement de preuve de $(S_{a_1} [] U_{b_1} \sqsubseteq T_{a_2} [] V_{b_2})$ l'invariant de collage de $S_{a_1} \sqsubseteq T_{a_2}$ (resp. $U_{b_1} \sqsubseteq V_{b_2}$). Dans le cas contraire la stratégie de réutilisation n'est pas justifiée.
2. Les conditions spécifiées dans les points (1) à (4) sont des conditions suffisantes de réutilisation et non nécessaires à la correction du raffinement. Une autre solution sans réutilisation peut éventuellement exister.
3. Si les substitutions S_{a_1} et U_{b_1} sont déterministes alors les conditions spécifiées dans les points (1) à (4) sont vérifiées. En effet, il existe une seule valeur de a'_1 (resp. b'_1) qui satisfait le raffinement $(S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2})$ (resp. $(U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2})$, celle-ci est déterminée par $Prd(S_{a_1})$ (resp. $Prd(U_{b_1})$). De plus, une solution possible du raffinement $(S_{a_1} [] U_{b_1} \sqsubseteq_{J(c_1,c_2)} T_{a_2} [] V_{b_2})$ doit satisfaire le $Prd(S_{a_1})$ (resp. $Prd(U_{b_1})$) qui lui même n'admet qu'une seule solution: les deux raffinements admettent donc le même espace de solutions.
4. Si les prédicats $J(a_1,a_2)$ et $J(b_1,b_2)$ sont déterministes (fonctionnels) en a_1 et b_1 respectivement alors les conditions spécifiées dans les points 1 à 4 sont vérifiées. En effet, une solution possible du raffinement $(S_{a_1} [] U_{b_1} \sqsubseteq_{J(c_1,c_2)} T_{a_2} [] V_{b_2})$ doit satisfaire le prédicat $J(c_1,c_2)$. D'après (i) et (ii) cette solution satisfait les prédicats $J(a_1,a_2)$ et $J(b_1,b_2)$. Or ces derniers n'admettent qu'une solution unique: cette solution unique est donc solution de $(S_{a_1} [] U_{b_1} \sqsubseteq_{J(c_1,c_2)} T_{a_2} [] V_{b_2})$.

B.3 Preuve par réutilisation du raffinement de la substitution conditionnée

Soient $S_{a_1}, T_{a_2}, U_{b_1}$ et V_{b_2} quatre substitutions et $J(a_1,a_2), J(b_1,b_2)$ deux prédicats tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2} \tag{H$_1$}$$

$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2} \tag{H$_2$}$$

Soient P_{c_1}, Q_{c_2} et $J(a_1 \cup b_1 \cup c_1, a_2 \cup b_2 \cup c_2)$ trois prédicats tels que:

$$(\mathbf{IF}\ P_{c_1}\ \mathbf{THEN}\ S_{a_1}\ \mathbf{ELSE}\ U_{b_1}\ \mathbf{END}) \sqsubseteq_{J(d_1,d_2)} (\mathbf{IF}\ Q_{c_2}\ \mathbf{THEN}\ T_{a_2}\ \mathbf{ELSE}\ V_{b_2}\mathbf{END}) \tag{G$_1$}$$

où: $d_1 = a_1 \cup b_1 \cup c_1$ et $d_2 = a_2 \cup b_2 \cup c_2$.

On voudrait réutiliser les preuves des raffinement (H$_1$) et (H$_2$) lors de la preuve du raffinement (G$_1$). Cela revient à définir les conditions nécessaires et suffisantes pour que cette réutilisation soit possible.

Le problème qu'on se pose est donc le suivant:

Hypothèses: par définition du raffinement et de la condition nécessaire à la correction du raffinement d'une substitution **IF**, on a les hypothèses suivantes:

$$(\mathrm{H}_1) \equiv (Trm(S_{a_1}) \wedge J(a_1,a_2)) \rightarrow [Trm(T_{a_2}) \wedge \forall a''_2.(Prd(T_{a_2}) \rightarrow \exists a'_1.(Prd(S_{a_1}) \wedge J(a'_1,a''_2)))] \tag{H$_3$}$$

$$(\mathrm{H}_2) \equiv (Trm(U_{b_1}) \wedge J(b_1,b_2)) \rightarrow [Trm(V_{b_2}) \wedge \forall b''_2.(Prd(V_{b_2}) \rightarrow \exists b'_1.(Prd(U_{b_1}) \wedge J(b'_1,b''_2)))] \tag{H$_4$}$$

$$(\mathrm{G}_1) \rightarrow (J(d_1,d_2) \rightarrow (P_{c_1} \leftrightarrow Q_{c_2})) \tag{H$_5$}$$

But: établir une preuve, par réutilisation de (H_1) et (H_2), de:

$$(Trm(Sub_{abs}) \wedge J(d_1,d_2)) \rightarrow [Trm(Sub_{con}) \wedge \forall\, d_2''.(Prd(Sub_{con}) \rightarrow \exists\, d_1'.(Prd(Sub_{abs}) \wedge J(d_1',d_2'')))] \tag{G$_2$}$$

avec:
Sub_{abs}=**IF** P_{c_1} **THEN** S_{a_1} **ELSE** U_{b_1} **END**
Sub_{con}=**IF** Q_{c_2} **THEN** T_{a_2} **ELSE** V_{b_2} **END**
On pose:

$$e_1 = (a_1 \cup c_1) - b_1 \qquad e_2 = (a_2 \cup c_2) - b_2$$
$$f_1 = (b_1 \cup c_1) - a_1 \qquad f_2 = (b_2 \cup c_2) - a_2$$

On a (par définition):

$$Trm(Sub_{abs}) = (P_{c_1} \rightarrow Trm(S_{a_1})) \wedge (\neg P_{c_1} \rightarrow Trm(U_{b_1}))$$
$$Trm(Sub_{con}) = (Q_{c_2} \rightarrow Trm(T_{a_2})) \wedge (\neg Q_{c_2} \rightarrow Trm(V_{b_2}))$$

et

$$Prd(Sub_{abs}) = (P_{c_1} \wedge Prd(S_{a_1}) \wedge f_1' = f_1) \vee (\neg P_{c_1} \wedge Prd(U_{b_1}) \wedge e_1' = e_1)$$
$$Prd(Sub_{con}) = (Q_{c_2} \wedge Prd(T_{a_2}) \wedge f_2' = f_2) \vee (\neg Q_{c_2} \wedge Prd(V_{b_2}) \wedge e_2' = e_2)$$

En rapportant ces définitions dans (G_2), puis par l'application des règles de déduction et de l'élimination du quantificateur universel, on obtient les sous buts suivants:

$$Q_{c_2} \rightarrow Trm(T_{a_2}) \tag{G$_3$}$$
$$\neg Q_{c_2} \rightarrow Trm(V_{b_2}) \tag{G$_4$}$$
$$Prd(Sub_{con}) \rightarrow \exists\, d_1'.(Prd(Sub_{abs}) \wedge J(d_1',d_2'')) \tag{G$_5$}$$

sous les hypothèses supplémentaires:

$$P_{c_1} \rightarrow Trm(S_{a_1}) \tag{H$_6$}$$
$$\neg P_{c_1} \rightarrow Trm(U_{b_1}) \tag{H$_7$}$$
$$J(d_1,d_2) \tag{H$_8$}$$

Preuve de G_3: en appliquant la règle de déduction sur (G_3), le but à prouver devient:

$$Trm(T_{a_2}) \tag{G$_{3.1}$}$$

sous l'hypothèse supplémentaire:

$$Q_{c_2} \tag{H$_9$}$$

(H_9), (H_8) et (H_5) donnent:

$$P_{c_1} \tag{H$_{10}$}$$

Pour prouver ($G_{3.1}$) par réutilisation de (H_3), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(S_{a_1})$ et $J(a_1,a_2)$. (H_{10}) et (H_6) donnent:

$$Trm(S_{a_1}) \tag{H$_{11}$}$$

Il reste à générer de notre environnement d'hypothèses, constitué de (H_8), (H_{10})et (H_{11}), l'hypothèse:

$$J(a_1,a_2) \tag{H$_{12}$}$$

Une condition nécessaire et suffisante à la réutilisation est:

$$\boxed{(J(d_1,d_2) \wedge P_{c_1} \wedge Trm(S_{a_1})) \rightarrow J(a_1,a_2)}$$

(H_{12}), (H_{11}) et (H_3) prouvent alors ($G_{3.1}$) et génèrent une hypothèse supplémentaire :

$$\forall\, a_2''.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2''))) \tag{H$_{13}$}$$

Preuve de G_4: en appliquant la règle de déduction sur (G_4), le but à prouver devient:

$$Trm(V_{b_2}) \tag{G$_{4.1}$}$$

sous l'hypothèse supplémentaire:

$$\neg Q_{c_2} \tag{H$_{14}$}$$

(H_{14}), (H_8) et (H_5) donnent:

$$\neg P_{c_1} \tag{H$_{15}$}$$

Pour prouver ($G_{4.1}$) par réutilisation de (H_4), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(U_{b_1})$ et $J(b_1,b_2)$. (H_{15}) et (H_7) donnent:

$$Trm(U_{b_1}) \tag{H$_{16}$}$$

Il reste à générer de notre environnement d'hypothèses, constitué de (H_8), (H_{15}) et (H_{16}), l'hypothèse:

$$J(b_1,b_2) \tag{H$_{17}$}$$

Une deuxième condition nécessaire et suffisante à la réutilisation est:

$$\boxed{(J(d_1,d_2) \wedge \neg P_{c_1} \wedge Trm(U_{b_1})) \rightarrow J(b_1,b_2)}$$

(H_{17}), (H_{16}) et (H_4) prouvent alors ($G_{4.1}$) et génèrent l'hypothèse:

$$\forall\, b_2''.(Prd(V_{b_2}) \rightarrow \exists\, b_1'.(Prd(U_{b_1}) \wedge J(b_1',b_2''))) \tag{H$_{18}$}$$

Preuve de (G_5): par l'application de la règle de déduction, le but à prouver devient:

$$\exists\, d_1'.(((P_{c_1} \wedge Prd(S_{a_1}) \wedge f_1' = f_1) \vee (\neg P_{c_1} \wedge Prd(U_{b_1}) \wedge e_1' = e_1)) \wedge J(d_1',d_2'')) \tag{G$_{5.1}$}$$

sous l'hypothèse supplémentaire:

$$(Q_{c_2} \wedge Prd(T_{a_2}) \wedge f_2'' = f_2) \vee (\neg Q_{c_2} \wedge Prd(V_{b_2}) \wedge e_2'' = e_2) \tag{H$_{19}$}$$

Compte tenu de cette dernière hypothèse, nous procédons par cas:

- $Q_{c_2} \wedge Prd(T_{a_2}) \wedge f_2'' = f_2$
  l'hypothèse (H_9) étant vraie, alors on récupère toutes les hypothèses générées lors de la preuve de (G_3). $Prd(T_{a_2})$ étant vrai, l'instanciation des variables a_2'' par elles-mêmes dans (H_{13}) donne:

$$\exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2'')) \tag{H$_{20}$}$$

Soit $sol_{a'_1}$ une des valeurs de a'_1 qui vérifie (H_{20}). On a donc:

$$[a'_1 := sol_{a'_1}]Prd(S_{a_1}) \quad (H_{21})$$

$$[a'_1 := sol_{a'_1}]J(a'_1,a''_2) \quad (H_{22})$$

Le but ($G_{5.1}$) à prouver est équivalent à:

$$\exists\, a'_1,f'_1.(P_{c_1} \wedge Prd(S_{a_1}) \wedge f'_1 = f_1 \wedge J(a'_1 \cup f'_1,d''_2)) \quad (G_{5.2})$$

Réutiliser la preuve du raffinement $(S_{a_1} \sqsubseteq T_{a_2})$ revient à vérifier que la solution $sol_{a'_1}$ est aussi une solution du raffinement à prouver. Pour cela, on doit vérifier que $sol_{a'_1}$ satisfait le but ($G_{5.2}$), c.à.d:

$$\exists f'_1.(P_{c_1} \wedge [a'_1 := sol_{a'_1}]Prd(S_{a_1}) \wedge f'_1 = f_1 \wedge [a'_1 := sol_{a'_1}]J(a'_1 \cup f'_1,d''_2) \quad (G_{5.3})$$

En prenant f'_1 égale à f_1, ce but se décompose en trois sous buts:

$$P_{c_1} \quad (G_{5.4})$$

$$[a'_1 := sol_{a'_1}]Prd(S_{a_1}) \quad (G_{5.5})$$

$$[a'_1 := sol_{a'_1}]J(a'_1 \cup f_1,a''_2 \cup f_2) \quad (G_{5.6})$$

($G_{5.4}$) et ($G_{5.5}$) sont prouvés par (H_{10}) et (H_{21}) respectivement. Le but ($G_{5.6}$) doit pouvoir être déduit de l'ensemble des hypothèses constituant notre environnement de preuve. On doit donc prouver:

$$\boxed{\begin{array}{r}(Q_{c_2} \wedge J(d_1,d_2) \wedge Trm(S_{a_1}) \wedge Prd(T_{a_2}) \wedge [a'_1 := sol_{a'_1}]Prd(S_{a_1})\wedge \\ [a'_1 := sol_{a'_1}]J(a'_1,a''_2)) \rightarrow [a'_1 := sol_{a'_1}]J(a'_1 \cup f_1,a''_2 \cup f_2)\end{array}}$$

– $\neg Q_{c_2} \wedge Prd(V_{b_2}) \wedge e''_2 = e_2$
l'hypothèse (H_{14}) étant vraie, alors on récupère toutes les hypothèses générées lors de la preuve de (G_4). $Prd(V_{b_2})$ étant vrai, l'instanciation des variables b_2'' par elles-mêmes dans (H_{18}) donne:

$$\exists\, b'_1.(Prd(U_{b_1}) \wedge J(b'_1,b''_2)) \quad (H_{23})$$

Soit $sol_{b'_1}$ la valeur de b'_1 qui vérifie (H_{23}). On a donc:

$$[b'_1 := sol_{b'_1}]Prd(U_{b_1}) \quad (H_{24})$$

$$[b'_1 := sol_{b'_1}]J(b'_1,b''_2) \quad (H_{25})$$

Le but ($G_{5.1}$) à prouver est équivalent à:

$$\exists\, b'_1,e'_1.(\neg P_{c_1} \wedge Prd(U_{b_1}) \wedge e'_1 = e_1 \wedge J(d'_1,d''_2)) \quad (G_{5.2})$$

Réutiliser la preuve du raffinement $(U_{b_1} \sqsubseteq V_{b_2})$ revient à vérifier que la solution $sol_{b'_1}$ est aussi une solution du raffinement à prouver. Pour cela, on doit vérifier que $sol_{b'_1}$ satisfait le but ($G_{5.2}$), c.à.d:

$$\exists\, e'_1.(\neg P_{c_1} \wedge [b'_1 := sol_{b'_1}]Prd(U_{b_1}) \wedge e'_1 = e_1 \wedge [b'_1 := sol_{b'_1}]J(b'_1 \cup e'_1,d''_2)) \quad (G_{5.3})$$

En prenant e'_1 égale à e_1, ce but se décompose en trois sous buts:

$$\neg P_{c_1} \quad (G_{5.4})$$

$$[b'_1 := sol_{b'_1}]Prd(U_{b_1}) \quad (G_{5.5})$$

$$[b'_1 := sol_{b'_1}]J(b'_1 \cup e_1,b''_2 \cup e_2) \quad (G_{5.6})$$

($G_{5.4}$) et ($G_{5.5}$) sont prouvés par (H_{15}) et (H_{24}) respectivement. Le but ($G_{5.6}$) doit pouvoir être déduit de l'ensemble des hypothèses constituant notre environnement de preuve. On doit donc prouver:

$$\boxed{\begin{array}{r}(\neg Q_{c_2} \wedge Trm(U_{b_1}) \wedge Prd(V_{b_2}) \wedge J(d_1,d_2) \wedge [b_1' := sol_{b_1'}]Prd(U_{b_1}) \wedge \\ [b_1' := sol_{b_1'}]J(b_1',b_2'')) \rightarrow [b_1' := sol_{b_1'}]J(b_1' \cup e_1, b_2'' \cup e_2)\end{array}}$$

Résumé

Soient S_{a_1}, T_{a_2}, U_{b_1}, V_{b_2} quatre substitutions et $J(a_1,a_2)$, $J(b_1,b_2)$ deux prédicats tels que :

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$$
$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$$

Soient P_{c_1}, Q_{c_2} et $J(a_1 \cup b_1 \cup c_1, a_2 \cup b_2 \cup c_2)$ un prédicat vérifiant ($f_i = (b_i \cup c_i) - a_i$, $e_i = (a_i \cup c_i) - b_i$ et $d_i = a_i \cup b_i \cup c_i$):

$i.$ $J(d_1,d_2) \rightarrow (P_{c_1} \leftrightarrow Q_{c_2})$

$ii.$ $(P_{c_1} \wedge Trm(S_{a_1}) \wedge J(d_1,d_2) \rightarrow J(a_1,a_2)$

$iii.$ $(\neg P_{c_1} \wedge Trm(U_{b_1}) \wedge J(d_1,d_2) \rightarrow J(b_1,b_2)$

$iv.$ si une valeur a_1' vérifiant:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$$

vérifie:

$$(P_{c_1} \wedge J(d_1,d_2) \wedge Trm(S_{a_1}) \wedge Prd(T_{a_2}) \wedge Prd(S_{a_1}) \wedge J(a_1',a_2'')) \rightarrow J(a_1' \cup f_1, a_2'' \cup f_2)$$

alors la valeur a_1' peut être réutilisée pour prouver: (**IF** P_{c_1} **THEN** S_{a_1} **ELSE** U_{b_1}) $\sqsubseteq_{J(d_1,d_2)}$ (**IF** Q_{c_2} **THEN** T_{a_2} **ELSE** V_{b_2}) si P_{c_1} est vrai.

$v.$ si une valeur b_1' vérifiant:

$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$$

vérifie:

$$(\neg Q_{c_2} \wedge Trm(V_{b_2}) \wedge Prd(V_{b_2}) \wedge J(d_1,d_2) \wedge Prd(U_{b_1}) \wedge J(b_1',b_2'')) \rightarrow J(b_1' \cup e_1, b_2'' \cup e_2)$$

alors la valeur b_1' peut être réutilisée pour prouver: (**IF** P_{c_1} **THEN** S_{a_1} **ELSE** U_{b_1}) $\sqsubseteq_{J(d_1,d_2)}$ (**IF** Q_{c_2} **THEN** T_{a_2} **ELSE** V_{b_2}) si P_{c_1} est faux.

Remarques

1. La condition (i) est une condition nécessaire pour la correction du raffinement conditionné.
2. La condition (ii) (resp. (iii)) est une condition nécessaire de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $S_{a_1} \sqsubseteq T_{a_2}$ (resp. $U_{b_1} \sqsubseteq V_{b_2}$), alors il faut être capable d'exhiber dans l'environnement de preuve de la substitution conditionnée l'invariant de collage de $S_{a_1} \sqsubseteq T_{a_2}$ (resp. $U_{b_1} \sqsubseteq V_{b_2}$). Dans le cas contraire la stratégie de réutilisation n'est pas justifiée.
3. Les deux dernières conditions sont nécessaires que pour la réutilisation. Une autre solution du raffinement sans réutilisation peut éventuellement exister.
4. La réutilisation de la preuve de raffinement de $S_{a_1} \sqsubseteq T_{a_2}$ (resp. $U_{b_1} \sqsubseteq V_{b_2}$) est réalisable pour toute valeur de a_1' (resp. b_1') qui vérifie la condition (iv) (resp. (v)): on n'exige pas que toutes les valeurs a_1' (resp. b_1') vérifient cette condition.
5. Si la substitution S_{a_1} (resp. U_{b_1}) est déterministe alors la condition (iv) (resp. (v)) est vérifiée. En effet, il existe une seule valeur de a_1' qui satisfait le raffinement ($S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$) (resp. ($U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$)), celle-ci est déterminée par $Prd(S_{a_1})$ (resp. $Prd(U_{b_1})$). De plus si P_{c_1} est vrai (resp. faux), une solution possible du raffinement conditionné doit satisfaire $Prd(S_{a_1})$ (resp. $Prd(U_{b_1})$) qui lui même n'admet qu'une seule solution: les deux raffinements admettent donc le même espace de solutions.

6. Si le prédicat $J(a_1,a_2)$ (resp. $J(b_1,b_2)$) est déterministe (fonctionnel) en a_1 (resp. b_1) alors la condition (iv) (resp. (v)) est vérifiée. En effet, une solution possible du raffinement conditionné doit satisfaire le prédicat $J(d_1,d_2)$. D'après (ii) (resp.(iii)) cette solution satisfait le prédicat $J(a_1,a_2)$ (resp. $J(b_1,b_2)$). Or $J(a_1,a_2)$ (resp. $J(b_1,b_2)$) n'admet qu'une solution unique a_1': cette solution unique est donc réutilisable.

B.4 Preuve par réutilisation du raffinement de la substitution séquentielle

Soient $S_{a_1}, T_{a_2}, U_{b_1}$ et V_{b_2} quatre substitutions et $J(a_1,a_2), J(b_1,b_2)$ deux prédicats tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2} \quad (\mathrm{H}_1)$$
$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2} \quad (\mathrm{H}_2)$$

Soit $J(a_1 \cup b_1, a_2 \cup b_2)$ un prédicat tel que:

$$(S_{a_1};\ U_{b_1}) \sqsubseteq_{J(c_1,c_2)} (T_{a_2};\ V_{b_2}) \quad (\mathrm{G}_1)$$

où: $c_i = a_i \cup b_i$.

On voudrait réutiliser les preuves des raffinements (H_1) et (H_2) lors de la preuve du raffinement (G_1). Cela revient à définir les conditions nécessaires et suffisantes pour que cette réutilisation soit possible.

Le problème qu'on se pose est donc le suivant:

Hypothèses: par définition du raffinement, on a les hypothèses suivantes:

$$(\mathrm{H}_1) \equiv (Trm(S_{a_1}) \wedge J(a_1,a_2)) \rightarrow [Trm(T_{a_2}) \wedge \forall\, a_2'.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2')))] \quad (\mathrm{H}_3)$$
$$(\mathrm{H}_2) \equiv (Trm(U_{b_1}) \wedge J(b_1,b_2)) \rightarrow [Trm(V_{b_2}) \wedge \forall\, b_2''.(Prd(V_{b_2}) \rightarrow \exists\, b_1''.(Prd(U_{b_1}) \wedge J(b_1'',b_2'')))] \quad (\mathrm{H}_4)$$

But: établir une preuve, par réutilisation de (H_1) et (H_2), de:

$$\begin{aligned}(Trm(S_{a_1};\ U_{b_1}) \wedge J(c_1,c_2)) \rightarrow [Trm(T_{a_2};\ V_{b_2}) \wedge \forall\, c_2''.(Prd(T_{a_2};\ V_{b_2}) \rightarrow \\ \exists\, c_1''.(Prd(S_{a_1};\ U_{b_1}) \wedge J(c_1'',c_2'')))]\end{aligned} \quad (\mathrm{G}_2)$$

On a (par définition):

$$Trm(S_{a_1};\ U_{b_1}) = Trm(S_{a_1}) \wedge \forall\, x.([a_1' := x]Prd_{a_1,a_1'}(S_{a_1}) \rightarrow [a_1 := x]\,Trm(U_{b_1}))$$
$$Trm(T_{a_2};\ V_{b_2}) = Trm(T_{a_2}) \wedge \forall\, y.([a_2' := y]Prd_{a_2,a_2'}(T_{a_2}) \rightarrow [a_2 := y]\,Trm(V_{b_2}))$$

et

$$Prd(S_{a_1};\ U_{b_1}) = Trm(S_{a_1}) \rightarrow \exists\, z.([a_1' := z]Prd_{a_1,a_1'}(S_{a_1}) \wedge [a_1 := z]Prd_{c_1,c_1''}(U_{b_1}))$$
$$Prd(T_{a_2};\ V_{b_2}) = Trm(T_{a_2}) \rightarrow \exists\, t.([a_2' := t]Prd_{a_2,a_2'}(T_{a_2}) \wedge [a_2 := t]Prd_{c_2,c_2''}(V_{b_2}))$$

En rapportant ces définitions dans (G_2), puis par application de la règle de déduction, on obtient les sous buts suivants:

$$Trm(T_{a_2}) \quad (\mathrm{G}_3)$$
$$\forall\, y.([a_2' := y]Prd_{a_2,a_2'}(T_{a_2}) \rightarrow [a_2 := y]\,Trm(V_{b_2})) \quad (\mathrm{G}_4)$$
$$\forall\, c_2''.(Prd(T_{a_2};\ V_{b_2}) \rightarrow \exists\, c_1''.(Prd(S_{a_1};\ U_{b_1}) \wedge J(c_1'',c_2''))) \quad (\mathrm{G}_5)$$

sous les hypothèses supplémentaires:

$$Trm(S_{a_1}) \tag{H$_5$}$$
$$\forall\, x.([a'_1 := x]Prd_{a_1,a'_1}(S_{a_1}) \rightarrow [a_1 := x]\,Trm(U_{b_1})) \tag{H$_6$}$$
$$J(c_1,c_2) \tag{H$_7$}$$

Preuve de G$_3$: pour prouver (G$_3$) par réutilisation de (H$_3$), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(S_{a_1})$ et $J(a_1,a_2)$. $Trm(S_{a_1})$ étant l'hypothèse (H$_5$), il reste à générer de l'ensemble d'hypothèses (H$_5$) et (H$_7$), l'hypothèse:

$$J(a_1,a_2) \tag{H$_8$}$$

Une condition nécessaire et suffisante à la réutilisation est:

$$\boxed{(Trm(S_{a_1}) \wedge J(c_1,c_2)) \rightarrow J(a_1,a_2)}$$

(H$_5$), (H$_8$) et (H$_3$) prouvent alors (G$_3$) et génèrent une hypothèse supplémentaire:

$$\forall\, a'_2.(Prd(T_{a_2}) \rightarrow \exists\, a'_1.(Prd_{a_1,a'_1}(S_{a_1}) \wedge J(a'_1,a'_2))) \tag{H$_9$}$$

Preuve de G$_4$: par l'élimination du quantificateur universel et l'application de la règle de déduction, (G$_4$) est équivalent à:

$$[a_2 := y]\,Trm(V_{b_2}) \tag{G$_{4.1}$}$$

sous l'hypothèse supplémentaire:

$$[a'_2 := y]Prd_{a_2,a'_2}(T_{a_2}) \tag{H$_{10}$}$$

En instanciant a'_2 par y dans l'hypothèse (H$_9$), puis en utilisant (H$_{10}$) on obtient:

$$\exists\, a'_1.(Prd_{a_1,a'_1}(S_{a_1}) \wedge [a'_2 := y]J(a'_1,a'_2)) \tag{H$_{11}$}$$

Soit $sol_{a'_1}$ une des valeurs de a'_1 qui vérifie cette dernière hypothèse, on a donc:

$$[a'_1 := sol_{a'_1}]Prd_{a_1,a'_1}(S_{a_1}) \tag{H$_{12}$}$$
$$[a'_1 := sol_{a'_1};\; a'_2 := y]J(a'_1,a'_2) \tag{H$_{13}$}$$

Remarque: dans cette dernière hypothèse les variables communes de a_1 et a_2 ont nécessairement les mêmes valeurs: $sol_{a'_1} = y$. En effet, une variable commune à a_1 et a_2 est une variable de S_{a_1} raffinée par elle-même. Or, une telle variable ne doit pas changer de valeur, car sinon le raffinement ne serait pas correct.

(H$_{12}$) et l'instanciation des variables $a_1{}'$ par $sol_{a'_1}$ dans (H$_6$) donne:

$$[a_1 := sol_{a'_1}]\,Trm(U_{b_1}) \tag{H$_{14}$}$$

Pour prouver (G$_{4.1}$) par réutilisation de (H$_4$), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(U_{b_1})$ et $[a_2 := y]J(b_1,b_2)$ pour n'importe quelle valeur de a_1: $sol_{a'_1}$ dans notre cas. Ayant $[a_1 = sol_{a'_1}]\,Trm(U_{b_1})$ comme hypothèse (cf. H$_{14}$), il reste à générer de notre environnement de preuve, constitué de (H$_5$), (H$_7$), (H$_{10}$), (H$_{12}$), (H$_{13}$) et (H$_{14}$) l'hypothèse:

$$[a_1 = sol_{a'_1};\; a_2 = y]J(b_1,b_2) \tag{H$_{15}$}$$

Une deuxième condition nécessaire et suffisante à la réutilisation est:

$$\boxed{\begin{array}{r}(Trm(S_{a_1}) \wedge J(c_1,c_2) \wedge [a_2' := y]Prd_{a_2,a_2'}(T_{a_2}) \wedge [a_1' := sol_{a_1'}]Prd_{a_1,a_1'}(S_{a_1}) \wedge \\ [a_1' := sol_{a_1'};\ a_2' := y]J(a_1',a_2') \wedge [a_1 := sol_{a_1'}]Trm(U_{b_1})) \rightarrow [a_1 = sol_{a_1'};\ a_2 = y]J(b_1,b_2)\end{array}}$$

(H_{14}), (H_{15}) et (H_4) prouvent alors ($G_{4.1}$) et génère l'hypothèse supplémentaire:

$$\forall\, b_2''.([a_2 := y]Prd(V_{b_2}) \rightarrow \exists\, b_1''.([a_1 := sol_{a_1'}]Prd_{b_1,b_1''}(U_{b_1}) \wedge J(b_1'',b_2''))) \qquad (H_{16})$$

Preuve de (G_5): par élimination du quantificateur universel et l'application de la règle de déduction, le but à prouver devient:

$$\exists\, c_1''.(Trm(S_{a_1}) \rightarrow \exists\, z.([a_1' := z]Prd_{a_1,a_1'}(S_{a_1}) \wedge [a_1 := z]Prd_{c_1,c_1''}(U_{b_1})) \wedge J(c_1'',c_2'')) \qquad (G_{5.1})$$

sous l'hypothèse supplémentaire:

$$Trm(T_{a_2}) \rightarrow \exists\, t.([a_2' := t]Prd_{a_2,a_2'}(T_{a_2}) \wedge [a_2 := t]Prd_{c_2,c_2''}(V_{b_2})) \qquad (H_{17})$$

(G_3) et (H_{17}) donnent:

$$\exists\, t.([a_2' := t]Prd_{a_2,a_2'}(T_{a_2}) \wedge [a_2 := t]Prd_{c_2,c_2''}(V_{b_2})) \qquad (H_{18})$$

Soit sol_t la valeur de t qui satisfait cette dernière hypothèse. On a donc:

$$[a_2' := sol_t]Prd_{a_2,a_2'}(T_{a_2}) \qquad (H_{19})$$
$$[a_2 := sol_t]Prd_{c_2,c_2''}(V_{b_2}) \qquad (H_{20})$$

(H_{19}) et l'instanciation des variables a_2' par sol_t dans (H_9) donnent:

$$\exists\, a_1'.(Prd_{a_1,a_1'}(S_{a_1}) \wedge [a_2' := sol_t]J(a_1',a_2')) \qquad (H_{21})$$

Soit $sol_{a_1'}$ une des valeurs de a_1' qui vérifie l'hypothèse (H_{21}), on a donc:

$$[a_1' := sol_{a_1'}]Prd_{a_1,a_1'}(S_{a_1}) \qquad (H_{22})$$
$$[a_1' := sol_{a_1'};\ a_2' := sol_t]J(a_1',a_2') \qquad (H_{23})$$

Comme $c_2 = a_2 \cup b_2$, en posant ($d_2 = a_2 - b_2$), l'hypothèse (H_{20}) se réécrit en:

$$[a_2 := sol_t]Prd_{b_2,b_2''}(V_{b_2}) \qquad (H_{24})$$
$$d_2'' = d_2' \qquad (H_{25})$$

La preuve de (G_4) est valable pour tout y, en particulier pour $y = sol_t$. (H_{24}) et l'instanciation des variables b_2'' par elles-mêmes dans (H_{16}) donnent:

$$\exists\, b_1''.([a_1 := sol_{a_1'}]Prd_{b_1,b_1''}(U_{b_1}) \wedge J(b_1'',b_2'')) \qquad (H_{26})$$

Soit $sol_{b_1''}$ la valeur de b_1'' qui vérifie cette dernière hypothèse, on a donc:

$$[b_1'' := sol_{b_1''};\ a_1 = sol_{a_1'}]Prd_{b_1,b_1''}(U_{b_1}) \qquad (H_{27})$$
$$[b_1'' := sol_{b_1''}]J(b_1'',b_2'') \qquad (H_{28})$$

En posant ($d_1 = a_1 - b_1$), réutiliser la preuve des deux raffinements ($S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$) et ($U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$) revient à vérifier que $(sol_{d_1{}'},sol_{b_1''})$ est une solution du raffinement de la séquence. En réécrivant $Prd_{c_1,c_1''}(U_{b_1})$ en ($Prd_{b_1,b_1''}(U_{b_1}) \wedge d_1'' = d_1'$), on doit vérifier que $(sol_{d_1{}'},sol_{b_1''})$ satisfait le but ($G_{5.1}$), c.à.d:

$$\begin{aligned} &Trm(S_{a_1}) \rightarrow \exists\, z.([a_1' = z]Prd_{a_1,a_1'}(S_{a_1}) \wedge [b_1'' := sol_{b_1''};\ a_1 = z]Prd_{b_1,b_1''}(U_{b_1}) \wedge \\ &\quad [d_1'' := sol_{d_1{}'}](d_1'' = d_1')) \wedge [d_1'' := sol_{d_1{}'};\ b_1'' := sol_{b_1''}]J(d_1'' \cup b_1'',a_2'' \cup b_2'') \end{aligned} \qquad (G_{5.2})$$

Comme $Trm(S_{a_1})$ est vrai (d'après H_5), en prenant z égal à $sol_{a_1'}$ on aura à prouver les sous buts suivant:

$$[a_1' := sol_{a_1'}]Prd_{a_1,a_1'}(S_{a_1}) \qquad (G_{5.3})$$
$$[b_1'' := sol_{b_1''};\ a_1 = sol_{a_1'}]Prd_{b_1,b_1''}(U_{b_1}) \qquad (G_{5.4})$$
$$[d_1'' := sol_{d_1{}'}](d_1'' = d_1') \qquad (G_{5.5})$$
$$[d_1'' := sol_{d_1'};\ b_1'' := sol_{b_1''}]J(d_1'' \cup b_1'',a_2'' \cup b_2'') \qquad (G_{5.6})$$

($G_{5.3}$) et ($G_{5.4}$) sont prouvés par (H_{22}) et (H_{27}) respectivement. ($G_{5.5}$) est vrai car les variables d_1 sont les variables de a_1 n'apparaissant pas dans b_1. Ces variables ne changent pas de valeurs après exécution de U_{b_1}. Il reste à prouver le but ($G_{5.6}$) qui doit pouvoir être déduit à partir de l'ensemble des hypothèses constituant l'environnement de preuve. On doit donc prouver:

$$\boxed{\begin{aligned} &(Trm(S_{a_1}) \wedge J(c_1,c_2) \wedge [a_1' := sol_{a_1'}]Prd_{a_1,a_1'}(S_{a_1}) \wedge [a_2' = sol_t]Prd_{a_2,a_2'}(T_{a_2}) \wedge \\ &\quad [a_2 = sol_t]Prd_{c_2,c_2''}(V_{b_2}) \wedge [b_1'' := sol_{b_1''};\ a_1 = sol_{a_1'}]Prd_{b_1,b_1''}(U_{b_1}) \wedge \\ &\quad [a_1' := sol_{a_1'};\ a_2' := sol_t]J(a_1',a_2') \wedge \\ &\quad [b_1'' := sol_{b_1''}]J(b_1'',b_2'')) \rightarrow [d_1'' := sol_{d_1{}'};\ b_1'' := sol_{b_1''}]J(d_1'' \cup b_1'',a_2'' \cup b_2'') \end{aligned}}$$

Résumé

Soient S_{a_1}, T_{a_2}, U_{b_1}, V_{b_2} quatre substitutions et $J(a_1,a_2)$, $J(b_1,b_2)$ deux prédicats tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$$
$$U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$$

Soit $J(a_1 \cup b_1,a_2 \cup b_2)$ un prédicat vérifiant ($c_i = a_i \cup b_i$ et $d_i = a_i - b_i$):

$i.$ $(Trm(S_{a_1}) \wedge J(c_1,c_2)) \rightarrow J(a_1,a_2)$

$ii.$ pour tout couple (a_2',a_1'), on a:

$$(Trm(S_{a_1}) \wedge J(c_1,c_2) \wedge Prd_{a_2,a_2'}(T_{a_2}) \wedge Prd_{a_1,a_1'}(S_{a_1}) \wedge [a_1 := a_1']Trm(U_{b_1}) \wedge J(a_1',a_2')) \rightarrow [a_1 := a_1';\ a_2 := a_2']J(b_1,b_2)$$

alors:
pour chaque valeur c_2'' vérifiant:

$$Prd_{c_2,c_2''}(T_{a_2};\ V_{b_2})$$

associée à la valeur t de a_2' vérifiant:

$$[a_2' = t]Prd_{a_2,a_2'}(T_{a_2}) \wedge [a_2 = t]Prd_{c_2,c_2''}(V_{b_2})$$

on a:
pour tout couple $(sol_{a_1'},sol_{b_1''})$ vérifiant:

- $sol_{a_1'}$ vérifie le raffinement $S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$ pour a_2' égale à t,
- $sol_{b_1'}$ vérifie le raffinement $U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2}$ pour a_2 égale à t et a_1 égale à $sol_{a_1'}$,

– $(Trm(S_{a_1}) \wedge J(c_1,c_2) \wedge [a_1' := sol_{a_1'}]Prd_{a_1,a_1'}(S_{a_1}) \wedge [a_2' := sol_t]Prd_{a_2,a_2'}(T_{a_2}) \wedge$ $[a_2 := sol_t]Prd_{c_2,c_2''}(V_{b_2}) \wedge [b_1'' := sol_{b_1''};\ a_1 := sol_{a_1'}]Prd_{b_1,b_1''}(U_{b_1}) \wedge [a_1' := sol_{a_1'};\ a_2' := sol_t]J(a_1',a_2') \wedge$ $[b_1'' := sol_{b_1''}]J(b_1',b_2'')) \rightarrow [d_1{}'' := sol_{d_1{}'};\ b_1'' := sol_{b_1''}]J(d_1'' \cup b_1'',a_2'' \cup b_2'')$

le couple $(sol_{d_1{}'},sol_{b_1''})$ permet de prouver: $(S_{a_1};\ U_{b_1}) \sqsubseteq_{J(c_1,c_2)} (T_{a_2};\ V_{b_2})$.

Remarques

1. La condition (i) est une condition nécessaire de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $S_{a_1} \sqsubseteq T_{a_2}$, alors il faut être capable d'exhiber dans l'environnement de preuve l'invariant de collage de $S_{a_1} \sqsubseteq T_{a_2}$. Dans le cas contraire la stratégie de réutilisation n'est pas justifiée.
2. La condition (ii) est une condition nécessaire de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $U_{b_1} \sqsubseteq V_{b_2}$, alors il faut être capable d'exhiber dans l'environnement de preuve après exécution de S_{a_1} et T_{a_2} l'invariant de collage de $U_{b_1} \sqsubseteq V_{b_2}$. Dans le cas contraire la stratégie de réutilisation n'est pas justifiée.
3. La dernière condition n'est une condition nécessaire que pour la réutilisation des solutions des raffinements de $S_{a_1} \sqsubseteq T_{a_2}$ et $U_{b_1} \sqsubseteq V_{b_2}$. Une autre solution du raffinement sans réutilisation peut éventuellement exister.
4. La réutilisation des preuves de raffinement de $S_{a_1} \sqsubseteq T_{a_2}$ et et $U_{b_1} \sqsubseteq V_{b_2}$ sont réalisables pour tout couple de valeurs (a_1',b_1'') qui vérifie la dernière condition: on n'exige pas que tous les couples vérifient cette condition.
5. Si les substitutions S_{a_1} et U_{b_1} sont déterministes alors la dernière condition est vérifiée. En effet, il existe une seule valeur du couple (a_1',b_1'') qui satisfait les raffinements $(S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2})$ et $(U_{b_1} \sqsubseteq_{J(b_1,b_2)} V_{b_2})$, celle-ci est déterminée par $(Prd(S_{a_1}),Prd(U_{b_1}))$. De plus, une solution possible du raffinement de la séquence doit satisfaire les prédicats $Prd(S_{a_1})$ et $Prd(U_{b_1})$ qui n'admettent qu'une seule solution: les deux raffinements admettent donc le même espace de solutions.
6. Si les prédicats $J(a_1,a_2)$ et $J(b_1,b_2)$ sont fonctionnels en a_1 et b_1 respectivement alors la dernière condition est vérifiée. En effet, une solution possible du raffinement en séquence doit satisfaire le prédicat $J(c_1,c_2)$. D'après (i) et (ii) cette solution satisfait les prédicat $J(a_1,a_2)$ et $J(b_1,b_2)$. Or $J(a_1,a_2)$ et $J(b_1,b_2)$ n'admettent qu'une solution unique (a_1',b_1''): cette solution unique est donc solution de $(S_{a_1};\ U_{b_1} \sqsubseteq_{J(c_1,c_2)} T_{a_2};\ V_{v_2})$.

B.5 Preuve par réutilisation du raffinement de la substitution ANY

Soient S_{a_1} et T_{a_2} deux substitutions, et $J(a_1,a_2)$ un prédicat tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2} \tag{H$_1$}$$

Soient P_{b_1}, Q_{b_2} et $J(a_1 \cup b_1,a_2 \cup b_2)$ trois prédicats tels que:

$$(@z.(P_{b_1} \Rightarrow S_{a_1})) \sqsubseteq_{J(a_1 \cup b_1,a_2 \cup b_2)} (@z.(Q_{b_2} \Rightarrow T_{a_2})) \tag{G$_1$}$$

L'ensemble des variables z est nécessairement inclus dans l'ensemble des variables b_1. De plus, cet ensemble est inclus dans l'ensemble des variables fixes de a_1 et a_2. En effet, chaque variable de cet ensemble ne peut être ni modifiée, ni raffinée.

On voudrait réutiliser la preuve du raffinement (H$_1$) lors de la preuve du raffinement (G$_1$). Cela revient à définir les conditions nécessaires et suffisantes pour que cette réutilisation soit possible.

Le problème qu'on se pose donc est le suivant (on pose $c_1 = a_1 \cup b_1, c_2 = a_2 \cup b_2$):

Hypothèses: par définition du raffinement et de la condition nécessaire à la correction du raffinement de la garde, on a les hypothèses suivantes:

$$(\mathrm{H}_1) \equiv [Trm(S_{a_1}) \wedge J(a_1,a_2)] \rightarrow [Trm(T_{a_2}) \wedge \forall\, a_2''.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2'')))] \quad (\mathrm{H}_2)$$

$$(\mathrm{G}_1) \rightarrow ((Q_{b_2} \wedge J(c_1,c_2)) \rightarrow P_{b_1}) \quad (\mathrm{H}_3)$$

But: établir une preuve, par réutilisation de (H_1), de:

$$[Trm(@z.(P_{b_1} \Rightarrow S_{a_1})) \wedge J(c_1,c_2)] \rightarrow [Trm(@z.(Q_{b_2} \Rightarrow T_{a_2})) \wedge \forall\, c_2''.(Prd(@z.(Q_{b_2} \Rightarrow T_{a_2})) \rightarrow \exists\, c_1'.(Prd(@z.(P_{b_1} \Rightarrow S_{a_1})) \wedge J(c_1',c_2'')))] \quad (\mathrm{G}_2)$$

Soient d_1 (resp. d_2) l'ensemble des variables libres de P_{b_1} (resp. Q_{b_2}) qui n'apparaissent pas dans S_{a_1} (resp. T_{a_2}):

$$d_1 = b_1 - a_1$$
$$d_2 = b_2 - a_2$$

on a (par définition):

$$Trm(@z.(P_{b_1} \Rightarrow S_{a_1})) = \forall\, z.(P_{b_1} \rightarrow Trm(S_{a_1}))$$
$$Prd(@z.(P_{b_1} \Rightarrow S_{a_1})) = \exists\, z.(P_{b_1} \wedge Prd(S_{a_1}) \wedge d_1' = d_1)$$

$$Trm(@z.(Q_{b_2} \Rightarrow T_{a_2})) = \forall\, z.(Q_{b_2} \rightarrow Trm(T_{a_2}))$$
$$Prd(Q_{b_2} \mid T_{a_2}) = \exists\, z.(Q_{b_2} \wedge Prd(T_{a_2}) \wedge d_2' = d_2)$$

En rapportant ces définitions dans (G_2), puis par application du théorème de déduction, on obtient les buts suivants:

$$\forall\, z.(Q_{b_2} \rightarrow Trm(T_{a_2})) \quad (\mathrm{G}_3)$$

$$\forall\, c_2''.(Prd(@z.(Q_{b_2} \Rightarrow T_{a_2})) \rightarrow \exists\, c_1'.(Prd(@z.(P_{b_1} \Rightarrow S_{a_1})) \wedge J(c_1',c_2''))) \quad (\mathrm{G}_4)$$

sous les hypothèses supplémentaires:

$$\forall\, z.(P_{b_1} \rightarrow Trm(S_{a_1})) \quad (\mathrm{H}_4)$$

$$J(c_1,c_2) \quad (\mathrm{H}_5)$$

Preuve de (G_3): par élimination du quantificateur universel et l'application du théorème de déduction, il s'agit de prouver:

$$Trm(T_{a_2}) \quad (\mathrm{G}_{3.1})$$

sous l'hypothèse:

$$Q_{b_2} \quad (\mathrm{H}_6)$$

Pour prouver ($\mathrm{G}_{3.1}$) par réutilisation de (H_2), on doit pouvoir déduire de notre environnement de preuve les deux termes $Trm(S_{a_1})$ et $J(a_1,a_2)$. (H_5) (H_6) et (H_3) donnent:

$$P_{b_1} \quad (\mathrm{H}_7)$$

(H_7) et l'instanciation des variables z par elles-mêmes dans (H_4) donnent:

$$Trm(S_{a_1}) \tag{H_8}$$

Il reste à générer de notre environnement d'hypothèses, constitué de (H_5), (H_6) et (H_8), l'hypothèse:

$$J(a_1,a_2) \tag{H_9}$$

Une condition nécessaire et suffisante à la réutilisation est:

$$\boxed{(J(c_1,c_2) \wedge Q_{b_2} \wedge Trm(S_{a_1})) \rightarrow J(a_1,a_2)}$$

(H_8), (H_9) et (H_2) prouvent (G_3) et génèrent l'hypothèse supplémentaire:

$$\forall\, a_2''.(Prd(T_{a_2}) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J(a_1',a_2''))) \tag{H_{10}}$$

Preuve de (G_4): par élimination du quantificateur universel et application du théorème de déduction, on doit prouver:

$$\exists\, c_1'.(\exists\, z.(P_{b_1} \wedge Prd(S_{a_1}) \wedge d_1' = d_1) \wedge J(c_1',c_2'')) \tag{$G_{4.1}$}$$

sous l'hypothèse supplémentaire:

$$\exists\, z.(Q_{b_2} \wedge Prd(T_{a_2}) \wedge d_2'' = d_2) \tag{H_{11}}$$

Soit z_0 une valeur de z qui vérifie cette dernière hypothèse, On a donc:

$$[z := z_0]Q_{b_2} \tag{H_{12}}$$
$$[z := z_0]Prd(T_{a_2}) \tag{H_{13}}$$
$$d_2'' = d_2 \tag{H_{14}}$$

En instanciant les variables a_2'' et z par elles-mêmes et z_0 respectivement dans (H_{10}), puis en utilisant (H_{13}), on obtient:

$$\exists\, a_1'.([z := z_0]Prd(S_{a_1}) \wedge [z := z_0]J(a_1',a_2'')) \tag{H_{15}}$$

Soit $sol_{a_1'}$ une des valeurs de a_1' qui satisfait cette dernière hypothèse, on a donc:

$$[z := z_0;\ a_1' := sol_{a_1'}]Prd(S_{a_1}) \tag{H_{16}}$$
$$[a_1' := sol_{a_1'};\ z := z_0]J(a_1',a_2'') \tag{H_{17}}$$

Réutiliser la preuve du raffinement ($S_{a_1} \sqsubseteq T_{a_2}$) revient à vérifier que la solution $sol_{a_1'}$ est aussi une solution du raffinement à prouver. Pour cela, on doit vérifier que $sol_{a_1'}$ satisfait le but ($G_{4.1}$), c.à.d:

$$\exists\, d_1'.(\exists\, z.(P_{b_1} \wedge [a_1' = sol_{a_1'}]Prd(S_{a_1}) \wedge d_1' = d_1) \wedge [a_1' = sol_{a_1'}]J(a_1' \cup d_1',a_2'' \cup d_2'')) \tag{$G_{4.2}$}$$

Comme d_2'' est égal à d_2 (d'après (H_{14})), en prenant d_1' égal à d_1, le but à prouver devient:

$$\exists\, z.(P_{b_1} \wedge [a_1' = sol_{a_1'}]Prd(S_{a_1})) \wedge [a_1' = sol_{a_1'}]J(a_1' \cup d_1,a_2'' \cup d_2) \tag{$G_{4.3}$}$$

En prenant z_0 comme valeur de z, les buts à prouver sont les suivants:

$$[z := z_0]P_{b_1} \tag{$G_{4.4}$}$$
$$[a_1' = sol_{a_1'};\ z := z_0]Prd(S_{a_1}) \tag{$G_{4.5}$}$$
$$[a_1' = sol_{a_1'}]J(a_1' \cup d_1,a_2'' \cup d_2) \tag{$G_{4.6}$}$$

Le but ($G_{4.4}$) se prouve par (H_3), (H_5) et (H_{12}). Le but ($G_{4.5}$) se prouve par (H_{16}), car la solution $sol_{a_1'}$ dépend de z_0 et non pas de z. Le but ($G_{4.6}$) doit pouvoir être déduit de l'ensemble des hypothèses constituant l'environnement de preuve. On doit donc prouver que:

$$\boxed{\begin{array}{r}([z := z_0]Q_{b_2} \wedge [z := z_0]Prd(T_{a_2}) \wedge J(c_1,c_2) \wedge [z := z_0;\ a_1' := sol_{a_1'}]Prd(S_{a_1}) \wedge \\ [a_1' := sol_{a_1'};\ z := z_0]J(a_1',a_2'')) \rightarrow [a_1{}' = sol_{a_1'}]J(a_1{}' \cup d_1, a_2{}'' \cup d_2)\end{array}}$$

Résumé

Soient S_{a_1}, T_{a_2} deux substitutions et $J(a_1,a_2)$ un prédicat tels que:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$$

Soient P_{b_1}, Q_{b_2} et $J(a_1 \cup b_1, a_2 \cup b_2)$ trois prédicats tels que ($c_i = a_i \cup b_i$ et $d_i = b_i - a_i$):

$i.$ $(Q_{b_2} \wedge J(c_1,c_2)) \rightarrow P_{b_1}$

$ii.$ $(Q_{b_2} \wedge Trm(S_{a_1}) \wedge J(c_1,c_2)) \rightarrow J(a_1,a_2)$

Soit z un sous-ensemble des variables non raffinées de $(a_1{}^f \cap b_1)$. Pour toute valeur z_0 de z vérifiant:

$$Q_{b_2} \wedge Prd(T_{a_2})$$

si l'une des valeurs a_1', associée à z_0, vérifiant:

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2}$$

vérifie

$$([z := z_0]Q_{b_2} \wedge [z := z_0]Prd(T_{a_2}) \wedge J(c_1,c_2) \wedge [z := z_0]Prd(S_{a_1}) \wedge J(a_1',a_2'')) \rightarrow J(a_1{}' \cup d_1, a_2{}'' \cup d_2{}'')$$

alors celle-ci peut être réutilisée pour prouver $(@z.(P_{b_1} \Rightarrow S_{a_1})) \sqsubseteq_{J(c_1,c_2)} (@z.(Q_{b_2} \Rightarrow T_{a_2}))$ pour la même valeur z_0 de z.

Remarques

1. La condition (i) est une condition nécessaire pour la correction du raffinement:
$$(@z.(P_{b_1} \Rightarrow S_{a_1})) \sqsubseteq_{J(c_1,c_2)} (@z.(Q_{b_2} \Rightarrow T_{a_2}))$$
2. La condition (ii) est une condition nécessaire de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $S_{a_1} \sqsubseteq T_{a_2}$, alors il faut être capable d'exhiber dans l'environnement de preuve de $(@z.(P_{b_1} \Rightarrow S_{a_1})) \sqsubseteq_{J(c_1,c_2)} (@z.(Q_{b_2} \Rightarrow T_{a_2}))$. Dans le cas contraire la stratégie de réutilisation n'est pas justifiée.
3. La dernière condition est une condition nécessaire uniquement pour la réutilisation de la solution du raffinement de $S_{a_1} \sqsubseteq T_{a_2}$. Une autre solution du raffinement sans réutilisation peut éventuellement exister.
4. La réutilisation de la preuve de raffinement de $S_{a_1} \sqsubseteq T_{a_2}$ est réalisable pour toute valeur de a_1' qui vérifie la dernière condition: on n'exige pas que toutes les valeurs a_1' vérifient cette condition.
5. Si la substitution S_{a_1} est déterministe alors la dernière condition est vérifiée. En effet, il existe une seule valeur de a_1' qui satisfait le raffinement $(S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2})$, celle-ci est déterminée par $Prd(S_{a_1})$. De plus, une solution possible du raffinement $(@z.(P_{b_1} \Rightarrow S_{a_1})) \sqsubseteq_{J(c_1,c_2)} (@z.(Q_{b_2} \Rightarrow T_{a_2}))$ doit satisfaire le $Prd(S_{a_1})$ (à moins que le raffinement soit une tautologie) qui lui-même n'admet qu'une seule solution: les deux raffinements admettent donc le même espace de solutions.
6. Si le prédicat $J(a_1,a_2)$ est fonctionnel en a_1 alors la dernière condition est vérifiée. En effet, une solution possible du raffinement $(@z.(P_{b_1} \Rightarrow S_{a_1})) \sqsubseteq_{J(c_1,c_2)} (@z.(Q_{b_2} \Rightarrow T_{a_2}))$ doit satisfaire le prédicat $J(c_1,c_2)$. D'après (ii) cette solution satisfait le prédicat $J(a_1,a_2)$. Or $J(a_1,a_2)$ n'admet qu'une solution unique a_1' pour toute valeur de a_2: cette solution unique est donc solution de $(@z.(P_{b_1} \Rightarrow S_{a_1})) \sqsubseteq_{J(c_1,c_2)}$ $(@z.(Q_{b_2} \Rightarrow T_{a_2}))$.

Annexe C

Règles d'implémentation et traduction SQL

Cette annexe complète le chapitre 5. On présente la structure complète de la machine formalisant une table relationnelle. Puis, on donne les règles d'implémentation de la substitution d'ajout pour un attribut séquentiel ainsi que celle de mise à jour d'un attribut. Enfin, la traduction en relationnel de quelques invariants spécifiques **B** est présentée.

C.1 Formalisation B d'une table relationnelle

Dans cette partie, nous décrivons la structure des machines B permettant l'implémentation des tables relationnelles. Nous distinguons deux types de tables. Dans ces machines sont spécifiées les opérations suivantes :

• L'ajout d'un tuple qui correspond à l'ajout d'un objet dans une classe ou une association. Les paramètres d'entrée de cette opération sont les valeurs des différents champs.

• Contrairement à l'opération d'ajout, plusieurs cas sont à distinguer pour l'opération de suppression.
  – **Suppression dans une classe:** les tuples à supprimer sont désignés par la valeur de leur attribut clé.
  – **Suppression dans une association:** comme nous l'avons vu dans le chapitre 3, deux types de suppression sont possibles: suppression d'un lien désigné par les deux objets concernés, ou suppression d'un ensemble de liens désignés par un ensemble d'objets de l'une des deux classes mises en œuvre par l'association.

• Mise à jour de la valeur d'un champ pour un tuple donné. Ce tuple est défini par la valeur de son attribut clé.

En plus de ces opérations, on spécifie pour chaque classe n'ayant pas initialement de clés une opération qui renvoie une valeur non encore attribuée de la clé . Cette opération correspond au terme *generer(TT_c-ran($v_{clé}$))* de la règle de raffinement relative à l'introduction de la clé.

La structure générale de la machine importée *Rel_a* associée à la variable T_a est la suivante :

MACHINE *Rel_a*
//la machine *Contexte* définit les ensembles abstraits correspondant aux attributs dont le type est un ensemble abstrait
SEES *Contexte*
DEFINITIONS *TABLE* $\widehat{=}$ **struct**(f_1: $T_1, \ldots, f_n$: T_n)
VARIABLES *Table_a*
INVARIANT
    Table_a $\subseteq$ *TABLE*

INITIALISATION $Table\_a := \emptyset$

OPERATIONS

//opération d'ajout d'un objet dans une classe ou une association

Insert_a(val$_1$,...,val$_n$)=

PRE $val_1 \in T_1 \wedge \ldots \wedge val_n \in T_n$ **THEN**

$Table\_a := Table\_a \cup \{\mathbf{rec}(val_1, \ldots, val_n)\}$

END;

//opération de suppression d'un objet d'une classe de clé f_1

Delete_a(val$_1$)=

PRE $val_1 \in T_1$ **THEN**

LET *ens* **BE**

$ens = \bigcup y.(y \in Table\_a \wedge y'f_1$[2] $= val_1 \mid \{y\})$

IN

Table_a:=Table_a - ens

END

END;

//opération de suppression d'un lien d'une association de clé (f_1,f_2)

Delete_a(val$_1$, val$_2$)=

PRE $val_1 \in T_1 \wedge val_2 \in T_2$ **THEN**

LET *ens* **BE**

$ens = \bigcup y.(y \in Table\_a \wedge y'f_1 = val_1 \wedge y'f_2 = val_2 \mid \{y\})$

IN

Table_a:=Table_a - ens

END

END;

//opération de suppression d'un ensemble de liens définis par l'un des attributs de la clé (f_1 ou f_2)

// d'une association

Delete_a_f_i(val)=

PRE $val \in T_i$ **THEN**

LET *ens* **BE**

$ens = \bigcup y.(y \in Table\_a \wedge y'f_i = val \mid \{y\})$

IN

Table_a:=Table_a - ens

END

END;

//opération de mise à jour d'un attribut f_i pour un tuple d'une classe défini par la valeur de sa clé f_1

Update_f$_i$(val$_1$, v)=

PRE $val_1 \in T_1 \wedge v \in T_i$ **THEN**

$Table\_a :\in (Table\_a \subseteq TABLE \wedge$

$\forall xx. (xx \in Table\_a\$0$[3] $\wedge xx'f_1 \neq val_1 \Rightarrow xx \in Table\_a) \wedge$

$\forall xx. (xx \in Table\_a\$0 \wedge xx'f_1 = val_1) \Rightarrow rec(xx'f_1, \ldots, xx'f_{i-1}, v, \ldots, xx'f_n) \in Table\_a) \wedge$

$\forall xx.(xx \in Table\_a \Rightarrow (xx'f_1 \neq val_1 \wedge xx \in Table\_a\$0) \vee (xx'f_1 = val_1 \wedge xx'f_i = v)))$

END;

//opération renvoyant une valeur non attribuée de l'attribut clé f_1

val $\longleftarrow$ NouvelleValeurClé=

BEGIN

$val : (val \in T_1 \wedge \neg(\exists xx.(xx \in Table\_a \wedge xx'f_1 = val)))$

END

END

2. $y'f$ désigne la valeur du champ f pour l'enregistrement y

3. $v\$0$ désigne la valeur avant de la variable v.

C.2 Règle d'implémentation de la séquence

L'implémentation d'ajout dans une séquence est réalisée comme suit :

RULE *Implément_Seq_Ajout* **IS**
REFINES
    $Rang\_Att := Rang\_Att \cup \{cc\} \times dom(val)$;
    $T_{Seq} := T_{Seq} \cup NouveauTuple$
REFINEMENT
  VAR x **IN**
    $x := 0$;
    WHILE $(x < size(val))$ **DO**
      $x := x+1$;
      $Insert\_Rang\_Att(cc,x,val(x))$
    INVARIANT
      $x \in NAT \wedge$
      $x \geq 0 \wedge$
      $x \leq size(val) \wedge$
      $Table\_Rang\_Att \subseteq$ **struct**$(Rang\_Att_c\text{: } T_{c_{clé}},\ Rang\_Att_{val} : NAT,\ Valeur : T_{Att}) \wedge$
      $\forall\, (a_1, a_2, a_3).\ (a_1 \mapsto a_2 \mapsto (a_1 \mapsto a_2 \mapsto a_3) \in T_{Rang\_Att} \cup NouveauTuple \Rightarrow$
        rec$(a_1, a_2, a_3) \in Table\_Rang\_Att) \wedge$
      $\forall\, (a_1, a_2, a_3).(a_1\ T_{c_{clé}} \in \wedge\ a_2 \in NAT \wedge a_3 \in T_{Att} \wedge$
        rec $(Rang\_Att_c\text{: } a_1,\ Rang\_Att_{val}\text{: } a_2,\ Valeur\_Att\text{: } a_3) \in Table\_Rang\_Att \Rightarrow$
          $a_1 \mapsto a_2 \mapsto (a_1 \mapsto a_2 \mapsto a_3) \in T_{Rang\_Att} \cup NouveauTuple) \wedge$
      $\forall\, n.(n \in NAT \wedge n \geq 1 \wedge n \leq x \Rightarrow$
          rec$(Rang\_Att_c\text{: } cc,\ Rang\_Att_{val}\text{: } n,\ Valeur\text{: } val(n)) \in Table\_Rang\_Att)$
    VARIANT $size(val) - x$
    END
  END

où:

- $nouveauTuple \,\widehat{=}\, ((\{cc\} \times dom(val) \times \{c_{clé}(cc)\}) \otimes prj_2(\{cc\},dom(val)) \otimes (prj_2(\{cc\},dom(val));\ val))$
- T_{Att} et $T_{c_{clé}}$ désignent respectivement les types de l'attribut séquentiel et de la clé associée à la classe sur laquelle est défini cet attribut,
- x: permet de parcourir la séquence,
- $Insert\_Rang\_Att$: c'est l'opération d'insertion d'un tuple dans la table $Rang\_Att$,
- les quatre premiers invariants permettent de typer les variables utilisées dans la boucle.
- le cinquième et sixième invariants de collage précisent la relation existant entre la variable abstraite T_{Rang_Att} et la variable concrète $Table\_Rang\_Att$,
- le dernier invariant de collage précise ce que la boucle doit réaliser.

C.3 Règle d'implémentation de la substitution de mise à jour

La substitution de mise à jour est implémentée par l'opération *Update* de la machine importée correspondant à la variable. Formellement, la règle d'implémentation de cette substitutions est la suivante :

RULE *Implément_Modif_*f_1 **IS**
REFINES
    $T_a := f'_1 \otimes \ldots \otimes f'_{i-1} \otimes (f'_i \lhd\!\!+ \{a_1 \mapsto val\}) \otimes \ldots f_n$
REFINEMENT
    $Update\_f_i(a_1, val)$
END

C.4 Traduction SQL des contraintes relationnelles

Dans cette section, nous présentons la traduction des contraintes liées aux domaines et rangs des associations. Nous présentons également la traduction des contraintes liées à l'héritage. Rappelons que cette traduction n'est pas obligatoire car les programmes générés respectent bien ces contraintes. Cette génération peut constituer une meilleure documentation pour le schéma de la base de données.

Contraintes liées aux domaine et rang d'une relation Une relation f définie de A dans D peut être obligatoire (totale) sur A ($dom(f) = A$) ou D ($ran(f) = D$). Ces contraintes, exprimées par des invariants B, sont raffinées automatiquement et sont traduites en SQL par les assertions suivantes:

- $dom(f) = A$:

  CREATE ASSERTION *Domaine_f*
  CHECK(
  NOT EXISTS (**SELECT** $A_{clé}$
  FROM $Table\_A$
  WHERE $A_{clé}$ **NOT IN** (**SELECT** f_A **FROM** $Table\_f$)))

  $A_{clé}$ désigne la clé de la table *Table_A* associée à la variable A, et f_A la clé étrangère de la table $Table\_f$ associée à la variable f. La clé étrangère f_A est liée par une contrainte référentielle à $A_{clé}$.
- $ran(f) = D$: idem cas précédent.

Contrainte de la séquence. Une séquence est une fonction totale particulière. Le domaine d'une séquence est un intervalle continu d'entiers. Sur chaque table correspondant à une séquence doit être exprimé le fait que l'ensemble des valeurs de l'attribut $Rang\_Att_{val}$, associé à une valeur donnée de l'attribut $Rang\_Att_c$, est un intervalle continu d'entiers. Pour chaque attribut, Att de type $Seq(T)$, défini sur la variable de classe c, est associée une table *Table_Rang_Att* définie par:

CREATE TABLE *Table_Rang_Att* (
Rang_Att$_c$ **TT**$_{c_{clé}}$,
Rang_Att$_{valeur}$ **INT**,
valeur **T**$_{Att}$ **NOT NULL**,
PRIMARY KEY (*Rang_Att*$_c$, *Rang_Att*$_{valeur}$))

Sur cette table doit être spécifiée la contrainte SQL suivante (on utilise l'expression de la somme des k premiers termes d'une suite arithmétique d'entiers de premier terme 1 et de raison 1) :

CREATE ASSERTION *AttSéquence*
CHECK(
NOT EXISTS (**SELECT** *Rang_Att*$_c$
FROM *Table_Rang_Att*
GROUP BY *Rang_Att*$_c$
HAVING SUM(*Rang_Att*$_{valeur}$) $\neq$ (**COUNT**$(*) \times ($**COUNT**$(*) + 1))/2)$))

Contraintes liées à l'élimination des sous-classes. Le raffinement d'une structure d'héritage par élimination des spécialisations est basé sur l'extension des informations, i.e attributs et associations, associées à chaque sous-classe sur la super classe. Néanmoins, nous devons garder le fait que chaque information n'est définie que pour sa sous-classe d'origine. Pour cela cinq cas sont à distinguer :

- **cas d'un attribut monovalué obligatoire**: un tel attribut doit être non nul pour tous les objets de la sous-classe (*Est_c*$_i$=*TRUE*). En SQL, cette contrainte est exprimée par:

  CHECK ((*Est_c*$_i$=*false* **OR** *att* **IS NOT NULL**) **AND** (*Est_c*$_i$=*true* **OR** *att* **IS NULL**))
- **cas d'un attribut monovalué optionnel**: un tel attribut ne peut être valué que pour les objets de la sous-classe (*Est_c*$_i$=*TRUE*). En SQL, cette contrainte est exprimée par:

  CHECK (*att* **IS NULL OR** *Est_c*$_i$=*true*)

Remarque: ces deux cas englobent le cas d'associations monovaluées dont la sous-classe C_i est source.

- **cas d'une association monovaluée dont la sous-classe est cible**: une telle association est traduite par une clé étrangère dans la table correspondant à la classe source. On doit spécifier que la valeur d'un tel attribut doit correspondre à la clé d'un objet de la sous-classe (*Est_*c_i*=TRUE*). En SQL, cette contrainte est exprimée par l'assertion (c désigne la variable de classe associée à la super-classe) :

  CREATE ASSERTION $AssMono$
  CHECK (
  att_Mig **IN** (**SELECT** $c_{clé}$
  FROM c
  WHERE *Est_*c_i*=1*))

- **cas d'un attribut multivalué optionnel** : idem cas précédent.
- **cas d'un attribut multivalué obligatoire**: idem $ran(Ass) = D$

Remarque: ces deux derniers cas englobent le cas d'associations multivaluées dont la sous-classe C_i est cible.

Contraintes liées à la conservation d'une hiérarchie d'héritage. Lors de la phase du raffinement de l'héritage, une solution était de garder l'ensemble des classes constituant la hiérarchie. On doit rajouter dans ce cas une contrainte indiquant que l'ensemble des valeurs de la clé de chaque sous-classe est inclus dans l'ensemble des valeurs de la clé de la super-classe. C'est une manière d'indiquer qu'un objet de chaque sous-classe est aussi un objet de la super-classe. Cette contrainte est spécifiée en SQL par ($c_{i\,clé}$ désigne la clé de la table *Table_*c_i associée à la sous-classe c_i, $c_{clé}$ désigne la clé de la table *Table_c* associée à la super-classe c) :

CREATE ASSERTION $C_i - sousclasseC$
CHECK(
NOT EXISTS (**SELECT** $c_{i\,clé}$
FROM *Table_*c_i
WHERE $c_{i\,clé}$ **NOT IN** (**SELECT** $c_{clé}$ **FROM** *Table_c*)))

Définition des vues. Les vues permettent de garder la notion de super/sous classes lors de la phase du raffinement de l'héritage. Elles permettent de traduire les définitions générées lors du raffinement.

- **élimination de la super classe**: En B, l'élimination de la super classe est remplacée par une définition: c=$\bigcup_{i=1..n} c_i$. En SQL, cette définition est traduite par la vue :

  CREATE VIEW c **AS**
  SELECT *
  FROM c_1 **FULL OUTER JOIN** ... **FULL OUTER JOIN** c_n

- **élimination des sous classes**: En B, l'élimination des sous classes est remplacée par un ensemble de définitions (une par sous classe): $c_i = Est\_c_i^{-1}([\{TRUE\}])$. En SQL, cette définition est traduite par la vue :

  CREATE VIEW c_i **AS**
  SELECT $att_1, \ldots, att_n$
  FROM *Table_c*
  WHERE *Est_*c_i*=1*

  où Att_i désignent les fonctions pour lesquelles la sous-classe C_i ou la super-classe C sont sources.

Annexe D

Étude de cas : Gestion d'un club vidéo

Nous présentons dans cette annexe l'étude de cas de gestion d'un club vidéo. Nous donnons l'énoncé du cas, puis sa spécification B. Nous illustrons les différentes étapes de raffinement présentées dans ce manuscrit. Les différentes classes **JAVA** implémentant l'application sont également données.

D.1 Présentation du cas

On considère un club vidéo de gestion de prêts d'un ensemble de cassettes. Chaque cassette identifiée par un titre (*Titre_K7*) est caractérisée par le nombre de copies disponibles (*NbLibre*). Un client décrit par un nom (*Nom_Cl*) appartient à un unique groupe. Un groupe identifié par un numéro (*Num_Gr*) détermine le tarif (*Tarif*) de prêt de ses clients. Le prêt d'une cassette par un client est caractérisée par une date (*Date_E*) et un statut (*Statut*) (voir figure D.1).

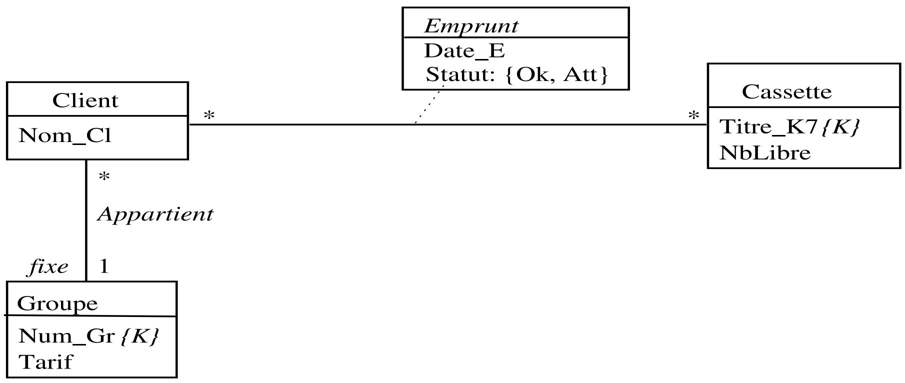


FIG. D.1 – *Diagramme de classes du club vidéo*

Une instance de la classe *Emprunt* peut être, entre autres, en cours (*EmpruntEnCours*) ou en attente (*EmpruntEnAttente*). À la réception de l'événement de création d'un nouvel emprunt (*CréationEmprunt*) deux actions sont possibles : si une copie de la cassette demandée est disponible alors on ajoute le prêt avec un statut 'Ok', sinon on l'ajoute avec un statut en attente 'Att' (Voir figure D.2).

On s'intéresse à la spécification de la transaction d'enregistrement d'un prêt pour un nouveau client *AjoutEmprunt* (Voir figure D.3). Cette transaction reçoit, en entrée, quatre paramètres qui sont : le nom du client à créer (no), son groupe (gr), la cassette à prêter (ca) et la date du prêt (da). Après avoir choisi un client cl parmi les client non existants ($Client - client$), cette transaction déclenche l'événement *CréationEmprunt* du diagramme d'états/transitions de la classe *Emprunt*, ajoute le nouveau client (invocation de l'opération *BasicAjoutClient*) et met à jour le nombre de copies de cassettes disponibles si le prêt n'est pas en attente (invocation de l'opération *BasicChangeNbLibre*).

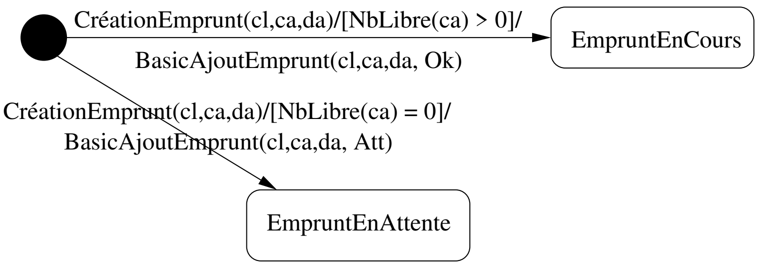


FIG. D.2 – *Diagramme états/transitions de la classe Emprunt*

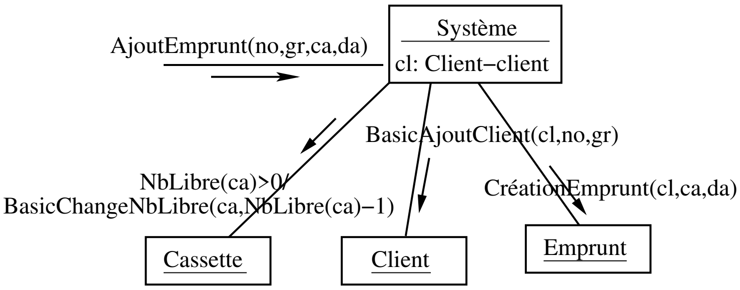


FIG. D.3 – *Diagramme de collaboration décrivant le prêt d'une cassette à un nouveau client*

D.2 Spécification formelle B

L'application des différentes règles de traduction du chapitre 3 génère une spécification abstraite B dont l'architecture est représentée par la figure D.4.

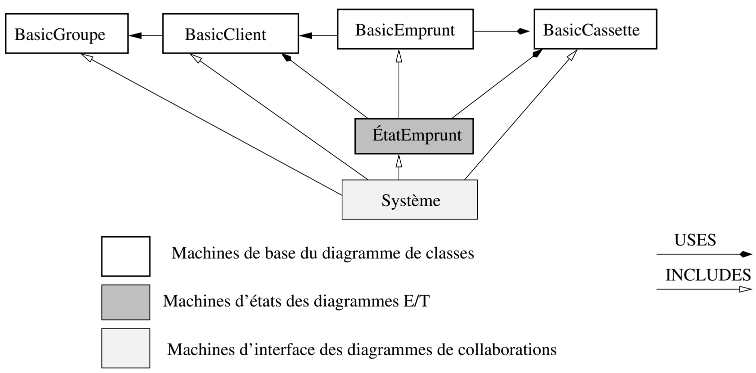


FIG. D.4 – *Diagramme états/transitions de la classe Emprunt*

Nous donnons ci-après le contenu des différentes machines constituant cette architecture. Seules les opérations de base nécessaires à l'illustration de notre approche sont reprises.

- Machine traduisant la classe *Cassette*

MACHINE *BasicCassette*
SETS *Cassette*
VARIABLES *cassette, Titre_K7, NbLibre*
INVARIANT
$cassette \subseteq Cassette \wedge Titre\_K7 \in cassette \rightarrowtail STRING \wedge NbLibre \in cassette \rightarrow NAT$
INITIALISATION *cassette, Titre_K7, NbLibre:=*$\emptyset$,$\emptyset$,$\emptyset$
OPERATIONS
BasicChangeNbLibre(ca,nb)=
PRE $ca \in cassette \wedge nb \in NAT$ **THEN**
$NbLibre(ca) := nb$
END
END

- Machine traduisant la classe *Client*

MACHINE *BasicClient*
SETS *Client*
USES *BasicGroupe*
VARIABLES *client, Nom_Cl, Appartient*
INVARIANT
$client \subseteq Client \wedge Nom\_Cl \in client \rightarrow STRING \wedge Appartient \in client \rightarrow groupe$
INITIALISATION *client, Nom_Cl, Appartient:=* $\emptyset$,$\emptyset$
OPERATIONS
BasicAjoutClient(cl, no, gr)=
PRE $cl \in Client\text{-}client \wedge no \in STRING \wedge gr \in groupe$ **THEN**
$client := client \cup \{cl\} ||$
$Nom\_Cl := Nom\_Cl \cup \{cl \mapsto no\} ||$
$Appartient := Appartient \cup \{cl \mapsto gr\}$
END
END

- Machine traduisant la classe association *Emprunt*

MACHINE *BasicEmprunt*
USES *BasicClient, BasicCassette*
SETS *STATUT={Ok, Att}*
VARIABLES *Emprunt, Date_E, Statut*
INVARIANT
$Emprunt \in client \leftrightarrow cassette \wedge Date\_E \in Emprunt \rightarrow NAT \wedge Statut \in Emprunt \rightarrow STATUT$
INITIALISATION *Emprunt, Date_E, Statut:=* $\emptyset$,$\emptyset$,$\emptyset$
OPERATIONS
BasicAjoutEmprunt(cl, ca, da,st)=
PRE $cl \in Client \wedge ca \in Cassette \wedge da \in NAT \wedge st \in STATUT$ **THEN**
$Emprunt := Emprunt \cup \{cl \mapsto ca\} ||$
$Date\_E := Date\_E \cup \{cl \mapsto ca \mapsto da\} ||$
$Statut := Statut \cup \{cl \mapsto ca \mapsto st\}$
END
END

- Machine traduisant le diagramme d'états/transitions de la classe *Emprunt*

```
MACHINE    ÉtatEmprunt
INCLUDES   BasicEmprunt
USES    BasicCassette, BasicClient
DEFINITIONS
    EmpruntEnCours(emp) ≙ emp ∈ Statut⁻¹[{Ok}]
    EmpruntEnAttente(emp) ≙ emp ∈ Statut⁻¹[{Att}]
OPERATIONS
CréationEmprunt(cl, ca, da)=
PRE   cl ∈ client ∧ ca ∈ cassette ∧ da ∈ NAT    THEN
    SELECT   NbLibre(ca) >0   THEN   BasicAjoutAmprunt(cl, ca, da,Ok)
    WHEN   NbLibre(ca) =0   THEN   BasicAjoutAmprunt(cl, ca, da,Att)
    END
END
END
```

- Machine traduisant les diagrammes de collaborations

```
MACHINE    Système
INCLUDES   BasicCassette, BasicClient, ÉtatEmprunt, BasicGroupe
OPERATIONS
AjoutEmprunt(no, gr, ca, da)=
PRE   Client -client ≠ ∅ ∧ no ∈ STRING ∧ gr ∈ group ∈ ca ∈ cassette ∧ da ∈ NAT    THEN
    ANY   cl ∈ Client − client   THEN
      IF   NbLibre(ca) >0   THEN   BasicChangeNbLibre (ca, NbLibre(ca)-1)   END||
      BasicAjoutClient(cl, no, gr) ||
      CréationEmprunt(cl, ca, da)
    END
END
END
```

D.2.1 Construction de la machine de sûreté *RobusteSystème*

L'opération *AjoutEmprunt* telle que exprimée ne garantit une exécution correcte que si celle-ci est appelée sous ses préconditions. Ces dernières doivent donc être testées avant l'appel de l'opération en question. Nous construisons pour cela la machine *RobusteSystème* qui remplit cette fonction. La spécification de cette machine est la suivante :

```
MACHINE    RobusteSystème
INCLUDES   Système
OPERATIONS
resultat ⟵ RobusteAjoutEmprunt(no, gr, ca, da)=
PRE   no ∈ STRING ∧ gr ∈ Groupe ∈ ca ∈ Cassette ∧ da ∈ NAT    THEN
  IF Client − client ≠ ∅    THEN
    IF   gr ∈ groupe    THEN
      IF   ca ∈ cassette   THEN
        AjoutEmprunt(no,gr,ca,da)||
        resultat:=”succes”
      ELSE
        resultat:=”CassetteInexistante”
      END
    ELSE
      resultat:=”GroupeInexistant”
    END
  ELSE
    resultat:=”NombreMaximalClientAtteint”
```

END
END
END

D.3 Processus de raffinement

Dans cette section, nous présentons les différentes étapes de raffinement qui permettent la génération d'une implémentation relationnelle. Comme on l'avait noté au chapitre 4, une étape préalable à ce processus de raffinement est la construction de la machine expansée.

D.3.1 Expansion de la spécification initiale

Dans cette étape, on construit la machine équivalente à l'ensemble de la spécification de la figure D.4. La machine expansée *Système_exp* a la structure suivante :

MACHINE $Systeme\_exp$
SETS $STATUT = \{Ok, Att\};\ Client;\ Groupe;\ Cassette$
VARIABLES
$cassette_1, Titre\_K7_1, NbLibre_1,$
$client_1, Nom\_Cl_1, Appartient_1,$
$groupe_1, Num\_Gr_1, Tarif_1,$
$Emprunt_1, Date\_E_1, Statut_1$
INVARIANT
$client_1 \subseteq Client \wedge Nom\_Cl_1 \in client_1 \rightarrow STRING \wedge groupe_1 \subseteq Groupe \wedge$
$Num\_Gr_1 \in groupe_1 \rightarrowtail NAT \wedge Tarif_1 \in groupe_1 \rightarrow NAT \wedge Appartient_1 \in client_1 \rightarrow groupe_1 \wedge$
$cassette_1 \subseteq Cassette \wedge Titre\_K7_1 \in cassette_1 \rightarrowtail STRING \wedge NbLibre_1 \in cassette_1 \rightarrow NAT \wedge$
$Emprunt_1 \in client_1 \leftrightarrow cassette_1 \wedge Date\_E_1 \in Emprunt_1 \rightarrow NAT \wedge Statut_1 \in Emprunt_1 \rightarrow STATUT$
INITIALISATION
$cassette_1, Titre\_K7_1, NbLibre_1, client_1, Nom\_Cl_1, groupe_1, Num\_Gr_1, Tarif_1, Appartient_1, Emprunt_1,$
$Date\_E_1, Statut_1 := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$
DEFINITIONS
BasicChangeNbLibre(ca,nb) $\widehat{=}$
BEGIN
$NbLibre_1(ca) := nb$
END;

BasicAjoutClient(cl, no, gr) $\widehat{=}$
BEGIN
$client_1 := client_1 \cup \{cl\} ||$
$Nom\_Cl_1 := Nom\_Cl_1 \cup \{cl \mapsto no\} ||$
$Appartient_1 := Appartient_1 \cup \{cl \mapsto gr\}$
END;

BasicAjoutEmprunt(cl, ca, da,st) $\widehat{=}$
BEGIN
$Emprunt_1 := Emprunt_1 \cup \{cl \mapsto ca\} ||$
$Date\_E_1 := Date\_E_1 \cup \{cl \mapsto ca \mapsto da\} ||$
$Statut_1 := Statut_1 \cup \{cl \mapsto ca \mapsto st\}$
END;

CreationEmprunt(cl, ca, da) $\widehat{=}$
BEGIN
SELECT $NbLibre_1(ca) > 0$ **THEN** $BasicAjoutEmprunt(cl, ca, da, Ok)$

WHEN $NbLibre_1(ca) = 0$ **THEN** $BasicAjoutEmprunt(cl, ca, da, Att)$
END
END

OPERATIONS
AjoutEmprunt(no, gr, ca, da)=
PRE $Client - client_1 \neq \emptyset \wedge no \in STRING \wedge gr \in groupe_1 \wedge ca \in cassette_1 \wedge da \in NAT$
THEN
ANY cl **WHERE** $cl \in Client - client_1$ **THEN**
IF $NbLibre_1(ca) > 0$ **THEN** $BasicChangeNbLibre(ca, NbLibre_1(ca) - 1)$ **END** ||
$BasicAjoutClient(cl, no, gr)$||
$CreationEmprunt(cl, ca, da)$
END
END
END

D.3.2 Identification des objets : introduction des clés

La classe *Client* ne possédant pas de clé (pas de fonction totale injective sur la variable $client_1$), on ajoute à cette étape une variable Num_Cl_1 définie comme une fonction injective de l'ensemble des clients existants vers un type *TT_Client* qui sera déclaré comme un ensemble abstrait dans le composant raffinant. La machine *Système_exp* est donc raffinée par (seules les parties modifiées sont reprises) :

REFINEMENT *Systeme_exp_C*
REFINES *Systeme_exp*
SETS *TT_Client*
VARIABLES
$\ldots, Num\_Cl_1$
INVARIANT
$Num\_Cl_1 \in client_1 \rightarrowtail TT\_Client$
INITIALISATION
$\ldots, Num\_Cl_1 := \ldots, \emptyset$
DEFINITIONS
BasicAjoutClient(cl, no, gr) $\widehat{=}$
BEGIN
ANY num **WHERE** $num \in TT\_Client - ran(Num\_Cl_1)$ **THEN**
$client_1 := client_1 \cup \{cl\};$
$Num\_Cl_1 := Num\_Cl_1 \cup \{cl \mapsto num\};$
$Nom\_Cl_1 := Nom\_Cl_1 \cup \{cl \mapsto no\};$
$Appartient_1 := Appartient_1 \cup \{cl \mapsto gr\}$
END
END
$\ldots$
END

D.3.3 Élimination des associations monovaluées

L'association $Appartient_1$ étant monovaluée, celle-ci est remplacée par une nouvel attribut, pour la classe *Client*, dont la valeur est égale à la clé du groupe concerné. Cet attribut est défini par :

$$AppartientMig_1 \in client_1 \rightarrow NAT \wedge$$
$$AppartientMig_1 = (Appartient_1;\ Num\_Gr_1)$$

Le composant *Système_exp_C* est raffiné par le composant *Système_exp_Mig*, l'opération *BasicAjoutClient* est raffinée par :

BasicAjoutClient(cl, no, gr) $\widehat{=}$
BEGIN
  ANY num **WHERE** $num \in TT\_Client - ran(Num\_Cl_1)$ **THEN**
    $client_1 := client_1 \cup \{cl\};$
    $Num\_Cl_1 := Num\_Cl_1 \cup \{cl \mapsto num\};$
    $Nom\_Cl_1 := Nom\_Cl_1 \cup \{cl \mapsto no\};$
    $Appartient_1 := Appartient_1 \cup \{cl \mapsto Num\_Gr_1(gr)\}$
  END
END

D.3.4 Raffinement des associations multivaluées

L'association *Emprunt* étant multivaluée, cette dernière est transformée en une classe ayant, en plus de ses propres attributs, deux attributs supplémentaires *Emprunt_Ca* et *Emprunt_Cl* liés par des contraintes référentielles aux clés $Titre\_K7_1$ et Num_Cl_1 des classes *Cassettes* et *Client* respectivement. Le couple formé par ces deux attributs constitue une clé pour la classe ainsi créée. Les attributs $Emprunt\_Ca_1$ et $Emprunt\_Cl_1$ sont définis par :

$$Emprunt\_Cl_1 = Emprunt_1 \lhd (prj_1(client_1, cassette_1);\ Num\_Cl_1) \wedge$$
$$Emprunt\_Ca_1 = Emprunt_1 \lhd (prj_2(client_1, cassette_1);\ Titre\_K7_1) \wedge$$
$$Emprunt\_Cl_1 \otimes Emprunt\_Ca_1 \in Emprunt_1 \rightarrowtail TT\_Client \times STRING$$

Le composant *Systeme_exp_Mig* est raffiné par le composant *Système_exp_Asso*. L'opération *BasicAjoutEmprunt* est raffinée par :

BasicAjoutEmprunt(cl, ca, da,st) $\widehat{=}$
BEGIN
  $Emprunt_1 := Emprunt_1 \cup \{cl \mapsto ca\};$
  $Date\_E_1 := Date\_E_1 \cup \{cl \mapsto ca \mapsto da\};$
  $Emprunt\_Cl_1 := Emprunt\_Cl_1 \cup \{cl \mapsto ca \mapsto Num\_Cl_1(cl)\};$
  $Emprunt\_Ca_1 := Emprunt\_Ca_1 \cup \{cl \mapsto ca \mapsto Titre\_K7_1(ca)\};$
  $Statut_1 := Statut_1 \cup \{cl \mapsto ca \mapsto st\}$
END

Dans ce cas précis, en plus du raffinement de l'opération de base *BasicAjoutEmprunt*, nous devons réécrire l'opération *AjoutEmprunt*. Les appels aux opérations *BasicAjoutClient* et *CreationEmprunt* ne doivent plus être effectués en parallèle mais en séquence. En effet, le raffinement de l'opération *BasicAjoutEmprunt* (utilisée dans l'opération *CreationEmprunt*) nécessite la connaissance des clés des objets ca et cl. Cette opération doit donc être appelée après l'opération *BasicAjoutClient*. Pour cela, l'opération *AjoutEmprunt* est réécrite en (les préconditions sont éliminées) :

AjoutEmprunt(no, gr, ca, da) $\widehat{=}$
BEGIN
  ANY cl **WHERE** $cl \in Client - client_1$ **THEN**
    IF $NbLibre_1(ca) > 0$ **THEN** $BasicChangeNbLibre(ca, NbLibre_1(ca) - 1)$ **END** ||
    BEGIN
      $BasicAjoutClient(cl, no, gr);$
      $CreationEmprunt(cl, ca, da)$
    END
  END
END
END

D.3.5 Passage d'un modèle fonctionnel à un modèle ensembliste

Le but de cette étape est la définition des structures des différentes tables (relations). On rassemble sous le nom d'une variable (traduite en une table), les caractéristiques concernant une même entité. L'application de cette règle, permet la définition des quatre variables suivantes : $T\_Cassette_1$, T_Client_1, $T\_Emprunt_1$, T_Groupe_1. Par exemple, la variable T_Client_1 remplace les trois variables initiales Num_Cl_1, Nom_Cl_1 et $Appartient\_Mig_1$. Ceci est exprimé par l'invariant suivant :

$$T\_Client_1 \in client_1 \rightarrow TT\_Client \times STRING \times NAT \wedge$$
$$T\_Client_1 = Num\_Cl_1 \otimes Nom\_Cl_1 \otimes AppartientMig_1$$

De même, les variables $Titre\_K7_1$ et $NbLibre_1$ sont remplacées par la variable $T\_Cassette_1$ définie par :

$$T\_Cassette_1 \in cassette_1 \rightarrow STRING \times NAT \wedge$$
$$T\_Cassette_1 = Num\_K7_1 \otimes NbLibre_1$$

Afin de faciliter l'écriture des substitutions consultant la valeur des différents attributs, nous exprimons chacun d'eux par une définition. Afin d'éviter le conflit avec les invariants de collage, on donne à ces définition des noms différents de ceux des attributs auxquels elles sont associées.

Le composant *Système_exp_Asso* est raffiné par le composant *Système_exp_Struct* suivant :

REFINEMENT *Systeme_exp_Struct*
REFINES *Systeme_exp_Asso*
VARIABLES
$cassette_1, T\_Cassette_1, client_1, T\_Client_1, Emprunt_1, T\_Emprunt_1, groupe_1, T\_Groupe_1$
INVARIANT
$T\_Cassette_1 \in cassette_1 \rightarrow STRING \times NAT \wedge$
$T\_Cassette_1 = Num\_K7_1 \otimes NbLibre_1 \wedge$

$T\_Groupe_1 \in groupe_1 \rightarrow NAT \times NAT \wedge$
$T\_Groupe_1 = Num\_Gr_1 \otimes Tarif_1 \wedge$

$T\_Client_1 \in client_1 \rightarrow TT\_Client \times STRING \times NAT \wedge$
$T\_Client_1 = Num\_Cl_1 \otimes Nom\_Cl_1 \otimes AppartientMig_1 \wedge$

$T\_Emprunt_1 \in Emprunt_1 \rightarrow TT\_Client \times STRING \times NAT \times STATUT \wedge$
$T\_Emprunt_1 = Emprunt\_Cl_1 \otimes Emprunt\_Ca_1 \otimes Date\_E \otimes Statut_1 \wedge$
INITIALISATION
$cassette_1, T\_Cassette_1, client_1, T\_Client_1, groupe_1, T\_Groupe_1, Emprunt_1, T\_Emprunt_1 :=$
$\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset$
DEFINITIONS
$Num\_Gr_2 \widehat{=} (T\_Groupe;\ prj_1(NAT, NAT));$

$Num\_Cl_2 \widehat{=} (T\_Client;\ prj_1(TT\_Client \times STRING, NAT);\ prj_1(TT\_Client, STRING));$

$NbLibre_2 \widehat{=} (T\_Cassette;\ prj_2(STRING, NAT));$

$Titre\_K7_2 \widehat{=} (T\_Cassette;\ prj_1(STRING, NAT));$

...
BasicChangeNbLibre(ca,nb) $\widehat{=}$
BEGIN
$T\_Cassette_1 := Titre\_K7_2 \otimes (NbLibre_2 \Leftarrow \{ca \mapsto nb\})$
END ;

BasicAjoutClient(cl, no, gr) $\widehat{=}$
BEGIN
ANY num **WHERE** $num \in TT\_Client - ran(Num\_Cl_2)$ **THEN**
$client_1 := client_1 \cup \{cl\};$
$T\_Client_1 := T\_Client_1 \cup \{cl \mapsto (num \mapsto no \mapsto NumGr_2(gr))\}$

END
END;

BasicAjoutEmprunt(cl, ca, da,st) $\widehat{=}$
BEGIN
$Emprunt_1 := Emprunt_1 \cup \{cl \mapsto ca\}$;
$T\_Emprunt_1 := T\_Emprunt_1 \cup \{(cl \mapsto ca) \mapsto (Num\_Cl_2(cl) \mapsto Titre\_K7_2(ca) \mapsto da \mapsto st)\}$
END;

CreationEmprunt(cl, ca, da) $\widehat{=}$
BEGIN
SELECT $NbLibre_2(ca) > 0$ **THEN** $BasicAjoutEmprunt(cl,ca,da,Ok)$
WHEN $NbLibre_2(ca) = 0$ **THEN** $BasicAjoutEmprunt(cl,ca,da,Att)$
END
END

OPERATIONS
AjoutEmprunt(no, gr, ca, da)=
BEGIN
ANY cl **WHERE** $cl : Client - client_1$ **THEN**
IF $NbLibre_2(ca) > 0$ **THEN** $BasicChangeNbLibre(ca,NbLibre_2(ca) - 1)$ **END** $||$
BEGIN
$BasicAjoutClient(cl,no,gr)$;
$CreationEmprunt(cl,ca,da)$
END
END
END
END

D.3.6 Implémentation de la machine *Système_exp*

Le dernier niveau de raffinement obtenu à la phase précédente est implémenté par l'importation d'un ensemble de machines. Pour chaque produit cartésien *T_a* une machine *Rel_a* est importée. Dans notre cas, l'implémentation du raffinement *Système_exp_Struct* importe les machines *Rel_Cassette*, *Rel_Client*, *Rel_Groupe* et *Rel_Emprunt*. Par exemple, les machines *Rel_cassette* et *Rel_Client* sont spécifiées par :

MACHINE $Rel\_Cassette$
VARIABLES
$Table\_Cassette$
INVARIANT
$Table\_Cassette \subseteq struct(Num\_K7_3 : STRING, NbLibre_3 : NAT)$
INITIALISATION
$Table\_Cassette := \emptyset$
OPERATIONS
Update_NbLibre(nu,bb)=
PRE $nu \in STRING \wedge bb \in NAT$ **THEN**
ANY cas **WHERE** $cas \in Table\_Cassette \wedge cas'Num\_K73 = nu$ **THEN**
$Table\_Cassette := Table\_Cassette - \{cas\} \cup \{rec(cas'Num\_K7_3,bb)\}$
END
END ;

nu $\longleftarrow$ **Read_NbLibre(ca)=**
PRE $ca \in STRING$ **THEN**
ANY cas **WHERE** $cas \in Table\_Cassette \wedge cas'Num\_K7_3 = ca$ **THEN**

$nu := cas'NbLibre_3$
END
END ;
END

et

MACHINE Rel_Client
SEES *Contexte* //la machine *Contexte définit l'ensemble abstrait* TT_Client
VARIABLES
$Table\_Client$
INVARIANT
$Table\_Client \subseteq struct(Num\_Cl_3 : TT\_Client, Nom\_Cl_3 : STRING, AppartientMig_3 : NAT)$
INITIALISATION
$Table\_Client := \emptyset$
OPERATIONS
Insert_Client(nu, bb,cc)=
PRE $nu \in TT\_Client \wedge bb \in STRING \wedge cc \in NAT$ **THEN**
$Table\_Client := Table\_Client \cup \{rec(nu,bb,cc)\}$
END ;

//opération renvoyant une nouvelle valeur de l'attribut clé
val $\longleftarrow$ NewValueKey_Client=
BEGIN
$val \in (val \in TT\_Client \wedge not(\exists\, xx.(xx \in Table\_Client \wedge xx'Num\_Cl_3 = val)))$
END
END

Cette étape permet, entre autres, le passage d'un modèle orienté objets (qui est **B**) vers un modèle orienté valeurs (relationnel). Chaque objet doit être confondu à la valeur de son attribut clé. Les ensembles abstraits associés aux classes doivent être valués aux types des clés définies sur ces classes. Dans notre cas, les ensembles abstraits *Cassette*, *Client* et *Groupe* sont valués par :

VALUES
$Cassette = STRING;$
$Client = TT\_Client;$
$Groupe = NAT$

En plus de la valuation des ensembles abstraits, les variables abstraites du dernier niveau de raffinement doivent être exprimées en fonction des variables des machines importées. Ceci est réalisé dans la clause **INVARIANT** du composant *IMPLEMENTATION*. Le composant *Système_exp_Struct* est implémenté comme suit :

IMPLEMENTATION *Systeme_exp_Imp*
REFINES *Systeme_exp_Struct*
IMPORTS
Rel_Client, Rel_Cassette, Rel_Groupe, Rel_Emprunt
VALUES $Client = TT\_Client;\ Cassette = STRING;\ Groupe = NAT$
INVARIANT
$(T\_Client_1;\ prj_1(TT\_Client \times STRING, NAT);\ prj_1(TT\_Client, STRING)) = id(client_1) \wedge$
$\forall (xx,yy,zz).(xx \mapsto (xx \mapsto yy \mapsto zz) \in T\_Client_1 \Rightarrow rec(xx,yy,zz) \in Table\_Client) \wedge$
$\forall (xx,yy,zz).(xx \in TT\_Client \wedge yy \in STRING \wedge zz \in NAT \wedge rec(xx,yy,zz) \in Table\_Client \Rightarrow$
$xx \mapsto (xx \mapsto yy \mapsto zz) \in T\_Client_1) \wedge$

$(T\_Cassette_1;\ prj_1(STRING, NAT)) = id(cassette_1) \wedge$
$\forall (xx,yy).(xx \mapsto (xx \mapsto yy) \in T\_Cassette_1 \Rightarrow rec(xx,yy) \in Table\_Cassette) \wedge$
$\forall (xx,yy).(xx \in NAT \wedge yy \in NAT \wedge rec(xx,yy) \in Table\_Cassette \Rightarrow xx \mapsto (xx \mapsto yy) \in T\_Cassette_1) \wedge$

```
    (T_Groupe1; prj1(NAT,NAT)) = id(groupe1)∧
    ∀(xx,yy).(xx ↦ (xx ↦ yy) ∈ T_Groupe1 ⇒ rec(xx,yy) ∈ Table_Groupe)∧
    ∀(xx,yy).(xx ∈ NAT ∧ yy ∈ NAT ∧ rec(xx,yy) ∈ Table_Groupe ⇒ xx ↦ (xx ↦ yy) ∈ T_Groupe1)∧

  ∀(xx,yy,zz,tt).((xx ↦ yy) ↦ (xx ↦ yy ↦ zz ↦ tt) ∈ T_Emprunt1 ⇒ rec(xx,yy,zz,tt) ∈ Table_Emprunt)
    ∀(xx,yy,zz,tt).(xx ∈ TT_Client ∧ yy ∈ NAT ∧ zz ∈ NAT ∧ tt ∈ STATUT∧
                rec(xx,yy,zz,tt) ∈ Table_Emprunt ⇒ (xx ↦ yy) ↦ (xx ↦ yy ↦ zz ↦ tt) ∈ T_Emprunt1)
DEFINITIONS
BasicChangeNbLibre(ca,nb) ≘
BEGIN
    Update_NbLibre(ca,nb)
END;

BasicAjoutClient(cl, no, gr) ≘
BEGIN
    Insert_Client(cl,no,gr)
END;

BasicAjoutEmprunt(cl, ca, da,st) ≘
BEGIN
    Insert_Emprunt(cl,ca,da,st)
END;

CreationEmprunt(cl, ca, da) ≘
BEGIN
    VAR   nb   IN
      nb ⟵ Read_NbLibre(ca);
      IF   nb > 0   THEN   BasicAjoutEmprunt(cl,ca,da,Ok)
      ELSIF nb = 0 THEN   BasicAjoutEmprunt(cl,ca,da,Att)
      END
    END
END

OPERATIONS
AjoutEmprunt(no, gr, ca, da)=
BEGIN
  VAR   nn,cl   IN
    cl ⟵ NewValueKey_Client;
    nn ⟵ Read_NbLibre(ca);
    BasicAjoutClient(cl,no,gr);
    CreationEmprunt(cl,ca,da);
    IF   nn ≤ 1   THEN
      BasicChangeNbLibre(ca,nn − 1)
    END
  END
END
END
```

Comme on peut le remarquer, l'implémentation de l'opération *AjoutEmprunt* est structurellement assez différente de son abstraction. Cet exemple (choisi volontairement) montre que l'implémentation des opérations de la machine *Système* nécessite la prise en compte des variables modifiées et lues de chaque substitution. En effet, un raffinement naïf de cette opération serait :

```
AjoutEmprunt(no, gr, ca, da)=
BEGIN
  VAR  nn,cl  IN
    cl ⟵ NewValueKey_Client;
    nn ⟵ Read_NbLibre(ca);
    IF  nn ≤ 1  THEN
      BasicChangeNbLibre(ca,nn − 1)
    END;
    BasicAjoutClient(cl,no,gr);
    CreationEmprunt(cl,ca,da)
  END
END
END
```

générant 36 obligations de preuve dont une seule est fausse. L'obligation de preuve erronée concerne le cas où le nombre de cassettes disponibles n, avant l'exécution de cette opération, est égal à 1. L'abstraction de cette opération exécute simultanément les opérations de création de l'emprunt et de mise à jour du nombre de cassettes disponibles. Donc, les deux opérations sont exécutées pour $n = 1$, et l'emprunt est enregistré avec un statut 'Ok'. En revanche, l'implémentation de cette même opération exécute ces opérations en séquence. L'opération de création d'un emprunt est donc exécutée avec $n = 0$ car ce nombre a été préalablement décrémenté par l'opération de mise à jour de *NbLibre*. Par conséquent, l'emprunt est enregistré avec un statut 'Att'. Cette implémentation ne vérifiant pas son abstraction est donc incorrecte.

De manière générale, une solution possible à ce problème est d'exécuter toutes les substitutions lisant une variable avant celles qui la modifient. Dans le cas où des variables lues par une substitution T_1 sont modifiées par une substitution T_2, et réciproquement, des variables lues par la substitution T_2 sont modifiées par la substitution T_1, l'utilisation des variables intermédiaires devient indispensable.

D.3.7 Implémentation de la machine *RobusteSystème*

L'implémentation de la machine *RobusteSystème* nécessite la spécification d'opérations additionnelles dans la machine *Système*. Ces opérations permettent l'évaluation des prédicats des substitutions **IF**. Dans notre cas, la machine *RobusteSystème* est implémentée comme suit :

```
IMPLEMENTATION  RobusteSystèmeImp
REFINES  RobusteSystème
IMPORTS  Système_exp
INVARIANT
//prédicat d'égalité de chaque variable v de la machine Système et de sa renommée v1 de la machine Système_exp
    cassette1 = cassette∧
    Titre_K71 = Titre_K7∧
    NbLibre1 = NbLibre∧
        ...
OPERATIONS
resultat ⟵ RobusteAjoutEmprunt(no, gr, ca, da)=
BEGIN
  VAR  nbmaxclient_Attteint IN
    nbmaxclient_Attteint ⟵ Test_NbMaxClient;
      IF  nbmaxclient_Attteint = FALSE  THEN
        VAR  existe_gr IN
          existe_gr ⟵ Test_Existe_Gr(gr)
          IF  existe_gr = TRUE  THEN
            VAR  existe_ca IN
              existe_ca ⟵ Test_Existe_Ca(ca)
              IF  existe_ca = TRUE  THEN
```

$AjoutEmprunt(no,gr,ca,da)$;
resultat:="succes"
ELSE
resultat:="CassetteInexistante"
END
END
ELSE
resultat:="GroupeInexistant"
END
END
ELSE *resultat:="NombreMaximalClientAtteint"*
END
END
END
END

avec $Test\_NbMaxClient$, $Test\_Existe\_Gr$ et $Test\_Existe\_Ca$ désignant des opérations testant la possibilité de création d'un nouveau client, l'existence d'un groupe et d'une cassette respectivement. Ces opérations sont ajoutées à la machine expansée. Elles ne nécessitent aucune réécriture lors des différentes étapes de raffinement et sont implémentées par des opérations similaires spécifiées dans les machines importées. Par exemple, l'opération $Test\_Existe\_Ca$ spécifiée par :
res $\longleftarrow$ Test_Existe_Ca(ca)=
PRE
$ca \in Cassette$
THEN
$res := bool(ca \in cassette)$
END
est implémentée par l'appel à l'opération *Existe_Ca* spécifiée, dans la machine importée *Rel_Cassette*, par :
nu $\longleftarrow$ Existe_Ca(ca)=
PRE $ca \in STRING$ **THEN**
$nu := bool(\exists\, cas.(cas \in Table\_Cassette \wedge cas'NumK7_3 = ca))$
END

D.4 Preuve

Afin de valider les différentes étapes du développement de ce cas d'étude, un ensemble d'obligations de preuve a été établi. Ces obligations de preuve concernent aussi bien la spécification abstraite initiale que les différentes étapes de raffinement. Nous avons utilisé pour cela le prouveur de l'AtelierB version 3.5. Le tableau D.5 donne un bilan général de cette phase de preuve, où :
- *SpecAbs* : Spécification abstraite,
- *Obv* : nombre d'obligations de preuve triviales,
- *nPO* : nombre d'obligations de preuve non triviales,
- *%PrAuto* : pourcentage d'obligations de preuve automatiquement déchargées,
- *nUn* : nombre d'obligations de preuve restant à prouver interactivement.

Comme on peut le remarquer, certaines règles de raffinement ont généré un nombre important d'obligations de preuve. Néanmoins, certaines d'entre elles sont identiques. Pour chaque partie (formule logique) de l'invariant, selon le nombre de cassettes disponibles (supérieur à 0 ou égal à 0) le prouveur de l'AtelierB génère une obligation de preuve. Or certaines d'entre elles ne dépendent pas du nombre de cassettes disponibles, et donc se prouvent exactement de la même manière. Pour cela, nous avons utilisé la commande *te* (*Try EveryWhere*) qui permet de rejouer le script d'une obligation de preuve sur une autre obligation.

Les preuves de raffinement générées pour la correction des différents composants peuvent être classées en deux catégories. La première catégorie concerne la satisfaction des condition nécessaires au raffinement des opérateurs de substitutions. Par exemple, pour le raffinement du **IF**, nous devons prouver que les deux prédicats sont équivalents. Ces preuves

	Composant	**Obv**	**nPO**	**nUn**	**%PrAuto**	**%Pr**
S	BasicCassette	5	6	0	100	100
p	BasicClient	3	7	0	100	100
e	BasicEmprunt	1	9	2	77	100
A	ÉtatEmprunt	18	0	0	100	100
b	Système	11	24	2	91	100
s	RobusteSystème	15	0	0	100	100
R	Système_exp	96	45	6	86	100
a	Système_exp_C	70	21	4	80	100
ff	Système_exp_Mig	57	19	5	73	100
i	Système_exp_Asso	76	29	12	58	100
n	Système_exp_Struct	79	55	21	61	100
e	Rel_Cassette	8	5	0	100	100
m	Rel_Client	11	1	0	100	100
e	Rel_Emprunt	10	1	0	100	100
n	Système_expImp	89	75	30	60	100
t	RobusteSystèmeImp	88	45	3	93	100
	Totaux	**637**	**342**	**83**	**86**	**100**

FIG. D.5 – *Tableau récapitulatif de la phase de preuve*

sont triviales quand les variables apparaissant dans ces prédicats ne sont pas raffinées. La complexité de ces preuves est proportionnelle à la complexité de l'invariant de collage relatif à ces variables. La deuxième catégorie concernent la correction des raffinement des opérations de base. Il serait donc intéressant de définir une base de preuves contenant la preuve générique de chaque règle de raffinement, puis de l'instancier par les données effectives de l'application pour obtenir des preuves de base qui seront donc utilisées lors de l'établissement des preuves de la seconde catégorie.

D.5 Traduction en code JAVA

Dans cette section est donné le code des différentes classes **JAVA** permettant l'implémentation de notre application. Quatre types de classes sont à distinguer :

- les classes implémentant les machines importées représentant les tables.
- les classes implémentant les opérations générées des diagrammes états/transitions.
- la classe implémentant la machine *Système_exp*.
- la classe implémentant la machine *RobusteSystème*

On commence par donner le schéma de la base de données généré puis le code des classes **JAVA**.

D.5.1 Structure des tables relationnelles

Chaque machine importée définie une table relationnelle. L'exemple du club vidéo définit donc quatre tables dont le script de création est le suivant :

CREATE TABLE $Cassette($
$Titre\_K7$ **CHAR(40) PRIMARY KEY**,
$NbLibre$ **INT NOT NULL**);

CREATE TABLE $Groupe($
Num_Gr **INT PRIMARY KEY**,
$Tarif$ **INT NOT NULL**);

CREATE TABLE $Client($
Num_Cl **INT PRIMARY KEY**,// TT_Client est traduit par le type **INT**
Nom_Cl **CHAR(40) NOT NULL**,
$Appartient$ **INT NOT NULL REFERENCES** $Groupe$);

CREATE TABLE $Emprunt($
$Emprunt\_Cl$ **INT REFERENCES** $Client$,
$Emprunt\_Ca$ **CHAR(40) REFERENCES** $Cassette$,
$Date\_E$ **INT NOT NULL**,
$Statut$ **CHAR(3) NOT NULL check** ($Statut$ **IN** ($'Ok'$,$'Att'$)),
PRIMARY KEY ($Emprunt\_Cl$,$Emprunt\_Ca$));

L'opération de génération de la valeur de la clé spécifiée dans la machine *Rel_Client* est implémentée par la procédure stockée suivante :

```
CREATE FUNCTION  NewValueKey_Client()  RETURNS INT  AS'

 DECLARE j INT ;
 DECLARE k INT;

BEGIN
 SELECT count(*) INTO k FROM Client ;

 //tester s'il y a des tuples déjà existants
 IF k=0 THEN
   RETURN 1;
 END IF ;

 SELECT  max(Num_Cl) INTO j FROM Client ;
 FOR  i IN 1..j  LOOP
  SELECT Num_Cl INTO k  FROM Client WHERE Num_Cl=i;
  IF NOT FOUND THEN
    RETURN i;
  END IF ;
 END LOOP;

 IF FOUND THEN
  RETURN j+1;
 END IF;
END;
'LANGUAGE 'plpgsql';
```

D.5.2 Classes JAVA

Chaque machine importée est traduite par une classe **JAVA** dont les méthodes correspondent aux opérations de la machine considérée. Les machines importées *Rel_Cassette*, *Rel_Groupe*, *Rel_Client* et *Rel_Emprunt* sont traduites par les classes **JAVA** suivantes :

1. *La classe Cassette*

```
public class  Cassette{
   private PreparedStatement stmtUpdateNbLibre;
   private PreparedStatement stmtReadNbLibre;
```

```
    private PreparedStatement stmtExisteCa;
    private Connection conn;

   /*Creation d'une instance*/
   public Cassette(Connection conn) throws SQLException {
    this.conn = conn;
    stmtUpdateNbLibre =
     conn.prepareStatement("update cassette set NbLibre=? WHERE Titre_K7=?");
    stmtReadNbLibre =
     conn.prepareStatement("select NbLibre from Cassette where Titre_K7= ?");
    stmtExisteCa =
     conn.prepareStatement("select count(*) from Cassette where Titre_K7 = ?");

 }

   /*Retourner la connexion associée*/
   public Connection getConnection(){
      return conn;
   }

    /*Mise à jour de l'attribut NbLibre*/
    public void Update_NbLibre (String nu, int bb)
     throws SQLException
      {
       stmtUpdateNbLibre.setString(2,nu);
       stmtUpdateNbLibre.setInt(1,bb);
       stmtUpdateNbLibre.executeUpdate();
      }

    /*Lecture du nombre de cassettes disponibles*/
    public int ReadNbLibre(String nu)
     throws SQLException
      {
       stmtReadNbLibre.setString(1,nu);
       ResultSet rst=stmtReadNbLibre.executeQuery() ;
       if (rst.next())
         return (rst.getInt(1));
       else return 0;
      }

    /*Test d'existance d'une cassette*/
    public int Existe_Ca(String nu)
     throws SQLException
      {
       stmtExisteCa.setString(1,nu);
       ResultSet rst=stmtExisteCa.executeQuery();
       if (rst.next())
         return (rst.getInt(1));
       else return 0;
      }
   }
```

2. *La classe Client :* on suppose que la capacité du club est de 50 clients.

```
public class  Client{
```

```
 private PreparedStatement stmtInsert;
 private PreparedStatement stmtNewValueKey;
 private PreparedStatement stmtFullClient ;
 private Connection conn;

/*Creation d'une instance*/
public Client(Connection conn) throws SQLException {
 this.conn = conn;
 stmtInsert =
  conn.prepareStatement("Insert into client values (?,?,?)");
 stmtNewValueKey =
  conn.prepareStatement ("select NewValueKey_Client();");
 stmtFullClient =
  conn.prepareStatement ("select count(*) from client;");
}

/*Retourner la connexion associée*/
public Connection getConnection() {
  return conn;
}

/*Inserer un nouveau client*/
public void Insert_Client(int nu, String bb, int cc)
 throws SQLException
  {
   stmtInsert.setInt(1,nu);
   stmtInsert.setString(2,bb);
   stmtInsert.setInt(3,cc);
   stmtInsert.executeUpdate();
  }

/*Générer une nouvelle valeur de la clé pour un client à créér*/
public int NewValueKey_Client()
 throws SQLException
  {
   ResultSet rst= stmtNewValueKey.executeQuery();
   if (rst.next())
       return rst.getInt(1);
   else return 0;
  }

/*Tester la possibilité de création d'un nouveau client*/
  public int Full_Client()
   throws SQLException
    {
     ResultSet rst= stmtFullClient.executeQuery();
     if (rst.next() & rst.getInt(1)<49)
        return 1;
     else return 0;
    }
 }
```

3. *La classe Emprunt*

```
public class  Emprunt{
```

```
    private PreparedStatement stmtInsert;
    private Connection conn;

   /*Creation d'une instance*/
   public Emprunt(Connection conn) throws SQLException {
    this.conn = conn;
     stmtInsert =
      conn.prepareStatement("insert into emprunt values (?,?,?,?)");
    }

   /*Retourner la connexion associée*/
   public Connection getConnection() {
    return conn;
   }

   /*Insertion d'un emprunt*/
   public void Insert_Emprunt(int cl,String ca, int da, String st)
    throws SQLException
     {
      stmtInsert.setInt(1,cl);
      stmtInsert.setString(2,ca);
      stmtInsert.setInt(3,da);
      stmtInsert.setString(4,st);
      stmtInsert.executeUpdate();
     }
}
```

4. *La classe EtatEmprunt*

```
public class EtatEmprunt {
   private Emprunt emprunt;
   private Cassette cassette;
   Connection conn;

  /*Creation d'une instance*/
  public EtatEmprunt(Cassette cassette, Emprunt emprunt) {
   if (cassette.getConnection() != emprunt.getConnection())
      throw new AppException
         ("Les instances n'utilisent pas la même connexion au serveur");
      this.conn = cassette.getConnection();
      this.cassette= cassette;
      this.emprunt = emprunt;
  }
  /*Création d'un emprunt*/
  public void CreationEmprunt(int cl, String ca, int da)
   throws SQLException
    {
        int nn=cassette.ReadNbLibre(ca);
        if (nn>0)
         emprunt.Insert_Emprunt(cl,ca,da,"Ok");
        else  emprunt.Insert_Emprunt(cl,ca,da,"Att");
     }
  }
```

5. *La classe Système*

```
public class Systeme {
```

```
    private GestionEmprunt gestionemprunt;
    private Emprunt emprunt;
    private Client client;
    private Cassette cassette;
    private Groupe groupe;
    Connection conn;

  /*Creation d'une instance*/
  public Systeme(Emprunt emprunt, Client client,
          GestionEmprunt gestionemprunt, Cassette cassette,Groupe groupe) {
 if (client.getConnection() != emprunt.getConnection()||
     cassette.getConnection() != client.getConnection() ||
     cassette.getConnection() != groupe.getConnection() )
 throw new AppException
           ("Les instances n'utilisent pas la même connexion au serveur");
  this.conn = client.getConnection();
  this.client = client;
  this.emprunt = emprunt;
  this.gestionemprunt=gestionemprunt;
  this.cassette=cassette;
  this.groupe=groupe;
  }

  /*Ajout d'un emprunt*/
  public void AjoutEmprunt(String no, int gr, String ca, int da)
     throws SQLException
      {

         int cl=client.NewValueKey_Client();
         int nn= cassette.ReadNbLibre(ca);
         client.Insert_Client(cl,no,gr);
         gestionemprunt.CreationEmprunt(cl,ca,da);
         if (nn >= 1)
          cassette.Update_NbLibre(ca,nn-1);
      }

  /*Test d'existance d'une cassette*/
  public int Test_Existe_Ca(String ca)
     throws SQLException
      {
       return (cassette.Existe_Ca(ca));
      }
   /*Test d'existance d'un groupe*/
   public int Test_Existe_Gr(int gr)
      throws SQLException
     {
      return (groupe.Existe_Gr(gr));
     }
  /*Test du nombre de clients existants*/
   public int Test_NbMaxClient()
      throws SQLException
     {
      return (client.Full_Client());
```

```
        }
    }
```

6. *RobusteSysteme*

```
public class RobusteSysteme {
    private GestionEmprunt gestionemprunt;
    private Emprunt emprunt;
    private Cassette cassette;
    private Groupe groupe;
    private Client client;
    private Systeme systeme;
    Connection conn;

  /*Creation d'une instance*/
  public RobusteSysteme(GestionEmprunt gestionemprunt, Emprunt emprunt,
         Cassette cassette,Groupe groupe,Client client, Systeme systeme) {

   if (client.getConnection() != emprunt.getConnection()||
       cassette.getConnection() != client.getConnection() ||
       cassette.getConnection() != groupe.getConnection())
   throw new AppException
("Les instances n'utilisent pas la même connexion au serveur");
   this.conn = client.getConnection();
   this.client = client;
   this.emprunt = emprunt;
   this.gestionemprunt=gestionemprunt;
   this.cassette=cassette;
   this.groupe=groupe;
   this.systeme=systeme;
  }

  /*Opération d'ajout d'un emprunt*/
  public String RobusteAjoutEmprunt(String no, int gr, String ca, int da)
   throws SQLException, AppException{
     try {
          int full_cl=systeme.NbMaxClient();
          if (full_cl!=0)
            {int existe_gr=systeme.Test_Existe_Gr(gr);
             if (existe_gr!=0)
                {int existe_ca=systeme.Test_Existe_Ca(ca);
                 if (existe_ca!=0)
                     {systeme.AjoutEmprunt(no, gr, ca, da);
                      conn.commit();
                      return "succes";
                     }
                 else
                     return ("CasseInext");
                }
            else return ("GroupeInext");
           }
       else return ("CapacitéClientDépasée");
      }
      catch (SQLException e)
           {
```

```
                    conn.rollback();
                    throw new AppException("La transaction n'a pas été traitée");
                   }
            }
     }
```

Les erreurs d'exécution des ordres SQL lèvent des exceptions de type SQLException. Quand celles-ci sont levées, le bloc catch est exécuté. On annule les différentes éventuelles modifications effectuées sur les données de la base et on signale à l'utilisateur que la transaction a échoué.

La génération du code des différentes classes **JAVA** étant réalisée en dehors de tout système formel, sa correction ne peut être établie de manière sûre. À travers cet exemple, on remarque qu'une grande partie des classes **JAVA** correspondant aux machines de base peuvent être générées automatiquement car celles-ci ont une structure standard. Le code donné dans cette section ne concerne que la partie traitement proprement dit de la transaction. Les détails annexes comme la gestion de l'exécution simultanée de plusieurs transactions n'est pas considérée. Ils sont traités par le gestionnaire de connexions à la base de données.

Bibliographie

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Press Syndicate of the University of Combridge, 1996.

[2] P. Anderson. Representing Proof Transformations for Program Optimization. In *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 575–589. Springer-Verlag, June 1994.

[3] B-CORE. *B-Toolkit Release 3.2. Manual*. Oxford, U.K, 1996.

[4] B. Barras and all. *The Coq Proof Assistant, Reference Manual (7.1)*. INRIA Rocquencourt, 2001.

[5] R. S. M. Barros. Deriving Relational Database Programs from Formal Specifications. In M. Naftalin, T. Denvir, and M. Bertran, editors, *Proceedings of FME'94: Industrial Benefit of Formal Methods*, pages 703–723, Berlin, Heidelberg, 1994. Springer.

[6] R.S.M. Barros. *On the Formal Specification and Derivation of Relational Database Applications*. PhD thesis, Department of Computing Science, The University of Glasgow, November, 1994.

[7] G. Barthe and O. Pons. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In F. Honsell and M. Miculan, editor, *"Proceedings of FOSSACS'01"*, volume 2030 of *LNCS*, pages 57–71, 2001.

[8] P. Behm, P. Benoit, A. Faivre, and J.M. Meynadier. METEOR : A Successful Application of B in a Large Project. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 369–387, 1999.

[9] M. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.

[10] G. Booch. *Object Oriented Design with Applications*. Benjamin Cummings, Redwood City, 1991.

[11] A. Borgida, J. Mylopoulos, and J.W. Schmidt. *Final Version on TDL Design, DAIDA delivrable*, 1987.

[12] J-M. Bruel and R. B. France. A Formal Object-Oriented CASE Tool for the Development of Complex Systems. In *Proceedings of the 7th European Workshop on Next Generation of Case Tools (NGCT'96)*, May 1996.

[13] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[14] R. Curien. *Outils pour la Preuve par Analogie*. PhD thesis, Université de Nancy, 1990.

[15] D. Delahaye, R. Di Cosmo, and B. Werner. Recherche dans une Bibliothèque de Preuves Coq en Utilisant le Type et Modulo Isomorphismes. In *Proceedings of PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, November 1997.

[16] DIGILOG groupe STERIA. *Atelier B, Manuel de Référence du langage B*. BP 16000, 13791 Aix-en-Provence Cedex 3 France, 1998.

[17] S. Dupuy. *Couplage de Notations Semi-Formelles et Formelles pour la Spécification des Systèmes d'Information*. PhD thesis, Université Joseph Fourier, September 2000.

[18] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An Overview of Roz: a Tool for Integrating UML and Z Specifications. In *Proceedings of 12th Conference on Advanced information Systems Engineering-CAiSE'2000*, volume 1789. Lecture Notes in Computer Science, Springer-Verlag, June 2000.

[19] D. Edmond. *Information Modeling*. Prentice Hall, 1992.

[20] D. Edmond. Refining Database Systems. In J.P. Bowen and M.G. Hinchey, editor, *ZUM'95: The Z Formal Specification Notation*, pages 25–44. Springer Verlag Lecture Notes in Computer Science, September 1995.

[21] T. Ellman. Explanation-Based Learning: A Survey of Programs and Perspectives. *ACM Computing Survey*, 21(2):163–221, 1989.

[22] P. Facon, R. Laleau, and A. Mammar. Combining UML with the B Formal Method for the Specification of Database Applications. Technical Report 87, CEDRIC Laboratory, Paris, September 1999.

[23] P. Facon, R. Laleau, A. Mammar, and F. Polack. Formal Specification of the UML Metamodel for Building Rigorous Caise Tools. Technical Report 91, CEDRIC Laboratory, Paris, September 1999.

[24] A. Felty and D. Howe. Generalization and Reuse of Tactic Proofs. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 1–15, Kiev, Ukraine, 1994. Springer-Verlag LNAI 822.

[25] B. Fraikin. Preuve de Raffinement Automatique dans la Methode B. DEA de logique et fondements informatiques, Univ. Denis Diderot, September 1999.

[26] M. Frappier and R. St-Denis. Specifying Information Systems Using Input-Output Traces. In *OOPSLA Workshop on Behavioral Semantics of OO Business and System Specifications*, 1998.

[27] B. Ginoux. *DESCARTES en 32 Questions, 32 Réponses et 32 Pages*, 1996. Note Technique, EDF/DER HI29/96/036.

[28] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 2000.

[29] T. Gunther, Schewe K. D., and I. Wetzel. On the Derivation of Executable Database Programs from Formal Specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe, Odense, Denmark*, volume 670, pages 351–366. Springer-Verlag, New York, NY, 1993.

[30] H. Habrias. *Spécification Formelle avec B*. Hermes/Lavoisier, 2001.

[31] R. P. Hall. Computational Approaches to Analogical Reasonning: A Comparative Analysis. *Artificial Intelligence*, 39:39–120, 1989.

[32] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[33] C. A. R. Hoar. An Axiomaric Bases For Computer Programming. *Communications of ACM*, 12(10):576–580, 1969.

[34] I. Jacobson. *Object Oriented software engineering*. Addison-Wesley, 1992.

[35] M. Jamnik. Analogy and Automated Reasoning. Technical Report CSRP-99-14, School of Computer Science, 1999.

[36] M. Jaume. *Contribution à la Sémantique de la Programmation Logique*. PhD thesis, Ecole Nationale des Ponts et Chaussées, 1999.

[37] S. Jean, D. Donsez, and S. Lecomte. Utilisation de Données pour la Flexibilité de Services Coopérant dans la Carte à Microprocessuer. In *XVII congrès INFORSID*, May 2000.

[38] S-K. Kim and D. Carrington. Formalizing the UML Class Diagram Using Object-Z. In *Int. Conference on the Unified Modeling Language UML99, LNCS 1723*, 1999.

[39] T. Kolbe and J. Brauburger. Plagiator - A Learning Prover. In *Conference on Automated Deduction*, pages 256–259, 1997.

[40] T. Kolbe and C. Walther. Reusing Proofs. In *European Conference on Artificial Intelligence*, pages 80–84, 1994.

[41] T. Kolbe and C. Walther. Proof Analysis, Generalization and Reuse. In W. Bibel and P.H Schmidt, editor, *Automated Deduction: A Basis for Applications*, volume II, pages 189–219. Kluwer Academic Publishers, 1998.

[42] R. Laleau and A. Mammar. Using a Formal Refinement to Derive Relational Database Implementations from B Specifications . Technical Report 86, CEDRIC laboratory, 2000. Extended version of the ZB2000 paper.

[43] P. Madden. The Specialization and Transformation of Constructive Existence Proofs. In *Proceeding of the Eleventh International Joint Conference on Artificial Inteligence*, pages 131–148. Morgan Kaufmann, 1989.

[44] N. Magaud and Y. Bertot. Changement de Représentation des Données dans le Calcul des Constructions Inductives. Technical Report 4039, INRIA, October 2000.

[45] N. Magaud and Y. Bertot. Changement de Représentation des Structures de Données en Coq: le cas des entiers naturels. In *Proceedings of JFLA'2001*, 2001.

[46] Z. Manna and R. Waldinger. Special Relations in Automated Deduction. *ACM*, 33(1), January 1986.

[47] Z. Manna and R. Waldinger. Fundamentals of Deductive Program Synthesis. In *Combinatorial Algorithms on Words, NATO ISI*. Springer-Verlag, 1991.

[48] R. Marcano and N. Lévy. Transformation d'Annotations OCL en Expressions B. In *Proceedings of "Approches Formelles dans l'Assistance au Développement de Logiciels" (AFADL'2001)*, June 2001.

[49] E. Melis and J. Whittle. External Analogy in Inductive Theorem Proving. In *Proceedings of 21st German Conference on Artificial Intelligenc, KI97*, 2001.

[50] J. Melton and A. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993.

[51] E. Meyer and J. Souquieres. A Systematic Approach to Transform OMT Diagrams to a B Specification. In *FM99*, volume 1708, pages 875–895, 1999.

[52] T.M. Mitchell, R.M. Keller, and S. T. Kedar-Cabelli. Explanation-based Generalization: A Unifiying Views. *Machine Learning*, 1:47–80, 1986.

[53] H. P. Nguyen. *Dérivation de Spécifications Formelles B à Partir de Spécifications Semi-formelles*. PhD thesis, CEDRIC Laboratory, December 1998.

[54] OMG. *Unified Modeling Language Specification, BetaR1 Release*. "Object Management Group., April 1999.

[55] S. Owen. *Analogy for Automated Reasoning*. Academic Press, 1990.

[56] O. Pons. Generalization in Type Theory Based Proof Assistants. In *Proceedings of TYPES'00*, December 2000.

[57] X. Qian. The Deductive Synthesis of Database Transactions. *ACM Transactions on Database Systems*, 18(4):626–677, 1993.

[58] T. Quatrany. *Visual Modeling with Rational Rose 2000 and UML.* Addison Wesley Professional, 1999.

[59] C. Rich and R.C. Waters. *The Programmer's Apprentice*. ACM Press and Addison Wesley, 1990.

[60] J.D.C. Richardson. Automating Changes of Data Type in Functional Programs. In *Proceedings of the 10th Conference on Knowledge-Based Software Engineering (KBSE)*, November 1995.

[61] M. Rittri. Using Types as Search Keys in Function Libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[62] E. R. Rollins and J.M. Wing. Specifications as Search Keys for Software Libraries. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 173–187, Paris, France, 1991. The MIT Press.

[63] J. Rouyer. Développement de l'Algorithme d'Unification dans le Calcul des Constructions avec Types Inductifs. Technical Report 1795, INRIA-Lorraine, October 1992.

[64] J. Rumbaugh and all. *Object Oriented Modeling and Design*. Prentice-Hall, 1991.

[65] K. D. Schewe. *Specification of Data-Intensive Application Systems*. PhD thesis, BTU Cottbus, 1994.

[66] K.D. Schewe, J.W. Schmidt, and I. Wetzel. Specification and Refinement in an Integrated Database Application Environment. In *Proc. VDM Conference 1991, Noordwijkerhout*, volume 551. Springer-Verlag, 1991.

[67] K.D. Schewe, I. Wetzel, and J.W. Schmidt. Towards a Structured Specification Language for Data-Intensive Applications. In D. Harper and M. Norrie, editors, *"Proceedings of Specification of Database Systems" Workshops in Computer Science*. Springer, 1992.

[68] J.W. Schmidt, H. Eckhardt, and F. Matthes. *DBPL Report, DBPL Memo*, 1988.

[69] D. R. Smith. KIDS : A Semi-Automatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[70] C. Snook and M. Butler. Using a Graphical Design Tool for Formal Specification. In *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG02)*, 2002.

[71] Y. Srinivas and J. McDonald. The Architecture of Specware, a Formal Software Development System, 1996.

[72] Y.V. Srinivas and R. Jullig. "Specware(TM): Formal Support for Composing Software". In , editor, *the Conference on Mathematics of Program Construction.* , July 1995.

[73] B. Thalheim. Extending the Entity-Relationship Model for a High-Level. *Spring Verlag LNC 504 Theory-Based Database Design. East/West Database*, pages 161–184, 1990.

[74] A. Villafiorita and R. Sebastiani. Proof Planning by Abstraction. In *ECAI-94 workshop on interactive systems*, 1994.

[75] C. Walther and T. Kolbe. Proving Theorems by Reuse. *Artificial Intelligence*, 116(1-2):17–66, 2000.

[76] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling with Uml* . Addison-Wesley Object Technology Series, 1999.

[77] I. Wetzel, K.D. Schewe, and J. W. Schmidt. Specification and Refinement of Databases. In M. Jarke, editor, *Database Application Engineering with DAIDA*, volume 1, pages 283–318. Springer-Verlag, 1993.

[78] J. Whittle. Analogy in *CLAM*. Master's thesis, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1995.