Couplage de notations semi-formelles et formelles pour la spécification des systèmes d'information

Sophie Dupuy

Université Joseph Fourier - Grenoble I

Thèse

pour obtenir le grade de

Docteur de l'Université Joseph Fourier

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Discipline : Informatique

présentée par

Sophie DUPUY

épouse CHESSA

le 22 Septembre 2000

Couplage de notations semi-formelles et formelles pour la spécification des systèmes d'information

Composition du jury

Président :	M.-F. Bruandet, CLIPS-IMAG, Université Grenoble I
Rapporteurs :	H. Habrias, IRIN, Université de Nantes
	M. Léonard, CUI, Université de Genève
Examinateur :	A. Finkelstein, University College London
Directeurs de thèse :	Y. Ledru, LSR-IMAG, Université Grenoble I
	M. Chabre-Peccoud, LSR-IMAG, Université Grenoble I

Thèse préparée au sein du laboratoire Logiciels, Systèmes et Réseaux, IMAG

Remerciements

Je voudrais exprimer ma profonde gratitude à Yves Ledru et Monique Chabre-Peccoud pour avoir su diriger cette thèse par leurs nombreux conseils et leur aide tout en me laissant une grande liberté dans le travail de recherche. Leur compétence et leurs encouragements m'ont été des plus précieux. Mes plus sincères remerciements.

Je tiens ensuite à remercier Marie-France Bruandet pour l'intérêt qu'elle a témoigné à ma thèse. Son avis "extérieur" et ses encouragements m'ont permis de surmonter certaines des difficultés rencontrées. Enfin merci pour sa participation au jury de cette thèse.

Je voudrais également remercier Henri Habrias et Michel Léonard pour avoir accepté de juger ce travail. Leurs commentaires constructifs ont amplement contribué à son amélioration et à celle de ce manuscrit.

Ma reconnaissance va à Anthony Finkelstein pour m'avoir si bien accueilli au sein de son équipe à Londres et avoir accepté de faire partie du jury de ce travail.

Je remercie aussi vivement les membres du laboratoire LSR et de l'équipe de Génie Logiciel de Londres. Tous m'ont permis d'une manière ou d'une autre de travailler dans la joie et la bonne humeur. En particulier un grand merci à Marlon, Stéphane, Héléna, Andrea, Sofia et les autres... Enfin j'ai une pensée particulière à Lydie pour son écoute et son soutien au cours de cette thèse. N'oublions pas non plus toute l'équipe administrative du LSR pour son soutien logistique sans défaut.

Merci également à Laurent Trilling pour m'avoir incité à essayer de faire de la recherche, à Catherine Parent pour son aide dans le travail de preuve, à mes étudiants pour m'avoir appris à formuler et reformuler et re-reformuler...

Finalement sans pouvoir exprimer avec des mots mes sentiments envers eux, je voudrais dédier ce travail à ma famille : mes parents pour tellement de choses dont je ne saurais donner la liste, Guillaume pour m'avoir toujours montré les possibilités d'un esprit curieux et m'avoir appris à persévérer, Régis pour sa (très grande !) patience, sa compréhension et son aide dans la vie de tous les jours et enfin Léa pour avoir embelli les derniers mois de ma thèse. Avec mes amis les plus proches, ils sont la source du bonheur sans laquelle ce travail n'aurait pas été possible. Qu'ils trouvent dans cette thèse l'aboutissement de leur soutien durant toutes mes années d'études.

Table des matières

Index

Chapitre 1

Introduction

1.1 Contexte et motivations

Les systèmes d'information (SI) deviennent de plus en plus complexes. En effet, les applications telles que la conception assistée par ordinateur ou la cartographie représentent une activité croissante par rapport aux applications de gestion qui évoluent elles aussi vers la manipulation d'objets et de traitements complexes. De plus, les frontières entre des domaines comme la gestion, la bureautique, les systèmes d'aide à la décision s'estompent pour laisser la place à des activités groupées du SI. Ces modifications nécessitent une participation plus importante des utilisateurs dans les différentes phases d'analyse, de conception et de développement. La détermination des besoins devient alors une activité clé dans le développement d'un SI. Les besoins et les problèmes doivent être bien compris et analysés car la majorité des erreurs est introduite pendant la phase d'analyse du système. Ces erreurs sont coûteuses en particulier si elles sont détectées tard. Or les coûts de développement sont déjà élevés et ceux d'évolution ou de maintenance peuvent être supérieurs. Les enjeux et les risques sont donc considérables. Pour diminuer les coûts et limiter les risques, la nécessité de rationaliser la conception et le développement des SI s'impose. Il est donc nécessaire de disposer de méthodes et d'outils.

Les méthodes utilisées pour l'analyse et la conception des SI peuvent être classées en trois catégories suivant le niveau de formalisation de leurs notations [FKV94]. Ainsi on distingue les méthodes informelles, les semi-formelles et les formelles.

Les méthodes informelles correspondent à l'usage de la langue naturelle (contrôlée ou non) pour la spécification des systèmes. Elles sont très souvent employées pour leur simplicité d'utilisation et leur potentiel de communication avec les utilisateurs. Néanmoins leur manque de précision est tel qu'il augmente le risque d'une mauvaise compréhension des besoins ou du système.

Les méthodes généralement utilisées pour l'analyse et la conception des SI sont des méthodes semi-formelles à base de modèles graphiques (diagrammes entité/relation, diagrammes de flux de données...). Ces modèles ont pour avantage de représenter le système sous forme de graphiques qui permettent une vue synthétique du SI. Leur aspect intuitif permet aussi une communication plus aisée avec les utilisateurs. Mais elles manquent parfois de précision. En effet, leurs notations sont

ambiguës si bien que l'interprétation d'un modèle est difficile. Cela rend aussi difficile la construction d'outils complets comprenant en particulier les aspects validation du modèle, limitant ainsi la confiance dans la fiabilité des systèmes contruits.

Les méthodes formelles sont basées sur des notations mathématiques. Elles fournissent des notations précises et non-ambiguës qui permettent la construction de preuves et le raffinement vers du code exécutable. Leur utilisation permet d'augmenter la qualité d'un système sans nécessairement augmenter les coûts et les délais. Mais bien que ces méthodes aient montré leur utilisabilité en pratique, elles sont très peu utilisées et leur domaine d'application se limite généralement aux systèmes critiques. Les raisons de cette faible utilisation sont nombreuses [Sai96]. Les méthodes formelles souffrent d'une certaine faiblesse méthodologique. En effet, certaines sont plus des langages que des méthodes proprement dites. Leur formalisme mathématique peut paraître rebutant. Et elles ont souvent été présentées comme une révolution dans le développement des systèmes. Or pour être pratiquée dans l'industrie, une nouvelle technologie doit pouvoir s'intégrer aux pratiques existantes. Craigen et al [CGR95] suggèrent qu'il faut améliorer l'intégration des méthodes formelles avec les autres techniques pratiquées en génie logiciel comme les autres techniques d'assurance ou les méthodes de conception. Elles doivent aussi apporter un support outil qui permette une utilisation plus aisée et faire appel à d'autres technologies pour remplir le rôle de support de communication. Une des approches proposées pour cela consiste à utiliser conjointement les méthodes semi-formelles et formelles.

Tel est l'objet d'étude de ce travail dont la motivation est d'exploiter au mieux la complémentarité des notations semi-formelles et formelles pour améliorer la précision dans le développement d'un système : les notations semi-fromelles doivent permettre de spécifier le système intuitivement fournissant ainsi un bon support de communication avec les utilisateurs alors que les notations formelles apportent précision et concision à la spécification nécessaires à tout raisonnement de vérification. Ce couplage permet aussi de diminuer les inconvénients de chacune d'entre elles. La sémantique des méthodes semi-formelles est définie plus précisément alors que les méthodes formelles deviennent plus facilement accessibles à une audience plus large.

1.2 Objectif et contribution

Dans cette optique, ce travail porte sur le couplage de notations semi-formelles et formelles afin de répondre à la question suivante :

Comment coupler des notations semi-formelles et formelles en cumulant leurs avantages respectifs tout en diminuant leurs inconvénients ?

Pour répondre à cette question, nous avons choisi parmi les différentes approches de couplage de notations, une stratégie de traduction des modèles semi-formels en spécifications formelles. Cette approche nous paraît la démarche la plus naturelle car le système est d'abord décrit graphiquement avant d'être précisé. Elle apporte aussi une aide à la construction de spécifications formelles permettant ainsi de les produire plus rapidement. Elle assure une certaine équivalence sémantique entre les deux types de spécifications qui garantit que tout raisonnement effectué sur l'une des spécifications est valide pour l'autre.

Toutefois, nous avons aussi étudié une autre stratégie de couplage de notations : la vérification de cohérence par méta-modélisation. Sa comparaison avec la traduction doit servir à mieux cerner les implications induites par le choix d'une stratégie de couplage.

L'approche de couplage étant déterminée, il est naturel de choisir les notations semi-formelles et formelles étudiées. Dans le contexte de la modélisation des SI, les spécifications semi-formelles se concentrent prioritairement sur les aspects statiques du système. Nous nous sommes donc intéressée principalement à l'étude d'un modèle objet bien que nous proposions aussi en annexe la traduction d'un modèle dynamique basé sur les diagrammes d'états-transitions de Harel [Har87]. Le langage formel choisi comme cible de la traduction doit donc servir en premier lieu à exprimer les concepts d'un modèle semi-formel objet et à offrir des outils de validation. Notre sélection s'est portée sur deux langages cibles, Z [Spi92] et une de ses extensions orientées-objet, Object-Z [DKRS91], pour leur lisibilité et leur adéquation aux concepts du modèle objet.

Nous avons décrit des règles de traduction du modèle objet en Z et Object-Z avec le but d'obtenir une spécification formelle conservant le plus possible la structure de la spécification semi-formelle et suffisamment claire pour être facilement lisible et exploitable. L'une des difficultés de ce travail repose sur le compromis entre l'expression de la sémantique de la spécification semi-formelle et la simplicité de la spécification formelle produite. Ce problème porte sur les caractéristiques du langage formel cible. Ici l'utilisation de deux langages nous a permis de mieux cerner les implications sur la traduction du choix d'un langage formel et en particulier de répondre à la question : faut-il un langage formel orienté objet pour décrire un modèle objet ?

Pour montrer l'intérêt de la traduction, nous avons cherché à aller plus loin que la simple énumération théorique de ses bénéfices en en développant quelques uns. Les apports généralement cités portent sur la précision sémantique des modèles semi-formels et l'utilisation du potentiel de raisonnement des langages formels ; nous les avons illustrés en spécifiant le sens de trois modèles objet et en étudiant la validation des contraintes d'un modèle objet grâce à la validation des gardes des opérations. De plus, ce travail met en évidence un bénéfice inattendu qui concerne l'aide à l'écriture d'annotations à un modèle objet.

Néanmoins toute approche est bénéfique seulement si elle peut s'effectuer efficacement grâce à des outils qui la supportent. L'étude et le développement d'un atelier de modélisation couplant les notations semi-formelles et formelles sont donc nécessaires à l'adoption de notre démarche de traduction. Dans ce travail, nous proposons un outil qui étend l'environnement Rational Rose en faisant cohabiter les notations objet et Z.

Ce travail a donc pour objectif d'étudier le couplage de notations semi-formelles et formelles par une approche de traduction dont nous tentons de cerner les bénéfices et les limites.

1.3 Organisation de la thèse

Cette thèse est organisée en 8 chapitres qui développent trois grands thèmes : les deux premiers chapitres présentent la multi-modélisation (l'utilisation de plusieurs types de modélisation) ; les deux suivants concernent la traduction d'un modèle objet en langage formel ; la troisième partie essaie de mettre en évidence certains des apports d'un couplage de notations semi-formelles et formelles.

Les personnes intéressées par le couplage de notations trouveront dans le chapitre 2 la présentation des différents types de modélisations (informelle, semi-formelle et formelle) ainsi qu'un bilan des différentes approches qui ont été proposées pour coupler des notations semi-formelles et formelles. Ayant choisi une stratégie de traduction parmi ces approches, le chapitre 3 porte plus particulièrement sur nos choix de formalismes pour la traduction : le modèle objet pour les notations semi-formelles, Z et d'Object-Z pour les langages formels. Pour le modèle objet, sa présentation est l'occasion de donner une idée de la sémantique des concepts qui vont être traduits.

La deuxième partie concerne la traduction du modèle objet en langage formel. Le chapitre 4 dresse un état de l'art des propositions de traduction. Sa lecture n'est pas indispensable à la compréhension du reste de la thèse, mais permet ultérieurement de situer notre travail par rapport aux travaux existants. Nous donnons ensuite dans le chapitre 5 nos règles de traduction pour chacun des concepts du modèle objet en Z et Object-Z. Ce chapitre se termine par la comparaison de ces deux langages formels comme langage cible.

Nos propositions de traduction étant décrites, nous montrons dans le chapitre 6 trois exemples de bénéfices qu'elles apportent : la définition d'un guide méthodologique pour l'expression des contraintes sur un modèle objet, l'aide à la validation de contraintes grâce à la systématisation de preuves et la précision sémantique de trois exemples de modèles objet. Le chapitre 7 montre comment tirer profit de nos propositions de traduction et de ses bénéfices en créant un environnement de modélisation où notations semi-formelles et formelles cohabitent. La lecture de ces deux chapitres ne nécessite pas une compréhension fine de nos propositions de traduction bien que cette dernière la facilite.

Le chapitre 8 étudie un autre type de couplage de notations et le compare avec notre stratégie de traduction. Il peut être vu de façon totalement indépendante du reste de la thèse si l'intérêt du lecteur concerne seulement les caractéristiques des deux stratégies de couplage étudiées.

Enfin la conclusion résume les deux avancées principales de ce travail qui sont des propositions permettant un couplage utile de notations semi-formelles et formelles et une étude de deux méthodes de couplage de notations. Elle se termine par une discussion concernant les difficultés liée à la formalisation des modèles semi-formels et propose quelques perspectives.

Chapitre 2

Multi-modélisation

2.1 Définitions

Le génie logiciel (GL) a été défini pour la première fois en 1969 [NR69] comme étant "l'établissement et l'utilisation des principes valides pour obtenir d'une manière économique un logiciel fiable qui marche efficacement avec des machines réelles". Depuis, cette définition a évolué pour aboutir à une standardisation [IEE90] : le GL est "l'application d'une approche systématique, disciplinée et quantifiable pour le développement, le fonctionnement et la maintenance de logiciel". Pour réussir à maîtriser développement, fonctionnement et maintenance du logiciel, les moyens suivants sont disponibles [Pre94] :

- **les méthodes :** elles spécifient comment construire un logiciel en recouvrant un grand nombre de tâches telles que le planning du projet, l'analyse des besoins et du système, la conception, l'algorithmique, le codage, le test ou la maintenance. Elles introduisent souvent **un langage spécifique ou une notation graphique** et des critères de qualité du logiciel.

- **les outils :** ils constituent le support automatique ou semi-automatique pour les méthodes.

- **les procédures :** elles permettent un développement cohérent en précisant comment combiner les méthodes et les outils. Elles définissent la séquence selon laquelle les méthodes sont appliquées, les documents sont rendus, les contrôles sont effectués etc.

Les méthodes sont donc le composant du GL dont l'objectif est de capturer les connaissances concernant un problème, d'en construire ou de valider une solution. Cela s'effectue en suivant des étapes, chaque étape correspondant à un document résultat, souvent appelé spécification. Une spécification est en particulier composée d'un ensemble de modèles qui peuvent être utilisés pour représenter un système de l'analyse des besoins à l'implantation. Un modèle est une représentation abstraite du système traité qui est construite à partir d'un ensemble de concepts de modélisation (par exemple pour UML : classe, association, états ...) qui sont représentés par des notations. En résumé, une méthode est définie comme [Rum95] :

- **un ensemble de concepts fondamentaux de modélisation** qui capturent la connaissance sémantique d'un problème ou d'une solution,

- **un ensemble de vues et de notations** pour présenter les informations de la modélisation sous-jacente,

- **un processus itératif** pour construire les modèles et les implanter,

- **une collection de suggestions et de règles** pour guider le développement.

Bien que les systèmes d'information SI se distinguent des logiciels par leurs aspects organisationnel, évolutif et leur orientation gestion de données, leur modélisation est basée sur les mêmes éléments de définition. Elle utilise des notations servant à construire, examiner et manipuler les modèles. Elle peut être informelle, semi-formelle ou formelle selon le langage utilisé. Cette classification des modélisations correspond à celle des langages suivant le niveau de formalisation de leur syntaxe et de leur sémantique [FKV94] :

- **langage informel** qui n'a ni syntaxe, ni sémantique précisément définies. C'est par exemple le cas de la langue naturelle.

- **langage semi-formel** qui a une syntaxe définie pour décrire les conditions selon lesquelles les constructions sont permises. Ils sont en général basés sur des notations graphiques telles que les diagrammes Entité-Relation [Che76], SADT [MM88] ou UML [BJR98].

- **langage formel** qui possède une syntaxe et une sémantique définies rigoureusement. Pour ces langages, il existe un modèle théorique pour valider une construction. Des exemples de langages formels sont certains langages de programmation, les réseaux de Pétri [Pet77], Lustre [CHPP87] ou Z [Spi92].

2.2 Les différents types de modélisation

2.2.1 Modélisation informelle

Une modélisation informelle est construite en langue naturelle avec ou sans règle de structuration. Son usage introduit des risques d'ambiguïtés car ni sa syntaxe, ni sa sémantique ne sont parfaitement définies. Pour réduire ces risques, plusieurs approches ont été proposées. La première consiste à apporter un soin particulier à la rédaction de la modélisation en suivant des recommandations. La rédaction peut par exemple se réduire à remplir des formulaires. La seconde est de restreindre la langue naturelle utilisée. Cette approche est généralement connue sous le nom de langue naturelle "contrôlée". Les langues contrôlées limitent le lexique et la syntaxe de la langue naturelle afin de diminuer le nombre de formulations possibles d'une spécification. Dans les deux cas, la restriction de la langue naturelle n'enlève rien à sa lisibilité.

Points forts d'une modélisation informelle

Une modélisation informelle a pour principal avantage sa facilité apparente de compréhension qui offre une manière familière de communiquer entre les différents acteurs du développement d'un logiciel ou d'un système. Elle est donc souvent utilisée soit comme composant unique de la description d'un système (par exemple pour écrire un cahier des charges), soit comme complément à des modélisations semi-formelles ou formelles.

Le deuxième point fort est le faible coût de formation nécessaire pour savoir rédiger une modélisation informelle (ce qui ne diminue pas le travail de modélisation). L'utilisation de la langue naturelle ne requiert aucune formation particulière et les restrictions proposées pour son utilisation nécessitent simplement la compréhension des règles de rédaction à suivre.

Points faibles d'une modélisation informelle

Les difficultés relatives à l'utilisation d'une modélisation informelle sont liées à l'étendue de la langue naturelle et notamment aux ambiguïtés qui en découlent.

Ces ambiguïtés peuvent mener à une mauvaise compréhension du système. Ainsi il peut exister un décalage entre les besoins du client et ceux compris par les concepteurs ou un malentendu entre concepteurs et développeurs. Dans ce cas, le logiciel ou le système obtenu ne correspond pas aux attentes du client.

Les ambiguïtés rendent imprécis tout raisonnement pour analyser ou vérifier la spécification [FKV91]. Il existe alors de fortes possibilités d'erreurs dans la modélisation du système. De plus, même en restreignant la langue naturelle par des conseils ou par une syntaxe moins riche, des imprécisions persistent. Il est donc difficile d'exploiter ensuite la modélisation produite et les outils de traitement de la langue restent fortement interactifs et limités à un domaine d'application.

2.2.2 Modélisation semi-formelle

Les méthodes semi-formelles sont généralement basées sur des langages graphiques qui ont une syntaxe précise, mais une sémantique assez faible. Leurs prémices sont apparues dans les années 60 avec des manuels de conduite de projet pour analyser les applications de gestion. Dans les années 70 qui marquent le point de départ du génie logiciel, c'est le développement des méthodes d'analyse et de conception proprement dites. Elles peuvent être classées chronologiquement en trois catégories [Gir95] :

- **les méthodes cartésiennes** (SA/SD [YC79], JSD/JSP [Jac83], SADT [MM88])
- **les méthodes systémiques** (Merise [TRR83], Remora [RFB88], IDA [BP83])
- **les méthodes objets** (OMT [RBP+91], OOD [Boo94], OOSE [JCO92], Fusion [CAB+96])

Les méthodes cartésiennes

Elles sont basées sur une approche fonctionnelle du SI : elles décrivent le système en termes d'opérations qui doivent être décomposées et de processus qui sont organisés par étapes. Elles se focalisent sur une description hiérarchique fonctionnelle généralement décrite par des flots de données et de contrôle.

Les diagrammes de flots de données ont comme composants des *entités externes* qui sont les éléments externes au système et avec lesquels celui-ci communique, des *fonctions ou des processus* représentant la partie du système qui transforme les entrées en sorties, *les répertoires de données* modélisant les données en attente ou persistentes et *les flots de données* qui représentent le mouvement des données dans le système. Ils peuvent définir plusieurs niveaux d'abstraction à travers la décomposition d'un processus dans un nouveau diagramme. Un exemple de diagramme de flot de données est présenté dans la figure 2.1. Ce diagramme est issu d'un exemple de gestion de conférences que nous allons développé tout au long de cette thèse. Il décrit les processus découlant de la demande d'inscription d'un participant à la conférence.

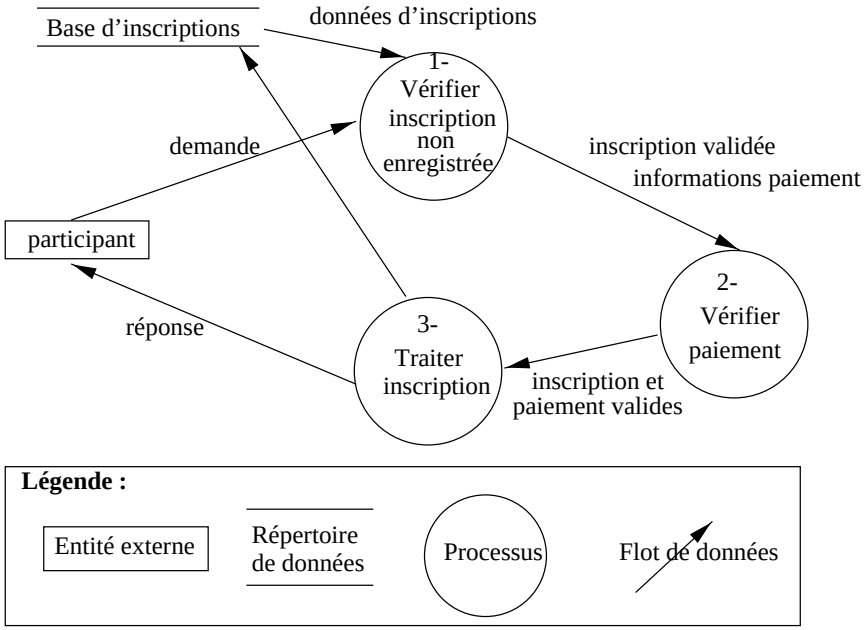


FIG. 2.1: Diagramme de Flots de Données en SA-SD

Ces méthodes ont pour avantage de proposer une méthode de travail descendante par étapes. La complexité du système est réduite par la décomposition qui est organisée grâce à la hiérarchisation (non illustrée ici). Dans leurs premières versions qui ne contenaient que des diagrammes fonctionnels, la structure des données n'était pas représentée de manière explicite et dépendait des traitements. Les versions suivantes ont alors ajouté une description de la structure du système sous forme de diagrammes Entité-Relation (ER [Che76]). Mais cela pose le problème de savoir dans quelle mesure un diagramme de flot de données ou de contrôle est cohérent avec un diagramme ER.

Les méthodes systémiques

Elles sont apparues dans les années 80. Elles marquent une rupture avec les méthodes antérieures pour privilégier une approche conceptuelle globale du SI basée sur la recherche des éléments pertinents et de leurs relations.

La plupart des méthodes systémiques utilisent pour la représentation des données des modèles graphiques basés sur le modèle Entité-Relation et pour la représentation du comportement des modèles basés sur les automates.

Les modèles ER sont composés *d'entités* et *de relations*. Les entités représentent un ensemble d'éléments du monde réel ayant les caractéristiques suivantes : une identification unique, un rôle dans le système en développement et une description exprimée par une ou plusieurs valeurs structurées. Les relations représentent les connexions entre entités et peuvent être décrites avec des cardinalités. Un exemple de modèle ER pour la gestion de conférences est présenté figure 2.2. Il exprime qu'un chercheur peut s'inscrire à plusieurs conférences dont le programme est constitué d'articles. La cardinalité 1 entre "Article" et "Conférence" précise qu'un article ne peut être au programme que d'une seule conférence.

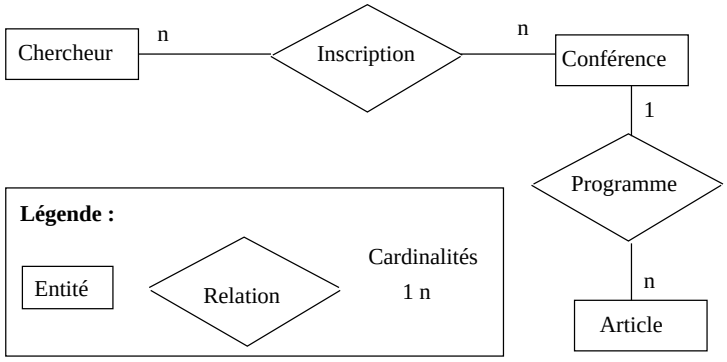


FIG. 2.2: Diagramme Entité-Relation

Ces méthodes proposent de représenter à la fois la structure et le comportement du système. De plus, elles prennent en compte les composantes non-informatisées d'un SI grâce à des modèles tels que le Modèle Organisationnel des Traitements de Merise. Cependant ces méthodes ont été conçues en pensant que les données et des traitements devaient être décrits de manière disjointe. Enfin le modèle de données est souvent contraint au niveau conceptuel par des contraintes du modèle logique relationnel.

Les méthodes objets

Les méthodes objets peuvent être considérées comme une évolution de l'approche systémique avec une plus grande cohérence entre les entités et leur dynamique. Elles ont pour concept fondamental l'objet qui regroupe à la fois des structures de données et un comportement. Le système est représenté sous forme d'objets interagissant les uns avec les autres. Comme les caractéristiques exactes qui définissent un modèle objet n'obtiennent pas de consensus, nous avons choisi les quatre aspects suivants :

- l'encapsulation
  L'idée est de voir l'objet comme une boîte noire dont les mécanismes internes restent cachés et qui est manipulable exclusivement par ses aspects externes.

- l'identité
  L'objet doit être un, accessible de manière unique par un identifiant quelconque. Chaque objet possède une clé unique qui permet de le référencer sans ambiguïté.

- la classification
  Les objets ayant la même structure et le même comportement sont regroupés en une classe. Une classe est une abstraction qui décrit les propriétés pertinentes pour l'application. Elle décrit un ensemble d'objets, chaque objet étant une instance de la classe.

- l'héritage et le polymorphisme
  L'héritage induit une relation hiérarchique entre classes. Une classe est définie suivant de grands traits (caractéristiques communes) et ensuite affinée dans des sous-classes ayant leurs propres caractéristiques.

  Le polymorphisme signifie qu'une même opération peut se comporter différemment suivant les classes. C'est le cas dans les hiérarchies d'héritage où les sous-classes peuvent affiner ou modifier une opération pour la rendre plus adéquate.

L'exemple (Fig. 2.3) montre la représentation du modèle de données d'une gestion de conférence dans une notation objet. Les particularités sont que les classes ont des opérations ("Chercheur" a, par exemple, une opération "inscrire"), qu"'Inscription" est maintenant une classe associative caractérisée par un couple (chercheur,conférence) et qu'une relation d'agrégation lie "Conférence" et "Article".

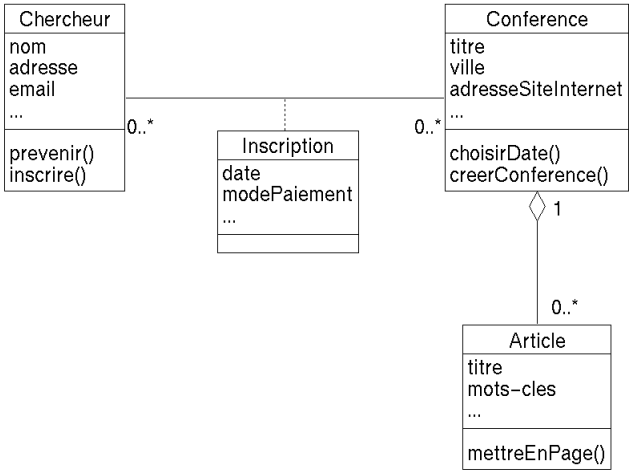


FIG. 2.3: Diagramme Objet en UML

Les méthodes objet bénéficient des nombreux avantages de l'utilisation d'une technologie à objets [Obj95]. Comme certaines des méthodes cartésiennes telles que

JSD, elles peuvent être utilisées de manière continue dès la phase d'analyse jusqu'à la phase de codage et de test ; elles permettent une meilleure modularité et une réutilisation des composants du système. En mettant l'accent sur ce qu'est un objet et sur ce qu'il fait, elles offrent la possibilité de représenter plus directement les entités du monde réel avec moins de déformation : il n'y a qu'un petit fossé sémantique entre la réalité et le modèle, ce qui facilite la compréhension du modèle. Mais du fait de leur complexité, les méthodes objets manquent de théorie, leur sémantique est mal définie. C'est par exemple le cas pour des concepts complexes tels que l'héritage ou l'agrégation. Il est aussi difficile de combiner les différentes vues d'un système (vue statique, vue dynamique, vue fonctionnelle quand elle existe).

Les méthodes objets constituent une avancée importante pour la modélisation des SI. De plus, dernièrement, une unification des différentes notations objets, Unified Modeling Language (UML) a été proposée et standardisée par l'OMG [UML97]. Depuis UML évolue toujours en vue d'une standardisation internationale par l'International Organization for Standardization ISO [Kob99].

Cette unification permet de diminuer la multiplicité des représentations, ce qui facilite grandement la communication et l'intégration des outils. Cependant cette approche d'unification tend à faire croire qu'une seule méthode est suffisante pour tout type de problème. Or les méthodes "généralistes" sont rarement les plus efficaces pour répondre à des buts spécifiques et il est plus raisonnable d'appliquer en chaque lieu la méthode la plus appropriée [Bou97, Jac99b]. Bien qu'UML ne soit pas une méthode, ses notations peuvent faire l'objet de la même remarque.

De plus, de nombreux points sont à approfondir ou à revoir dans UML. Dans [SG99], une liste de ces points est proposée et comprend :

- des contradictions entre les modèles d'UML,
- des ambiguïtés,
- des lacunes qui ne permettent pas de représenter certains concepts,
- des malentendus qui font que les développeurs interprètent mal les modèles UML.

UML n'est donc pas (encore ?) la panacée des notations semi-formelles même si des progrès ont été effectués avec en particulier la définition de la sémantique par un méta-modèle en UML. Cependant pour obtenir des modèles non-ambigus, compréhensibles et échangeables, UML doit être plus précis avec peut-être moins de concepts, mais un noyau bien défini qui serait composable pour obtenir de nouvelles notions [ECM+99] ; ce qui est effectivement en partie pris en considération dans la dernière version 1.3 à travers des mécanismes d'extension tels que les stéréotypes qui permettent de définir de nouveaux concepts en UML.

Points forts d'une modélisation semi-formelle

Les modèles semi-formels aident à élaborer et à structurer des idées en représentant le système sous forme de diagrammes. Les notations graphiques permettent d'avoir une vision claire et abstraite du système. Les concepteurs peuvent les utiliser

pour mieux comprendre les besoins et soumettre leurs propositions aux utilisateurs. En effet, les notations graphiques sont de bons vecteurs de communication tant entre concepteurs et utilisateurs qu'entre concepteurs et développeurs. Elles ont donc trois avantages principaux : elles permettent une vue synthétique, structurante et intuitive du système.

De plus, l'utilisation de modèles objets améliore la modularité et réutilisation des composants du système. Elle permet aussi d'accroître la traçabilité entre les différentes phases de développement d'un système.

Points faibles d'une modélisation semi-formelle

Un des points faibles d'une modélisation semi-formelle est son manque de précision. Les notations demeurent ambiguës si bien que l'interprétation d'un modèle est parfois difficile ou incorrecte. C'est ce manque de sémantique précise qui conduit à des ambiguïtés ou des malentendus. Le problème se pose en particulier pour des concepts tels que la composition de flots dans les diagrammes de flots de données [PdRHP94] et l'agrégation dans les notations à objets.

Ce manque de sémantique limite aussi le support de raisonnement et de simulation possible. Les techniques de validation et de vérification sont donc peu développées, en particulier pour les méthodes objets. Il est alors difficile de construire des outils qui permettraient d'automatiser (au moins en partie) la validation ou la vérification des modèles. Enfin à partir d'un même modèle, la génération de code par des outils différents peut donner des codes forts différents. La confiance dans la fiabilité des systèmes construits est alors nécessairement limitée.

Enfin en proposant divers modèles pour modéliser les différentes vues d'un système, les méthodes introduisent des problèmes de gestion de cohérence entre ces modèles.

2.2.3 Modélisation formelle

Les langages formels sont basés sur des notations mathématiques qui fournissent un cadre précis et non-ambigu pour la modélisation. Ils sont apparus dans les années 70 et se sont depuis développés en fonction des activités ciblées si bien qu'ils sont aujourd'hui nombreux. Ils peuvent être groupés selon les catégories suivantes [Mar94] :

- **les langages orientés modèles** (Z [Spi92], VDM [ABH+95], B [Abr96]...)
- **les langages orientés propriétés** (Larch [GHW85], Act One, Lotos [SGO91]...)

Cette classification peut être complétée par une classification fonctionnelle complémentaire en deux catégories : les langages pour les systèmes séquentiels et ceux pour les systèmes réactifs.

Les langages orientés modèles

Les langages orientés modèles sont basés sur la notion d'état. Ils expriment une définition explicite de l'état du système en construisant un modèle en termes de

structures mathématiques telles que les ensembles, les fonctions ou les prédicats. La définition des opérations sur le système est basée sur la manière par laquelle elles affectent le modèle.

Par exemple, en Z, l'état du système est représenté par un schéma groupant des variables avec des prédicats. Dans sa partie supérieure, le schéma *Acceptations* comprend deux variables : *soumissions* qui représente l'ensemble des articles soumis à la conférence, *ResSoumissions* qui donne le résultat d'une soumission c'est-à-dire si une soumission est acceptée ou refusée. Dans la deuxième partie du schéma, les prédicats imposent qu'une soumission dont on connaît le résultat fasse partie de l'ensemble des soumissions reçues.

$$STATUT ::= Accepte \mid Refuse$$

Acceptations

$$soumissions : \mathbb{F}\, SOUMISSION$$
$$ResSoumissions : SOUMISSION \pfun STATUT$$
$$\dom ResSoumissions \subseteq soumissions$$

Le schéma peut également être utilisé pour spécifier une opération. Dans ce cas, la partie supérieure décrit les entrées (notées par ?), les sorties (notées par !) et les variables modifiées par l'opération. La partie inférieure du schéma exprime un prédicat qui est satisfait par l'état des variables avant et après l'opération. Par exemple, l'opération *Accepter* qui enregistre l'acceptation d'une soumission modifie les variables du schéma *Acceptations* (Δ) et a comme entrée la soumission acceptée. Le prédicat impose que cette soumission fasse partie de l'ensemble des soumissions reçues, mais sans résultat. Enfin il spécifie que cette soumission est ajoutée aux résultats des soumissions en précisant que la valeur de *ResSoumission* après l'opération (notée par ') est égale à celle de *ResSoumission* avant l'opération à laquelle le couple *(uneSoumission ?,Accepte)* est ajouté.

Accepter

$$\Delta Acceptations$$
$$uneSoumission? : SOUMISSION$$
$$uneSoumission? \in soumissions \land uneSoumission? \notin \dom ResSoumissions$$
$$ResSoumissions' = ResSoumissions \cup \{uneSoumission? \mapsto Accepte\}$$

Dernièrement sont aussi apparus des langages formels à objets. Ils introduisent une structure, la classe, qui regroupe la structure de données et les opérations réalisables sur un objet. En particulier des extensions orientées objet des langages orientés modèle ont été proposées (VDM++ [Dür94], OOZE [AG91], Object-Z [DKRS91]...) . Ainsi les schémas Z donnés précédemment pour specifier les acceptations d'une soumission et une de leurs opérations sont regroupés dans une seule classe en Object-Z.

Acceptations

$$soumissions : \mathbb{F}\, SOUMISSION$$
$$ResSoumissions : SOUMISSION \pfun STATUT$$
$$\mathrm{dom}\, ResSoumissions \subseteq soumissions$$

Accepter

$$\Delta ResSoumissions$$
$$uneSoumission? : SOUMISSION$$
$$uneSoumission? \in soumissions$$
$$ResSoumissions' = ResSoumissions \cup \{uneSoumission? \mapsto Accepte\}$$

...

Les langages orientés modèles sont riches en constructeurs [Som92], permettant ainsi d'écrire des spécifications assez concises. Par contre, cela rend plus difficile leur apprentissage et leur maîtrise.

Les langages orientés propriétés

Avec un langage orienté propriétés, le système est décrit en posant un ensemble de propriétés, habituellement sous la forme d'un ensemble d'axiomes que le système doit satisfaire, ce qui tend à préserver la liberté d'implantation.

Les langages orientés propriétés peuvent être divisés en deux catégories nommées axiomatiques et algébriques. Les langages axiomatiques découlent des travaux de Hoare sur les preuves et la correction des implantations de types abstraits de données où les pré- et post-conditions en logique du premier ordre sont utilisées pour la spécification de chaque opération du type. Dans un langage algébrique, des axiomes sont utilisées pour spécifier les propriétés d'un système, mais ces axiomes sont réduits à des équations. Une spécification algébrique est composée d'une signature et d'un ensemble d'axiomes qui sont des formules logiques. La signature comprend les noms des types et les opérations définies par leur profil. Dans cette approche, la description des données se limite à un nom qui n'a pas de structure sous-jacente. Pour la gestion de conférences, la spécification comporte une sorte *Article*. Ses opérations (définies par leur profil) sont *titre, mots-cles et Créer* . Les axiomes spécifient les opérations par des formules logiques.

Sortes : Article
Opérations :
titre(Article) → String
mots-cles(Article) → String
...
Creer(String,String,..) → Article
Axiomes :
titre(Creer(t,m,...))=t
mots-cles(Creer(t,m,...))=m

Les spécifications orientées propriétés proposent une approche plus déclarative et plus abstraite que les spécifications orientées modèles. Mais leurs mécanismes d'utilisation sont très lourds. Comme il n'existe pas d'état, un nouvel élément doit être créé à chaque modification.

Points forts d'une modélisation formelle

Contrairement à un mythe répandu, les méthodes formelles ne fournissent pas de solution à tous les problèmes de sécurité, de sûreté et de fiabilité [Voa99]. Les avantages des langages formels ont été largement présentés [Som92, Win90, Hab95, And95]... En bref, précision, abstraction et formalismes permettent raisonnement, vérification, tests et preuves [And95].

Grâce aux aspects formels, des propriétés peuvent être établies par raisonnement [Sai96, Hab95, FKV94, Hal90a]. Les intuitions sont donc démontrables par une argumentation stricte. Il devient alors possible de maîtriser le processus conduisant de la spécification au programme et de montrer qu'une implantation dans un programme est conforme à sa spécification. De plus, il est aussi possible de générer automatiquement du code ou des cas de tests à partir des spécifications.

La précision des spécifications formelles, c'est-à-dire le niveau de détails qu'elles fournissent, permet de mieux percevoir le problème [ZJ97, HS99, HRW93, Win90, Hal90a]. L'écriture de spécifications formelles aide à cristalliser les idées vagues du client, à révèler ou éviter des imprécisions, des ambiguïtés ou des contradictions, aidant ainsi à identifier les besoins du client. La modélisation est donc mieux comprise et assimilée.

Leur niveau d'abstraction élevé permet de retenir seulement les caractéristiques essentielles et préserve la liberté d'implantation. La modélisation est donc plus concise. De plus, les différents niveaux d'abstraction sont clairement identifiés. Cela évite leur mélange empêchant ainsi la sur-spécification [Hab95].

L'utilisation des méthodes formelles est donc préconisée de l'analyse des besoins, à l'implantation en passant par les phases de spécification et de conception. Elle se traduit par des réductions de coûts en aval de la spécification et une rentabilité accrue des investissements en amont de la réalisation, la vérification étant un point clé de l'assurance qualité.

Points faibles d'une modélisation formelle

De nombreuses idées reçues entourent les langages formels [Hal90a, BH95]. Ils concernent :

- **leur formalisme ;** selon ces idées reçues, les langages formels nécessitent des mathématiciens de haut niveau et sont inacceptables pour les utilisateurs ;
- **leur utilisation ;** les langages formels ne seraient utilisables que pour des systèmes critiques ou des logiciels de petite taille ;
- **leurs désavantages par rapport aux méthodes traditionnelles ;** les spécifications formelles augmenteraient les coûts et provoqueraient des retards

dans le processus de développement ; elles auraient pour vocation de remplacer les méthodes traditionnelles alors que leur utilisation n'est pas nécessaire.

Bien que des exemples d'applications industrielles [CW96] et des études comparatives [LFB96] aient montré que la plupart de ces idées reçues étaient infondées, les langages formels demeurent peu utilisés. En effet, il existe des points faibles avérés tels que le niveau de formation nécessaire à leur maîtrise, des notations abstraites donc peu intuitives, un manque de méthodes et d'outils de support et des problèmes de passage à l'échelle. De plus, les analyses effectuables sont limitées par la complexité des formalismes : du fait de l'utilisation courante de la logique du premier ordre, les preuves ne sont que semi-décidables.

C'est pourquoi de nombreuses propositions ont été effectuées pour augmenter leur utilisation [FKV94, Mar94, Sai96]. Il est en particulier recommandé par tous de :

- faire un effort pour **former** le plus de personnes possible aux langages formels ;
- développer des **outils plus accessibles et plus robustes** pour gérer de larges spécifications ou réaliser des analyses sémantiques plus efficaces afin de détecter plus d'erreurs ;
- **mieux intégrer** les langages formels dans le processus de développement, en particulier en les combinant avec les méthodes traditionnelles.
- **améliorer leur interface** en proposant des notations plus facilement compréhensibles.

2.2.4 Bilan

Les sections précédentes ont montré que chaque type de modélisation a des avantages et des inconvénients. Pour les comparer, nous avons choisi de nous intéresser aux critères suivants : précision de la modélisation, utilisation, facilité de communication et coût de formation. Le tableau 2.1 résume les résultats de cette comparaison suivant le bârème ci-dessous :

+++ point très positif

++ point positif

\+ point assez positif

— point très négatif

– point négatif

\- point assez négatif

	précision	utilisation	facilité de communication	coût de formation
modélisation informelle	-	+++	+++	++
modélisation semi-formelle	+	++	++	+
modélisation formelle	+++	-	—	—

TAB. 2.1: Comparaison des différents types de modélisations

Les différentes modélisations sont donc complémentaires. D'une part, les modélisations informelles et semi-formelles qui pêchent par leur imprécision, sont de bons vecteurs de communication avec un coût de formation peu élevé. D'autre part, la modélisation formelle apporte la précision manquant aux modélisations informelle et semi-formelle. Dans ce travail, nous nous intéressons au couplage de modélisations semi-formelle et formelle avec pour objectif de gommer leurs inconvénients et de cumuler leurs avantages.

2.3 Approches de couplage de notations

2.3.1 Présentation des approches de couplage existantes

Chaque type de notations présentant des avantages et des inconvénients, de nombreux travaux ont essayé de coupler des notations différentes afin d'obtenir une vue du système la plus complète possible. Ces travaux peuvent être regroupés en trois approches :

- **créer un nouveau formalisme à partir de concepts de différents langages** pour qu'il cumule les qualités des notations choisies.
  C'est le principe qui a mené à la définition de nouveaux langages formels tels que Lotos [SGO91] qui combine ActOne et CCS, ZCCS [GS97] qui intègre Z et CCS ou les langages formels orientés objet comme Object-Z.

- **compléter les notations existantes** pour diminuer leurs inconvénients
  Cette approche a été utilisée pour représenter les concepts objets en Z [Hal90b, Hal94] ou pour préciser la sémantique de notations semi-formelles en décrivant formellement leur méta-modèle [Zha97, MM97, EFLR98].

- **utiliser plusieurs notations existantes** pour cumuler leurs avantages.
  En suivant cette approche, chaque notation est utilisée pour représenter la partie du système pour laquelle elle est la plus adaptée. C'est ce qui est à la base de la création d'UML ou d'une nouvelle méthode couplant Fusion et Object-Z [AS97]. De même, Büsssow et Weber [BW96] ont proposé des spécifications d'un système utilisant les statecharts et Z alors que Wieringa et al. [WDH97] combinent des spécifications semi-formelles et des spécifications formelles. Le lien entre les différentes notations peut alors être effectué :

  - soit en générant systématiquement des spécifications à partir d'autres spécifications afin de **disposer de vues complémentaires** qui permettent

de vérifier des propriétés du système.
Dans ce cas, la génération s'effectue suivant des règles de traduction entre les notations. Ainsi on peut citer la méthode SAZ [PWM93] combinant SSADM et Z.

- soit **en vérifiant a posteriori la cohérence** entre des spécifications développées dans des notations différentes.
  La vérification s'effectue soit en traduisant les différentes notations dans un langage commun, soit en définissant des cadres qui permettent d'exprimer les liens entre les différentes notations. La traduction a, par exemple, été utilisée entre Z et Lotos [BDS96] alors que les ViewPoints [FKN+92] sont l'exemple typique de cadres définis pour représenter les notations et leurs liens.

La traduction est donc une technique de couplage de notations qui peut être utilisée aussi bien pour générer des spécifications dans le but de valider certains aspects d'un système que pour vérifier la cohérence a posteriori de différentes spécifications.

Ces trois approches proposent des moyens de coupler différents types de notations. Elles peuvent donc être utilisées dans le cas de couplage de notations semi-formelles et formelles.

2.3.2 Créer un nouveau formalisme

Appliquée aux notations semi-formelles et formelles, la première démarche consiste à définir une méthode graphique sur de bonnes bases formelles.

Des exemples de **méthodes créées en s'inspirant de notations semi-formelles et formelles** sont Fusion [CAB+96] ou Alloy [Jac99a]. Fusion est une méthode graphique orientée-objet influencée par les méthodes formelles. En particulier, dans la phase d'analyse les opérations sont spécifiées en partie en termes de pré et post-conditions comme dans certains langages formels. Mais la partie principale de la spécification du système reste encore non formelle.

Alloy suit plutôt la démarche inverse en essayant de donner une apparence graphique à des parties de notations formelles. Elle couple UML et Z en proposant une nouvelle notation textuelle, inspirée de Z, dont un sous-ensemble peut être exprimé graphiquement dans des notations proches de celles d'UML.

2.3.3 Compléter les méthodes existantes

Cette approche préconise d'améliorer les méthodes existantes en les complétant pour diminuer leurs désavantages. Ainsi il est proposé de donner une sémantique formelle aux notations semi-formelles ou de définir une syntaxe graphique à des notations formelles.

Pour **rendre formelles des notations semi-formelles**, il faut leur donner une sémantique formelle. Pour cela, un méta-modèle est généralement défini dans un langage formel [Zha97, MM97, EFLR98]. En particulier, le groupe pUML (Precise

UML) essaie de rendre UML formel [FELR98] afin de pouvoir ensuite raisonner sur les modèles UML, en vérifier la cohérence et construire des outils supportant ces activités. Pour cela, il propose d'écrire un méta-modèle d'UML en Z [EFLR98] et de définir la sémantique d'OCL par un méta-modèle en UML et des contraintes en OCL [KGR99] .

Pour **donner une apparence graphique aux spécifications formelles**, Dick et Loubersac [DL91] représentent sous forme de diagrammes entité/structure, extension de diagrammes entité/relation et de diagrammes d'état d'opérations des spécifications exprimées en VDM.

2.3.4 Utiliser plusieurs notations existantes

Traduction d'une notation en une autre

Cette approche de couplage de notations consiste à traduire une notation vers une autre. Dans le cas de notations semi-formelles et formelles, on peut soit dériver des spécifications semi-formelles à partir de spécifications formelles, soit faire l'inverse.

Dans le cadre de la dérivation de spécifications semi-formelles à partir de spécifications formelles, Gogolla et Richters [GR98] présentent comment des constructions de spécifications algébriques en TROLL sont liées à des éléments graphiques d'UML. La traduction des modèles en TROLL permet de représenter des diagrammes de classes et d'états-transitions en UML même si tous les aspects des spécifications formelles ne peuvent pas être décrits graphiquement.

Mais l'approche de traduction la plus développée est celle allant **des spécifications semi-formelles aux spécifications formelles**. Pratiquement tous les types de méthodes semi-formelles selon la classification chronologique de [Gir95] ont été étudiés. Les méthodes cartésiennes ont été couplées avec des méthodes formelles telles que Z ou VDM. On peut par exemple citer l'intégration de SSADM et de Z ([PWM93]) ou de SA et de VDM ([PKP91]). Le couplage des méthodes systémiques avec les méthodes formelles est peu développé. On peut néanmoins citer l'intégration de Merise et de Z [DS96] pour les modèles conceptuels des données et des traitements. La plupart des travaux de recherche s'intéressent maintenant à intégrer des annotations formelles aux modèles des méthodes orientées objet. Ce type de travail a été effectué pour des notations de type OMT vers du VDM [LHR95], du B [Ngu98, MS99] ou vers des spécifications algébriques [BC95, WRC97], de Fusion vers Z [FB97] ou plus récemment d'UML vers Z [SF97, FBLPS97] ou vers des spécifications axiomatiques [LB98]. Certains travaux s'orientent aussi vers la traduction dans des langages de spécifications formelles orientées objet [LG96, AS97, KC99]. Ainsi Achatz et Schulte [AS97] proposent une méthode complète pour écrire parallèlement des spécifications en Fusion et Object-Z.

La traduction de spécifications semi-formelles vers des spécifications formelles comprend la traduction implicite ou explicite des concepts du modèle et celle des concepts du domaine. Elle peut donc s'effectuer selon deux démarches [FL95] :

- dérivation directe (Fig. 2.4)
  Tout concept du modèle est traduit directement dans le langage formel.

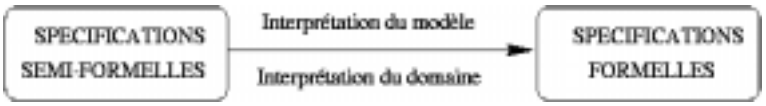


FIG. 2.4: Dérivation directe

– génération par méta-modélisation (Fig. 2.5)
Le principe de cette approche est de spécifier formellement une fois pour toute le méta-modèle du modèle semi-formel. Puis pour chaque application, on effectue la traduction de la partie propre à chaque application en utilisant la formalisation du méta-modèle.

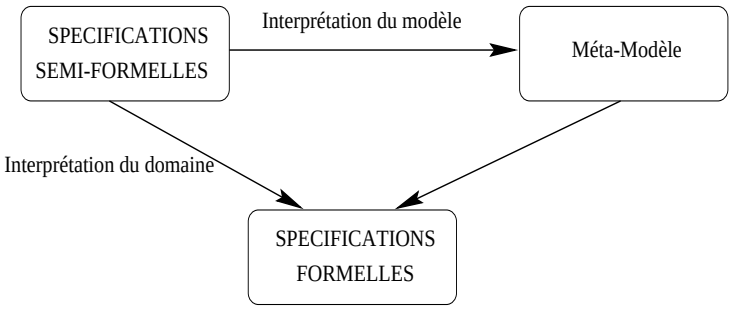


FIG. 2.5: Génération par méta-modélisation

La deuxième approche a été utilisée pour traduire les diagrammes de flux de données en Z [Ran90]. Mais généralement, le méta-modèle n'est utilisé que pour formaliser certains concepts dont la sémantique n'est pas traduisible directement. C'est par exemple le cas pour les événements [Ham94] en Z ou les associations en VDM [FL96].

Vérifier la cohérence entre spécifications

La dernière approche de couplage de notations consiste à développer indépendamment des vues d'un système dans des formalismes différents, puis à vérifier la cohérence entre ces vues. Bien que ce processus soit couramment utilisé dans les langages composés de différents modèles tels qu'UML, la façon de gérer la cohérence entre les vues n'est pas souvent abordée. Néanmoins la recherche autour de la vérification de cohérence entre les vues se développe. Dans ce domaine, deux approches ont été proposées : exploiter une représentation canonique dans laquelle les vues seront comparées ou gérer la cohérence en termes de règles entre les méta-modèles des notations utilisées.

La première manière de gérer la cohérence entre des spécifications est **d'utiliser une représentation canonique**. Les spécifications sont traduites dans la représentation canonique qui capture donc les différentes perspectives dans un modèle commun. Les incohérences éventuelles entre les spécifications peuvent ensuite être détectées au niveau de la représentation canonique. De nombreuses représentations canoniques ont été utilisées dans ce but. Des exemples sont la logique du premier ordre [ZJ93], les graphes conceptuels [Del92] ou Telos [SC95].

Une approche alternative pour vérifier la cohérence de vues multiples est d'**utiliser un langage de méta-représentation**. Ce langage fournit des notations pour décrire les structures de représentation et spécifier les règles de cohérence entre les différentes vues. Chaque type de notation a donc un méta-modèle dans le langage de méta-représentation et les règles de cohérence définissent des liens entre les parties des différents méta-modèles. Cette démarche a été proposée dans [Tha99] pour vérifier la cohérence entre les points de vue dans le cadre des ViewPoints [FKN+92]. Dans ce travail, les graphes conceptuels servent de langage de méta-représentation. [EEF+99] se base aussi sur cette démarche pour vérifier la cohérence entre différents types de documents. Dans ce cas, le langage de méta-représentation est XML, actuel standard reconnu pour la représentation de documents et qui permet de créer physiquement les liens de cohérence entre des documents.

2.3.5 Choix d'une approche

Avantages et inconvénients des approches proposées

Afin de motiver la démarche choisie, nous présentons ici les avantages et les inconvénients des approches présentées précédemment :

- La création d'une méthode semi-formelle sur des bases formelles cumule les avantages des notations semi-formelles et formelles : les vues du système sont graphiques, synthétiques et précises. Mais la nécessité de modifier les habitudes de travail existantes en apprenant une nouvelle notation, pose des problèmes humains.

- Compléter des méthodes existantes pour diminuer leurs inconvénients permet de rester proche des habitudes de travail existantes. Mais si cela consiste à rendre formelles des notations semi-formelles, cela nécessite que les concepteurs comprennent le méta-modèle formel, ce qui est difficile. Il n'est donc pas évident que cette approche permette aux concepteurs de mieux comprendre la sémantique de leurs modèles. La seconde solution qui consiste à donner une apparence graphique aux spécifications formelles demandent toujours d'avoir une bonne maîtrise du langage formel.

- L'utilisation de différentes notations existantes permet de cumuler les avantages des différentes spécifications en utilisant chaque notation dans le contexte qui lui convient le mieux. Mais cette démarche nécessite de définir les liens entre différents types de spécifications. Les avantages et les inconvénients dépendent alors de l'approche choisie.

  - Une approche de traduction offre une équivalence sémantique entre les spécifications semi-formelles et formelles, permettant ainsi l'exploitation du meilleur des deux types de spécifications. En particulier, les analyses ou raisonnements effectués sur les spécifications formelles sont aussi valables pour les spécifications semi-formelles. Il faut aussi gérer lors de la traduction les pertes ou ajouts d'information éventuels puisque les spécifications formelles contiennent en général des informations qui

ne sont pas toutes décrites dans des langages graphiques comme UML. Mais la principale difficulté d'une traduction porte sur l'équivalence sémantique. Même si on peut imaginer de prouver certaines propriétés, l'équivalence repose sur la vision du traducteur et il est parfois difficile de la conserver lors du processus de traduction. De plus, toute traduction dépend totalement des formalismes étudiés.

La visualisation graphique de spécifications formelles convient plutôt aux personnes maîtrisant déjà les langages formels puisqu'elle nécessite l'écriture complète des spécifications formelles.

La démarche qui propose de dériver des spécifications semi-formelles vers des spécifications formelles est plus facilement accessible aux concepteurs puisqu'elle se base, en premier lieu, sur les notations semi-formelles déjà répandues. De plus, elle aide à écrire des spécifications formelles, permettant ainsi de les produire plus rapidement en assurant une certaine cohérence entre les deux types de spécifications.

– La vérification de cohérence entre spécifications a posteriori a pour avantage d'être totalement indépendante des formalismes. Les travaux dans ce domaine présentent un cadre dans lequel tous types de spécification peuvent être couplés. Ils laissent donc à l'utilisateur une totale liberté quant aux choix des formalismes utilisés et au processus de développement des spécifications. Mais le degré de cohérence qui lie les différents types de spécifications n'est pas clairement défini ; en particulier, rien ne garantit l'équivalence sémantique entre les spécifications. Enfin la démarche n'apporte pas d'aide pour l'écriture de spécifications formelles.

Pour le couplage de spécifications semi-formelles et formelles, l'approche de traduction nous paraît la plus appropriée car elle établit un lien fort entre les spécifications qui garantit une exploitation optimale de chacune d'elles. La traduction du semi-formel vers le formel semble la plus naturelle puisque le système est d'abord décrit graphiquement avant d'être précisé. Enfin, elle facilite la construction de spécifications formelles.

Il reste donc à choisir entre la traduction directe et la génération par méta-modélisation. La dérivation directe permet une utilisation directe du langage formel. Par contre, la formalisation implicite des notions du méta-modèle ne permet pas de séparer le raisonnement sur le méta-modèle de celui sur l'application. La génération par méta-modélisation a pour avantage de formaliser les notions du méta-modèle. Elle oblige à définir plus explicitement la sémantique des notations semi-formelles. Les spécifications d'une application sont aussi plus courtes et moins répétitives. Mais leur développement est moins intuitif et nécessite de bien connaître le méta-modèle. Dans le cadre d'une aide à la construction de spécifications formelles, la dérivation directe semble la plus naturelle. C'est pourquoi, elle a été choisie pour ce travail qui se situe dans le cadre d'une approche de **traduction de spécifications semi-formelles vers des spécifications formelles par dérivation directe**. Néanmoins, afin de comparer plus en détails notre approche avec d'autres propositions,

nous avons aussi étudié **le couplage de notations semi-formelles et formelles dans un cadre de vérification de cohérence par méta-modélisation**, travail décrit au chapitre 8.

2.4 Conclusion

Dans ce chapitre, nous avons présenté les différents types de modélisation (informelle, semi-formelle et formelle), ainsi que leurs principaux avantages et inconvénients. Nous avons conclu en la complémentarité de ces modélisations. Partant de cette constation, nous avons présenté les approches proposées pour coupler différents types de notations. Enfin nous avons choisi parmi ces approches notre cadre de travail qui consiste en la traduction de notations semi-formelles vers des notations formelles.

Chapitre 3

Présentation de notre approche et des notations utilisées

3.1 Présentation du projet Champollion

3.1.1 Approche du projet Champollion

Etant donné notre préférence pour une approche de traduction de notations semi-formelles en notations formelles, nous avons à prendre en compte les compléments d'information ajoutés lors du passage des notations objet aux spécifications formelles. Ces compléments étant généralement exprimés en langue naturelle, nous devons adopter une approche qui intègre aussi les spécifications informelles. C'est ce que propose le projet Champollion (Fig. 3.1) qui est basé sur la traduction de modélisations informelle et semi-formelle vers une modélisation formelle. Son processus comprend les activités suivantes :

1. Dans un premier temps, une spécification semi-formelle et des annotations en langue naturelle sont développées à partir du problème initial.

2. La spécification semi-formelle est le point de départ d'un processus de traduction qui produit des squelettes de spécifications formelles. Dans le même temps, les annotations qui expriment des contraintes sont formulées en langage formel.

3. Les squelettes de spécifications formelles sont complétés par les contraintes afin de produire une spécification formelle complète.

4. La spécification formelle est utilisée pour vérifier la cohérence de la spécification semi-formelle et de ses annotations. On peut, par exemple, valider un modèle et ses contraintes en calculant les pré-conditions des opérations d'un modèle objet [Led98].

5. Grâce au processus de traduction, les raisonnements effectués sur la spécification formelle peuvent être reportés à la spécification semi-formelle et à ses contraintes.

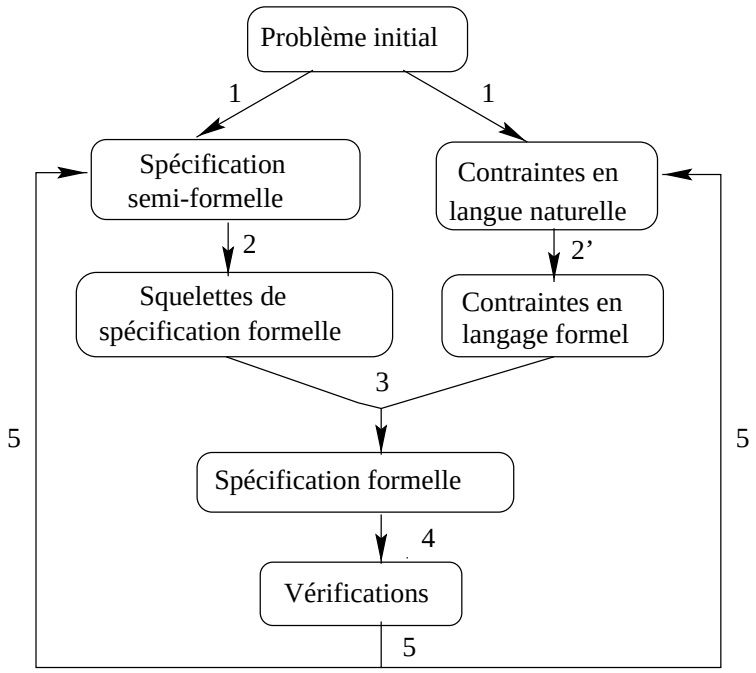


FIG. 3.1: L'approche du projet Champollion

La traduction des annotations en langue naturelle en annotations formelles est traitée dans [Che99]. Pour notre part, nous nous intéressons ici aux autres étapes du processus : le chapitre 5 et l'annexe B.3 présentent nos propositions pour la traduction de notations semi-formelles en spécifications formelles ; l'intégration d'annotations est décrite au chapitre 6 et des exemples de raisonnement sur la spécification formelle et leur impact sur les spécifications semi-formelles sont aussi donnés au chapitre 6.

L'approche du projet Champollion pourrait s'appliquer à n'importe quel type de spécifications semi-formelles et formelles. Mais nous avons choisi d'utiliser les notations semi-formelles orientées objet et les langages Z et Object-Z pour les raisons suivantes.

3.1.2 Choix du langage semi-formel

Le choix du langage semi-formel est dicté par la capacité des méthodes à représenter les SI. Sur ce point, les méthodes objets sont une évolution importante dont nous avons présenté les avantages dans la section précédente. En particulier, leurs notations, en apportant une plus grande cohérence entre les aspects statiques et dynamiques d'un système, contribuent à mieux maîtriser la modélisation des SI.

De plus, les méthodes objets sont de plus en plus utilisées en partie grâce à l'adoption de notations communes (UML) et à leur standardisation. Dans ce travail, nous utilisons donc des notations objets inspirées de celles d'UML.

Nous nous intéressons aussi aux compléments de ces notations qui permettent d'enrichir un modèle en ajoutant des informations telles que les contraintes de domaine ou la spécification des opérations. Dans ce domaine, nous avons choisi d'étu-

dier un langage de contraintes basé sur la logique du premier ordre et la théorie des ensembles : OCL (Object Constraints Language [WK98]) qui sert à annoter les modèles UML. Son intérêt vient du fait qu'il contribue à ajouter de la formalisation aux modèles objets.

3.1.3 Choix du langage formel cible

Dans le contexte d'un couplage de modèles semi-formels objets et formels, deux aspects nous paraissent importants pour choisir le langage formel cible. Tout d'abord, ce langage doit être le plus proche possible des concepts semi-formels ; en particulier, dans l'optique de spécification de SI, le langage formel cible doit être bien adapté à l'expression des concepts du modèle de données. Le deuxième point concerne les outils facilitant l'utilisation des langages formels. En effet, ceux-ci conditionnent les possibilités d'exploitation de l'augmentation de précision apportée par les spécifications formelles pour détecter automatiquement des incohérences, produire des prototypes ou des séries de tests. Nous avons donc comparé différents langages formels, qui nous semblent représentatifs du domaine, suivant les critères d'adéquation aux modèles semi-formels et d'outils de support. Le tableau 3.1 résume ce travail. Dans celui-ci, "modèle statique" fait référence aux représentations des données alors que "modèle dynamique" se réfère aux comportements des objets d'un système.

Afin de faciliter la traçabilité, le langage formel cible doit être le plus proche possible des concepts des modèles semi-formels à préciser. D'après le tableau 3.1, les langages à flots de données tels que Lustre ne correspondent pas à nos besoins dans la mesure où ils sont plus proches du modèle dynamique que du statique.

Pour la statique, les langages orientés propriétés pourraient convenir. Mais même si leurs concepts sont proches des concepts du modèle de données, ils s'orientent plus vers la spécification des propriétés du système que vers celle du modèle proprement dit car il n'y a pas de représentation explicite de l'état des données. La façon de spécifier un système est donc très différente entre les langages orientés propriétés et les modèles semi-formels. Nous avons donc plutôt orienté notre choix vers les langages orientés modèles tels que Z ou VDM, leurs extensions orientées objet et B pour des raisons historiques et pour leur intégration de la notion d'état.

Le deuxième point qui guide le choix du langage formel cible concerne les outils facilitant l'utilisation des modélisations formelles. Z, B ou VDM disposant de plusieurs outils de support intéressants, notre choix s'effectue suivant des critères plus subjectifs. Notre préférence va vers Z et ses extensions orientées objet, d'une part pour leur lisibilité, d'autre part pour leur adéquation avec le langage de contraintes d'UML, OCL [WK98]. En effet Z étant la base de la définition d'OCL, il permet d'exploiter facilement les contraintes exprimables sur un modèle de données.

Il reste à choisir entre Z et une de ses extensions objet dont les concepts sont a priori plus proches des modèles statiques orientés objet. Parmi les nombreuses extensions orientées objet de Z (ZEST [CR92], Z++ [LH94], Mooz [MC92], OOZE [AG91], Object-Z [DKRS91]), nous avons choisi Object-Z car il semble s'imposer comme le standard de ces extensions. De plus, Object-Z est une référence dans le monde des spécifications formelles à objets [And95].

Langage	Caracté- -ristique	Adéquation aux modèles	Outils
Larch	O. propriétés	statique	type checker outil de preuve
Lotos	O. propriétés	statique dynamique	simulateurs générateur de C, de code parallèle
Lustre	flot données	dynamique	outil de preuve outil de test générateur de code
Z	O. modèle	statique	typechecker animateur, simulateur générateur de tests outil de preuve
VDM	O. modèle	statique	typechecker générateur de C++ couverture de tests (outil de preuve)
B	O. modèle	statique	type checker outil d'animation outil de preuve générateur de C
VDM++	O.O.	statique	typechecker générateur de C++
Object-Z	O.O.	statique	typechecker

TAB. 3.1: Caractéristiques des langages formels étudiés

Les sections suivantes présentent les concepts des langages semi-formels et formels que nous allons utiliser dans la suite de ce travail.

3.2 Modèle objet

Les notations semi-formelles objet telles que UML fournissent un ensemble de modèles pour décrire les aspects statiques et dynamiques d'un système. Nous nous intéressons principalement au modèle objet bien qu'une étude d'un modèle dynamique basé sur les diagrammes d'états-transitions soit donnée dans l'annexe B.

Le modèle objet, qui décrit les objets d'un système en terme de classes et de leurs associations, est une extension du modèle Entité-Relation ([Che76]) vers les concepts de l'approche objet (classe, opération, héritage, agrégation et composition).

3.2.1 Classe et concepts associés

Objet Le concept d'objet est une notion très vaste qui recouvre toute abstraction du monde réel qui est identifiée comme pertinente pour l'application considérée. C'est une entité caractérisée par un identifiant et un état qui se comporte (interagit avec d'autres objets) par le biais d'opérations. Il est représenté par des propriétés, des opérations et des contraintes sur ces propriétés.

Classe Les objets ayant une structure similaire, un comportement commun, des relations communes avec les autres objets ainsi qu'une même sémantique sont regroupés dans une classe. Une classe regroupe des attributs représentant la structure et des opérations décrivant le comportement d'une collection d'objets isomorphes, chaque objet étant alors une instance de la classe.

Dans les méthodes semi-formelles orientées-objet telles que UML, une classe est graphiquement représentée par une boîte à tiroirs, zones support d'informations similaires en terme de rôle : la plus haute contient le nom de la classe, celle du milieu la liste des attributs et la plus basse la liste des opérations (Fig. 3.2).

FIG. 3.2: Format d'une classe et classe *Article*

Dans l'exemple de la figure 3.2, "Article" est une classe avec deux attributs : le titre de l'article ("titre") et un ou plusieurs mots-clés ("mots-cles"). La classe a une seule opération "MettreEnPage".

Pour Atkinson [ABW+89] et Hall [Hal94], le concept de classe contient deux aspects :

- Une classe joue un rôle d'entrepôt d'objets qui permet aux classes d'être rattachées à leurs "extensions" c'est-à-dire l'ensemble des instances. Ce réservoir peut être manipulé en appliquant les opérations de la classe. Par exemple, l'extension de "Article" est l'ensemble des articles existants dans la base de données de gestion de conférences.

- Une classe joue également un rôle de fabrique d'objets en permettant de créer des objets de la classe. Ce rôle correspond à celui de générateurs d'instances dans [BGV94]. Ainsi "Article" est une fabrique d'objets ayant un titre, des mots-clés etc.

Attribut Un attribut peut être défini formellement comme une fonction d'un ensemble d'entités vers une ensemble de valeurs ou un produit cartésien de valeurs [Che76]. Son type, sa valeur initiale par défaut et sa cardinalité (par défaut 1..1) peuvent être définis. La cardinalité précise :

- si un attribut est obligatoire ou optionnel. S'il est obligatoire, l'attribut doit toujours avoir une valeur. Pour un article, l'attribut "mots-clés" est obligatoire car un article doit toujours avoir au moins un mot-clé. Par contre, "Article pourrait avoir un attribut optionnel "pageDepart" dont la valeur ne serait connue qu'après la mise en page.

- si un attribut est multi-valué, il peut avoir plusieurs valeurs simultanées. Par exemple, plusieurs mots-clés peuvent être proposés pour un article.

Dans le cas de l'attribut "mots-clés", il est noté entre crochets que sa cardinalité est 1..n c'est-à-dire qu'un article a plusieurs mots-clés (au moins 1).

Opération Une opération est une transformation qui peut être appliquée aux objets d'une classe. Elle a des paramètres qui sont écrits entre parenthèses après le nom de l'opération et qui sont séparés par des virgules. Les paramètres de l'opération (ou de sa signature) peuvent comporter des arguments qui ne sont pas des attributs de l'objet cible, mais par exemple, des attributs de l'objet appelant l'opération.

De plus comme une classe est à la fois une fabrique et un réservoir d'objets, les opérations sur une classe peuvent être divisées en deux catégories : les opérations sur un objet et celles sur l'ensemble des instances de la classe. Par exemple, l'opération "MiseEnPage" qui permet de numéroter les pages d'un article en fonction de la numérotation des autres articles d'une conférence est une opération sur l'extension d'"Article".

3.2.2 Association et concepts relatifs

Association Les associations relient deux ou plusieurs classes non nécessairement distinctes. Elles sont instanciées par des liens qui représentent les connexions entre des instances d'objets. Elles comportent des rôles (extrémité d'une association) et des cardinalités. Pour la gestion de conférences, "Ecrire" est une association entre "Chercheur" et "Article" (Fig 3.3). Le premier caractère d'une cardinalité représente le nombre minimum d'objets impliquées dans le rôle alors que le dernier décrit le nombre maximum de ces objets (le caractère "*" signifiant un nombre n quelconque supérieur à 1). Ainsi les cardinalités expriment qu'un chercheur écrit zéro ou plusieurs articles (0..*) publiés dans les conférences gérées et un article est écrit par un ou plusieurs chercheurs (1..*). Ici le rôle d' "Ecrire" allant de "Article" à "Chercheur" est nommé "auteur".

FIG. 3.3: Association entre "Chercheur" et "Article"

Classe associative Une association peut être modélisée comme une classe si on lui ajoute une description d'attributs et d'opérations. On parle alors de classe associative qui représente le n-uplet des instances mises en jeu dans l'association et des attributs. Par exemple, un chercheur peut s'inscrire à plusieurs conférences et pour chaque inscription, on enregistre la date de l'inscription et son mode de paiement. "Inscription" est donc une classe à part entière bien qu'elle lie "Chercheur" et "Conférence" (Fig. 3.4).

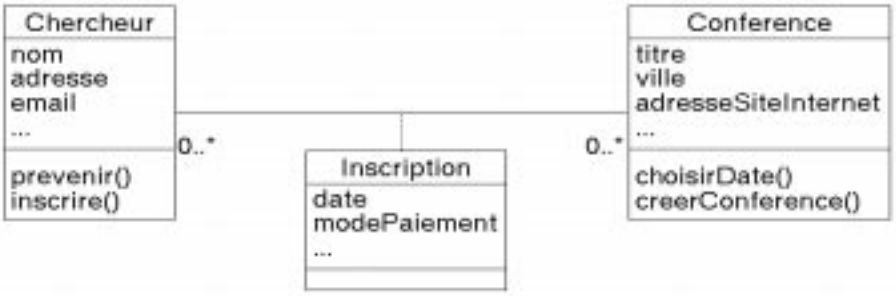


FIG. 3.4: Classe associative "Inscription"

Agrégation et composition Les termes d'agrégation et de composition sont employés en référence à une relation "tout-partie" entre un agrégat (le tout) et un composant (la partie). Ce concept a donné lieu à de nombreuses interprétations qui divergent suivant les caractéristiques jugées nécessaires pour parler de relation "tout-partie". Parmi ces caractéristiques, les propriétés suivantes sont souvent citées comme distinguant une relation "tout-partie" d'une association :

- **la dépendance du comportement** [SFLP98, Bru98, Civ93] ; cette propriété se réfère au fait qu'un agrégat doit avoir un cycle de vie chevauchant celui de son composant. Elle est aussi connue sous une forme plus restrictive, **la dépendance existentielle** [UML97, Ous97, Bla93] qui impose que la création ou la destruction d'une instance d'un agrégat entraîne la création ou la suppression de ses composants. Par exemple, une salle d'un centre de conférences n'a pas d'existence en dehors de celle de son centre : elle ne peut être créée que s'il a été créé et doit être détruite s'il est détruit.

- **l'exclusivité** qui est le fait de partager ou non des composants entre plusieurs agrégats [Lan95]. Ainsi une salle ne peut faire partie que d'un seul centre de conférences. Cette caractéristique est souvent citée [UML97, Lan95, SFLP98, Ous97, Civ93, KR94], mais cela sert en général à distinguer différents types de relation "tout-partie" [UML97, Civ93, Bla93]. Ainsi, en UML, l'exclusivité est une des propriétés qui différencie la composition de l'agrégation.

- **la propagation des propriétés (attributs, opérations, références)** [MC94, Bla93] qui sert à propager les valeurs des propriétés d'un agrégat à ses composants ; par exemple, si un centre de conférences est situé à une certaine adresse, ses salles ont la même valeur d'adresse.

- **la transitivité** ; cette propriété exprime que si un objet A est un composant d'un objet B et si l'objet B est un composant de l'objet C, alors l'objet A est un composant de C. Ainsi si une salle est composée de stands, ces stands sont aussi des parties du centre de conférences auquel appartient leur salle. Bien que la transitivité soit une caractéristique principale d'une relation "tout-partie" dans [UML97, SFLP98, Bla93], la transitivité ne fait pas l'unanimité puisque [MC94] juge qu'une relation "tout-partie" est en général non-transitive et que [Ous97] précise que la transitivité n'a de sens que pour les relations "tout-partie" de même type sémantique.

- **l'anti-symétrie** qui signifie que si un objet A est un composant d'un objet B alors l'objet B ne peut pas être un composant de A ; par exemple, une salle qui est une partie d'un centre de conférences, ne peut pas être composée de ce même centre. L'anti-symétrie est considérée comme l'une des caractéristiques principales dans [UML97, SFLP98, Bla93, MC94].

A partir de ces caractéristiques, nous distinguons deux types de relation "tout-partie" que nous nommons agrégation et composition pour nous conformer à la terminologie d'UML. Par rapport aux autres associations, **l'agrégation** a la particularité d'être **anti-symétrique et transitive**. Mais il n'existe pas d'exclusivité ou de dépendance existentielle déclarées entre un objet composant et son agrégat. Une agrégation est notée par un losange vide reliant la classe agrégat à ses composants.

Pour la gestion des conférences, un comité de programme est constitué de plusieurs membres (Fig. 3.5). Mais un chercheur peut faire partie de plusieurs comités de programme : il n'y a donc pas d'exclusivité. Enfin si un comité de programme est supprimé de la base de données, ses membres ne le sont pas puisqu'ils peuvent être enregistrés à d'autres titres que celui de membre de ce comité.

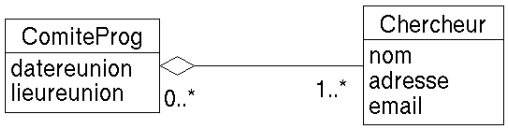


FIG. 3.5: Agrégation entre "ComiteProg" et "Chercheur"

Une composition est une agrégation qui possède deux propriétés supplémentaires : l'exclusivité et la dépendance existentielle. Elle dénote une appartenance forte entre un agrégat et son composant c'est-à-dire qu'un composant est inclus dans un agrégat unique (**exclusivité**), cet agrégat pouvant changer au cours du temps. Une composition se distingue aussi d'une agrégation par **la dépendance existentielle entre un composant et son agrégat** : un composant ne peut pas exister en dehors d'un agrégat et si l'agrégat est détruit, le composant doit aussi être supprimé. La notation d'une composition se différence de celle de l'agrégation par le fait que le losange est plein (noir).

Par exemple, un centre de conférences est composé de plusieurs salles, mais une salle ne fait partie que d'un seul centre (Fig. 3.6). De plus, l'existence d'une salle dépend de celle de son centre de conférences car si on détruit un centre, il semble normal d'en détruire aussi les salles.

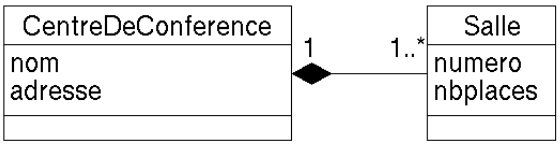


FIG. 3.6: Composition entre "CentreDeConferences" et "Salle"

3.2.3 Généralisation-spécialisation

Nous considérons la généralisation-spécialisation telle qu'elle est définie dans [SS77]. La spécialisation est définie comme une relation qui, à une classe dite super-classe, fait correspondre une classe plus affinée, dite sous-classe. Les attributs et les opérations communs à un groupe de sous-classes sont liés à la super-classe et partagés par chaque sous-classe. On dit qu'une sous-classe **hérite des propriétés** (attributs et opérations) de sa super-classe. Une sous-classe a aussi des propriétés spécifiques qui lui sont propres. Tout objet de la sous-classe est objet de la super-classe avec éventuellement des propriétés supplémentaires. Ainsi l'ensemble des objets d'une sous-classe est inclus dans l'ensemble des objets de la super-classe. Il existe donc une relation hiérarchique entre les classes. Les sous-classes peuvent être non-disjointes c'est-à-dire qu'un même objet appartient à deux familles de spécialisation.

La spécialisation est notée par un triangle reliant une super-classe à ses sous-classes. Le sommet du triangle pointe sur la super-classe. Par exemple (Fig. 3.7), lors d'une conférence, une présentation peut être soit une présentation d'article (sous-classe "PresentationDArticle"), soit une présentation invitée (sous-classe "PresentationInvitee"). Une présentation invitée a en plus des attributs de "Presentation" ("date", "heureDebut", "duree" et "media"...) l'attribut "cout" qui représente les frais engagés pour la présentation. "changerHoraire" et "changerSalle" sont deux opérations de "Presentation" dont héritent "PresentationDArticle" et "PresentationInvitee". "PresentationInvitee" a en plus une opération spécifique : "Chiffrer" qui évalue le coût d'une présentation invitée.

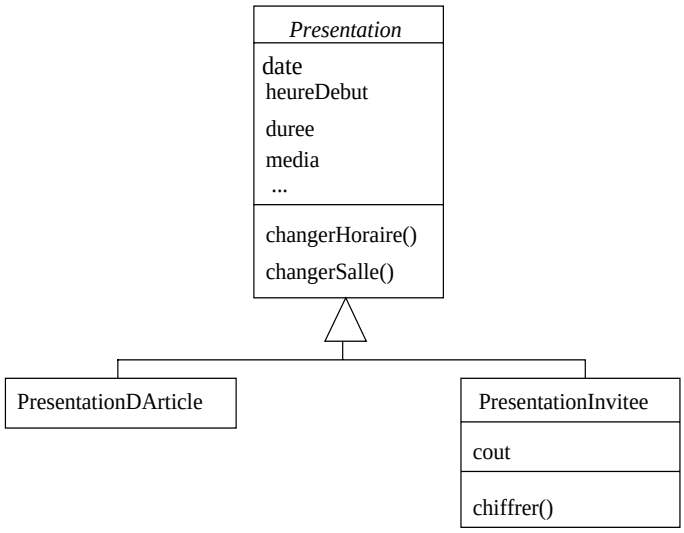


FIG. 3.7: Spécialisation de "Presentation"

Une super-classe peut ne pas avoir d'instances directes, on dit alors qu'elle est **non-instanciable ou abstraite**. Elle sert à organiser les propriétés communes à plusieurs classes (ses sous-classes). Ce concept se dénote par l'écriture du nom de la classe en italique. Par exemple, "Presentation" est une classe abstraite puisqu'une présentation est soit invitée, soit d'article.

Lors d'une spécialisation, la même opération peut s'appliquer à plusieurs classes. Elle peut prendre des formes différentes dans les différentes classes c'est-à-dire que son implantation varie suivant la classe. Cette opération est dite **polymorphe**. Dans ce travail, nous considérons qu'une opération d'une super-classe peut être redéfinie dans une sous-classe seulement si sa signature reste compatible avec celle de la super-classe.

3.2.4 Expression de contraintes

Les concepts exposés précédemment permettent de représenter les relations statiques entre les différents objets du système. Mais toutes les contraintes de domaine ne peuvent pas être décrites ainsi. C'est pourquoi, un modèle est généralement complété par l'expression d'annotations en langue naturelle. Or comme nous l'avons évoqué dans la section 2.2.1, la langue naturelle est imprécise et ambiguë. Plusieurs propositions ont donc été faites afin de représenter les contraintes statiques sous une autre forme. Les premières [NECH92, Ou98, Ken97] proposent d'étendre les modèles pour pouvoir exprimer graphiquement les contraintes. Les secondes [CD94, WK98] prévoient d'ajouter en commentaires aux diagrammes des contraintes exprimées dans un formalisme proche de la logique du premier ordre. Une troisième [KH99] présente un moyen de combiner des contraintes graphiques et textuelles.

Contrairement aux premières propositions, la deuxième approche a pour avantage de ne pas surcharger les notations graphiques et de ne pas limiter les contraintes à quelques types pré-définis. Par rapport à la dernière approche, elle permet de représenter les contraintes de manière uniforme, même si l'utilisation de notations graphiques est parfois plus intuitive.

Parmi les langages d'expression de contraintes, nous nous sommes intéressée à **OCL** (Object Constraint Language [WK98]) qui a été développé pour annoter les diagrammes UML. OCL peut servir à spécifier des invariants, décrire des pré et post-conditions des opérations ou comme langage de navigation sur un diagramme UML. Nous nous intéressons plus particulièrement à son rôle d'expression de contraintes sur un modèle objet. Dans ce cadre, son expressivité se limite aux contraintes statiques : OCL est en particulier incapable de décrire des contraintes dynamiques ou temporelles. Il a néanmoins pour avantage d'apporter des notations précises, mais plus facilement compréhensibles que les symboles mathématiques de Z ou B.

OCL est issu de Syntropy [CD94] dans laquelle Z est utilisé pour écrire les contraintes. Il est donc basé sur la théorie des ensembles et la logique du premier ordre, mais pour faciliter la compréhension les symboles mathématiques sont remplacés par des opérations pré-définies. Dans le cas de la figure 3.2, pour spécifier que l'ensemble des mots-clés d'un article ne doit pas supérieur à 5, -> permet d'accéder à l'opération "size" de l'ensemble "motscles" qui correspond à l'opérateur de cardinalité # de Z :

Article
self.mots-cles -> size <= 5 ;

Comme le montre l'expression précédente, une contrainte OCL est définie dans **le contexte** d'une instance d'un type spécifié. Le contexte doit être un élément du diagramme UML annoté. Ici le contexte est la classe "Article". "self" se réfère alors à l'instance contextuelle. Si le contexte est "Article", "self" correspond à une instance d'article.

Les propriétés (attributs, opérations, associations) d'un contexte sont accessibles en utilisant la notation "." suivi du nom de la propriété. Par exemple "self.mots-cles" est la valeur de la propriété "motscles" de l'article considéré. Le même principe sert à **naviguer** parmi les associations. A partir d'un objet spécifié, une association peut être suivie pour se référer à d'autres objets et à leurs propriétés. La navigation d'une association s'effectue en utilisant le nom du rôle opposé. Si la cardinalité du rôle est au maximum 1, la valeur de l'expression est un objet, sinon ce peut être un ensemble, une séquence etc. Par exemple, dans le contexte d'un article, "self.auteur" est l'ensemble des auteurs de cet article.

Nous ne détaillons pas ici toutes les primitives d'OCL, mais nous préciserons leur sens lors de leur utilisation. De plus, une description complète est disponible dans [WK98].

3.3 Notations formelles utilisées : Z et Object-Z

3.3.1 Z

Z est un langage de spécification formelle à base de modèles. A l'origine, il a été conçu à Grenoble par Jean-Raymond Abrial [Abr77]. Depuis, il a été développé à Oxford et sa standardisation est en cours à l'ISO [Z95].

Z est basé sur la logique du premier ordre et sur la théorie des ensembles. En Z, le système est décrit sous forme d'une collection de variables et d'une liste d'opérations qui peuvent changer la valeur de ces variables. Les variables représentent l'état interne du système qui peut être accédé ou modifié uniquement par les opérations. Il y a donc encapsulation de l'état du système.

La notion d'ensemble

Définition Les spécifications Z sont basées sur la notion d'ensemble, qui est proche de celle de type en langages de programmation. Un ensemble est une collection d'entités distinctes qui ont toutes le même type.

Comment définir un ensemble ? Z ne propose pas de nombreux **types prédéfinis**. En fait, il fournit seulement l'ensemble des entiers. Pour disposer de plus de types, on doit introduire des **ensembles de base**. Un ensemble de base correspond à un ensemble disjoint des autres ensembles c'est-à-dire qu'un élément ne peut pas appartenir à deux ensembles de base dans une spécification. Rien ne précise le nombre d'éléments de l'ensemble, les notations particulières ou les opérations qui

peuvent être faites sur ces éléments. Introduire un ensemble de base assure juste qu'il existe un tel ensemble d'éléments types. Par exemple, la déclaration suivante introduit les ensembles de base DATE et NOM :

$$[DATE, NOM]$$

Une autre manière d'introduire un ensemble est de décrire ses éléments en en donnant la liste. Par exemple, Booleen est un **type énuméré** qui a deux valeurs Vrai ou Faux :

$$Booleen ::= Vrai \mid Faux$$

A partir des types pré-définis, des ensembles de base et des types énumérés, de nouveaux ensembles peuvent être construits par les opérations ensemblistes : inclusion, extension d'éléments d'un type de base...

- **sous-ensemble**

  La construction d'un sous-ensemble est utilisée pour introduire un ensemble d'éléments dont le type est déjà connu.

  Par exemple, on peut définir l'ensemble constant des prénoms PRENOM comme un sous-ensemble fini, non-vide des noms :

  $$PRENOM : \mathbb{F}\, NOM$$

- **ensemble défini en extension**

  A partir d'un ensemble donné, il est toujours possible de définir un sous-ensemble en donnant son extension. L'ensemble des entiers naturels pairs inférieurs ou égaux à 10 est, par exemple :

  $$NatIEDix == \{0, 2, 4, 6, 8, 10\}$$

  NatIEDix peut être vue comme une "macro".

- **intervalle**

  Il est aussi possible de définir un sous-ensemble en spécifiant l'intervalle considéré sur des entiers. Ceci suppose l'existence d'une relation d'ordre sur cet ensemble. NOTE est ainsi défini comme un sous ensemble des entiers.

  $$NOTE == 0..20$$

- **ensemble défini en compréhension ou en intension**

  Une dernière façon de définir un sous-ensemble d'un ensemble consiste à le caractériser par un prédicat cohérent avec le type de l'ensemble de base. Un tel ensemble est défini par trois éléments :

  $$\{decl \mid pred \bullet expression\}$$

  où decl introduit une variable d'un ensemble déjà défini
  pred est un prédicat qui filtre les éléments de cet ensemble
  expression est une fonction à appliquer aux éléments résultants

  Par exemple, l'ensemble de nombre pairs positifs est représenté par :

  $$PairPos == \{n : \mathbb{Z} \mid n \geqslant 0 \bullet 2 * n\}$$

Relations entre ensembles Une relation n-aire sert à décrire la façon dont les éléments des ensembles sont liés. Le lien R entre les ensembles A, B et C correspond à l'ensemble des n-uplets constitués d'un élément de A, d'un élément de B et d'un élément de C :

$R : \mathbb{P}(A \times B \times C)$

Les relations les plus souvent utilisées sont les relations binaires qui expriment le lien entre des paires d'éléments. En Z, on appelle **relation** un ensemble de paires ordonnées. Si A et B sont des ensembles, alors $A \leftrightarrow B$ dénote l'ensemble des relations entre A et B, le symbole $\leftrightarrow$ étant une abréviation du sous-ensemble de produit cartésien $\mathbb{P}(A \times B)$.

Par exemple, il existe une relation qui lie un chercheur a son nom (Fig. 3.8). Un chercheur a un nom, mais un nom peut correspondre à plusieurs chercheurs. En Z, cette relation est déclarée par :

$$nommer : Chercheur \leftrightarrow NOM$$

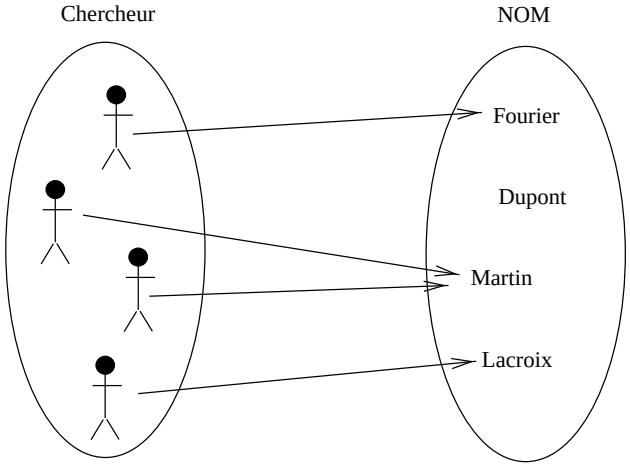


FIG. 3.8: Une relation entre chercheurs et noms

La relation étant un cas particulier de produit cartésien, elle peut être considérée comme un ensemble de paires. Soient *C1, C2, C3, C4* quatre chercheurs et *Fourier, Dupont, Martin et Lacroix* des noms :

$$C1, C2, C3, C4 : Chercheur$$
$$Fourier, Dupont, Martin, Lacroix : NOM$$

L'extension de *nommer* est l'ensemble de paires suivant :

$\{(C1, Fourier), (C2, Martin), (C3, Martin), (C4, Lacroix)\}$

Une syntaxe alternative pour les paires d'éléments est la notation **"maplet"** :

$\{C1 \mapsto Fourier, C2 \mapsto Martin, C3 \mapsto Martin, C4 \mapsto Lacroix\}$

Pour pouvoir extraire les informations contenues dans une relation, des fonctions de base sont incluses dans le langage mathématique. Les exemples les plus simples sont les fonctions **domaine** (dom) et **codomaine** (ran). Pour la relation *nommer*, le domaine est l'ensemble des éléments de *Chercheur* qui sont liés à un ou plusieurs

éléments de *NOM* c'est-à-dire l'ensemble $\{C1, C2, C3, C4\}$. Le codomaine de *nommer* est l'ensemble des noms qui sont liés à au moins un chercheur, soit Fourier, Martin, Lacroix.

Certains liens peuvent aussi être modélisés comme des cas particuliers de relations. Si un élément du domaine a au plus une image, alors la relation entre les deux ensembles est une **fonction** (notée $\rightarrow$). Suivant les propriétés des liens entre les ensembles, les fonctions peuvent être partielles, injectives, surjectives ou bijectives. Les caractéristiques et les notations de ces différents types de fonctions sont données dans l'annexe A.

Ainsi *nommer* est une fonction (une chercheur a un seul nom) totale (tout chercheur a un nom). Ceci est noté par $\rightarrow$.

$$nommer : Chercheur \rightarrow NOM$$

La notion de schéma

Définition Z est enrichi par une construction modulaire très puissante : **le schéma**. Un schéma Z est une structure de spécification de données ou de traitements. Il est caractérisé par un nom. Comme pour les variables et les constantes, le nom d'un schéma a un type [Z95] dont les constituants sont ceux cités dans la partie déclaration et qui vérifient les prédicats qui portent exclusivement sur les variables déclarées. La déclaration constitue le lexique des objets locaux au schéma qui est la définition de l'environnement d'application des prédicats de la deuxième partie. Les prédicats précisent les contraintes portant sur les variables ou la relation entre les états initiaux et finaux d'une opération.

$$\begin{array}{|l} \hline < NomDeSchema > \\ < DECLARATIONS > \\ \hline < PREDICATS > \\ \hline \end{array}$$

Schéma d'état et inclusion de schéma Un schéma d'état sert à exprimer les caractéristiques structurelles d'un modèle en décrivant l'état du système. Il consiste en la déclaration de variables et de prédicats qui contraignent ces variables. Par exemple, on introduit un schéma *Article* qui a comme variables *titre* et *mots-cles* et un prédicat qui précise que l'ensemble des mots-clés ne peut pas être vide :

$$\begin{array}{|l} \hline Article \\ titre : TITRE \\ motscles : \mathbb{F}\, MOT \\ \hline motscles \neq \varnothing \\ \hline \end{array}$$

Les schémas d'état peuvent importer d'autres schémas : le nom du schéma inclus fait alors partie des déclarations du schéma l'incluant. Par exemple, *Article2* importe le schéma *Article* et ajoute la contrainte que le nombre de mots-clés doit être inférieur ou égal à cinq :

Article2

$$Article$$

$$\#motscles \leq 5$$

L'inclusion d'un schéma I dans un schéma S correspond à l'inclusion des déclarations de I dans celles de S et les prédicats de I sont ajoutés à ceux de S, formant ainsi une ou plusieurs conjonctions supplémentaires. Les variables du schéma inclus sont donc disponibles dans l'environnement engloblant et vérifiant les prédicats inclus. Par exemple, le schéma *Article2* précédent est équivalent à :

Article2

$$titre : TITRE$$
$$motscles : \mathbb{F}\, MOT$$

$$motscles \neq \varnothing \wedge \#motscles \leq 5$$

Schéma d'opération Nous avons vu qu'un schéma peut servir à décrire un type. Une deuxième utilisation du concept de schéma est la description d'opérations. Pour ce cas, il faut définir la relation entre l'état avant l'opération et l'état après. Les opérations ont des variables d'entrée (notées par ?) et des variables de sortie (notées par !). Elles sont spécifiées par la relation entre ces deux types de variables et les deux états des instances du type : état initial (avant l'opération) et l'état final (après l'opération dénoté par '). Dans le cas d'une opération modifiant *Article*, l'état d'article est inclus avant et après l'opération :

ModifierArticle

$$Article$$
$$Article'$$

$$\ldots$$

Article' veut dire que toutes les variables de *Article* sont décorées d'un prime pour représenter les valeurs de ces variables après l'opération. La partie prédicat du schéma caractérise alors l'opération en décrivant son effet sur ces valeurs. Par exemple, on spécifie l'opération *AjouterMotcle* qui ajoute un mot-clé à l'ensemble des mots-clés d'un article. Le schéma *Article* est inclus dans le schéma de l'opération car ses instances vont être modifiées. On introduit une variable d'entrée *m ?* qui représente le nouveau mot-clé. *m ?* est ajouté aux mots-clés. L'autre variable (*titre*) d'un article n'est pas modifiée.

AjouterMotcle

$$motscles, motscles' : \mathbb{F}\, MOT$$
$$titre, titre' : TITRE$$
$$m? : MOT$$

$$motscles \neq \varnothing \wedge motcles' \neq \varnothing \wedge motscles' = motscles \cup \{m?\}$$
$$titre' = titre$$

Par convention, pour inclure deux copies d'un même schéma, l'un d'eux étant décoré d'un prime, on utilise la notation Δ. L'opération *ModifierArticle* devient alors :

AjouterMotcle

$\Delta Article$
$m? : MOT$

$motscles' = motscles \cup \{m?\} \land titre' = titre$

Il existe aussi une autre convention pour signifier que les variables du schéma inclus ne changent pas : Ξ précédant le nom du schéma.

Pré-condition Une opération est applicable sous certaines conditions : il doit exister un état final et si elles existent, des sorties satisfaisant la relation spécifée par l'opération. Ainsi Z propose un opérateur appelé précondition de schéma noté pre qui extrait **la plus faible pré-condition** d'une opération. Pour une opération *Op* qui modifie l'état *State* et qui a *Outs* comme paramètres de sortie, pre *Op* est défini comme [PST96] :

$\exists$ State' ; Outs ! $\bullet$ Op

Le prédicat résultant est une condition sur l'état initial et les paramètres d'entrée de l'opération. Si cette condition est vérifiée, l'existence de l'état final et des paramètres de sortie est garantie si l'opération se termine.

L'opérateur pre appliqué a l'opération *AjouterMotcle* donne :

$\exists$ titre' : TITRE ; motscles' : $\mathbb{F}$ MOT $\bullet$ AjouterMotcle

Ce prédicat correspond à l'expression suivante qui est toujours vraie :

$titre \in TITRE \land motscles \in \mathbb{F}\, MOT \land m? \in MOT$
$\land\ motscles \cup \{m?\} \in \mathbb{F}\, MOT \land \neg\ motscles = \{\}$

En d'autres termes, la pré-condition exprime le type du paramètre d'entrée et des variables d'état et la contrainte invariante sur les mots-clés.

Combinaison de schémas Un schéma peut aussi être défini en combinant d'autres schémas : pour cela, Z introduit un langage d'opérateurs logiques de schémas qui permet de définir de nouveaux schémas en utilisant des schémas existants. Ces opérateurs sont **la conjonction, la disjonction, la négation, la quantification et la composition** (cf. annexe A). Ils peuvent être appliqués à la fois aux schémas d'état et d'opération : un schéma d'état encapsule des variables dont les valeurs représentent l'état du système et un schéma d'opération peut être considéré comme une relation entre deux états.

Par exemple, la disjonction de schémas correspond à la fusion des déclarations et à la disjonction des prédicats des différents schémas. Supposons que S et T soient deux schémas où P et Q sont des prédicats sur les variables correspondantes :

S

$a : A$
$b : B$

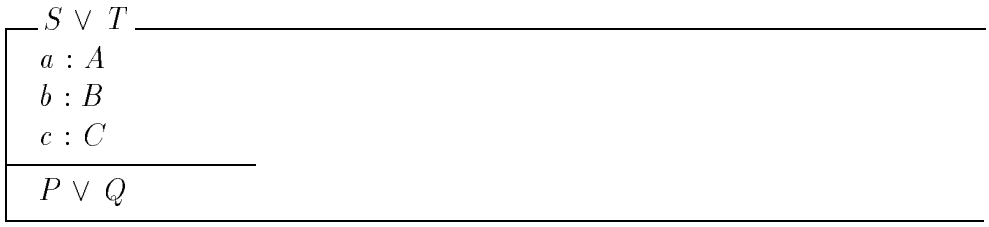
P

T

$b : B$
$c : C$

Q

Quand S et T sont combinés par une disjonction ($S \vee T$), cela correspond à créer un nouveau schéma :



Si une même variable est déclarée dans les deux schémas, comme c'est le cas pour b ci-dessus, les types doivent être identiques pour pouvoir mettre en commun les variables communes sinon, le schéma $S \vee T$ est indéfini.

Conclusion

Bien qu'il existe deux types de schémas en Z (les schémas d'état et les schémas d'opération) et de nombreuses façons de les combiner, il n'est pas possible de regrouper des opérations qui affectent un schéma d'état, ce qui correspond à la notion d'objet ou de type abstrait. Nous nous sommes donc intéressée à une extension orienté objet de Z qui permet de regrouper un état et ses opérations.

3.3.2 Object-Z

Object-Z est une extension de Z aux concepts à objets. Il a été conçu à l'Université du Queensland (Australie) sous l'impulsion de Roger Duke et G. Gordon [DKRS91]. L'objectif initial est de structurer les spécifications Z. Pour cela, Object-Z introduit la notion de classe qui encapsule un schéma d'état avec ses opérations. Des classes complexes peuvent être spécifiées à partir de classes plus simples en utilisant en particulier l'héritage.

Classe

L'extension majeure d'Object-Z est la classe. Syntaxiquement, une classe est constituée de :

- **la liste de visibilité**
  Elle sert à préciser quels sont les attributs et les opérations accessibles depuis une autre classe. Si elle est omise, tous les attributs sont visibles.

- **des classes héritées**
  Ce sont les noms des super-classes qui doivent être héritées dans cette sous-classe. Une sous-classe comprend alors toutes les caractéristiques de ses super-classes (attributs, opérations) et leurs invariants.

- **la définition de types et de constantes**
  On définit ici les types et les constantes locaux à la classe suivant la syntaxe de Z. Leur champ de définition est limité à la classe dans laquelle ils sont déclarés.

- **un schéma d'état**
  Il n'est pas nommé, mais il est constitué comme les schémas Z de déclarations de variables et d'un invariant d'état.
  Les déclarations sont divisées en variables primaires et variables secondaires ([DRS94]). Les variables secondaires peuvent être définies à partir des variables primaires et elles sont implicitement disponibles pour des modifications dans n'importe quelle opération de la classe. Par exemple, l'âge d'une personne est une variable secondaire qui peut être trouvée à partir d'une date de naissance et de la date du jour.
  Les variables et les constantes sont appelées les attributs d'une classe. Chaque attribut ne peut prendre que des valeurs cohérentes avec l'invariant de classe et des éventuelles super-classes.

- **un schéma d'état initial**
  Il spécifie l'état initial de la classe. Il est noté par le nom INIT.

- **des schémas d'opérations**
  Dans ces schémas, seuls les attributs modifiés figurent dans la liste du Δ, les autres ne changent pas.

- **un invariant d'histoire**
  L'ensemble des histoires représentant une classe est contraint par un invariant qui peut être exprimé en logique temporelle.
  Actuellement cette partie n'est pas gérée par le vérificateur de type d'Object-Z ([Joh96]) et semble avoir été abandonnée.

NomClasse[ParamètresGénériques]

liste de visibilité
classes héritées
définitions de types
définitions de constantes
schéma d'état
schéma d'état initial
schémas d'opérations

---

invariant d'histoire

Par exemple, on définit la classe *ArticleExt* (on suppose que le type ARTICLE a été défini précédemment). Le schéma d'état de *ArticleExt* a un attribut *Articles* qui représente un ensemble d'articles et un prédicat qui exprime que deux articles doivent avoir des titres différents. Dans l'état initial, l'ensemble est vide et on peut à l'aide des opérations *EnregisterArticle* ajouter un article à l'ensemble des articles.

ArticleExt

$$Articles : \mathbb{F}\, ARTICLE$$

$$\forall\, a1, a2 : Articles \mid a1 \neq a2 \bullet a1.titre \neq a2.titre$$

INIT

$$Articles = \varnothing$$

EnregistrerArticle

$$\Delta(Articles)$$
$$a? : ARTICLE$$

$$a? \notin Articles$$
$$Articles' = Articles \cup \{a?\}$$

Une classe n'est pas seulement une extension syntaxique. Dans une première version d'Object-Z, les instances d'une classe étaient les objets de la classe ([DKRS91]). Cependant, des travaux sur le langage ont mené à modifier la sémantique d'une classe ([DRS94]) : un schéma de classe définit un type dont les instances sont des références à des objets. Donc quand une variable a est déclarée être du type de la classe A, la valeur de a est un identifiant d'objet ([Gri95]).

Comme en Z, l'inclusion de schéma est possible en Object-Z. Un schéma ou une classe peuvent contenir un schéma, mais une classe ne peut pas être incluse dans un schéma ou dans une autre classe.

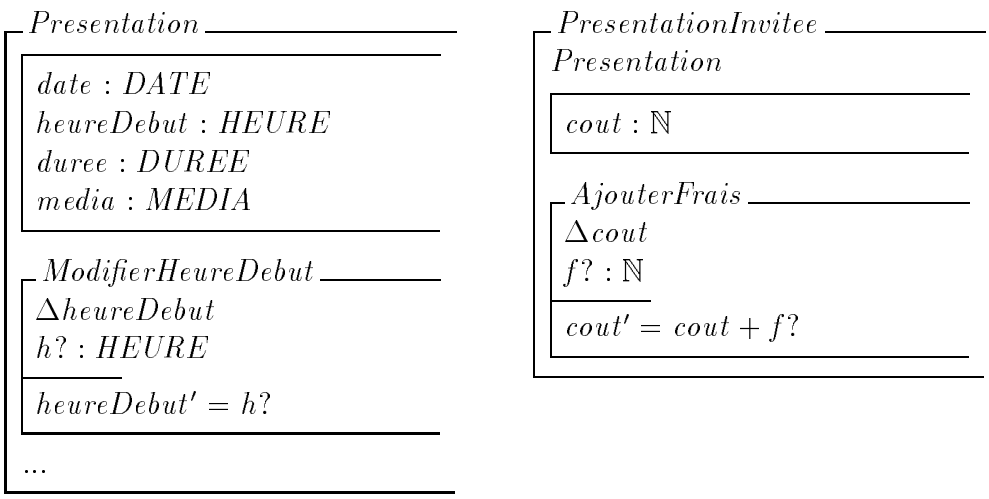
Héritage

L'héritage permet de réutiliser des spécifications. Comme nous l'avons vu précédemment, il est représenté par l'inclusion du nom de la super-classe dans la sous-classe. Il revient en fait à fusionner les éléments de la super-classe et les éléments propres à la sous-classe :

- les définitions de types et de constantes sont fusionnées ;
- les schémas de même nom sont joints ;
- les schémas définis par des expressions sont "évalués" avant l'héritage ;
- les invariants d'histoire sont joints.

Pour la gestion de conférences, une présentation se spécialise en présentation invitée qui a pour caractéristique propre son coût. La classe *Presentation* a comme attributs l'heure de début, la durée et le média utilisé. On introduit ensuite la classe *PresentationInvitee* qui hérite de la classe *Presentation*. Tous les attributs (resp. opérations) de *Presentation* sont aussi des attributs (resp. opérations) de *PresentationInvitee*. *PresentationInvitee* a pour attribut spécifique *cout* qui représente les frais engendrés par la présentation et une opération *AjouterFrais* qui ajoute des frais

au coût de la présentation.



L'héritage en Object-Z comporte aussi la notion de **polymorphisme** ([DKRS91]). $\downarrow$ C représente l'ensemble des objets possibles de C et de ses sous-classes. Etant donné la nouvelle sémantique des classes, on peut supposer que $\downarrow$ C est maintenant l'ensemble des références aux objets possibles de C et de ses sous-classes. c : $\downarrow$ C est donc une référence à un objet de C ou de l'une de ses sous-classes. Si une opération est appliquée à c, celle qui est appliquée est celle de la classe effective de c et non celle de C. Une opération héritée d'une super-classe peut être redéfinie ou supprimée. Cela est indiqué par les mots-clés *redefine* et *remove*.

Opérateurs spécifiques

Object-Z ajoute des opérateurs de schéma à Z :

- **le choix indéterministe** S [] T qui représente le choix non-déterministe d'une opération parmi celles dont les pré-conditions sont satisfaites ;
- **l'accessibilité des déclarations** $S \bullet T$ qui dénote l'enrichissement de l'environnement de l'opération T avec le contexte de S ;
- **l'opérateur parallèle** $S \parallel T$ qui se comporte comme la conjonction, mais qui égalise aussi les paramètres d'entrées et de sortie ayant le même nom.

Conclusion

Object-Z fournit les principaux concepts (encapsulation d'opérations, héritage etc...) de l'approche orientée objet. Mais sa sémantique n'est pas toujours bien définie. C'est par exemple le cas pour l'héritage qui de plus, n'est pas "propre" car une opération d'une sous-classe peut être totalement redéfinie ou supprimée. Nous n'utiliserons donc pas ces possibilités dans notre travail.

3.4 Conclusion

Dans ce chapitre, nous avons présenté le projet dans lequel s'inscrit ce travail. Dans ce cadre, nous avons choisi de nous intéresser aux modèles semi-formels objets ainsi qu'aux langages formels Z et Object-Z.

Nous avons introduit les principaux concepts des notations utilisées pour notre travail. Pour les modèles semi-formels, nous avons étudié les concepts du modèle objet. Concernant le modèle objet, les concepts les plus répandus (classe, association, héritage, agrégation et composition) ont été présentés. Nous avons éclairci certains points imprécis en particulier pour l'agrégation et la composition. Un autre concept inspiré du standard UML, le langage d'expression de contraintes, a également été abordé.

Enfin nous avons introduit le langage formel Z et une de ses extensions orientées objet Object-Z. Z étant basé sur la théorie des ensembles et la logique du premier ordre, nous avons présenté les notions d'ensemble et de schéma qui structure la spécification en groupant des variables et leurs contraintes écrites en logique. Pour Object-Z, nous avons abordé les concepts qui lui sont spécifiques par rapport à Z c'est-à-dire la classe, l'héritage et ses trois opérateurs spécifiques de composition de schémas.

Chapitre 4

Travaux de traduction similaires

Ce chapitre décrit les travaux déjà effectués concernant la traduction de modèles objet vers des langages formels orientés modèle ou orientés objet. Pour chaque concept, nous décrivons les différentes alternatives de traductions proposées. Enfin pour faciliter la comparaison des différents travaux, nous résumons l'apport de chacun d'entre eux.

Ce chapitre n'est pas indispensable à la compréhension de la suite du manuscrit et peut être oublié lors d'une première lecture.

4.1 Travaux étudiés

Les articles que nous étudions concernant la traduction de modèles objet vers un langage formel sont :

- [FB97, FBLP97] qui dérivent du Z à partir du modèle objet de Fusion ;
- [FBLPS97, SF97] qui effectuent le même travail à partir du diagramme de classes d'UML ;
- [NR94] qui part d'un modèle entité-relation étendu pour obtenir des spécifications en B ;
- [FL96, FLN96a, FLN96b, Ngu98] qui traduisent le modèle objet en des spécifications B ; par la suite, nous citons seulement [Ngu98] qui récapitule toutes les propositions faites dans ces articles ;
- [MS99] qui s'intéresse à la traduction des modèles d'OMT en B ;
- [Lan95, LG96, LHW96] qui dérivent des spécifications B, Z++ ou VDM++ à partir de modèles objet ; par la suite, nous ne citons plus [LG96] qui est tiré de [Lan95] ;
- [AS97] qui propose une méthode inspirée de Fusion et d'Object-Z ;
- [KC99] qui dérive le diagramme de classes d'UML en Object-Z.

Les modèles objet de ces travaux sont divers et peuvent différer de celui que nous avons choisi. En cas de nécessité, la sémantique des concepts formalisés est donc explicitée.

4.2 Classe

Soit "Classe" une classe ayant pour attribut "a" de type "T" et l'opération "Op".

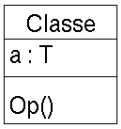


FIG. 4.1: Classe

Comme nous l'avons évoqué dans la partie 3.2, une classe joue deux rôles : celui de fabrique d'objets et celui d'entrepôt d'objets. Elle peut donc être définie en extension (ensemble de toutes les instances existantes) ou en intension (description des propriétés communes aux instances possibles) ([Hal94]). Pour représenter l'intension et l'extension d'une classe, il faut décrire les attributs, les opérations et les objets existants de cette classe. Nous étudions les représentations formelles proposées pour ces concepts, puis nous discutons des structures dans lesquelles elles peuvent être incluses c'est-à-dire de la modularité des spécifications formelles produites.

4.2.1 Représentation des attributs d'une classe

La proposition la plus naturelle pour représenter les attributs consiste à les décrire par **des variables dont le type est celui de l'attribut**. Par exemple, un attribut "a" de type "T" donne lieu à une variable a de type T [FB97, FBLPS97, FBLP97, SF97, AS97, KC99, Lan95] :

$a : T$

Dans le cas de l'utilisation d'un langage formel non-orienté objet, un attribut particulier est souvent introduit : il s'agit de l'identité d'objet [Ngu98, MS99, LHW96, FB97, FBLPS97, FBLP97, SF97]. Elle est dans ce cas spécifiée explicitement alors que dans l'approche objet, elle demeure toujours implicite. En Z par exemple, un type de base est déclaré pour représenter ces identités et une variable de ce type est ajoutée à la déclaration des attributs.

$[IDClasse]$

CLASSE

$id : IDClasse$

... [attributs de "Classe"]

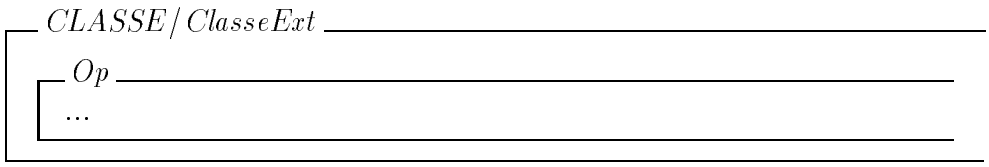
Une approche alternative à la représentation des attributs [NR94, Ngu98, MS99, LHW96] consiste à utiliser les identités d'objets : chaque attribut de la classe est

vu comme **une relation ou une fonction entre l'identité et la valeur d'une propriété de l'instance**. Soit *IDClasse* l'identité d'un objet de la classe "Classe", l'attribut "a" de type "T" est représenté par :

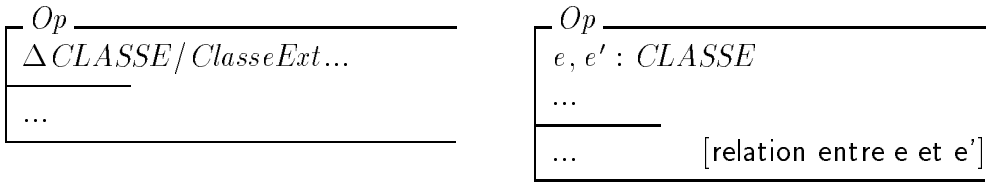
$a : ID \leftrightarrow T$

4.2.2 Représentation des opérations d'une classe

A partir d'un modèle objet seul tel que celui d'OMT, de Fusion ou d'UML, seules les signatures des opérations peuvent être traduites en langage formel. Pour les langages formels dotés du concept d'opération tels que B ou Object-Z, une opération "Op" donne lieu à une opération *Op* [KC99, Lan95, MS99]. Par exemple, en Object-Z, "Op" se traduit par un schéma d'opération dans l'une des classes représentant la classe.



En Z, une opération est décrite par un schéma d'opération [FBLPS97]. Mais la représentation des opérations de [FBLPS97] diverge des schémas traditionnels d'opération en Z (schéma de gauche) par le fait que le Δ n'est pas utilisé et qu'il y a une variable de type *CLASSE* pour l'état avant *e* et une autre *e'* pour l'état après l'opération (schéma de droite). Cela permet ensuite de représenter un objet par le lien entre son identité et le schéma d'opération (cf. page suivante). Ce choix ne se justifie que par son aspect technique, aspect qui rend la solution plus difficile à comprendre.



4.2.3 Représentation des objets existants

L'ensemble des objets existants d'une classe est défini **soit par l'ensemble des éléments du type de la classe, soit par l'ensemble des identités des objets de cette classe**.

La première approche nécessite d'avoir préalablement défini la classe comme un type en créant une structure particulière qui peut être un schéma Z [SF97, FB97, FBLP97], une classe Object-Z [AS97, KC99] etc. Si on nomme *CLASSE* la structure représentant le type d'une classe, les objets existants de cette classe sont spécifiés par l'ensemble ($\mathbb{P}$) des éléments existants du type *CLASSE* :

$classes : \mathbb{P}\ CLASSE$

Dans la deuxième approche, les objets existants se réduisent à l'ensemble de leurs identités. En B [LHW96, Ngu98, MS99], les identités des objets possibles sont

représentées par un ensemble *IDClasse* et les identités d'objets existants donnent lieu à une variable *classes* qui doit être incluse dans *IDClasse*.
Machine ...
Sets IDClasse
Variables classes
Invariant classes $\subseteq$ IDClasse

De façon similaire, en Z [FBLPS97], un type donné *IDClasse* décrit les identités possibles et l'ensemble des objets existants est un ensemble variable d'éléments de ce type :

$[IDClasse]$

$$\begin{array}{|l} \hline classes : \mathbb{P}\, IDClasse \\ \ldots \\ \hline \end{array}$$

4.2.4 Modularité des spécifications formelles

Certains travaux prennent en compte systématiquement l'intension et l'extension d'une classe ([LHW96, FBLPS97, AS97, Ngu98, KC99, MS99]) définissant pour chacune d'elles deux types de représentation. D'autres travaux ne décrivent que la définition en intension ([NR94, Lan95]). R.B.France et J-M.Bruel ([FB97]) pensent qu'une telle distinction n'est pas toujours nécessaire car les deux interprétations capturent le fait qu'une classe caractérise une collection d'instances. Ils recommandent donc de représenter explicitement l'aspect extensionnel seulement en cas de nécessité.

Pour les travaux représentant à la fois l'intension et l'extension d'une classe, certains utilisent une même structure pour ces deux aspects, alors que d'autres définissent des structures distinctes. Parmi les propositions utilisant **différentes structures pour décrire une classe**, l'une d'elles [FBLPS97] donne systématiquement lieu à des structures distinctes alors que d'autres regroupent les objets des différentes classes dans une même structure. Dans [FBLPS97], France et al. utilisent un schéma Z *ClasseAttr* pour représenter les attributs, un schéma Z *Op* pour chaque opération et un autre schéma *ClasseExt* qui décrit les instances (*instances*) sous forme d'identités d'objets et des fonctions liant chaque instance à ses données (*attributs*) et ses opérations (*op*) :

ClasseAttr
$$\begin{array}{|l} \hline a : T \\ \hline \end{array}$$

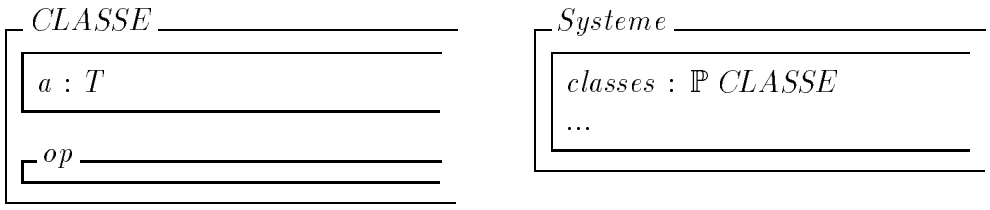
Op
$$\begin{array}{|l} \hline \\ \hline \end{array}$$

$[IDClasse]$

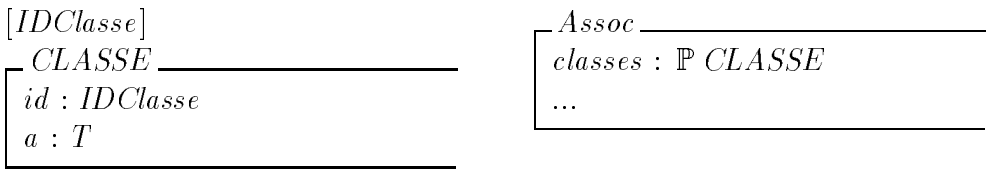
ClasseExt
$$\begin{array}{|l} \hline instances : \mathbb{P}\, IDClasse \\ attributs : IDClasse \pfun ClasseAttr \\ op : IDClasse \pfun Op \\ \hline \mathrm{dom}\, attributs = instances \\ \mathrm{dom}\, op = instances \\ \ldots \\ \hline \end{array}$$

Bien que cette solution soit peu naturelle (surtout pour les opérations), elle a pour avantage d'être proche de la notion d'objet en faisant correspond à une identité des ensembles d'attributs et d'opérations. Cela permet en particulier de "simuler" l'encapsulation d'opérations.

Contrairement à [FBLPS97], [AS97, KC99] ne représentent pas les objets existants d'une classe dans une structure à part, mais ils regroupent tous les les objets du système dans une même structure. L'intension donne systématiquement lieu à la création d'une classe Object-Z *CLASSE* alors que l'extension est toujours représentée dans une classe globale décrivant l'ensemble du système *Systeme*. Le risque de cette solution est de surcharger la classe *Systeme* si le nombre de classes est important.



Le même type de démarche est suivi dans [FB97, FBLP97, SF97] où l'intension donne lieu à un schéma Z spécifiant l'intension de la classe (*CLASSE*) alors que l'extension est représentée dans les schémas décrivant les associations. Cette solution qui est très proche de la précédente évite le problème de la surcharge, mais introduit le risque de multiplier les ensembles d'objets d'une même classe puisque chaque association définit l'ensemble des objets sur lesquels elle porte.



La démarche qui consiste à décrire **toutes les caractéristiques d'une classe dans une seule structure**, est basée sur la représentation de l'état complet des objets et non sur leur état interne. [Ngu98, MS99, LHW96] représentent la classe en intension comme un type en déclarant l'ensemble des identités d'objets possibles *IDClasse*. L'ensemble des objets existants est défini par l'ensemble des identités d'objets existants *classes* qui est un sous-ensemble des identités possibles ($classes \subseteq IDClasses$). Il n'y a donc pas la notion de type de la classe. Ces propositions utilisent le langage B et son concept de machine abstraite. Une machine abstraite consiste en un ensemble de variables, des propriétés sur ces variables (appelées invariant) et des opérations. L'état du système, i.e. l'ensemble des valeurs des variables, est modifiable seulement par les opérations. Dans [Ngu98, MS99, LHW96], une classe est représentée par une machine abstraite regroupant l'intension et l'extension :
Machine Classe
Sets IDClasse
Variables classes, a
Invariant classes $\subseteq$ IDClasse $\wedge$ $a \in classes \leftrightarrow$ T
Operations
...

Cette dernière solution a pour avantage d'être proche de l'approche objet : l'intension et l'extension sont représentées dans une seule structure qui correspond directement à celle de classe. Mais le risque est de confondre les concepts relatifs à une classe.

4.3 Association et concepts relatifs

4.3.1 Association

Soit l'association "A" liant les classes "C1" et "C2" :

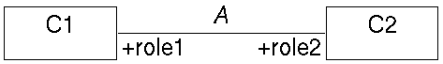
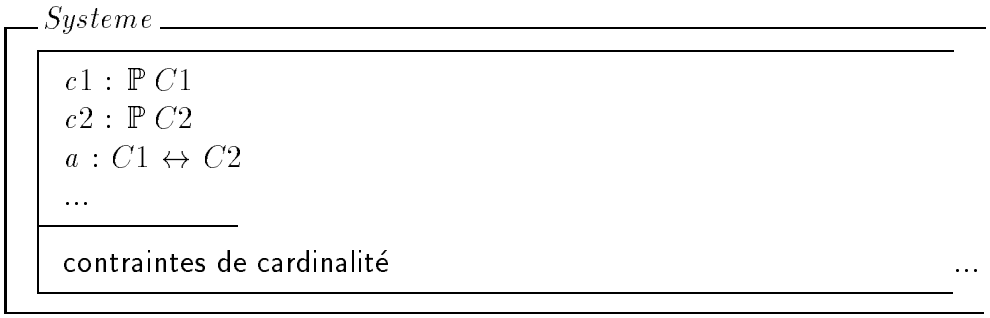


FIG. 4.2: Association

Il existe globalement trois façons de définir les associations. Elles peuvent être décrites comme **une relation ou un ensemble de fonctions**, comme **un ensemble d'objets** ou comme **une propriété de l'une des classes** de l'association.

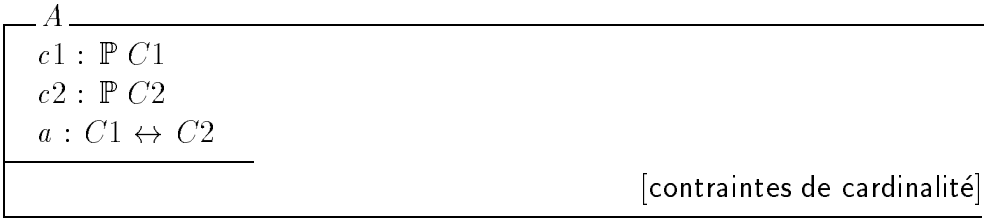
Représentation sous forme de relation La solution la plus répandue consiste à représenter une association par **une relation** [FB97, FBLPS97, FBLP97, SF97, Ngu98, MS99, KC99, AS97] ou par les fonctions correspondant à chacun des rôles de l'association [LHW96]. Dans ce cas, il se pose le problème de la modularité [Ngu98] pour savoir si la relation doit être dans une des structures de classes ou dans une structure propre à l'association.

La première solution est illustrée dans [AS97] où une association est représentée dans la classe Object-Z représentant le système global *Systeme*. Le risque de surcharge de *Systeme* déjà présent avec les ensembles d'objets des classes est augmenté par l'ajout des associations.

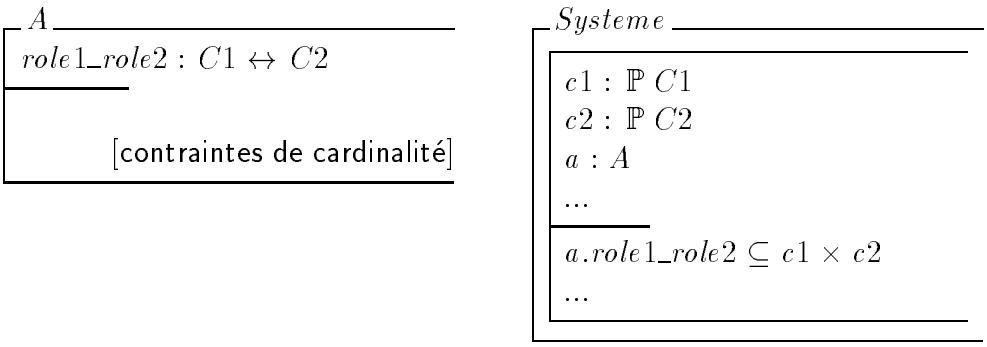


De façon assez similaire à la solution précédente, dans [KC99, FB97, FBLP97, SF97], une association donne toujours lieu à une structure à part dans laquelle elle est définie comme une relation entre des objets. L'introduction d'une structure spécifique à chaque association rend les spécifications plus modulaires, évitant ainsi toute surcharge.

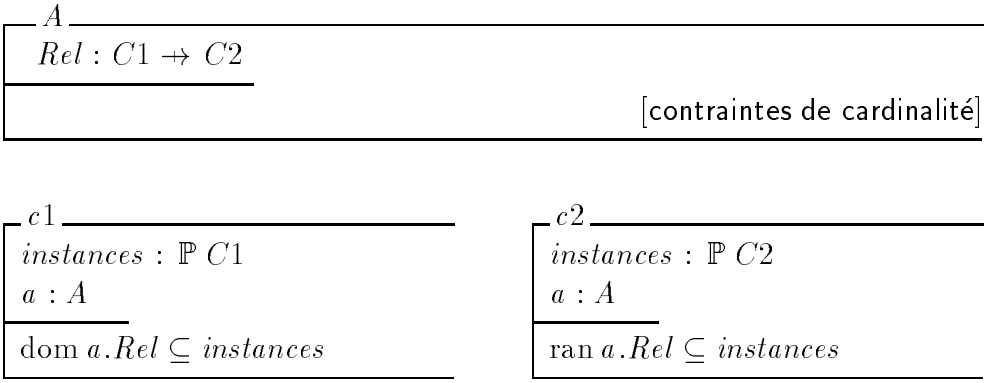
[FB97, FBLP97, SF97] introduisent un schéma Z contenant les objets existants des classes en relation *c1, c2*, une variable de type relation a et des contraintes pour spécifier les cardinalités :



Une solution intermédiaire [KC99] aux solutions de [AS97] et [FB97, FBLP97, SF97] doit permettre de conjuguer modularité et simplicité des spécifications. Dans [KC99], une association donne lieu à un schéma spécifique A qui est complété, dans la classe Object-Z *Systeme* représentant le système global, par des contraintes permettant d'exprimer les cardinalités en liant les objets existants des classes en relation. $a.role1\_role2 \subseteq c1 \times c2$ spécifie que le relation *role1_role2* associe des objets de *c1* et *c2* qui sont les ensembles d'objets existants de *C1* et *C2*.



[FBLPS97] va plus loin en proposant quatre schémas Z différents pour représenter une association : le premier A représentant un des rôles *Rel* de l'association et les cardinalités ; le deuxième *AssocStruct* représentant la structure de l'association ; les deux derniers étant des schémas *c1 et c2* contenant les ensembles d'objets des classes *instances* et un lien vers l'association a. Ces nombreux schémas induisent une forte complexité qui nuit à la lisibilité et à la compréhension des spécifications ainsi produites.



AssocStruct

$ca : c1$
$cb : c2$

$ca.a = cb.b$

Enfin dans le cadre des propositions représentant une association par une relation, Facon et al. [Ngu98] proposent une solution reprise dans [MS99] dont le but est d'optimiser la modularité des spécifications : si l'association est non-fixe ou a des attributs, une machine B indépendante est créée, sinon l'association est représentée dans la machine de l'une des classes (une association entre les classes "C1" et "C2" est fixe pour a si l'ensemble des liens concernant chaque objet de "C1" est créé en même temps qu'un objet de "C2" et ne peut être modifié ensuite).
Machine...
Variables..., a
Invariant $a \in c1 \leftrightarrow c2$... [contraintes de cardinalités]

Représentation par un ensemble d'objets La deuxième façon de représenter une association consiste à **la considérer comme un ensemble d'objets**. Chaque association a alors une structure indépendante qui contient une variable pour chaque classe impliquée dans l'association. Dans [NR94], toute association donne lieu à une machine abstraite A qui utilise (**Uses**) les machines de description des classes *C1* et *C2*. Le lien entre une association et les classes s'exprime par des fonctions entre la variable *Va* représentant l'association et les variables *Vc1a* et *Vc2a* décrivant les objets de type *C1* et *C2* participant à l'association.
Machine A
Uses $C1, C2$
Variables $Va, Vc1a, Vc2a$
Invariant $Vc1a \in Va \rightarrow Vc1 \wedge Vc2a \in Va \rightarrow Vc2$

Cette solution permet de représenter de manière identique les associations non binaires. En effet, bien que cela ne soit pas mentionné par l'auteur, il suffit de décrire des variables et des fonctions correspondant à chacune des classes liées pour représenter une association n-aire. L'inconvénient est que cette solution peut sembler moins naturelle que l'utilisation des concepts de relation ou de fonction.

Représentation par une référence Dans le dernier cas, une association est représentée par **une variable supplémentaire dans la description d'une classe**. Cette propriété fait le lien avec la classe associée. Pour obtenir la vision d'une association bidirectionnelle, il faut que chaque classe ait une propriété de ce type. Dans [Lan95], une variable *role2* est ajoutée à une des classes Z++ ou VDM++ représentant une des classes associées. Cette variable est une référence (symbole @) vers l'ensemble des objets de la classe cible liés par la relation.
class C1
values ... [déclaration des attributs]
instance variables role2 : @C2 ...
end C1

La référence aux objets liés par une association correspond à une vision bas niveau de l'approche objet qui ne reconnaît pas les associations comme des objets de première classe. Son seul avantage est de permettre d'accéder aux objets liés par navigation, ce qui est proche de l'approche objet et de OCL.

Bilan La manière la plus intuitive pour formaliser une association est l'utilisation de relation ou de fonction. Mais cette solution ne peut pas être étendue facilement pour représenter les associations non binaires. [Ngu98] propose néanmoins dans ce but de simuler une association non binaire en introduisant une classe qui est liée par des associations binaires aux classes initialement en relation. Cela demeure un palliatif de circonstance.

La formalisation d'une association par des références souffre du même défaut que l'utilisation de relation ou de fonction : il faudrait pouvoir faire référence à plusieurs objets pour représenter les associations non binaires ; mais dans ce cas, la proposition devrait être approfondie pour exprimer correctement les cardinalités et la réciprocité des rôles de l'association. De plus comme nous l'avons déjà évoqué, cette solution ne reconnaît pas les associations comme des objets de première classe.

En fait, la seule solution qui représente de façon simple les associations n-aires quel que soit le nombre de classes liées est celle qui considère une association comme un ensemble d'objets.

4.3.2 Classe associative

Soit la classe associative "CA" liant les classes "C1" et "C2" et ayant pour attribut "ai" et pour opération "opi" :

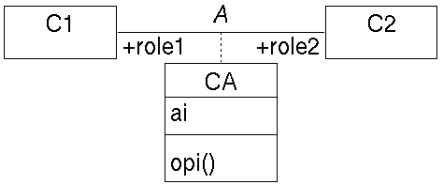


FIG. 4.3: Classe associative

Une classe associative peut être considérée **soit comme une classe, soit comme une association particulière**.

Représentation sous forme de classe particulière Dans [Ngu98], une classe associative est vue comme une classe dont la cardinalité des rôles de la classe associative vers les autres classes vaut 1 (Fig. 4.4). La spécificité des classes associatives n'est pas directement traitée mais exprimée en terme des propriétés d'une classe et d'associations monovaluées.

FIG. 4.4: Classe associative suivant [Ngu98]

Cela se traduit en B **comme une classe**, avec un invariant supplémentaire qui précise que la cardinalité de chaque rôle de la classe associative vers les autres classes vaut 1. Cet invariant utilise le produit direct $\otimes$ qui lie deux fonctions. Le produit direct est un opérateur sur les fonctions qui permet de lier un élément à un couple. Soit, par exemple, la fonction *Père* qui associe à chaque personne son père et la fonction *Mère* qui associe à chaque personne sa mère, alors le produit direct *Père* $\otimes$ *Mère* associe à chaque personne x le couple (Père de x, Mère de x).
Machine...
Sets $CA, C1, C2$
Variables $a, c1, c2, ai, role1, role2$
Invariant... $\wedge\ a \subseteq CA \wedge ai \in a \rightarrow Ti \wedge role1 \otimes role2 \otimes ai \in a \rightarrowtail c1 \times c2 \times T \wedge ...$

Représentation sous forme d'association particulière Les autres travaux [NR94, FBLP97, FBLPS97, SF97, MS99, KC99] ajoutent à la traduction de l'**association** les propriétés d'une classe. [NR94] ajoute à la machine traduisant l'association des variables *Vai* correspondant aux attributs de l'association. Comme pour la traduction des attributs d'une classe, *Vai* est une fonction entre la variable de l'association *Va* et le type de l'attribut *Ti*. La traduction d'une classe associative n'est qu'un cas particulier d'association pour laquelle des attributs sont ajoutées. Elle est donc naturelle par rapport à la proposition sur les associations.
Machine CA
Use $C1, C2$
Variables $Va, Vc1a, Vc2a, Vai$
Invariant $Vc1a \in Va \rightarrow Vc1 \wedge Vc2a \in Va \rightarrow Vc2 \wedge Vai \in Va \rightarrow Ti$

De même dans [MS99], une classe associative suit à la fois les règles de traduction pour une classe et pour une association. Toute classe associative donne lieu à une machine abstraite *CA*. L'association est définie par une variable *ca*, ses attributs par des variables *ai*, ses opérations par des opérations de *CA*.
Machine CA
Variables..., ca, ai
Invariant $ca \in c1 \leftrightarrow c2 \wedge ai \in ca \leftrightarrow Ti$... [contraintes de cardinalités]
Operations ... [opérations de la classe associative]

[FBLP97, FBLPS97, SF97, KC99] proposent une solution assez similaire à celle présentée précédemment en B. Ils introduisent une structure spécifique aux propriétés de la classe et ajoutent une fonction liant les classes et cette structure dans la représentation de l'association. France et al [FBLP97, FBLPS97, SF97] créent un schéma Z contenant les attributs de la classe associative *AAttr* et ajoutent une variable *a-attr* dans le schéma de l'association *A*. *a-attr* est une relation entre les couples d'objets de "C1" et "C2" et les attributs de la classe associative.

AAttr
$ai : Ti$

A

$c1 : \mathbb{P}\, C1$
$c2 : \mathbb{P}\, C2$
$a : C1 \leftrightarrow C2$
$a\_attr : (C1 \times C2) \leftrightarrow AAttr$

$\mathrm{dom}\, a\_attr = a$ [contraintes de cardinalité]

Dans [KC99], les propriétés (attributs et opérations) spécifiques à la classe associative sont exprimées dans une classe Object-Z *CACls* et l'association est représentée par un schéma *AAssoc*. *AAssoc* contient une variable *roles* qui lie les propriétés de la classe associative aux couples d'objets de "C1" et "C2" :

CACls

$ai : Ti$

opi

...

AAssoc

$roles : CACls \pfun (C1 \times C2)$

[contraintes de cardinalité]

Enfin dans la classe représentant le système, les objets existants des classes en relation et la classe associative sont liés : les objets existants de la classe associative sont ceux du domaine de l'association ($\mathrm{dom}\, a.roles = cas$) et les objets impliqués dans l'association sont des objets existants des classes ($\mathrm{ran}\, a.roles \subseteq c1 \times c2$).

Systeme

$c1 : \mathbb{P}\, C1$
$c2 : \mathbb{P}\, C2$
$cas : \mathbb{P}\, CACls$
$a : AAssoc$

$\mathrm{dom}\, a.roles = cas$
$\mathrm{ran}\, a.roles \subseteq c1 \times c2...$

Bilan La première proposition [Ngu98] ne traite pas directement du concept de classe associative : une classe associative est une classe dont les associations ont des spécificités. Dans les solutions [FBLP97, FBLPS97, SF97, MS99, KC99], une classe associative comporte bien les caractéristiques d'une classe et celles d'une association. Mais contrairement à la proposition de [NR94], le fait de faire le lien entre l'association et les propriétés de la classe par une relation ne met pas en évidence que les objets d'une classe associative sont des n-uplets comportant les valeurs des objets liés et des attributs spécifiques.

4.3.3 Agrégation

Soit "A" une relation "tout-partie" liant les classes "CA" et "CC" :

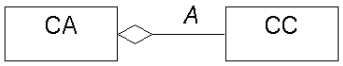


FIG. 4.5: Agrégation

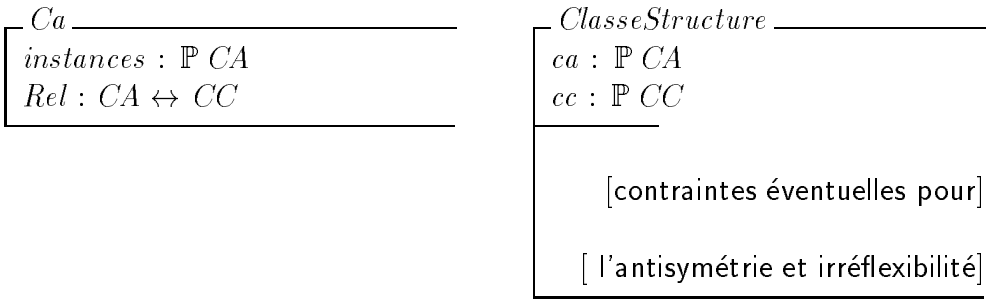
La sémantique de l'agrégation n'étant pas précise, les propositions de traduction vers des langages formels se basent sur des visions différentes de ce concept.

Représentation par une association quelconque La sémantique la moins contraingnante consiste à voir l'agrégation comme **un cas particulier d'association**. Dans ce cas, elle s'exprime comme une association quelconque [Ngu98]. L'agrégation perd alors toutes ses particularités pour être réduite au concept d'association.

Représentation des propriétés d'anti-symétrie et d'irréflexivité Dans [MS99, KC99, FB97, SF97, Lan95], différents types d'agrégation sont identifiés. [MS99, KC99, FB97, SF97] font la distinction entre une agrégation "faible" et une agrégation "forte" (aussi appelée composition) qui impose la dépendance existentielle et l'exclusivité des composants. Lano [Lan95] répertorie une classification complète des différents types d'agrégation et exprime les propriétés mathématiques associées à chacun d'eux. Mais cette classification n'est pas utilisée par la suite pour proposer des traductions vers les langages formels.

Dans [MS99, KC99, FB97, SF97], le type d'agrégation le moins restreint n'impose **ni la dépendance existentielle, ni l'exclusivité des composants**, mais peut avoir des **propriétés qui lui sont propres**. [MS99] le traduit alors comme une association qui n'a pas de particularité. Comme dans [Ngu98], l'agrégation est équivalente à toute autre association.

[FBLPS97] voit l'agrégation "minimale" comme une association à laquelle peuvent être ajoutées des contraintes spécifiques telles l'antisymétrie ou l'irréflexibilité. L'agrégation se traduit par un schéma contenant les instances de la classe agrégat (*instances*) et la relation entre "CA" et "CC" ; un schéma *ClasseStruct* représentant la structure de l'agrégation. Les contraintes pour exprimer l'anti-symétrie ou l'irréflexibilité de l'agrégation peuvent être ajoutées à *ClasseStructure*.



Cette proposition ([FBLPS97]) concernant l'agrégation souffre comme celle de l'association du nombre de schémas utilisés. De plus, la traduction ne semble pas totalement systématique car deux exemples d'agrégation ne donnent pas lieu exactement

aux mêmes schémas. Même si l'idée de la traduction reste la même, cela ne favorise pas l'assimilation de ces propositions.

[FBLP97] ne précise pas la sémantique de l'agrégation, mais sa représentation laisse penser qu'une agrégation a la particularité d'être anti-symétrique et transitive. Il représente l'inclusion des composants dans l'agrégat en décrivant les composants comme des attributs *cc1* et *cc2* de l'agrégat sans préciser d'autres contraintes. De plus, si les composants "CC1" et "CC2" sont liés par une relation, leur association "relcc1cc2" est aussi contenue dans la classe agrégat :

CA

$id : IDCA$

... [attributs de CA]

$cc1 : \mathbb{P}\, CC1$

$cc2 : \mathbb{P}\, CC2$

$relcc1cc2 : CC1 \leftrightarrow CC2$

De façon similaire, [KC99] propose d'inclure une variable *cc* représentant les composants dans la classe Object-Z de l'agrégat *CA*. De plus, il ajoute à l'ensemble des composants le symbole Ⓢ qui exprime que les relations cycliques entre l'agrégat et ses composés sont prohibées. Dans la classe *Systeme* décrivant le système global, une contrainte $\forall\, a : ca \bullet a.cc \subseteq cc$ précise que les composants d'autres objets doivent être des objets existants de leur classe.

CA

... [attributs de CA]

$cc : \mathbb{P}\, CC$ Ⓢ

Systeme

$ca : \mathbb{P}\, CA$

$cc : \mathbb{P}\, CC$

$\forall\, a : ca \bullet a.cc \subseteq cc$

L'ajout d'une variable correspondant aux composants dans l'agrégat est la façon la plus naturelle de représenter le caractère anti-symétrique et transitif d'une agrégation. Or si cette solution est toujours valide en Object-Z [KC99] où une variable est une référence vers un autre objet, ce n'est pas le cas pour Z [FBLP97] : dans le cas d'une agrégation réflexive, un agrégat *CA* devrait avoir une variable de type *CA*, ce qui n'est pas correct en Z.

Représentation de la dépendance existentielle et de l'exclusivité des composants Une vision intermédiaire de l'agrégation [Lan95, SF97] consiste à la définir comme **une relation "tout-partie" imposant la dépendance existentielle, mais pas forcément l'exclusivité des composants**.

Dans [Lan95], la traduction en Z++ introduit la référence aux objets composants comme un moyen de représenter l'agrégation. En Z++, cela signifie que le cycle de vie des objets composants dépend de celui de leur agrégat, mais cela n'impose pas l'exclusivité. La dépendance existentielle est ainsi exprimée de façon simple grâce à l'emploi d'un concept similaire en Z++.

class CA
functions
assoc : $\mathbb{F}\ \overline{CC}$

Shroff et France [SF97] traduisent cette sémantique "intermédiaire" de l'agrégation en incluant, dans le schéma Z d'intension de l'agrégat *CA*, une variable *cc* pour chaque composant et ajoutent un autre schéma *ClassStructure* représentant la structure de l'agrégation dans lequel sont exprimées des contraintes dont la dépendance existentielle :

CA

$id : IDCA$
... [attributs de CA]
$cc : \mathbb{P}\ CC$

ClasseStructure

$ca : \mathbb{P}\ CA$
$cc : \mathbb{P}\ CC$
...

$\forall\, c : cc \bullet (\exists\, a : ca \bullet c \in a.cc)$
[dépendance existentielle]

Cette traduction permet d'exprimer l'exclusivité des composants en ajoutant simplement un prédicat dans le schéma *ClasseStructure*. Si la classe "CA" est composée de "CC1" et de "CC2", si la cardinalité de "CA" vers "CC1" est 1 et celle de "CA" vers "CC2" est 1..n alors la composition est représentée en ajoutant à *ClasseStructure* la contrainte suivante qui signifie que si deux objets agrégats *a1* et *a2* ont les mêmes composants, ils sont égaux :

$\forall\, a1, a2 : ca \mid a1.cc1 = a2.cc1 \vee (a1.cc2 \neq \varnothing \wedge a1.cc1 = a2.cc2) \bullet a1 = a2$

Cette vision est une version plus restreinte de l'agrégation qui correspond à celle généralement répandue pour **la composition**. Dans ce cas, la relation "tout-partie" a les propriétés **de dépendance existentielle et d'exclusivité des composants**. Pour signifier ces restrictions entre une agrégation et une composition, [FBLPS97] contraint le codomaine de la relation. La composition se traduit comme un agrégation : un schéma pour chacune des classes avec un lien vers la relation pour la classe agrégat et un schéma *ClasseStructure* pour la structure de la composition. Pour exprimer l'exclusivité des composants, une contrainte $(\forall\, c : \mathrm{ran}\ Rel \bullet \#(Rel \rhd \{c\}) = 1)$ est ajoutée à *Ca*. Elle signifie que deux instances distinctes de "CA" ne peuvent pas partager de composants. De plus, toute instance de "CC" doit être liée à un agrégat $(\mathrm{ran}\ ca.Rel = cc.instances)$.

Ca

$instances : \mathbb{P}\ CA$
$Rel : CA \leftrightarrow CC$

$\forall\, c : \mathrm{ran}\ Rel \bullet \#(Rel \rhd \{c\}) = 1$

ClasseStructure

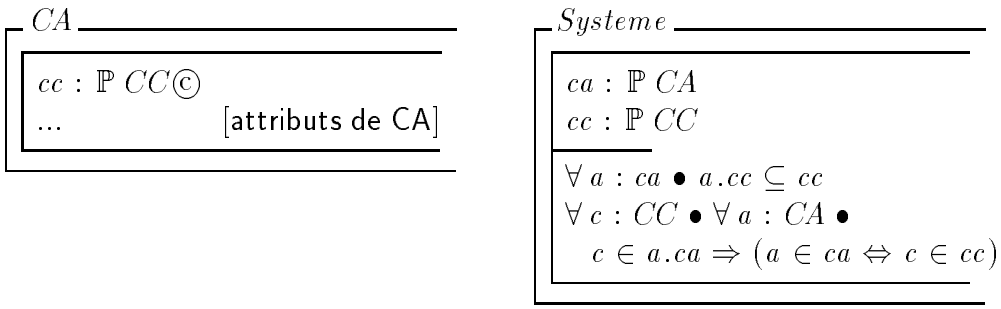
$ca : CA$
$cc : CC$

$\mathrm{ran}\ ca.Rel = cc.instances$
[contraintes éventuelles pour]

[l'anti-symétrie et irréflexibilité]

En Object-Z, [KC99] traduit aussi la composition par l'ajout d'une variable dans la classe agrégat *CA*, mais remplace le symbole Ⓢ par ⓒ qui exprime l'exclusivité des composants et l'absence de relations cycliques. La classe *Systeme* exprime toujours

que les objets composants sont des objets existants de leur classe $\forall\, a : ca \bullet a.cc \subseteq cc$. De plus, une dernière contrainte ajoute la dépendance existentielle des composants en spécifiant que pour toutes les instances du composant "CC" qui sont contenues dans des instances de "CA", si l'agrégat existe, alors le composant doit aussi exister dans le système $(\forall\, c : CC \bullet \forall\, a : CA \bullet c \in a.ca \Rightarrow (a \in ca \Leftrightarrow c \in cc))$.



Pour [NR94, MS99], la composition se traduit par l'inclusion (clause **Includes**) de la machine B représentant la classe composant dans la machine de l'agrégat. La clause **Includes** permet de définir la liste des machines à incorporer dans la machine courante. Les ensembles et les variables d'une machine incluse deviennent des ensembles et des valeurs de la machine courante. Cette clause est choisie par l'auteur pour représenter la dépendance entre un composant et son agrégat. La composition étant considérée comme une association contrainte, il y a toujours une description de la relation de composition comme une association $(a \in ca \leftrightarrow cc)$. Par exemple, l'agrégation dans [MS99] s'exprime par :

Machine *Composant*
Sets CC
Variables $cc, ...$
Invariant $cc \subseteq CC \wedge ...$

Machine *Agregat*
Includes *Composant*
Sets CA
Variables $..., ca, a$
Invariant $ca \subseteq CA \wedge a \in ca \leftrightarrow cc \wedge$
$...$

Dans [Lan95], la composition trouve une traduction directe par l'utilisation du concept d'héritage indexé de VDM++. Ici il signifie qu'il y a n copies distinctes de "CC" dans "CA".
class CA
is subclass of CC[1,..,n]

Bilan Les propositions concernant l'agrégation peuvent être divisées en deux catégaries : celles qui expriment ses propriétés par l'ajout de contraintes [FBLPS97, SF97] et celles qui utilisent des concepts équivalents pré-définis dans le langage formel cible [FBLP97, KC99, Lan95, NR94, MS99]. L'ajout de contraintes utilise des notations connues pour décrire de manière explicite chacune des propriétés. Cela permet de facilement identifier quelles propriétés restreignent une agrégation. En utilisant des concepts pré-définis, l'expression des propriétés est simplifiée, mais leur sémantique est sous-jacente : il faut connaître la notation employée pour bien cerner les caractéristiques de l'agrégation exprimées.

4.4 Généralisation/Spécialisation

Soit une relation de généralisation entre une classe “SuperClasse” et les classes “SousClasse1” et “SousClasse2” (Fig. 4.6).

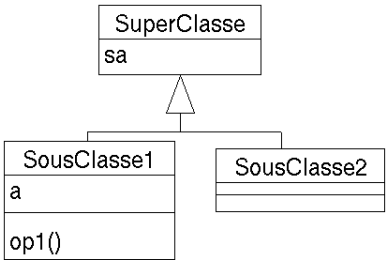
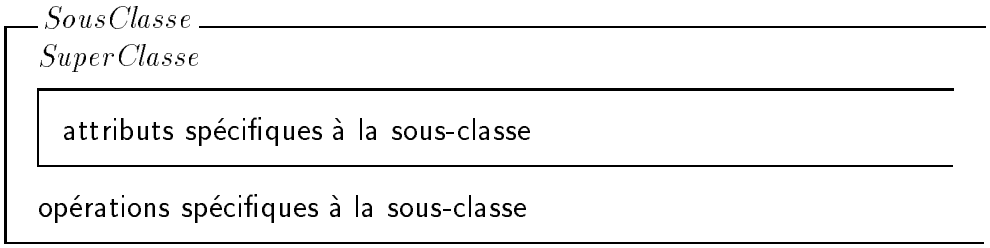


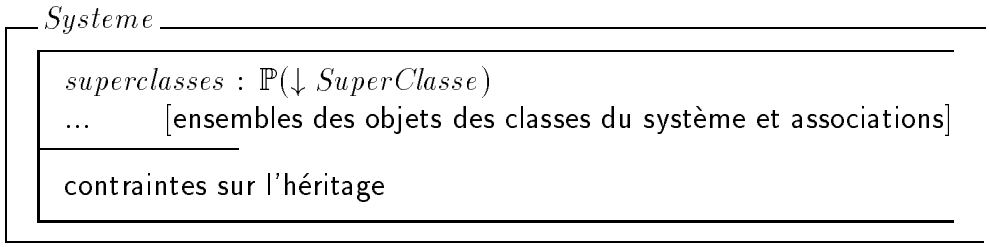
FIG. 4.6: Héritage

Représentation dans les langages formels orientés objet La traduction de l'héritage varie beaucoup suivant le langage formel cible. **Les langages formels orientés objet** tirent avantage de leurs concepts objet pour représenter directement l'héritage. En Z++, Lano [Lan95] utilise la fonctionnalité **Extends** pour signifier qu'une sous-classe hérite des propriétés de sa super-classe. En VDM++ [Lan95], le mot-clé à employer est **is-sub-class-of** alors qu'en Object-Z [AS97, KC99], il suffit de déclarer dans la liste des classes héritées le nom de la super-classe. Par exemple, en Object-Z, *SousClasse* déclare la classe *SuperClasse* dans ses classes héritées.

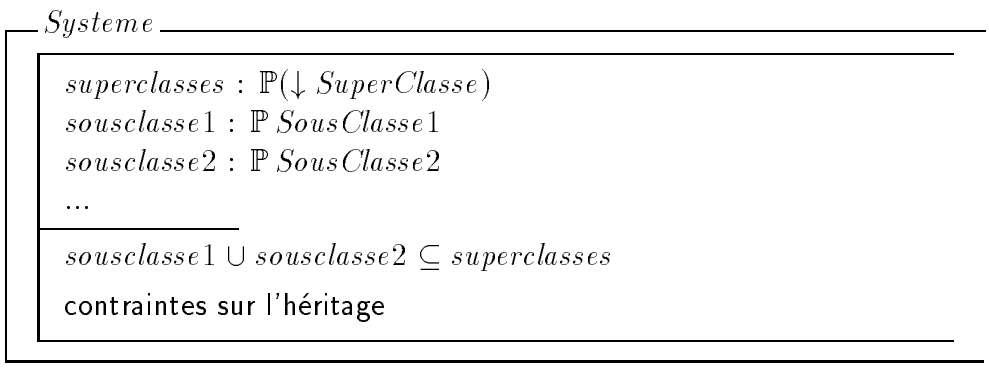


L'utilisation de l'héritage des langages formels orientés objet permet, a priori, d'exprimer de façon simple l'héritage des modèles objet. Néanmoins pour décrire l'héritage tel que nous le considérons (section 3.2.3), il faut aussi exprimer l'héritage au niveau des objets existants, ce que ne spécifie pas l'héritage des langages formels orientés objet. Les propositions [Lan95, AS97, KC99] doivent donc être complétées.

[AS97, KC99] ajoutent à la classe représentant le système global des contraintes pour spécifier que tout objet existant d'une sous-classe est un objet de sa super-classe. Achatz et Schulte [AS97] déclarent l'ensemble des objets existants de la super-classe comme des objets de la super-classe et de ses sous-classes grâce au symbole $\downarrow$. Par contre, il n'existe pas d'ensemble d'objets existants pour chaque sous-classe. Il n'est donc pas possible d'exprimer des contraintes ou des associations concernant les objets des sous-classes.



De la même façon, Kim et Carrington [KC99] déclarent les objets de la super-classe, mais ajoutent une variable pour représenter l'ensemble des objets existants de chaque sous-classe (*sousclasse1, souclasse2*...). Ils expriment ainsi que les objets existants des sous-classes sont inclus dans ceux de la super-classe ($sousclasse1 \cup sousclasse2 \subseteq superclasses$). On voit ici l'un des intérêts de regrouper toutes les classes dans une classe *Systeme* : les liens entre les objets d'une super-classe et ceux de ses sous-classes sont facilement exprimables.



Représentation dans les langages formels non orientés objet Pour les langages formels non-orientés objet, l'héritage n'existant pas, il faut utiliser les mécanismes d'inclusion de schéma ou de machine abstraite. Mais il est aussi possible d'éviter ces mécanismes en ne considérant pas la sous-classe comme une entité à part entière. La formalisation d'une sous-classe est alors directement incluse dans la structure de sa super-classe [LHW96]. [LHW96] ajoute dans la machine abstraite de la super-classe *SuperClasse*, une variable *sousclasse1* représentant les instances de "SousClasses1". L'invariant précise que ces instances constituent un sous-ensemble des instances de la super-classe. La relation d'héritage est ainsi complètement remise à plat en niant l'existence propre des sous-classes. La traduction perd donc des informations de structuration importantes du modèle objet.
Machine SuperClasse
Sets SUPERCLASSE
Variables superclasse, sousclasse1,...
Invariant sousclasse1 $\subseteq$ superclasse ...

Les autres propositions doivent utiliser les mécanismes de structuration des spécifications. En B, il existe plusieurs clauses permettant de construire, de hiérarchiser ou de composer des machines abstraites. Pour représenter l'héritage, les travaux existants préconisent l'utilisation des clauses **Includes** ou **Uses**. Ces deux clauses permettent de définir les constantes, les ensembles, les variables et les propriétés d'une

machine qui sont partagés par la machine courante. De plus, la clause **Includes** est transitive et autorise la modification des variables. Nagui-Raiss [NR94] utilise cette clause pour signifier qu'une sous-classe hérite des propriétés de sa super-classe. La machine d'une sous-classe inclut la machine de sa super-classe (**Includes** *SuperClasse*) et l'invariant précise que les objets de la sous-classe sont un sous-ensemble de ceux de la super-classe.

Machine SousClasse1
Includes SuperClasse
Variables sousclasse1
Invariant sousclasse1 $\subseteq$ superclasse

Par contre, [Ngu98, MS99] utilisent plutôt la clause **Uses**. Au lieu d'inclure la super-classe dans la sous-classe par la clause **Includes**, la sous-classe inclut sa super-classe en utilisant **Uses**.

Machine SousClasse1
Uses SuperClasse
Variables sousclasse1
Invariant sousclasse1 $\subseteq$ superclasse

Les machines de la hiérarchie d'héritage sont ensuite incluses (lien **Includes**) dans une autre machine afin de pouvoir exprimer les contraintes telles que la disjonction des sous-classes qui lient les sous-ensembles des sous-classes.

Machine HéritageStructure
Includes SuperClasse, SousClasse1, SousClasse2
Invariant [contrainte sur l'héritage]

Ces solutions B permettent uniquement de spécifier l'héritage d'attributs. Cependant [Ngu98] propose un moyen de simuler l'héritage d'opérations :

- Dans la machine de la sous-classe, des instances de la sous-classe peuvent être créées et des instances de la super-classe peuvent être spécialisées en tant qu'instances de la sous-classe. Dans la machine de structure de la hiérarchie, seules figurent les opérations qui créent réellement des instances de la sous-classe comme cas particulier d'instances de la super-classe (avec application parallèle des opérations de création des sous- et super-classes).

- Dans ce contexte, l'invariant $sousclasse1 \subseteq superclasse$ de la machine de structure hiérarchique peut alors être prouvé.

De même, [MS99] parle d'invoquer les opérations de création d'instances en parallèle.

En Z, un des mécanismes de structuration de schémas est l'inclusion qui permet à un schéma d'accéder aux variables et aux prédicats du schéma inclus. L'inclusion a donc été utilisée par [FB97, FBLPS97] pour représenter l'héritage d'attributs : le schéma *SousClasse1* décrivant les attributs d'une sous-classe inclut le schéma *SuperClasse* des attributs de sa super-classe.

SuperClasse

$id : IDSuperClasse$
$sa : T$
[attributs de la superclasse]

*SousClasse*1

SuperClasse
$a : T$
[attributs de la sous-classe]

Dans [FB97], chaque niveau d'héritage donne lieu à un schéma Z qui permet d'exprimer les contraintes entre les objets de la hiérarchie. Ce schéma *Niveau1* a pour variables les ensembles d'instances de la super-classe *superclasses*, de ses sous-classes *sousclasses1, sousclasses2* et les ensembles d'identités d'objets correspondant *superid, sous1id,sous2id*. Les contraintes sur les objets telles que le recouvrement des sous-classes ou le caractère abstrait de la super-classe, sont décrites à l'aide de ces ensembles d'identités. Ici par exemple, on spécifie que la super-classe est abstraite ($superid = sous1id \cup sous2id$).

*Niveau*1

$$
\begin{array}{l}
superclasses : \mathbb{P}\, SuperClasse \\
sousclasses1 : \mathbb{P}\, SousClasse1 \\
sousclasses2 : \mathbb{P}\, SousClasse2 \\
superid, sous1id, sous2id : \mathbb{P}\, IDSuperClasse \\
\hline
superid = \{s : superclasses \bullet s.id\} \\
sous1id = \{s1 : sousclasses1 \bullet s1.id\} \\
sous2id = \{s2 : sousclasses2 \bullet s2.id\} \\
superid = sous1id \cup sous2id
\end{array}
$$

Une fois l'héritage d'attributs effectué par inclusion de schéma, [FBLPS97] propose de traduire les objets existants comme pour les autres classesé : l'extension d'une sous-classe donne lieu à un schéma *SousClasse1Ext* contenant comme variables l'ensemble des identités d'objets de la sous-classe (qui sont des identités de la super-classe), la fonction liant les identités aux attributs *attributs* et pour chaque opération une fonction entre les identités et l'opération. De ce fait, cette proposition a pour intérêt de tenir aussi compte de l'héritage d'opérations.

*SousClasse*1*Ext*

$$
\begin{array}{l}
instances : \mathbb{P}\, IDSuperClasse \\
attributs : IDSuperClasse \pfun SousClasse1 \\
op1 : IDSuperClasse \pfun \mathbb{P}\, Op1
\end{array}
$$

La dernière façon proposée de représenter l'héritage en Z est de ne pas utiliser l'inclusion de schéma, mais de représenter la super-classe comme une variable de ses sous-classes [FBLP97, SF97]. Le schéma d'une sous-classe *SousClasse1* contient alors une variable *super* qui fait le lien avec le schéma de la super-classe *SuperClasse*. L'utilisation d'une variable est difficilement compréhensible pour représenter l'héritage puisqu'elle correspond plutôt à l'expression d'un rôle d'une association. Dans ce cas, on ne peut vraiment parler d'héritage d'attributs.

SuperClasse

$$
\begin{array}{l}
id : IDSuperClasse \\
sa : T \\
\qquad \text{[attributs de la superclasse]}
\end{array}
$$

*SousClasse*1

$$
\begin{array}{l}
super : SuperClasse \\
a : T \\
\qquad \text{[attributs de la sous-classe]}
\end{array}
$$

S'il est nécessaire d'ajouter des contraintes entre les objets existants de la hiérarchie, un schéma représentant la structure est créé :

$Niveau1$

$superclasses : \mathbb{P}\, SuperClasse$
$sousclasses1 : \mathbb{P}\, SousClasse1$

$\forall s1 : sousclasses1 \bullet s1.super \in superclasses$
[contraintes entre objets des sous-classes]

Bilan Les langages formels orientés objet offrent des traductions de l'héritage plus simples que celles des autres langages. Néanmoins, l'héritage des opérations est rarement évoqué. Si, dans les langages objets, il est possible d'imaginer que les opérations sur l'intension d'une classe sont héritées automatiquement. Rien n'est dit sur les opérations portant sur les objets existants. Pour les autres langages, il a seulement été proposé de le simuler dans [Ngu98, MS99].

4.5 Récapitulatif

Dans ce chapitre, nous avons décrit les différentes propositions existantes pour la traduction des concepts des modèles objet dans des langages formels orientés modèle ou objet. La présentation ayant porté sur les représentations proposées pour chaque concept, il est intéressant de fournir un point de vue complémentaire des travaux existants en récapitulant quels sont les concepts traités par les différents auteurs. Le tableau 4.1 indique, pour chaque travail les concepts étudiés et précise le niveau de traitement de chaque concept suivant le barème proposé dans [Ngu98] :

T : le concept est complètement traité
T$^-$: le concept n'est que partiellement traité
A : le concept est simplement abordé
P : le concept n'est pas traité mais pourrait l'être dans la démarche adoptée

Concepts / Articles	Classe	Attri--but	Opé--ration	Association binaire	Association n-aire	Classe associative	Agré--gation	Héri--tage
[NR94]	T$^-$	T	A	T	P	T	T$^-$	T$^-$
[LHW96]	T	T		T				T$^-$
[Ngu98]	T	T	A	T	T$^-$	T	T$^-$	T
[MS99]	T	T	A	T	P	T	T$^-$	T
[FB97]	T	T		T		P	T	T$^-$
[FBLP97]	T	T		T		T	T	T$^-$
[SF97]	T	T		T		T	T	T$^-$
[FBLPS97]	T	T	T	T		T	T	T$^-$
[Lan95]	T$^-$	T	T$^-$	T			T	T$^-$
[AS97]	T	T	P	T				T
[KC99]	T	T	T$^-$	T		T	T	T

TAB. 4.1: Récapitulatif des travaux sur le modèle objet

Ce tableau montre qu'aucun travail ne traite de façon complète tous les concepts d'un modèle objet. Aussi nous proposons nos propres règles de traduction.

Chapitre 5

Traduction du modèle objet en Z et Object-Z

Dans ce chapitre, nous présentons nos règles de traduction de chaque concept du modèle objet vers Z [DLCP00b] et Objet-Z [DLCP97]. Nous comparons nos propositions entre elles en étudiant les avantages respectifs de Z et d'Object-Z pour représenter le modèle objet. Enfin nous positionnons notre travail par rapport aux travaux similaires.

5.1 Traduction du modèle objet en Z

5.1.1 Traduction du concept de classe

Classe La spécification formelle d'une classe doit bien séparer son intension (description des propriétés communes aux instances possibles) de son extension (ensemble de toutes les instances existantes). Or ces deux aspects d'une classe ne sont pas exprimables directement par une structure de Z. Il faut donc utiliser deux représentations distinctes. Les caractéristiques des objets sont représentées par un schéma Z qui groupe les attributs de la classe. Un schéma Z pouvant être vu comme un type, cette première représentation considère une classe comme un type. Il faut ensuite préciser l'ensemble des objets existants c'est-à-dire l'ensemble des objets de ce type présents dans la base. Cela est représenté par un autre schéma.

Proposition 1 : Classe

Chaque **classe "CCC"** donne lieu à :

- **un schéma-type** de nom *CCC* contenant la déclaration des attributs,
- **un schéma** *CccExt* définissant l'ensemble des objets de la classe

  Il comprend une seule variable *Ccc* définissant l'ensemble des objets existants

  ($Ccc : \mathbb{F}\ CCC$).

Exemple 5.1

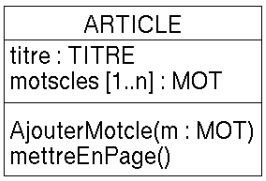


FIG. 5.1: Un exemple de classe

La classe "Article" donne lieu à deux schémas Z : ARTICLE qui contient les attributs et ArticleExt qui a pour variable l'ensemble fini ($\mathbb{F}$) des articles existants.

ARTICLE
...

ArticleExt
$Article : \mathbb{F}\, ARTICLE$

Attribut Un attribut est défini par son nom et optionnellement par son type. Une classe étant décrite par deux schémas, les attributs d'une classe sont représentés par des variables dans le schéma-type de la classe.

Proposition 2 : Attribut
Chaque **attribut "a"** de type "T_a" d'une classe "CCC" donne lieu à :

- la déclaration d'un type T_a si T_a n'est pas déjà défini
- la déclaration d'**une variable** a de type T_a dans le schéma CCC.
  - Si "a" est monovalué, la déclaration est $a : T_a$.
    Si "a" est optionnel, on introduit un élément particulier du type T_a qui est l'élément indéfini ($indefini T_a : T_a$). Si la valeur de "a" n'est pas connue, la variable a a pour valeur $indefini T_a$. Le caractère obligatoire est implicitement représenté si T_a n'a pas d'élément indéfini ou par la contrainte $a \neq indefini T_a$.
  - Si "a" est multivalué, la déclaration est $a : \mathbb{F}\, T_a$.
    Le caractère optionnel est implicitement représenté par le cas où l'ensemble $\mathbb{F}\, T_a$ est vide. Par contre, si l'attribut est obligatoire, la contrainte $a \neq \varnothing$ est ajoutée.

Exemple 5.2 *La définition des attributs complète le schéma ARTICLE. L'attribut titre est de type TITRE. "motscles" étant multivalué et obligatoire, il se traduit par une variable motscles qui est un ensemble fini de mots ($\mathbb{F}\, MOT$) et par la contrainte $motscles \neq \varnothing$ qui précise que cet ensemble ne peut pas être vide.*

ARTICLE

$titre : TITRE$
$motscles : \mathbb{F}\, MOT$

$motscles \neq \varnothing$

Opération Dans l'approche objet, les opérations sont encapsulées dans les classes. Le concept d'encapsulation n'existant pas en Z, il ne peut être que simulé en incluant dans le nom de chaque opération en Z le nom de la classe sur laquelle porte cette opération. Une opération sur une classe se traduit en Z par un schéma d'opération qui contient dans son nom le nom de la classe à laquelle elle appartient. De plus, si l'opération porte sur les attributs de la classe, l'opération doit être promue au niveau des objets existants de la classe c'est-à-dire qu'elle doit être définie sur l'extension de la classe afin de pouvoir être appelée à partir d'un objet existant.

Proposition 3 : Opération
Chaque opération "o" d'une classe "CCC" donne lieu à un schéma d'opération Z *CCCO*.
De plus, dans le cas d'une opération portant sur les attributs de la classe, elle est promue au niveau de l'extension de la classe en créant :

- un schéma générique *ChangeCcc* qui peut être combiné avec toutes les opérations sur les attributs de la classe,
- une opération composée de l'opération au niveau de l'intension de "CCC" et de l'opération générique :
$CccO == ChangeCcc \wedge CCCO$

ChangeCcc

$\Delta CccExt$
ΔCCC
$x? : CCC$

$x? \in Ccc$
$\theta CCC = x?$
$Ccc' = (Ccc \setminus \{x?\}) \cup \{\theta CCC'\}$

Exemple 5.3 *La classe "ARTICLE" a une opération qui ajoute un mot-clé. Le modèle objet ne permettant de générer que la signature de l'opération, le schéma est complété manuellement afin d'obtenir une spécification complète de l'opération. Les parties soulignées du schéma ARTICLEAjouterMotcle représentent ces compléments d'information.*

ARTICLEAjouterMotcle

$\underline{\Delta ARTICLE}$
$nouveaumotcle? : MOT$

$\underline{motcles' = motcles \cup \{nouveaumotcle?\}}$
$\underline{titre' = titre}$

L'opération "AjouterMotcle" portant sur l'attribut "motscles" de la classe, doit être promue au niveau de l'extension de la classe. Pour promouvoir cette opéra-

tion, on introduit l'opération ChangeArticle. ChangeArticle modifie ArticleExt et une instance de ARTICLE correspondant à x ?. θARTICLE est une variable de type ARTICLE et θARTICLE′ dénote sa valeur finale.

ChangeArticle

$$\Delta ArticleExt$$
$$\Delta ARTICLE$$
$$x? : ARTICLE$$

$$x? \in Article$$
$$\theta ARTICLE = x?$$
$$Article' = Article \setminus \{x?\} \cup \{\theta ARTICLE'\}$$

ChangeArticle peut être combinée avec toutes les opérations sur les attributs. Pour "AjouterMotcle", l'opération ArticleAjouterMotcle promeut l'opération ARTICLEAjouterMotcle au niveau de l'extension de "ARTICLE". La modification de l'attribut "motscles" est promue pour un élément donné x ? de Article. Cela modifie aussi l'ensemble Article comme l'ancien élément x ? est remplacé dans l'ensemble par la nouvelle valeur de θARTICLE.

$$ArticleAjouterMotcle == ChangeArticle \land ARTICLEAjouterMotcle$$

5.1.2 Traduction du concept d'association

Cas général

Une association est caractérisée par son nom et les contraintes de cardinalité sur chacun de ses rôles. Elle peut aussi être vue comme un groupe de liens, un lien étant une connexion entre des objets. Elle décrit donc un ensemble de liens de la même façon qu'une classe décrit un ensemble d'objets potentiels. Cette interprétation de l'association est très proche de celle d'une classe. La formalisation proposée consiste donc à voir toute association comme une classe : l'intension de l'association spécifie le type des objets participant à l'association et l'extension décrit les objets existants de l'association et les rôles de celle-ci. Pour les associations n-aires, les modèles semi-formels objet sont peu loquaces. Ils précisent qu'elles existent, mais ils ne spécifient pas leur sémantique et en particulier, ils ne disent rien concernant leurs cardinalités. Dans ce travail, nous suivons le même principe de définition des rôles que pour les associations binaires comme le montre notre exemple : un rôle est une fonction entre le produit cartésien des différentes classes en relation et la classe arrivant à ce rôle.

Le problème se pose ensuite de la modularité ([FL96, FLN96a, Ngu98]) pour savoir si l'association doit être dans un des schémas de classes ou dans un schéma spécifique propre à l'association. Il semble préférable pour la lisibilité de créer un nouveau schéma pour chaque association. Cela permet aussi de rester plus proche de la structure du modèle objet.

Exemple 5.4 *La relation "Présenter" (Fig. 5.2) est une association ternaire entre "CHERCHEUR", "ARTICLE" et "PRESENTATIONDARTICLE". Notre interprétation de ses rôles suit celle des modèles objets pour les associations bianires. Par*

Proposition 4 : Association
Chaque **association "A"** entre les classes "$CCC_1, .., CCC_n$" donne lieu à :

- **un schéma Z** *ARel* décrivant les liens entre les objets potentiels
  Pour chaque classe CCC_i impliquée dans l'association, il existe une variable ccc_i de type CCC_i (ccc_i : CCC_i).

- **un schéma Z** *ARelExt* définissant l'extension de l'association (liens existants).
  ARelExt contient l'inclusion de tous les schémas d'extensions Ccc_iExt des classes "CCC_i" participants à l'association. Elle contient la définition de l'association comme un ensemble d'objets de type *ARel* ($A : \mathbb{F}\, ARel$) et de chaque rôle de l'association sous forme de fonction. L'association est totalement spécifiée par la spécification de l'ensemble des fonctions. La partie prédicat précise que les objets liés sont des objets existants des classes participant à l'association ($\{x : CCC_i \bullet x.ccc_i\} \subseteq Ccc_i$).

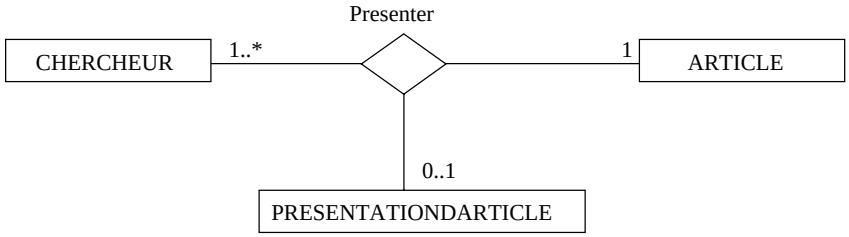


FIG. 5.2: Un exemple d'association

exemple, le rôle allant à "ARTICLE" signifie que pour tout chercheur et toute présentation d'article existants, il existe zéro ou un article correspondant.

"Présenter" est décrite par deux schémas Z PresenterRel et PresenterRelExt. PresenterRel spécifie les liens entre un chercheur, un article et une présentation quelconques. Tout élément de ce schéma contient un élément de CHERCHEUR, un élément d'ARTICLE et un élément de PRESENTATIONDARTICLE qui sont liés :

PresenterRel

$article : ARTICLE$
$chercheur : CHERCHEUR$
$presentationdarticle : PRESENTATIONDARTICLE$

Le deuxième schéma PresenterRelExt représente l'ensemble des liens de l'association. La variable Presenter décrit les éléments existants de type PresenterRel c'est-à-dire les tripets constitués d'un chercheur, d'un article et d'une présentation. Les trois premières lignes du prédicat servent à exprimer que les objets impliqués dans l'association font partie des extensions de CHERCHEUR, de ARTICLE et de PRESENTATIONDARTICLE.

PresenterRelExt

$ChercheurExt$; $ArticleExt$; $PresentationDArticleExt$
$Presenter : \mathbb{F}\ PresenterRel$
...

$\{x : Presenter \bullet x.article\} \subseteq Article$
$\{x : Presenter \bullet x.chercheur\} \subseteq Chercheur$
$\{x : Presenter \bullet x.presentationdarticle\} \subseteq PresentationDArticle$
...

Pour être plus précis, une association est décrite par ses rôles et leur cardinalité. La proposition 5 exprime comment les représenter en Z.

Proposition 5 : Rôle
Chaque **rôle "R"** de l'association "A" allant des classes "$CCC_1, ..., CCC_{b-1}, CCC_{b+1}, ..., CCC_n$" vers une classe "$CCC_b$" donne lieu à **une fonction** R. Les cardinalités sont spécifiées de la façon suivante :

- **la cardinalité minimale** s'exprime par des prédicats dans *ARelExt*.

  Si la cardinalité minimale est **1**, alors on ajoute un prédicat qui précise que pour tout nuplet dont les éléments sont de type $(CCC_1, ..., CCC_n)$, il existe un élément de type CCC_b. Si le rôle est monovalué, le prédicat est de la forme :
  $\forall\, c1 : Ccc1\ ; c_{b-1} : Ccc_{b-1}; c_{b+1} : Ccc_{b+1}; ...; cn : Cccn \bullet$
  $\exists\, cb : Cccb \bullet cb = R(c1, ..., cn)$

  Si le rôle est multi-valué, le prédicat est de la forme :
  $\forall\, c1 : Ccc1\ ; c_{b-1} : Ccc_{b-1}; c_{b+1} : Ccc_{b+1}; ...; cn : Cccn \bullet$
  $\exists\, cb : Cccb \bullet cb \in R(c1, ..., cn)$

- **la cardinalité maximale** s'exprime par les fonctions représentant les rôles.

  Si la cardinalité maximale de "R" est **égale à 1**, R :
  $CCC_1 \times ... \times CCC_{b-1} \times CCC_{b+1} \times ... \times CCC_n \nrightarrow CCC_b$

  Sinon (cardinalité maximale **supérieure à 1**)
  $R : CCC_1 \times ... \times CCC_{b-1} \times CCC_{b+1} \times ... \times CCC_n \nrightarrow \mathbb{F}\ CCC_b$.

Dans les prédicats, chaque rôle "R" est décrit en fonction des objets de l'extension A de l'association.
Si **"R" est monovalué**, sa définition est de la forme :
$R = \{x : A \bullet (x.ccc1, ...x.ccc_{b-1}, x.ccc_{b+1}, ..., x.cccn) \mapsto x.cccb\}$
Si **"R" est multivalué**, sa définition est de la forme :
$R = \{x : A \bullet (x.ccc1, ...x.ccc_{b-1}, x.ccc_{b+1}, ..., x.cccn) \mapsto$
$\{cb : Cccb \mid \exists\, y : A \bullet y.cccb = cb \wedge y.ccc1 = x.ccc1 \wedge ... \wedge y.cccn = x.cccn\}\}$

Exemple 5.5 *Le schéma d'extension de l'association PresenterRelExt permet de préciser les cardinalités et les rôles de l'association. Presenter représente l'ensemble*

des liens de l'association c'est-à-dire l'ensemble des objets en relation. Les trois rôles de Presenter sont représentés par trois fonctions presentation, presentateur et articlepresente définies à partir de Presenter.

$$
\begin{array}{|l}
\textit{PresenterRelExt} \\
\hline
\textit{ChercheurExt}; \textit{ArticleExt}; \textit{PresentationDArticleExt} \\
\textit{Presenter} : \mathbb{F}\, \textit{PresenterRel} \\
\textit{presentation} : \\
\quad (\textit{CHERCHEUR} \times \textit{ARTICLE}) \pfun \textit{PRESENTATIONDARTICLE} \\
\textit{presentateur} : \\
\quad (\textit{ARTICLE} \times \textit{PRESENTATIONDARTICLE}) \pfun \mathbb{F}\, \textit{CHERCHEUR} \\
\textit{articlepresente} : \\
\quad (\textit{CHERCHEUR} \times \textit{PRESENTATIONDARTICLE}) \pfun \textit{ARTICLE} \\
\hline
\ldots \\
\textit{presentation} = \{x : \textit{Presenter} \bullet \\
\quad (x.\textit{chercheur}, x.\textit{article}) \mapsto x.\textit{presentationdarticle}\} \\
\textit{presentateur} = \{x : \textit{Presenter} \bullet (x.\textit{article}, x.\textit{presentationdarticle}) \mapsto \\
\quad \{y : \textit{CHERCHEUR} \mid \exists\, z : \textit{Presenter} \bullet \\
\qquad z.\textit{chercheur} = y \land z.\textit{article} = x.\textit{article} \land \\
\qquad z.\textit{presentationdarticle} = x.\textit{presentationdarticle}\}\} \\
\textit{articlepresente} = \{x : \textit{Presenter} \bullet \\
\quad (x.\textit{chercheur}, x.\textit{presentationdarticle}) \mapsto x.\textit{article}\} \\
\hline
\end{array}
$$

Simplification

Le schéma de traduction, explicité dans la proposition 4, fonctionne quelle que soit l'arité de l'association. Cependant, dans la majorité des cas, les associations sont binaires et un schéma simplifié dans lequel on laisse de côté le "moule" (*ARel*) de l'association pour se concentrer sur la spécification de l'extension des deux rôles est proposé.

Proposition 6 : Association binaire

Chaque association A entre deux classes $CCC_1 et CCC_2$ donne lieu à un schéma décrivant chacun des rôles R_1 et R_2 de l'association sous forme de fonctions R_1 et R_2. Pour des raisons de simplicité, on commence toujours par spécifier un rôle monvalué, s'il existe. Dans cette proposition, on suppose que s'il existe un rôle monovalué, il s'agit de R_1, le rôle allant de $CCC_1 vers CCC_2$.
Les cardinalités des rôles sont spécifiées de la façon suivante :

- cardinalité minimale

  Si R_1 est monovalué (R_2 étant mono ou multivalué) :

  Si la cardinalité minimale de R_1 est au moins 1, cela signifie que tout objet de CCC_1 doit participer à l'association. Il y a alors égalité entre les objets existants Ccc_1 et les objets de CCC_1 participant à l'association ($\mathrm{dom}\, R_1 = Ccc_1$). Sinon (si la cardinalité minimale vaut 0) certains objets existants de CCC_1 peuvent ne pas participer à l'association : $\mathrm{dom}\, R_1 \subseteq C_1$.

  Si la cardinalité minimale de R_2 est au moins 1, alors le codomaine de R_1 est l'ensemble des objets existants de CCC_2 ($\mathrm{ran}\, R_1 = Ccc_2$) ; sinon c'est un sous-ensemble de ces objets ($\mathrm{ran}\, R_1 \subseteq Ccc_2$).

  Si R_1 et R_2 sont multivalués :
  Le domaine de R_1 est défini comme dans le cas précédent. Pour le codomaine, **si la cardinalité minimale de R_2 est au moins 1**, alors il est défini par $\bigcup(\mathrm{ran}\, R_1) = Ccc_2$; sinon il l'est par $\bigcup(\mathrm{ran}\, R_1) \subseteq Ccc_2$.

- cardinalité maximale
  La cardinalité maximale est en fait exprimée par les fonctions. Pour *R_1*, une fonction de type $CCC_1 \nrightarrow CCC_2$ exprime qu'à un élément de CCC_1 correspond un seul élément de CCC_2 (cardinalité maximale 1) et $CCC_1 \nrightarrow \mathbb{F}\ CCC_2$ exprime qu'à un élément de CCC_1 correspondent plusieurs éléments de CCC_2.

R_2 est spécifié à partir de R_1 :

- si R_1 et R_2 sont monovalués, R_2 est défini par :
  $R_2 = \{c2 : \mathrm{ran}\, R_1; c1 : \mathrm{dom}\, R_1 \mid c2 = R_1(c1) \bullet c2 \mapsto c1\}$

- si R_1 est monovalué et R_2 multi-valué, R_2 est défini par :
  $R_2 = \{c2 : \mathrm{ran}\, R_1 \bullet c2 \mapsto \{c1 : \mathrm{dom}\, R_1 \mid c2 = R_1(c1) \bullet c1\}\}$

- si R_1 et R_2 sont multivalués, R_2 est défini par : $R_2 = \{c2 : \bigcup(\mathrm{ran}\, R_1) \bullet c2 \mapsto \{c1 : \mathrm{dom}\, R_1 \mid c2 \in R_1(c1) \bullet c1\}\}$

Exemple 5.6

Pour la relation "Soumettre" entre "SOUMISSION" et "CONFERENCE" (Fig. 5.3), on spécifie les deux rôles "soumissiondeconference" et "conferencedesoumission". Les rôles sont déclarés comme étant des fonctions. Comme pour la proposition précédente, les prédicats servent à exprimer que l'association lie des objets existants de "SOUMISSION" et "CONFERENCE" : les deux premiers prédicats

FIG. 5.3: Un exemple d'association binaire

expriment les cardinalités minimales de "conferencedesoumission" et "soumissiondeconference" en contraignant le domaine et le co-domaine de l'un des rôles ; enfin le troisième prédicat exprime soumissiondeconference à partir de conferencedesoumission :

$$
\begin{array}{|l}
\textit{SoumettreRel} \\
\hline
SoumissionExt\ ;ConferenceExt \\
conferencedesoumission : SOUMISSION \pfun CONFERENCE \\
soumissiondeconference : CONFERENCE \pfun \mathbb{F}\, SOUMISSION \\
\hline
dom\, conferencedesoumission = Soumission \\
ran\, conferencedesoumission \subseteq Conference \\
soumissiondeconference = \{x : ran\, conferencedesoumission \bullet x \mapsto \\
\quad \{y : dom\, conferencedesoumission \mid conferencedesoumission(y) = x \bullet y\}\} \\
\hline
\end{array}
$$

5.1.3 Classe Associative

Une association se formalisant comme une classe, une classe associative est spécifiée en Z comme une association qui a en plus dans son schéma d'intension la déclaration des attributs spécifiques à l'association. De même, les éventuelles opérations de la classe associative suivent la règle 3 de traduction des opérations.

Proposition 7 : Classe associative
Une classe associative suit la règle de traduction 4 des associations. Ses attributs spécifiques sont déclarés dans l'intension de l'association suivant la proposition 2. Les opérations suivent la proposition 3 de traduction des opérations.

Exemple 5.7

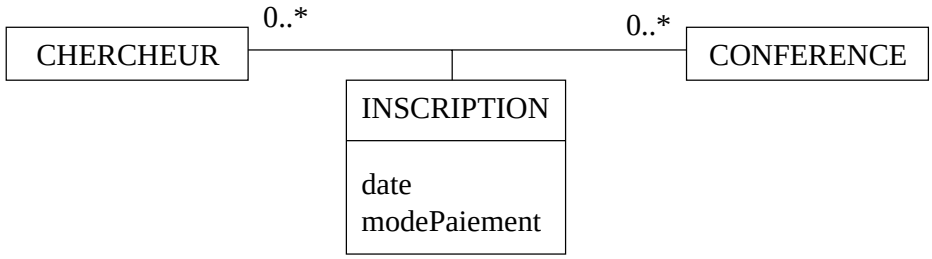


FIG. 5.4: Un exemple de classe associative

Comme pour une association, la classe associative "INSCRIPTION" donne lieu à un schéma InscriptionRel décrivant les liens potentiels entre "CHERCHEUR" et "CONFERENCE". InscriptionRel contient en plus des éléments de CHERCHEUR et de CONFERENCE, les attributs spécifiques à la classe associative date et modePaiement.

InscriptionRel

$$chercheur : CHERCHEUR$$
$$conference : CONFERENCE$$
$$date : DATE$$
$$modePaiement : MODEPAIEMENT$$

L'extension d' "Inscription" suit exactement la règle de traduction des associations (proposition 4 et 5).

InscriptionRelExt

$$ChercheurExt\ ;ConferenceExt$$
$$Inscription : \mathbb{F}\ InscriptionRel$$
$$conferencesdechercheur : CHERCHEUR \pfun \mathbb{F}\ CONFERENCE$$
$$chercheurinscrit : CONFERENCE \pfun \mathbb{F}\ CHERCHEUR$$

$$\{x : Inscription \bullet x.conference\} \subseteq Conference$$
$$\{x : Inscription \bullet x.chercheur\} \subseteq Chercheur$$
$$conferencesdechercheur = \{x : Inscription \bullet x.chercheur \mapsto \{c : Conference \mid \exists\, y : Inscription \bullet y.conference = c \land y.chercheur = x.chercheur\}\}$$
$$chercheurinscrit = \{x : Inscription \bullet x.conference \mapsto \{ch : Chercheur \mid \exists\, y : Inscription \bullet y.chercheur = ch \land y.conference = x.conference\}\}$$

5.1.4 Traduction de l'agrégation et de la composition

Agrégation Nous avons défini l'agrégation comme une association binaire particulière qui spécifie une relation "tout-partie" entre un agrégat (le tout) et un composant (la partie). Par rapport aux autres associations, l'agrégation a la particularité d'être antisymétrique et transitive. Cette particularité de l'agrégation pourrait se traduire en Z par l'inclusion dans le schéma de la classe agrégat d'une variable représentant le composant. Par exemple, si une classe "CA" est composée d'une classe "CC", le schéma d'intension *CA* a une variable *cc* représentant l'élément de "CC" qui compose un élément de "CA" :

CA

$$\ldots \qquad \text{[attributs de CA]}$$
$$cc : CC$$

Mais ce schéma de traduction ne permet d'exprimer des agrégations réflexives (un objet de "CA" ne pourrait pas être un agrégat de plusieurs autres objets de "CA"). En effet, Z ne permet pas la définition de schémas-types récursifs.

Nous devons donc envisager une solution moins naturelle pour représenter les propriétés transitives et antisymétriques d'une agrégation. On peut exprimer simplement ces propriétés de l'agrégation par "un objet ne peut pas être (transitivement) une partie de lui-même". C'est ce que nous allons exprimer en Z en posant des contraintes sur une association "simple".

Pour savoir si un objet est transitivement une partie de lui-même, il est nécessaire de connaître tous ses composants directs ou par transitivité. Cela revient à exprimer le caractère transitif des agrégations en générant, le cas échéant, la fermeture transitive du graphe d'agrégation : s'il existe une agrégation entre X et Y, et entre Y et Z (Z pouvant être égal à X), la spécification est complétée par une relation entre X et Z qui résulte de la composition des deux agrégations. S'il y a alors une agrégation réflexive "Agreg" dans le graphe résultant, on exprime son caractère transitif :
$\forall x1, x2, x3 : X \mid x2 = Agreg(x1) \land x3 = Agreg(x2) \bullet x3 = Agreg(x1)$

Une fois la fermeture transitive du graphe d'agrégation générée, le caractère antisymétrique peut être spécifié en précisant qu'un objet ne peut pas être une partie de lui-même. Ce cas peut se produire seulement s'il existe une agrégation réflexive sur sa classe, d'où la contrainte précisant qu'un objet ne peut pas être agrégat de lui-même :
$\forall x1, x2 : X \mid x2 = Agreg(x1) \bullet x1 \neq x2$

Proposition 8 : Agrégation
L'agrégation entre une classe agrégat "CA" et une classe composée "CC" se traduit par la génération d'**un schéma** traduisant les rôles de l'agrégation comme pour les associations binaires.
Le caractère transitif s'exprime par la génération en Z de la fermeture transitive du graphe d'agrégation ou de composition. S'il existe une agrégation réflexive dans la fermeture transitive, l'analyste doit écrire **les deux contraintes** suivantes :
$\forall x1, x2, x3 : X \mid x2 = Agreg(x1) \land x3 = Agreg(x2) \bullet x3 = Agreg(x1)$
$\forall x1, x2 : X \mid x2 = Agreg(x1) \bullet x1 \neq x2$

Exemple 5.8

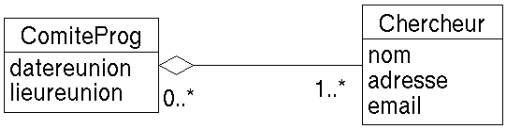


FIG. 5.5: Un exemple d'agrégation

L'agrégation entre "COMITEPROG" et "CHERCHEUR" n'est pas réflexive. Elle se traduit donc comme une association binaire. Le nom de ce schéma se termine par "Agreg" pour préciser qu'il s'agit d'une agrégation.

ComiteChercheurAgreg

$ComiteprogExt$; $ChercheurExt$
$membres : COMITEPROG \pfun \mathbb{F}\ CHERCHEUR$
$comiteprogdechercheur : CHERCHEUR \pfun \mathbb{F}\ COMITEPROG$

$dom\ membres = Comiteprog$
$\bigcup(ran\ membres) \subseteq Chercheur$
$comiteprogdechercheur = \{x : \bigcup(ran\ membres) \bullet x \mapsto$
$\quad \{y : dom\ membres \mid x \in membres(y) \bullet y\}\}$

ComiteChercheurAgreg illustre le cas d'une agrégation sans contrainte particulière. Un exemple d'agrégation transitive et réflexive est donné dans le paragraphe 6.3.2.

Composition Nous avons aussi introduit une variante de l'agrégation, la composition qui dénote une appartenance forte entre un agrégat et son composant. Une composition se distingue d'une agrégation par deux propriétés.

Dans une composition, un composé peut appartenir au plus à un agrégat alors que dans une agrégation, un composé peut appartenir à plusieurs agrégats. Cette différence ne modifie pas la règle de traduction de l'agrégation dans la mesure où le nombre d'agrégats pour un composant est exprimé lors de l'expression de l'association entre l'agrégat et le composant. Par contre, une erreur peut être détectée si dans une composition, la cardinalité du rôle de composition est supérieure à 1.

La seconde différence entre agrégation et composition porte sur la sémantique dynamique des objets. La sémantique dynamique de l'agrégat se propage à ses parties c'est-à-dire que si l'agrégat est détruit, ses composants doivent aussi être détruits. Ceci est équivalent au concept d'existence de domaine pour lequel une contrainte en B et une obligation de preuve sont proposées dans [HG97]. En Z, ceci ne peut se traduire par une contrainte statique, mais par une obligation de preuve sur l'opération de suppression. Par exemple, on considère une classe "CA" qui est composée de la classe "CC" et une opération "SupprimerCa" qui supprime un objet de "CA" de la base de données. L'obligation de preuve exprime que si on supprime un objet ca de "CA", tout objet cc composant de ca doit aussi avoir été supprimé de l'extension Cc de "CC" (Cc' dénote la valeur de Cc après l'opération) :

theorem SupprimerComposants

$$\forall ca : CA \bullet \forall cc : CC \mid cc = composant(ca) \bullet SupprimerCa(ca) \Rightarrow cc \notin Cc'$$

Proposition 9 : Composition
La composition entre une classe agrégat "CA" et une classe composée "CC" se traduit suivant la même proposition que pour l'agrégation, en ajoutant une obligation de preuve pour l'opération de suppression d'objets agrégat.
theorem SupprimerComposants

$$\forall ca : CA \bullet \forall cc : CC \mid cc = composant(ca) \bullet SupprimerCa(ca) \Rightarrow cc \notin Cc'$$

Exemple 5.9

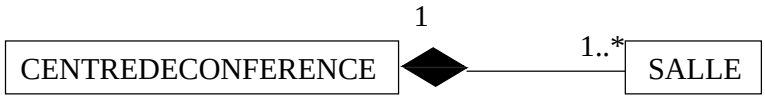


FIG. 5.6: Un exemple de composition

La composition entre "CENTREDECONFERENCE" et "SALLE" se traduit comme une association binaire mis à part que le nom du schéma se termine par "Comp" pour préciser qu'il s'agit d'une composition.

CentreSalleComp

$$CentredeconferenceExt; SalleExt$$
$$centredesalle : SALLE \pfun CENTREDECONFERENCE$$
$$salledecentre : CENTREDECONFERENCE \pfun \mathbb{F}\, SALLE$$

$$dom\, centredesalle = Salle$$
$$ran\, centredesalle = Centredeconference$$
$$salledecentre = \{x : ran\, centredesalle \bullet x \mapsto \{y : dom\, centredesalle \mid centredesalle(y) = x \bullet y\}\}$$

Il faut aussi ajouter une obligation de preuve qui spécifie que si un centre de conférence est détruit (opération SupprimerCentredeconference), toutes ses salles doivent aussi avoir été supprimées :

theorem *SupprimerSalles*

$$\forall cc : CENTREDECONFERENCE \bullet \forall s : SALLE \mid s \in centredesalle(cc) \bullet SupprimerCentredeconference(cc) \Rightarrow s \notin Salle'$$

5.1.5 Traduction du concept d'héritage

Pour clarifier la présentation de notre proposition concernant l'héritage, nous avons décomposé l'héritage en trois parties : l'héritage d'attributs, la substitution d'objets et l'héritage d'opérations.

Héritage d'attributs L'héritage des attributs est le fait qu'une sous-classe a implicitement les attributs de sa super-classe. Comme lors de la traduction d'une classe, les attributs de la super-classe peuvent être décrits dans un schéma Z. L'héritage d'attributs peut alors se traduire en Z par l'inclusion de ce schéma dans le schéma d'intension de la sous-classe. Nous avons choisi de créer un schéma pour les attributs spécifiques à chaque classe impliquée dans une relation d'héritage c'est-à-dire qu'il existe un schéma Z pour les attributs de la super-classe et un schéma pour les attributs de chacune des sous-classes. Cela permet de généraliser notre règle de traduction à l'héritage à n niveaux (une sous-classe peut aussi être super-classe d'une autre classe). De plus, cela facilite l'expression des opérations sur les attributs de la super-classe, comme nous le montrons dans l'exemple 5.12.

Sous-proposition 10.1 : Héritage d'attributs
Soit une relation d'héritage entre une super-classe CCC et ses sous-classes $CCC_1, ..., CCC_n$. Les attributs de la **super-classe CCC** sont exprimés en Z dans un schéma de nom $CCCATTR$. Les attributs d'une sous-classe CCC_i de CCC sont traduits en Z dans **un schéma de nom** CCC_iATTR. **L'héritage d'attributs** se traduit alors par la création d'un **schéma de nom** CCC_i incluant $CCCATTR$ et CCC_iATTR.

Exemple 5.10

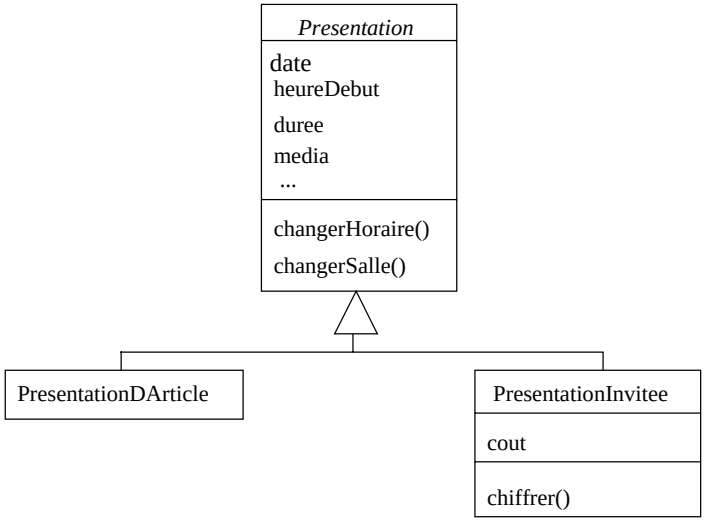


FIG. 5.7: Un exemple d'héritage

Le fait que "PRESENTATIONINVITEE" hérite des attributs de "PRESENTATION" se traduit par l'inclusion dans le schéma de l'intension PRESENTATIONINVITEE d'un schéma contenant les attributs de "PRESENTATIONINVITEE". On a donc :

- *un schéma PRESENTATIONATTR qui contient les attributs spécifiques à une présentation,*

$$\begin{array}{|l} PRESENTATIONATTR \\ \hline date : DATE \\ heureDebut : HEURE \\ duree : HEURE \\ media : MEDIA \\ \hline \end{array}$$

- *un schéma PRESENTATIONINVITEEATTR contenant les attributs spécifiques à une présentation invitée,*

$$\begin{array}{|l} PRESENTATIONINVITEEATTR \\ \hline cout : \mathbb{N} \\ \hline \end{array}$$

– *un schéma PRESENTATIONINVITEE qui inclut les deux schémas précédents, représentant ainsi l'intension de "PRESENTATIONINVITEE"*

PRESENTATIONINVITEE
PRESENTATIONATTR
PRESENTATIONINVITEEATTR

Substitution d'objets La sémantique de l'héritage précise qu'un objet d'une sous-classe doit être à tout moment substituable à un objet de sa super-classe. Au niveau des types, cela correspond au fait que tout élément de la sous-classe est élément de la super-classe. La super-classe doit donc offrir tous les types de ses sous-classes. Cela peut s'exprimer en Z par une union de types : un nouveau type est défini comme étant la disjonction d'autres types ; il a donc toutes les caractéristiques des types utilisés. La super-classe "CCC" est l'union des types de ses sous-classes "$CCC_1, ..., CCC_n$". Elle a ainsi tous les attributs de ses sous-classes. On peut remarquer que les types des sous-classes doivent être disjoints pour que plusieurs déclarations d'attributs identiques dans différentes sous-classes ne soient pas mis en commun lors de l'union de types de la super-classe.

$CCC == CCC_1 \lor ... \lor CCC_n$

Tout élément de la super-classe a donc tous les attributs de ses sous-classes. Il peut être un élément de n'importe laquelle des sous-classes. Mais certains de ses attributs qui correspondent aux attributs des autres sous-classes n'ont aucune signification. Si l'union de types permet la substitution des éléments d'une super-classe par ceux de ses sous-classes, elle leur ajoute aussi des caractéristiques qui n'ont pas lieu d'être. Elle n'offre donc pas une représentation totalement équivalente à l'héritage.

De plus, cette interprétation de l'héritage ne permet pas de traiter de manière satisfaisante l'héritage multiple. Par exemple, si la classe "X" hérite des classes "Y" et "Z", un élément de "X" aurait pour attributs ceux de "Y", ceux de "Z" et ceux qui lui seraient spécifiques. La définition du schéma Y aurait alors par l'intermédiaire du schéma X les attributs de "Z". Nous ne supportons donc pas l'héritage multiple.

Une fois que la substitution est définie au niveau des types, il faut la répercuter au niveau des objets existants (extension) de la base de données. Pour cela, il faut faire un lien entre les objets existants de la super-classe et ceux de ses sous-classes. Dans les éléments existants d'une super-classe, certains sont des éléments d'une sous-classe et cet ensemble correspond à l'ensemble des éléments existants de la sous-classe (Fig. 5.8).

Ainsi, lors de la définition de l'extension de la super-classe, on définit l'ensemble des éléments qui appartiennent à une sous-classe ($Ccc_i 2 : \mathbb{F}\, CCC$). L'extension d'une sous-classe exprime aussi que tout objet de la sous-classe correspond à un objet existant de sa super-classe.

Enfin on peut préciser que la super-classe est abstraite en ajoutant une contrainte qui exprime que tous les éléments de la super-classe "CCC" appartiennent à une des sous-classes "$CCC_1, ..., CCC_n$" :

$Ccc = Ccc2_1 \cup ... \cup Ccc2_n$

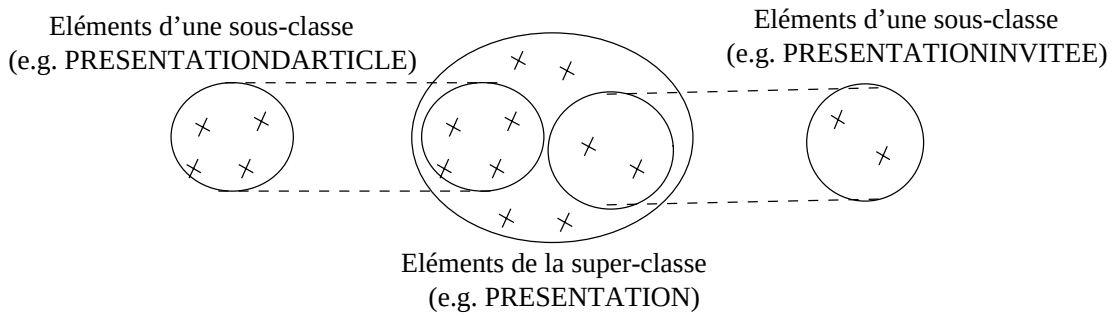


FIG. 5.8: Promotion de l'héritage à l'extension des classes

Notre proposition pour représenter la substitution d'objets au niveau des extensions est complexe et nécessite l'introduction de nombreuses variables et contraintes. Elle a néanmoins le mérite d'expliciter sa signification et ses implications.

Sous-proposition 11.2 : Substitution d'objets

Soit une relation d'héritage entre une super-classe CCC et ses sous-classes $CCC_1, ..., CCC_n$.
La substitution d'objets s'exprime en Z par :

- la définition de la super-classe comme la **disjonction de ses sous-classes** ($CCC == CCC_1 \lor ... \lor CCC_n$).
- **un schéma** *CccExt* définissant l'ensemble des objets existants de la super-classe CCC.
  Il a pour variables *Ccc* définissant l'ensemble des objets existants ($Ccc : \mathbb{F}\ CCC$) et $Ccc_1 2, ..., Ccc_n 2$ qui représentent les objets de CCC qui sont aussi des objets de $CCC_1, ..., CCC_n$ ($Ccc_i 2 : \mathbb{F}\ CCC$). Pour pouvoir importer les contraintes, les prédicats de ce schéma expriment l'inclusion des objets existants de la sous-classe dans sa super-classe ($Ccc_i 2 \subseteq Ccc$ et $\forall\, x : Ccc_i 2 \bullet \exists\, y : CCC_i \mid x.a_1 = y.a_1 \land ... \land x.a_n = y.a_n$) où les a_i sont les attributs de la sous-classe et de sa super-classe.

  Si la super-classe n'est pas instanciable, la contrainte $Ccc = Ccc2_1 \cup ... \cup Ccc2_n$ est ajoutée.
- **un schéma** $Ccc_i Ext$ définissant l'ensemble des objets existants de chaque sous-classe
  Il inclut le schéma d'extension de la super-classe *CccExt*. Il a pour variable Ccc_i définissant l'ensemble des objets existants ($Ccc_i : \mathbb{F}\ CCC_i$). Les prédicats de ce schéma expriment qu'à chaque objet existant de CCC_i correspond un objet existant de CCC ($Ccc_i = \{x : CCC_i \mid \exists\, y : Ccc_i 2 \bullet x.a_1 = y.a_1 \land ... \land x.a_n = y.a_n\}$).

Exemple 5.11 *Dans l'exemple 5.10, la substitution d'objets signifie qu'une présentation invitée (ou une présentation d'article) est une présentation. Pour pouvoir substituer une présentation invitée ou une présentation d'article à une présentation,*

on définit un schéma PRESENTATION dont les éléments peuvent être des éléments de PRESENTATIONINVITEE ou de PRESENTATIONDARTICLE.

$$PRESENTATION == PRESENTATIONINVITEE \lor PRESENTATIONDARTICLE$$

Tout élément du schéma PRESENTATION a donc tous les attributs d'une présentation invitée et d'une présentation d'article. On satisfait donc bien à l'interprétation de l'héritage : une présentation peut être à la fois une présentation invitée et une présentation d'article. Un élément de PRESENTATION n'est donc pas du même type qu'un élément de PRESENTATIONINVITEE.

On définit ensuite l'extension des présentations. A priori, les présentations existantes peuvent être des présentations (ni invitées, ni d'article), des présentations invitées ou des présentations d'article. Il faut définir dans **les éléments existants de** *PRESENTATION* **ceux qui correspondent aussi à des éléments de** *PRESENTATIONINVITEE. Ils sont représentés par l'ensemble (PresentationInvitee2) dans le schéma de l'extension de "PRESENTATION". On exprime par la contrainte* $PresentationInvitee2 \subseteq Presentation$ *que les présentations invitées sont un sous-ensemble des présentations.*

Enfin, il faut exprimer que toute présentation qui appartient à PresentationInvitee2 correspond à un élément de PRESENTATIONINVITEE qui a les mêmes date, heure de début, durée, média et coût et satisfait aux contraintes de la classe.

$$\forall x : PresentationInvitee2 \bullet \exists y : PRESENTATIONINVITEE \bullet x.date = y.date \land x.heureDebut = y.heureDebut \land x.duree = y.duree \land x.media = y.media \land x.cout = y.cout$$

L'extension de "PRESENTATION" est décrite dans le schéma Z suivant :

PresentationExt

$$Presentation : \mathbb{F}\, PRESENTATION$$
$$PresentationInvitee2 : \mathbb{F}\, PRESENTATION$$
$$\ldots$$

$$PresentationInvitee2 \subseteq Presentation$$
$$\forall x : PresentationInvitee2 \bullet \exists y : PRESENTATIONINVITEE \bullet x.date = y.date \land x.heureDebut = y.heureDebut \land x.duree = y.duree \land x.media = y.media \land x.cout = y.cout$$
$$\ldots$$

On peut lui ajouter que "PRESENTATION" est une classe abstraite c'est-à-dire qu'une présentation est soit une présentation invitée soit une présentation d'article.

PresentationExt

$$Presentation : \mathbb{F}\, PRESENTATION$$
$$PresentationInvitee2 : \mathbb{F}\, PRESENTATION$$
$$\ldots$$

$$Presentation = PresentationInvitee2 \cup PresentationDArticle2$$
$$\ldots$$

Les objets existants de la sous-classe PresentationInvitee sont construits à partir de l'extension correspondante PresentationInvitee2 dans la super-classe.

$$
\begin{array}{|l}
\textit{PresentationInviteeExt} \\
\hline
\textit{PresentationExt} \\
\textit{PresentationInvitee} : \mathbb{F}\, \textit{PRESENTATIONINVITEE} \\
\hline
\textit{PresentationInvitee} = \\
\quad \{x : \textit{PRESENTATIONINVITEE} \mid \exists\, y : \textit{PresentationInvitee2} \bullet \\
\qquad x.\textit{date} = y.\textit{date} \land x.\textit{heureDebut} = y.\textit{heureDebut} \land \\
\qquad x.\textit{duree} = y.\textit{duree} \land x.\textit{media} = y.\textit{media} \land x.\textit{cout} = y.\textit{cout}\} \\
\hline
\end{array}
$$

Héritage d'opérations L'héritage comporte aussi l'héritage des opérations de la super-classe dans les sous-classes. Les opérations portant sur les attributs de la super-classe, doivent aussi être applicables sur ces attributs au niveau de chaque sous-classe et les opérations sur les objets existants de la super-classe doivent être appelables sur des objets de l'une de ses sous-classes. Ces mécanismes se réfèrent à l'encapsulation des opérations qui sont appelées au travers des objets d'une classe. Etant donné qu'il n'existe pas d'équivalence en Z, la solution que nous proposons pour l'héritage d'opérations ne peut être qu'une simulation.

Pour les opérations modifiant l'extension de la super-classe, les contraintes posées sur les ensembles d'objets existants de la super-classe et de ses sous-classes permettent d'effectuer l'opération à la fois au niveau des objets existants de la super-classe et de ceux de ses sous-classes. En effet, ces contraintes garantissent que tout objet d'une sous-classe correspond à un objet de la super-classe. Il est donc possible d'employer un objet d'une sous-classe dans une opération définie sur la super-classe.

Pour les opérations portant sur les attributs de la super-classe, elles sont héritées en les promouvant au niveau des extensions des sous-classes. Cette promotion s'effectue en faisant la conjonction de l'opération sur les attributs de la super-classe et de l'opération générique de promotion de la sous-classe. La puissance des calculs de schémas en Z la rend donc étonnamment simple .

Sous-proposition 12.3 : Héritage d'opérations
L'héritage des opérations sur l'extension de la super-classe est automatique compte-tenu des contraintes sur les extensions.
Pour les opérations portant sur les attributs de la super-classe CCC, elles sont promues au niveau des sous-classes CCC_i en combinant l'opération qui modifie les attributs de la super-classe et l'opération de promotion des opérations de CCC_i ($ChangeCcc_i$).

Exemple 5.12 *Par exemple, l'opération de changement d'horaire d'une présentation est promue au niveau de l'extension des présentations invitées en combinant les opérations ChangePresentationInvitee et PRESENTATIONChangeHoraire. On peut noter que l'expression de PRESENTATIONChangeHoraire est*

simplifiée par l'existence des schémas d'attributs. En effet, ici pour spécifier la modification d'une présentation qui n'influe pas sur les attributs de "PRESENTATIONINVITEE" et "PRESENTATIONDARTICLE", il suffit d'inclure $\Xi PRESENTATIONINVITEEATTR$ *et* $\Xi PRESENTATIONDARTICLEATTR$.

PRESENTATIONChangeHoraire

$$\Delta PRESENTATION$$
$$\Xi PRESENTATIONINVITEEATTR$$
$$\Xi PRESENTATIONDARTICLEATTR$$
$$nvheure? : HEURE$$

$$heureDebut' = nvheure? \land date' = date \land duree' = duree \land media' = media$$

ChangePresentationInvitee

$$\Delta PresentationInviteeExt; \Delta PRESENTATIONINVITEE$$
$$x? : PRESENTATIONINVITEE$$

$$x? \in PresentationInvitee \land \theta PRESENTATIONINVITEE = x?$$
$$PresentationInvitee' = PresentationInvitee \setminus \{x?\} \cup \{\theta PRESENTATIONINVITEE'\}$$

$$PresentationInviteeChangeHoraire == ChangePresentationInvitee \land PRESENTATIONChangeHoraire$$

5.1.6 Traduction d'une vue

Une fois que toutes les classes et leurs associations sont spécifiées, elles peuvent être groupées pour offrir une vue partielle ou globale du diagramme. Cela est réalisé en créant un nouveau schéma Z qui inclut les schémas des associations (incluant eux-mêmes les schémas d'extension des classes).

Proposition 13 : Vue
Une vue partielle ou globale d'un diagramme est représentée par un schéma Z incluant les schémas des associations de cette vue.

Exemple 5.13 *Pour la gestion de conférences, la vue globale du diagramme de classes est représentée dans le schéma GestionConferencesBD :*

GestionConferencesBD

$$PresenterRelExt; SoumettreRelExt; InscriptionRelExt; ...$$

5.1.7 Conclusion

Nos propositions de traduction en Z couvrent les principaux concepts du modèle objet. Elles permettent de préciser la sémantique des concepts d'association n-aire, d'agrégation, de composition et d'héritage du modèle objet.

Mais si elles sont satisfaisantes pour les classes, les associations et les classes associatives, elles sont complexes ou peu intuitives pour les concepts typiquement objets tels que l'héritage ou l'agrégation. Elles sont aussi mieux adaptées aux variables qu'aux opérations.

5.2 Traduction du modèle objet en Object-Z

La section précédente montre que Z ne permet pas toujours de décrire les concepts du modèle objet de façon totalement satisfaisante. C'est pourquoi nous étudions la traduction de ces mêmes concepts vers une extension orientée objet de Z, Object-Z, qui semble a priori plus adéquate. Les règles de traduction du modèle objet vers Object-Z ([DLCP97]) suivent les mêmes principes de traduction que celles présentées pour Z. Nous ne détaillons dans cette section que celles qui sont fortement influencées par les concepts objets d'Object-Z c'est-à-dire la traduction des classes, des opérations, de l'héritage, de l'agrégation et de la composition. Pour les autres concepts traités en Z, les propositions suivent les mêmes règles en remplaçant les schémas par des classes.

5.2.1 Traduction du concept de classe

Object-Z offre une structure de classe qui encapsule les attributs de la classe et ses opérations et qui fournit implicitement le concept d'identité d'objet. La traduction naturelle d'une classe semble être une classe Object-Z. Mais les classes en Object-Z ne représentent pas les objets existants de la classe (extension). Il faut donc créer une nouvelle classe ou un nouveau schéma en Object-Z pour définir l'extension d'une classe. L'extension est aussi représentée par une classe car il peut exister des opérations spécifiques à l'ensemble des objets existants. Une classe est donc décrite par deux classes Object-Z la première représentant l'intension et la deuxième l'extension de la classe.

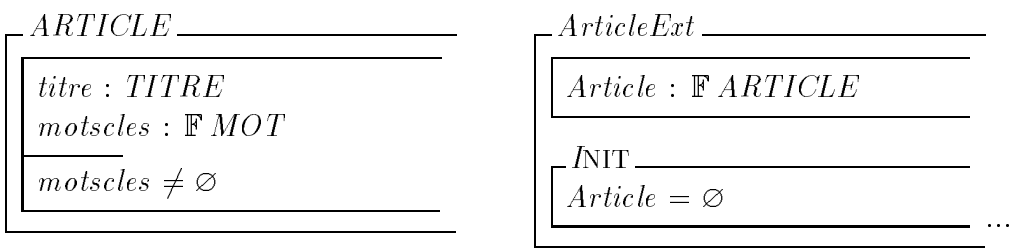
Comme pour la dérivation vers Z, les attributs sont décrits dans l'intension de la classe. Ils trouvent une traduction directe puisque Object-Z a dans le schéma d'état d'une classe des variables et des constantes qui sont appelées attributs. Le type d'un attribut peut être local à la classe ou global. On choisit de toujours utiliser pour les attributs des types globaux pour que tout type puisse être utilisable par une autre classe.

Les opérations ont aussi une traduction directe en Object-Z qui a pour avantage de les encapsuler dans une classe. Mais si elles portent sur l'intension de la classe, il faut penser à les promouvoir sur l'extension de la classe.

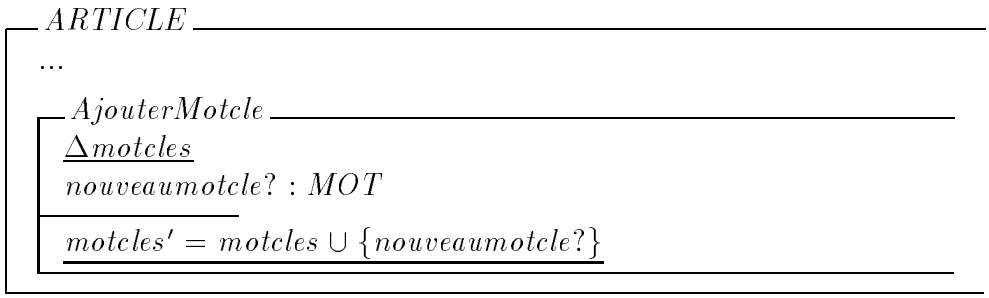
Les règles de traduction concernant les concepts de classe, d'attribut et d'opération sont similaires à celles proposées pour Z ; elles ne sont pas données ici mais sont illustrées par une exemple. Cependant en Object-Z, les opérations sont encapsulées dans les classes représentant l'intension et l'extension.

Exemple 5.14 *La classe "Article" de l'exemple 1.1 donne lieu à deux classes Object-Z : ARTICLE qui contient les attributs et ArticleExt qui a pour variable l'ensemble*

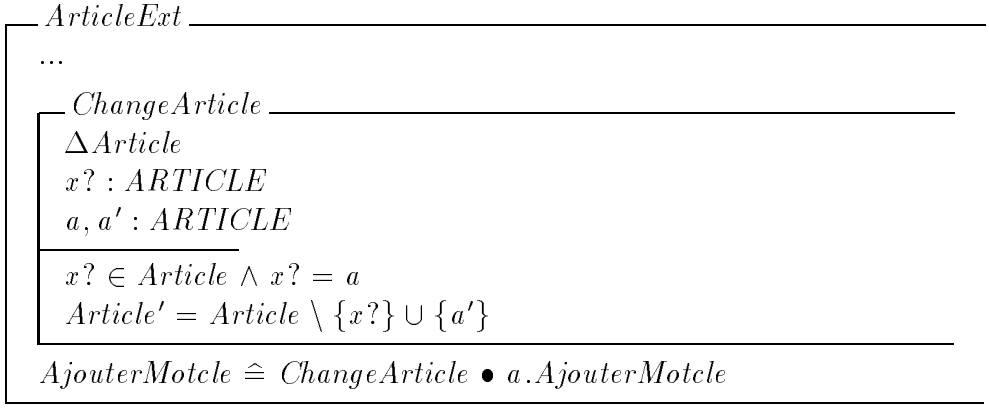
de articles existants.



L'opération "AjouterMotcle" qui ajoute un mot-clé se traduit par un schéma d'opération dans l'intension de la classe ARTICLE. Comme précédemment, les parties soulignées du schéma d'opération AjouterMotcle représentent ces compléments d'information au modèle objet.



L'opération "AjouterMotcle" portant sur l'attribut "motscles" de la classe, doit être promue au niveau de l'extension de la classe. Pour promouvoir cette opération, on introduit l'opération ChangeArticle. ChangeArticle modifie une instance de ARTICLE correspondant à x ?. a est une variable de type ARTICLE et a' dénote sa valeur finale. ChangeArticle peut être combinée avec toutes les opérations sur les attributs grâce à l'opérateur • qui est sémantiquement équivalent à la conjonction d'opérations. Pour "AjouterMotcle", l'opération AjouterMotcle promeut l'opération au niveau de l'extension de "ARTICLE". La modification de l'attribut "motscles" est promue pour un élément donné x ? de Article. Cela modifie aussi l'ensemble Article comme l'ancien élément x ?, correspondant à a est remplacé dans l'ensemble par la nouvelle valeur de a'.



5.2.2 Traduction de l'agrégation et de la composition

Agrégation Contrairement à Z, il est possible de traduire le caractère antisymétrique et transitif d'une agrégation en utilisant l'inclusion dans la classe agrégat d'une variable représentant le composant. Cela vient du fait qu'en Object-Z les variables représentent des références vers des objets. Mais à part cette particularité, l'agrégation se traduit comme une association binaire normale suivant la règle de traduction 6. L'agrégation est ainsi considérée comme un cas particulier d'association dont les caractéristiques sont définies simplement par l'ajout d'une variable.

Proposition 14 : Agrégation
L'agrégation entre une classe agrégat CA et une classe composée CC donne lieu à :

- l'inclusion dans la classe de l'intension de la classe agrégat CA d'une variable cc de type CC :
  - si l'agrégat comporte un seul objet composant (cardinalité maximale du rôle inverse de l'agrégation égale à 1), cc est un élément de CC ($cc : CC$). De plus, si l'agrégat peut ne comporter aucun objet composant (cardinalité minimale du rôle inverse de l'agrégation égale à 0), on introduit un élément particulier du type CC qui est l'élément indéfini ($indefinicc : CC$).
  - sinon (cardinalité maximale du rôle inverse de l'agrégation supérieure à 1) cc est l'ensemble des éléments de CC composant CA ($cc : \mathbb{F}\ CC$). Le cas de la cardinalité minimale du rôle inverse de l'agrégation égale à 0 est implicitement représenté par l'ensemble vide.
- la génération d'une classe traduisant les rôles de l'agrégation suivant la proposition 6 pour les associations binaires. Seule dans la spécification du rôle d'agrégation est modifiée pour faire le lien avec la variable ajoutée.

Exemple 5.15 *COMITEPROG a en plus des ses attributs, une variable chercheur qui représente les chercheurs composant un comité :*

COMITEPROG

$datereunion : DATE$
$lieureunion : LIEU$

Variable représentant l'agrégation

$chercheur : \mathbb{F}\ CHERCHEUR$

L'agrégation se traduit comme une association binaire, mais en précisant le lien entre le rôle "membres" et l'inclusion de variable faite dans COMITEPROG.

ComiteChercheurAgregExt

$comiteprogext : ComiteProgExt$
$chercheurext : ChercheurExt$
$membres : COMITEPROG \pfun \mathbb{F}\, CHERCHEUR$
$comiteprogdechercheur : CHERCHEUR \pfun \mathbb{F}\, COMITEPROG$

$dom\, membres = comiteprogext.Comiteprog$
$\bigcup(ran\, membres) \subseteq chercheurext.Chercheur$
$membres = \{c : comiteprogext.Comiteprog \bullet c \mapsto c.chercheur\}$
$comiteprogdechercheur = \{x : \bigcup(ran\, membres) \bullet x \mapsto$
$\quad \{y : dom\, membres \mid x \in membres(y) \bullet y\}\}$

Composition Une composition dénote une appartenance forte entre un agrégat et ses composés en imposant qu'un composant n'appartienne qu'à un seul agrégat et qu'il ne puisse exister sans son agrégat. La propriété d'exclusivité s'exprime en précisant que les objets existants d'une classe composante doivent impérativement être liés à un seul agrégat. On ajoute donc à l'extension d'une classe composante "CC_i" une contrainte qui impose que chacun de ses objets existants soit lié à un seul agrégat ca :
$\forall\, c : Cc_i \bullet \exists_1\, ca : Ca \bullet ca.cc_i = c$ ou $\forall\, c : Cc_i \bullet \exists_1\, ca : Ca \bullet c \in ca.cc_i$

De plus, cette contrainte impose qu'un composant ne puisse pas exister sans un agrégat. Elle exprime donc aussi la dépendance existentielle entre un composant et son agrégat.

La sémantique de la composition est explicitée de façon simple par l'écriture d'une contrainte statique. Afin de vérifier que les opérations la respectent, il est possible d'écrire, comme en Z, une obligation de preuve. Mais étant donnée l'existence de la contrainte, elle est implicite.

Proposition 15 : Composition
La composition d'une classe "CA" en classes "$CC_1, CC_2, .., CC_n$" est représentée en Object-Z comme une agrégation.
La dépendance existentielle et l'exclusivité des composants sont spécifiées par un prédicat dans l'extension la classe composante :
$\forall\, c : Cc_i \bullet \exists_1\, ca : Ca \bullet ca.c_i = c$ (s'il n'existe qu'un seul composant)
ou $\forall\, c : Cc_i \bullet \exists_1\, ca : Ca \bullet c \in ca.cc_i$ (s'il existe plusieurs composants)

où Cc_i est l'ensemble des objets de CC_i

Exemple 5.16 *Pour traduire le caractère agrégatif d'une composition, l'intension de la classe CENTREDECONFERENCE a une variable supplémentaire salle qui représente les salles d'un centre de conférences :*

CENTREDECONFERENCE

$nom : NOM$
$adresse : ADRESSE$

Variable représentant la composition

$salle : \mathbb{F}\, SALLE$

On exprime aussi l'association binaire correspondant à la composition :

ConstituerCompExt

$centredeconferenceext : CentreDeConferenceExt$
$salleext : SalleExt$
$centredesalle : SALLE \pfun CENTREDECONFERENCE$
$sallesdecentre : CENTREDECONFERENCE \pfun \mathbb{F}\, SALLE$

$dom\, centredesalle = salleext.Salle$
$ran\, centredesalle = centredeconferenceext.CentreDeConference$
$centredesalle = \{s : salleext.Salle;$
$\quad c : centredeconferenceext.CentreDeConference \mid s \in c.salle$
$\quad\quad \bullet\ s \mapsto c\}$
$sallesdecentre = \{c : ran\, centredesalle \bullet c \mapsto$
$\quad \{s : dom\, centredesalle \mid c = centredesalle(s) \bullet s\}\}$

Enfin dans l'extension de "SALLE", on exprime l'exclusivité entre un centre de conférences et une salle.

SalleExt

$centredeconference : CentreDeConferenceExt$
$Salle : \mathbb{F}\, SALLE$

$\forall s : Salle \bullet \exists_1 c : centredeconference.CentreDeConference \bullet s \in c.salle$

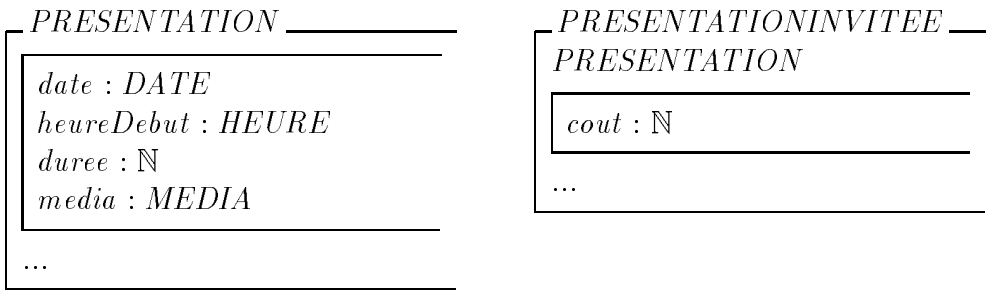
5.2.3 Traduction du concept d'héritage

La traduction la plus naturelle est l'utilisation de la structure d'héritage d'Object-Z. Mais sa sémantique est mal définie. Le fait que le concept d'héritage consiste en la fusion des éléments de la super-classe et de ceux de la sous-classe laisse penser qu'il correspond à ce que nous avons appelé précédemment l'héritage d'attributs.

Héritage d'attributs On utilise l'héritage Object-Z pour qu'une sous-classe ait les caractéristiques de sa super-classe. Cela se traduit dans la classe représentant l'intension de la sous-classe par la déclaration de sa super-classe.

Sous-proposition 16.4 : Héritage d'attributs
L'héritage d'attributs se traduit en Object-Z par la déclaration d'héritage de la super-classe dans l'intension des sous-classes.

Exemple 5.17 *Pour l'exemple des présentations, l'héritage d'attributs s'exprime par l'héritage de PRESENTATION dans PRESENTATIONINVITEE :*



Substitution d'objets A partir des classes définissant les objets possibles, on déclare les extensions des classes. Il faut pouvoir exprimer que les objets existants d'une sous-classe ont un lien avec les objets existants de sa super-classe. Pour lier les objets d'une super-classe et de ses sous-classes, on utilise tout d'abord le symbole $\downarrow$ exprimant que les objets de la super-classe sont ceux qui lui sont propres et ceux de ses sous-classes. Bien que l'héritage en Object-Z ne spécifie pas implicitement la substitution d'objets, l'existence du concept $\downarrow$ en facilite l'expression.

On peut ensuite exprimer l'inclusion des objets d'une sous-classe "CCC_i" dans sa super-classe "CCC" par un prédicat dans l'extension de la super-classe : $ccciext.Ccc_i \subseteq Ccc$

Si la super-classe "CCC" est non-instanciable, ces objets appartiennent à l'une de ses sous-classes "$CCC_1, ..., CCC_n$" :
$\forall\, c : Ccc \bullet c \in ccc_1 ext.Ccc_1 \lor ... \lor c \in ccc_n ext.Ccc_n$

Sous-proposition 17.5 : Substitution d'objets
La substitution des objets d'une sous-classe "CCC_i" dans l'ensemble des objets de la super-classe "CCC" se traduit lors de la déclaration de l'ensemble des objets de la super-classe Ccc par $\downarrow$ qui représente l'ensemble des objets possibles de CCC et de ses sous-classes (Ccc : $\downarrow \mathbb{F}$ CCC).
Enfin, on exprime des contraintes d'inclusion entre les objets de la super-classe Ccc et ceux des sous-classes "$CCC_1, ..., CCC_n$" ($ccciext.Ccc_i \subseteq$ Ccc).
Si la super-classe est non-instanciable, la contrainte $\forall\, c : Ccc \bullet c \in ccc_1 ext.Ccc_1 \lor ... \lor c \in ccc_n ext.Ccc_n$ est aussi ajoutée à l'extension de la super-classe.

Exemple 5.18 *Dans la spécification de l'extension de "PRESENTATION", on exprime en employant $\downarrow$ dans la déclaration de Presentation que l'ensemble des présentations existantes Presentation est constitué des objets de "PRESENTATION"*

mais aussi de ceux de "PRESENTATIONINVITEE" et de "PRESENTATIONDARTICLE". On précise aussi que l'ensemble des présentations invitées est inclus dans l'ensemble des présentations (presentationinviteeext.PresentationInvitee ⊆ Presentation) et qu'une présentation est soit invitée soit d'article.

PresentationExt

$$
\begin{array}{l}
presentationinviteeext : PresentationInviteeExt \\
presentationdarticleext : PresentationDArticleExt \\
Presentation : \mathbb{F} \downarrow PRESENTATION \\
\ldots \\
\hline
presentationinviteeext.PresentationInvitee \subseteq Presentation \\
presentationdarticleext.PresentationDArticle \subseteq Presentation \\
\forall\, p : Presentation \bullet \\
\quad p \in presentationinviteeext.PresentationInvitee \lor \\
\quad p \in presentationdarticleext.PresentationDArticle \\
\ldots \\
\ldots
\end{array}
$$

Héritage d'opérations Pour les opérations, l'héritage en Object-Z permet à une sous-classe d'hériter des opérations de sa super-classe. Mais ces opérations peuvent être totalement redéfinies ou supprimées dans la sous-classe. Cela n'étant pas autorisé dans le modèle objet, nous interdisons cette pratique lors du passage à Object-Z.

Les opérations sur l'intension d'une super-classe sont héritées au niveau de l'intension d'une sous-classe par le concept d'héritage d'Object-Z. Leur promotion au niveau de l'extension se fait aussi automatiquement en utilisant lors de la promotion au niveau de l'extension de la super-classe le symbole ↓ c'est-à-dire en écrivant l'opération générique de promotion de la façon suivante :

ChangeCcc

$$
\begin{array}{l}
\Delta Ccc \\
x? : \downarrow CCC \\
c, c' : \downarrow CCC \\
\hline
x? \in Ccc \land x? = c \\
Ccc' = (Ccc \setminus \{x?\}) \cup \{c'\}
\end{array}
$$

En composant ensuite *ChangeCcc* avec l'opération sur l'intension de la super-classe "O", on obtient une opération applicable aux instances de la superclasse et de ses sous-classes.

$OCccExt \mathrel{\widehat=} ChangeCcc \bullet c.O$

Une opération sur l'extension de la super-classe est héritée en déclarant que tous les objets manipulés lors de l'opération sont soit des objets de la super-classe, soit des objets des sous-classes c'est-à-dire en utilisant le symbole ↓ pour toutes les variables du type de la super-classe.

L'usage de l'héritage et du symbole $\downarrow$ simplifient l'expression de l'héritage d'opérations en Object-Z. De plus, on bénéficie toujours de la puissance des calculs de schémas pour effectuer la promotion des opérations sur l'intension de la super-classe.

Sous-proposition 18.6 : Héritage d'opérations
L'héritage des opérations de la super-classe dans une sous-classe s'effectue automatiquement grâce à l'utilisation de l'héritage et de l'opérateur $\downarrow$ d'Object-Z dans les spécifications des opérations de la super-classe.

Exemple 5.19 *Prenons comme exemple l'opération de changement d'horaire d'une présentation qui est au niveau de l'intension de "PRESENTATION" :*

PRESENTATION

...

ChangeHoraire

$\Delta heureDebut$
$nvheure? : HEURE$

$heureDebut' = nvheure?$

Cette opération est automatiquement héritée au niveau des intensions de "PRESENTATIONINVITEE" et "PRESENTATIONDARTICLE". Pour la promouvoir au niveau de leur extension, il faut modifier l'opération générique de promotion de leur super-classe "ChangePresentation". L'utilisation de $\downarrow$ permet d'exprimer que "ChangePresentation" est applicable à une présentation, à une présentation invitée ou à une présentation d'article :

PresentationExt

...

ChangePresentation

ΔCcc
$x? : \downarrow PRESENTATION$
$p, p' : \downarrow PRESENTATION$

$x? \in Presentation \land x? = p$
$Presentation' = (Presentation \setminus \{x?\}) \cup \{p'\}$

$ChangeHoraire \mathrel{\widehat=} ChangePresentation \bullet p.ChangeHoraire$

5.2.4 Conclusion

Nos propositions de traduction en Object-Z couvrent elles-aussi les principaux concepts du modèle objet. Nous avons décrit ici seulement celles (encapsulation d'opérations, agrégation, composition et héritage) qui se distinguent des propositions concernant Z grâce à l'utilisation des concepts objets d'Objet-Z. Elles nous semblent proposer des traductions simples et assez intuitives de ces concepts.

5.3 Comparaison

5.3.1 Comparaison entre les traductions en Z et Object-Z

Comparaison syntaxique Nous comparons Z et Object-Z suivant leur adéquation au modèle objet pour le schéma de traduction présenté ci-dessus (d'autres principes de traduction peuvent aussi être envisagés). Cette adéquation est évaluée en comparant la complexité des spécifications formelles obtenues suivant des règles de traduction équivalentes. Nous choisissons de mesurer la complexité en comptant pour chaque concept le nombre de structures nécessaires pour l'exprimer dans le langage formel cible et le nombre de lignes de spécifications formelles obtenues (au format Latex). Pour Z, la structure prise en compte est le schéma. Pour Object-Z, nous avons choisi la classe pour tous les concepts sauf pour les opérations où nous comptabilisons les schémas d'opération qui sont les équivalents des schémas Z.

Le tableau 5.1 résume les valeurs permettant la comparaison syntaxique de Z et d'Object-Z pour nos règles de traduction. Nous ne détaillons ici que les associations binaires dans la mesure où leur cas peut être généralisé aux associations n-aires.

Concept	Z		Object-Z	
	Nb. structures	Nb. lignes	Nb. structures	Nb. lignes
classe	3 schémas	16	2 classes	20
opération sur				
- l'intension	2 schémas	10	2 schémas	7
- l'extension	1 schéma	6	1 schéma	6
assoc. binaire	1 schéma	10	1 classe	13
agrégation	1 schéma	10	2 classes	14
héritage	6 schémas + 1 par op. sur attr. sur-classe	32 + 3 par op. sur attr. sur-classe	4 classes	29

TAB. 5.1: Tableau récapitulatif

Ce tableau montre qu'en termes de structures ou de nombres de lignes employés lors de notre principe de traduction, Z et Object-Z offrent des solutions équivalentes dans de nombreux cas (classe, opération, association et agrégation).

Z semble parfois mieux adapté qu'Object-Z ; c'est le cas pour notre proposition de traduction des associations. La spécification Object-Z peut être considérée comme moins lisible dans la mesure où une association se traduit dans une "double" structure (un schéma d'état dans une classe).

Par contre, il existe une nette différence dans la représentation de l'héritage. En effet, il est très difficile de traduire l'héritage en Z. De nombreux schémas doivent être ajoutés dans un certain ordre qui rend les spécifications Z difficilement traçables et lisibles. Object-Z permet de traduire l'héritage dans des spécifications formelles plus simples et plus claires.

Comparaison sémantique Sémantiquement nous nous sommes employée à proposer pour un concept des solutions équivalentes en Z et Object-Z. Cependant la

représentation en Object-Z est généralement plus proche de la sémantique souhaitée.

Pour les concepts objet (opérations, agrégation ou héritage), Object-Z semble plus approprié que Z. En effet, si Z et Object-Z sont équivalents pour représenter le type d'une classe, Object-Z est plus adéquat que Z en tenant compte des opérations. Son concept de classe permet d'encapsuler les opérations avec la structure de la classe. Le lien entre une classe du modèle objet et une classe Object-Z s'effectue sans difficulté. Z n'ayant pas ce concept d'encapsulation d'opérations, les schémas se multiplient sans qu'il y ait de lien direct entre les schémas décrivant l'intension et l'extension de la classe et les opérations. La traçabilité est donc plus difficile entre le modèle objet et Z. De même, la représentation de l'agrégation ou de la composition est plus intuitive en Object-Z car elle exprime de façon plus naturelle et systématique les particularités d'une agrégation ou d'une composition.

Enfin, l'utilisation de l'héritage d'Object-Z offre a priori une solution sémantiquement plus proche que celle de Z pour traduire l'héritage du modèle objet. Mais le flou sémantique de l'héritage en Object-Z oblige à nuancer cette remarque. La sémantique d'Object-Z n'étant pas totalement définie, il est exagéré d'affirmer que la représentation de l'héritage en Object-Z est sémantiquement plus satisfaisante.

Conclusion Z semble bien adapté pour traduire les concepts "relationnels" de type d'une classe ou d'association. Les expressions Z et Object-Z de ces concepts sont équivalentes. Par contre, la traduction en Z de l'encapsulation d'opérations ou de l'héritage est complexe. Object-Z est beaucoup mieux adapté pour traduire les concepts orientés objet car il dispose à la fois d'une structure regroupant l'état et les opérations, la classe, et du concept d'héritage. De plus, l'agrégation se traduit de façon plus naturelle en permettant de faire référence aux composants directement à partir d'un objet agrégat.

L'utilisation d'Oject-Z permet d'obtenir des spécifications formelles dont la structure est proche de celle du modèle objet correspondant. Les spécifications formelles en Object-Z sont plus lisibles et moins lourdes que celles en Z. Object-Z est donc plus adéquat comme langage cible de la traduction des modèles objet. Néanmoins, Object-Z souffre d'un flou sémantique (en particulier pour l'héritage) qui minimise le gain en précision apporté par les spécifications formelles.

Cependant, nous avons privilégié Z dans cette thèse car le manque d'outils d'Object-Z fait que ses spécifications peuvent difficile être exploitées. Dans l'attente d'outils d'analyse pour Object-Z, nous avons travaillé avec Z afin de pourvoir valider certaines idées sur les bénéfices apportés par le passage à des spécifications formelles.

5.3.2 Comparaison avec les travaux similaires

Nous reprenons le tableau récapitulant les travaux existants sur la traduction du modèle objet (chap. 4) en indiquant les concepts pris en compte dans ce travail.

Ce tableau appelle plusieurs remarques. Tout d'abord, nous nous sommes attachée à traiter les principaux concepts du modèle objet. Nos propositions sont plus complètes que toutes celles disponibles à ce jour. De plus, nous avons essayé de re-

Concepts / Articles	Classe	Attri--but	Opé--ration	Association		Classe associative	Agré--gation	Héri--tage
				binaire	n-aire			
[NR94]	T$^-$	T	A	T	P	T	T$^-$	T$^-$
[LHW96]	T	T		T				T$^-$
[Ngu98]	T	T	A	T	T$^-$	T	T$^-$	T
[MS99]	T	T	A	T	P	T	T$^-$	T
[FB97]	T	T		T		P	T	T$^-$
[FBLP97]	T	T		T		T	T	T$^-$
[SF97]	T	T		T		T	T	T$^-$
[FBLPS97]	T	T	T	T		T	T	T$^-$
[Lan95]	T$^-$	T	T$^-$	T			T	T$^-$
[AS97]	T	T	P	T				T
[KC99]	T	T	T$^-$	T		T	T	T
ce travail	T	T	T	T	T	T	T	T

TAB. 5.2: Synthèse des travaux sur le modèle objet

présenter toute la complexité de la sémantique des concepts du modèle en suivant les descriptions données dans des ouvrages tels que [UML99]. De ce point de vue, les travaux existants divergent du nôtre par au moins l'un des aspects suivants.

Certains travaux [NR94, LG96] ne décrivent que la définition en intension d'une classe. Nous pensons que le fait de ne pas représenter systématiquement l'extension d'une classe produit une spécification formelle incomplète. Au contraire, [FB97, SF97] estiment que la distinction entre intension et extension n'est pas toujours nécessaire car les deux interprétations capturent le fait qu'une classe caractérise une collection d'instances. Dans ces propositions, les extensions sont intégrées à des schémas servant à représenter les associations. Des ensembles d'objets d'une même classe se trouvent dans différents schémas sans qu'il y ait de liens entre eux. Ce manque de représentation explicite des objets existants d'une classe rend les spécifications Z moins traçables et moins lisibles.

Les identifiants d'objets sont souvent représentés de manière explicite [LHW96, FB97, SF97, Ngu98, MS99]. Cette notion existe dans le modèle objet mais nous ne jugeons pas nécessaire de la coder explicitement car elle est implicite dans les systèmes à objets. Néanmoins si on suppose que plusieurs objets d'une classe peuvent avoir des valeurs identiques pour tous les attributs (pas de notion de clé), l'expression explicite d'un identifiant est obligatoire en Z. Notre proposition de traduction d'une classe en Z peut alors être modifiée pour ajouter au schéma d'intension une variable représentant l'identifiant.

Notre proposition en couvrant les associations n-aires, est plus complète que celles des autres travaux existants qui se limitent au cas des associations binaires. Néanmoins [Ngu98] évoque les associations non binaires en utilisant une représentation équivalente grâce à l'usage d'une classe associative. Bien que [NR94] n'en parle pas, il pourrait les traiter dans la mesure où le principe de traduction proposé est similaire au nôtre.

La sémantique de l'héritage peut être différente de la nôtre : [Lan95, LG96] n'expriment pas que les objets existants d'une sous-classe sont des objets de la

super-classe. Mais la différence la plus importante porte sur le fait que l'héritage d'opérations n'est pas évoqué. Pour les travaux ayant comme langage cible un langage formel orienté objet [Lan95, AS97, KC99], on peut supposer que les opérations d'une super-classe sont héritées grâce au mécanisme d'héritage. Mais pour certains travaux utilisant Z [FB97, FBLP97, SF97] ou B [NR94, LHW96], rien ne dit comment le décrire. Enfin [Ngu98, MS99] ne pouvant l'exprimer en B, proposent un moyen de le simuler.

Nous avons essayé de faciliter la traçabilité entre spécifications semi-formelles et formelles en fournissant des spécifications Z et Object-Z les plus proches possible de la structure du modèle objet. Or certains travaux [FB97, FBLP97, SF97, FBLPS97] multiplient le nombre de schémas Z pour représenter des structures d'héritage ou d'agrégation. En particulier dans [FB97], pour pallier l'absence de concepts objet dans Z, la traduction des concepts simples demande des constructions complexes : une classe se traduit par un type donné représentant l'ensemble de ses identifiants, un schéma pour son intension, un schéma pour son extension ayant pour variables l'ensemble des identifiants existants, ses attributs et ses opérations ; une association se traduit dans un schéma de spécification de l'association pour représenter les rôles et leur cardinalité, un schéma pour la structure de l'association, une variable dans les schémas des extensions des classes de l'association. Il nous semble que cette complexité nuit à la lisibilité et à la traçabilité des spécifications Z ainsi produites.

5.4 Conclusion

Nous avons présenté dans ce chapitre nos propositions de traduction des principaux concepts du modèle objet vers Z et Object-Z. Nous avons tout d'abord décrit celles allant vers Z. Pour celles concernant Object-Z, nous n'avons donné que celles pour lesquelles le principe de traduction diffère de celui de Z.

Ces propositions nous ont permis de comparer l'adéquation de Z et d'Object-Z au modèle objet pour notre type de traduction et de conclure que l'utilisation d'Object-Z permettait d'obtenir des spécifications formelles plus simples et plus lisibles.

Enfin nous avons positionné notre travail par rapport aux travaux existants sur la traduction de modèle objet vers des langages formels orientés modèle ou objet.

Chapitre 6

Exploitation des spécifications Z produites

Dans le chapitre 3, nous avons décrit la démarche du projet Champollion que nous utilisons pour coupler divers types de notations. Elle est basée sur la traduction de spécifications semi-formelles en spécifications formelles et sur l'intégration de certaines contraintes dans les spécifications formelles. A partir des spécifications formelles ainsi produites, il est possible d'effectuer des analyses fines ou de raisonner. Dans la partie précédente, nous avons proposé des règles de traduction pour automatiser la génération de spécifications Z ou Object-Z. Dans ce chapitre, nous privilégions l'utilisation de Z vu la disponibilité d'outils d'analyse. Nous montrons comment des contraintes peuvent être intégrées aux spécifications Z produites. Puis nous donnons deux exemples d'utilisation des spécifications Z afin de mieux comprendre un modèle ou d'en corriger d'éventuelles erreurs.

6.1 Guide méthodologique pour l'expression des contraintes

6.1.1 Introduction

Un modèle objet ne permet pas d'exprimer toutes les contraintes de domaine ou de gestion nécessaires à une spécification correcte du système. C'est pourquoi des notations telles que l'Object Constraint Language (OCL [WK98]) ont été développées. Mais OCL offre aujourd'hui une sémantique pauvre et ne possède pas de support outil permettant d'exploiter les contraintes ; en conséquence, les contraintes peuvent être mutuellement incohérentes ou incohérentes avec le modèle qu'elles annotent. De plus, il n'existe pas de guide méthodologique pour savoir où exprimer une contrainte afin de la rendre exploitable. Lors de l'évolution d'un diagramme, il est alors difficile de cerner quelles sont les contraintes touchées par les modifications. Dans cette section, nous proposons une classification des contraintes qui permet de savoir où écrire chacune d'entre elles.

Dans le domaine des bases de données, de nombreuses classifications des contraintes d'intégrité ([AF94, Dat95, DA82]) ont déjà été proposées. Elles dis-

tinguent généralement les contraintes statiques des contraintes dynamiques. Dans ce travail, nous ne nous intéressons qu'aux contraintes statiques qui correspondent à celles que peut exprimer OCL. Des diverses classifications, nous retenons qu'il existe trois types de contraintes d'intégrité statiques : les contraintes intra-objet portant sur un ou plusieurs attributs d'un objet ; les contraintes intra-classe qui portent sur plusieurs objets d'une classe et les contraintes inter-classes portant sur plusieurs classes de la base de données. Nous pouvons prendre en compte ces différents types de contraintes statiques, mais nous adoptons une classification différente afin de pouvoir guider l'écriture ou la modification des contraintes dans une démarche de modélisation.

Notre classification [Dup00] est basée sur l'identification des limites d'expression du modèle objet grâce à son expression en spécifications formelles. En effet, la traduction du modèle objet en Z aide à localiser ces limites car les "vides" de squelettes Z correspondent aux aspects que le diagramme n'exprime pas. L'identification de ces "vides" permet de classer les contraintes pour créer un guide méthodologique d'aide à l'expression des contraintes.

Comme OCL semble avoir été adopté pour écrire des contraintes sur le diagramme de classes d'UML, nous tentons de rester proche des habitudes de travail des concepts en exprimant les contraintes en OCL même si cela multiplie les intermédiaires entre les spécifications semi-formelles et formelles. Nous identifions là où ces contraintes doivent être exprimées en complément d'un modèle objet. OCL tirant ses origines de Z, il est possible d'écrire les contraintes en OCL, puis de les reformuler en Z pour les intégrer aux squelettes de spécifications Z. Pour chaque type de contrainte, nous montrons comment une contrainte OCL peut être écrite en Z. La plupart ces contraintes ont été vérifiées syntaxiquement avec le parser d'OCL d'IBM [IBM]. Dans le cas contraire, nous expliquons pourquoi cela n'a pas été possible. Par souci de concision, certaines contraintes sont seulement exprimées en Z, mais leur expression correspondante en OCL peut être facilement produite.

6.1.2 Classification des contraintes

Contraintes sur une classe

Une classe comporte deux aspects : l'intension et l'extension. Les contraintes sur une classe peuvent donc concerner un seul objet ou les connexions mutuelles entre plusieurs objets. Les contraintes sur un objet sont une restriction des valeurs d'un ou de plusieurs attributs alors que les restrictions sur les connexions sont des contraintes sur l'extension de la classe. Nous illustrons ces deux types de contraintes sur la classe "SESSION" (Fig. 1(a)). Suivant nos règles de traduction en Z, la classe "SESSION" donne lieu à deux schémas Z *SESSION* et *SessionExt* dont la partie "prédicats" est vide (Fig.1(b)). On peut noter que le schéma *SESSION* a en plus des attributs de la classe, une variable *conference* qui est nécessaire car une conférence est un élément de la clé d'une session.

Contraintes sur l'intension d'une classe Ces contraintes sont divisées en deux catégories : celles concernant un seul attribut et celles portant sur plusieurs attributs.

(a) Classe "SESSION"

SESSION

$titre : TITRE$
$date : DATE$
$heureDebut : HEURE$
$heureFin : HEURE$
$type : TYPESESSION$
$nbpres : \mathbb{N}$
$prix : \mathbb{N}$
$conference : CONFERENCE$

...

SessionExt

$Session : \mathbb{F}\, SESSION$

...

(b) Schémas Z *SESSION* et *SessionExt*

FIG. 6.1: Classe "SESSION" et sa représentation en Z

Un exemple de contraintes sur un seul attribut est la contrainte spécifiant que le prix d'une session ne peut pas dépasser un montant donné (e.g. 5000 FF). Cette contrainte s'exprime en OCL dans le contexte de la classe "SESSION" : pour toute session donnée *self*, la valeur de l'attribut "prix" est inférieure à 5000 FF.

SESSION
self.prix < 5000 ;

Cette contrainte est réécrite en syntaxe Z ($prix < 5000$) pour compléter le squelette du schéma d'intension *SESSION* (Fig.1(b)).

Il est aussi possible d'exprimer une contrainte sur plusieurs attributs sur la classe "SESSION". Par exemple, l'heure de début d'une session doit être inférieure à son heure de fin. Pour écrire cette contrainte, il faut pouvoir comparer deux heures. Aussi nous considérons que le type "HEURE" est constitué de deux champs "hr" et "min" représentant les heures et les minutes. La comparaison entre deux heures s'effectue alors en comparant "hr" et "min" de chacune des heures. Comme elle correspond à une opération courante sur les heures, nous définissons en OCL une opération "EstSup" sur "HEURE" qui compare une heure "h1" à partir de laquelle l'opération est appelée avec une autre heure "h2" donnée en paramètre. La postcondition de "EstSup" spécifie que "h1" est supérieure à "h2" si les heures de "h1" sont supérieures aux heures de "h2" ou si les heures de "h1" et "h2" sont égales et les minutes de "h1" sont supérieures aux minutes de "h2".

h1.EstSup(h2 : HEURE) : Boolean
Post : h1.hr > h2.hr or (h1.hr=h2.hr and h1.min> h2.min) ;

La contrainte comparant l'heure de début et l'heure de fin d'une session correspond alors à l'utilisation de l'opération "EstSup". Elle s'exprime dans le contexte de "SESSION".

SESSION
self.heureFin.EstSup(self.heureDebut) ;

De façon similaire, un type *HEURE* et une relation *EstSup* sont introduits en Z. L'expression de la contrainte sur les heures en Z est alors équivalente à celle OCL :

$$(heureFin, heureDebut) \in EstSup$$

En ajoutant au schéma *SESSION* l'expression de ces contraintes, on obtient le schéma suivant :

SESSION

$titre : TITRE$
$date : DATE$
$heureDebut : HEURE$
$heureFin : HEURE$
$type : TYPESESSION$
$nbpres : \mathbb{N}$
$prix : \mathbb{N}$
$conference : CONFERENCE$

$prix < 5000$
$(heureFin, heureDebut) \in EstSup$

Contraintes sur l'extension d'une classe Les contraintes sur l'extension d'une classe concernent les connexions mutuelles entre les objets d'une même classe. Ainsi deux objets doivent avoir des valeurs distinctes pour leurs attributs clés. La contrainte de clé est un exemple typique de contrainte sur l'extension d'une classe. Pour la classe "SESSION", la clé est constituée de la conférence à laquelle appartient la session et de son titre. Elle spécifie que deux instances de "SESSION" (la session donnée "self" et une autre "s1") doivent avoir une conférence ou un titre distinct.

SESSION
self ->forAll(s1 | s1<>self implies
s1.conference<>self.conference or s1.titre <> self.titre) ;

Cette contrainte peut facilement être traduite en Z dans une expression équivalente :

$$\forall s1, s2 : Session \mid s1 \neq s2 \bullet s1.conference \neq s2.conference \lor s1.titre \neq s2.titre$$

Mais les contraintes sur les objets d'une classe ne se limitent pas aux contraintes de clé. Par exemple, le fait qu'une session pleinière n'ait pas d'autre session de la

même conférence en parallèle est une contrainte sur l'ensemble des sessions. Cette contrainte nécessite de savoir si deux plages horaires définies par leurs heures de début et de fin se recouvrent. A l'instar de la fonction de comparaison des heures, une fonction de recouvrement des plages horaires est définie. Elle spécifie que deux couples d'heures *(d1,f1)* et *(d2,f2)* se recouvrent si *d2* est compris entre *d1* et *f1* ou si *d1* est compris entre *d2* et *f2*.

$$HeureRecouvr : (HEURE \times HEURE) \leftrightarrow (HEURE \times HEURE)$$

$$\begin{array}{l}\forall\, d1, f1, d2, f2 : HEURE \mid \\ \quad ((d1, f1), (d2, f2)) \in HeureRecouvr \land (d1, f1) \in EstSup \land \\ \quad (d2, f2) \in EstSup \bullet d1 = d2 \lor \\ \qquad ((d2, d1) \in EstSup \land (f1, d2) \in EstSup) \lor \\ \qquad ((d1, d2) \in EstSup \land (f2, d1) \in EstSup)\end{array}$$

La contrainte sur les sessions pleinières exprime que si s et p sont deux sessions différentes d'une même conférence, si p est une session pleinière alors s doit être programmée à une date ou à une plage horaire différente de celle de s :

$$\begin{array}{l}\forall\, p, s : Session \mid p.conference = s.conference \land p.type = pleiniere \land s \neq p \bullet \\ p.date \neq s.date \lor \\ ((s.heureDebut, s.heureFin), (p.heureDebut, p.heureFin)) \notin HeureRecouvr\end{array}$$

Après ajout de la contrainte de clé et de celle sur les sessions pleinières, on obtient le schéma *SessionExt* suivant :

SessionExt

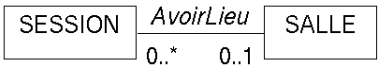
$$Session : \mathbb{F}\, SESSION$$

$$\begin{array}{l}\forall\, s1, s2 : Session \mid s1 \neq s2 \bullet \\ \quad s1.conference \neq s2.conference \lor s1.titre \neq s2.titre \\ \forall\, p, s : Session \mid p.conference = s.conference \land p.type = pleiniere \land s \neq p \bullet \\ p.date \neq s.date \lor \\ ((s.heureDebut, s.heureFin), (p.heureDebut, p.heureFin)) \notin HeureRecouvr\end{array}$$

Contraintes sur une association

Les contraintes sur les associations correspondent à des restrictions sur des objets liés par une association. Pour tous les types d'associations (n-aires, binaires, agrégation et composition), les contraintes portent sur les instances des classes liées par l'association, donc sur le schéma Z représentant l'extension de l'association par ses rôles. Ce schéma est celui qui est utilisé dans le cas de la traduction simplifiée des associations binaires. Ainsi un exemple d'association binaire tel que l'association "AvoirLieu" entre "SESSION" et "SALLE" (Fig. 2(a)) illustre aussi les cas des autres types d'associations. L'association "AvoirLieu" se traduit par un schéma Z *AvoirLieuRel* qui définit les rôles entre les sessions et les salles existantes.

Pour cette association, il faut s'assurer que deux sessions n'ont pas lieu en même temps dans la même salle. En OCL, comme les associations ne sont pas des objets de



(a) Association entre "SESSION" et "SALLE"

AvoirLieuRel

$SalleExt; SessionExt$

$salledesession : SESSION \pfun SALLE$

$sessiondesalle : SALLE \pfun \mathbb{F}\, SESSION$

...

(b) Schéma Z *AvoirLieuRel*

FIG. 6.2: Association entre "SESSION" et "SALLE" et sa représentation en Z

première classe, il faut choisir la classe dans laquelle la contrainte est écrite. Ici nous avons choisi de l'écrire dans la classe "SESSION" car son expression est facilitée par le fait que le rôle allant à "SALLE" est monovalué. L'expression OCL de cette contrainte signifie que pour une session, toute autre session qui a lieu à la même date et pendant des heures communes, ne se déroule pas dans la même salle. Elle se réfère donc à la notion de recouvrement des heures et peut à ce titre utiliser une opération équivalente à la fonction de recouvrement *HeureRecouvr* que nous avons définie en Z. Nous supposons qu'une telle opération existe, bien que nous ne sachions pas comment en OCL définir et utiliser une opération sur un couple d'objets (ici un couple d'heures) :

SESSION
self -> forAll (s1 | self<> s1 and s1.date=self.date and
(self.heureDebut,self.heureFin).HeureRecouvr(s1.heureDebut,s1.heureFin)
implies s1.salledesession <> self.salledesession) ;

En Z, cette contrainte est écrite dans le schéma de l'association "AvoirLieu". Il n'est donc pas nécessaire de choisir son contexte, précisé de facto par le schéma. De plus, nous nous servons de la fonction *HeureRecouvr* pour simplifier son écriture.

AvoirLieuRel

$SalleExt; SessionExt$

$salledesession : SESSION \pfun SALLE$

$sessiondesalle : SALLE \pfun \mathbb{F}\, SESSION$

...

$\forall s1, s2 : Session \mid s1 \neq s2 \land s1.date = s2.date \land$
$((s1.heureDebut, s1.heureFin), (s2.heureDebut, s2.heureFin))$
$\in HeureRecouvr$
$\bullet\ salledesession(s1) \neq salledesession(s2)$

Contraintes sur une classe associative

Comme une classe, une classe associative est définie en intension et en extension. Les contraintes sur une classe associative peuvent donc porter sur son intension s'il

s'agit de restrictions sur les valeurs d'attributs d'un seul objet ou sur son extension dans le cas de restrictions mutuelles entre les instances en relation.

Pour la gestion de conférences, nous avons introduit précédemment la classe associative "Inscription" qui lie un chercheur à une conférence (Fig. 6.3). Cette classe associative se traduit en Z par deux schémas *InscriptionRel* et *InscriptionRelExt*.

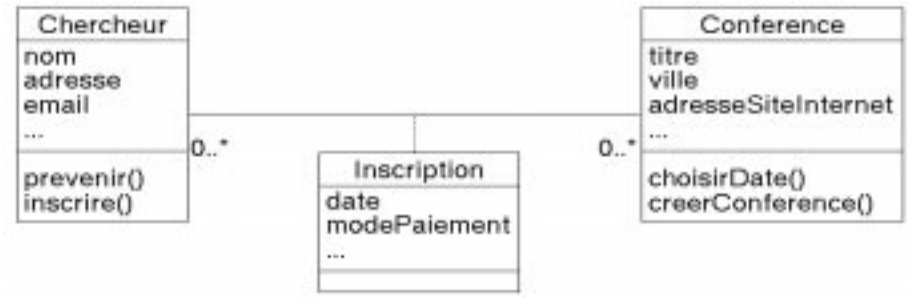


FIG. 6.3: Classe associative "Inscription"

InscriptionRel

$chercheur : CHERCHEUR$
$conference : CONFERENCE$
$date : DATE$
$modePaiement : MODEPAIEMENT$
$somme : \mathbb{N}$

...

InscriptionRelExt

$ChercheurExt; ConferenceExt$
$Inscription : \mathbb{F}\ InscriptionRel$
$chercheurinscrit :$
$\quad CHERCHEUR \pfun \mathbb{F}\ CONFERENCE$
$conferencedechercheur :$
$\quad CONFERENCE \pfun \mathbb{F}\ CHERCHEUR$

...

Contraintes sur l'intension de la classe associative Comme pour les classes, les contraintes sur l'intension de la classe associative restreignent les valeurs d'un n-uplet représentant les objets liés et les attributs spécifiques. Par exemple, la date d'une inscription à une conférence doit être antérieure à la date de début de cette conférence. Cette contrainte nécessite de savoir comparer deux dates. Comme pour la définition de "EstSup" sur les heures, il faut compléter la définition du type "DATE" par une opération "estAvant" qui renvoie vrai si la date considérée est antérieure à une date donnée en paramètre. Dans le cas d'une inscription, il faut que sa date ait lieu avant la date de début de la conférence :

INSCRIPTION
self.date.estAvant(self.conference.dateDebut) ;

En Z, la comparaison de deux dates nous fait compléter la définition du type de base *DATE* par une relation d'ordre total non-réflexive *DateInf* : deux dates d1 et d2 peuvent être comparées si d1 et d2 ne sont pas égales et $(d1, d2) \in DateInf$ signifie que d1 est antérieure à d2. La contrainte sur la date d'inscription s'écrit alors $(date, conference.dateDebut) \in DateInf$ et complète le schéma d'intension d' "INSCRIPTION".

InscriptionRel

$chercheur : CHERCHEUR$
$conference : CONFERENCE$
$date : DATE$
$modePaiement : MODEPAIEMENT$
$somme : \mathbb{N}$

$(date, conference.dateDebut) \in DateInf$

Contraintes sur l'extension de la classe associative Elles concernent les connexions mutuelles entre plusieurs objets liés par l'association. Par exemple, le nombre de participants à une conférence ne peut pas excéder la capacité maximale de cette conférence. Il suffit d'exprimer que la capacité d'une conférence donnée est supérieure au nombre de chercheurs inscrits. En OCL, nous avons choisi de tirer profit de l'association représentée par "INSCRIPTION" en naviguant à partir d'une conférence vers les chercheurs qui y sont inscrits. Cette contrainte est donc écrite à partir du contexte "CONFERENCE" ("size" représentant le nombre d'éléments d'un ensemble). Nous utilisons ici le symbole "->" pour accéder aux propriétés de l'ensemble des chercheurs inscrits et le "." pour se référer aux propriétés d'un seul objet, ici self.

CONFERENCE
self.capaMax >= self.chercheurinscrit -> size ;

Comme pour les contraintes sur les associations, il n'est pas nécessaire de choisir le contexte de cette contrainte en Z. Elle s'écrit de facto dans le schéma de l'extension d"'INSCRIPTION". Elle spécifie que pour toute conférence (dont on connaît des inscrits), le nombre des inscrits (la cardinalité #) est inférieur ou égal à sa capacité maximale.

InscriptionRelExt

$ChercheurExt; ConferenceExt$
$Inscription : \mathbb{F}\, InscriptionRel$
$chercheurinscrit : CHERCHEUR \pfun \mathbb{F}\, CONFERENCE$
$chercheurdeconference : CONFERENCE \pfun \mathbb{F}\, CHERCHEUR$

...
$\forall\, c : \mathrm{dom}\, chercheurdeconference \bullet$
$\quad \#(chercheurdeconference(c)) \leq c.capaMax$

Contraintes sur une relation d'héritage

Comme les autres propriétés d'une super-classe, ses contraintes sont héritées dans ses sous-classes. Dans [BM95], des règles d'héritage des contraintes sont définies comme des conditions à respecter lors de la redéfinition des contraintes héritées dans les sous-classes : les contraintes peuvent être redéfinies par restriction ou par augmentation. Une règle de restriction de contraintes correspond au cas où une contrainte de la super-classe est redéfinie par une restriction supplémentaire dans la

sous-classe. L'augmentation de contraintes est le fait qu'une contrainte de la super-classe soit conservée dans la sous-classe et enrichie à l'aide d'une ou plusieurs autres contraintes au niveau de la sous-classe. Il est bien sûr possible de conserver sans modification une contrainte de la super-classe dans sa sous-classe. Ces différents cas de règles d'héritage de contraintes correspondent à notre interprétation de l'héritage qui spécifie une sous-classe raffine les propriétés de sa super-classe et hérite donc des contraintes de sa super-classe. Mais dans [BM95], il est aussi possible de définir des règles d'exclusion de contraintes dans lesquelles une contrainte de la super-classe est remplacée par une contrainte de la sous-classe qui peut la contredire. Ce type de règles qui doit permettre d'exprimer les exceptions, va à l'encontre de notre vision de l'héritage si bien que nous ne l'avons pas traité. Nous nous restreignons donc à ce qui est appelé l'héritage fort de contraintes dans [AF94] (l'héritage fort signifie qu'une contrainte définie à un niveau n d'une hiérarchie vient compléter les contraintes définies aux niveaux supérieurs).

Dans notre approche, les règles de restriction ou d'augmentation correspondent à l'ajout de contraintes au niveau de l'intension et de l'extension de la sous-classe. Une sous-classe peut évidemment avoir des contraintes spécifiques qui sont sans rapport avec la super-classe. Nous ne nous intéressons ici qu'aux contraintes spécifiques à la relation d'héritage c'est-à-dire celles restreignant des propriétés de la super-classe. En particulier, un héritage liant plusieurs classes, il existe des contraintes portant sur les différents objets de la hiérarchie. Mais avant d'étudier l'enrichissement des contraintes au niveaux des sous-classes, il faut montrer que notre règle de traduction permet la conservation de contraintes.

Comme précédemment, nous utilisons l'exemple de la spécialisation de "PRESENTATION" (Fig 6.4).

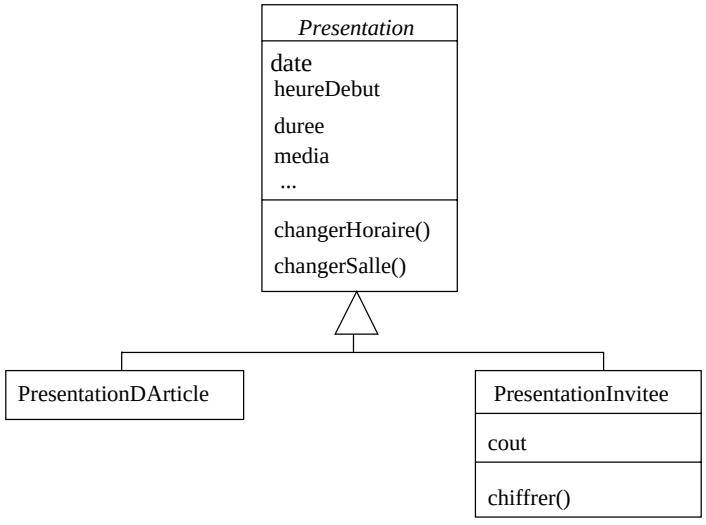


FIG. 6.4: Spécialisation de "Presentation"

Conservation de contraintes Nous avons identifié deux types de contraintes pour une classe : les contraintes sur son intension et sur son extension. Une sous-

classe peut donc hériter des contraintes sur l'intension et sur l'extension de sa super-classe.

A titre d'exemple de contrainte sur l'intension de la super-classe, on veut imposer que la durée d'une présentation ne doit pas dépasser une heure. Cette contrainte portant sur l'un de ses attributs de "PRESENTATION", s'exprime dans le schéma Z *PRESENTATIONATTR*. Ce schéma étant ensuite inclus dans le schéma d'intension de "PRESENTATIONINVITEE" et "PRESENTATIONDARTICLE", la contrainte est automatiquement héritée par les sous-classes :

PRESENTATIONATTR

$date : DATE$
$heureDebut : HEURE$
$duree : HEURE$
$media : MEDIA$

$duree.hr < 1 \lor (duree.hr = 1 \land duree.min = 0)$

PRESENTATIONINVITEEATTR

$cout : \mathbb{N}$

PRESENTATIONINVITEE

$PRESENTATIONATTR$
$PRESENTATIONINVITEEATTR$

De même, il faut garantir que les contraintes sur l'extension de la super-classe sont conservées dans les sous-classes. Comme notre règle de traduction de l'héritage assure qu'à tout instant un objet d'une sous-classe peut se substituer à un objet de sa super-classe, les contraintes sur l'extension d'une super-classe restreignent aussi les objets de ses sous-classes. Par exemple, si on veut limiter à 3 le nombre de présentations ayant lieu en même temps, il faut ajouter au schéma d'extension de "PRESENTATION" une contrainte spécifiant que les présentations qui ont la même heure de début sont inférieures ou égales à 3 : $\forall p1 : Presentation \bullet \#\{p2 : Presentation \mid p1 \neq p2 \land p1.date \neq p2.date \land p1.heureDebut \neq p2.heureDebut\} = 3$

Cette contrainte doit aussi s'appliquer aux instances de "PRESENTATIONINVITEE". Suivant notre règle de traduction, ces instances sont représentées par la variable *PresentationInvitee2* dans *PresentationExt* et par une variable *PresentationInvitee* dans *PresentationInviteeExt*. Les deux premières contraintes de *PresentationExt* qui sont générées automatiquement lors de la traduction garantissent alors l'héritage des contraintes entre les présentations et les présentations invitées. La contrainte sur le nombre de présentations est valide pour l'ensemble *PresentationInvitee2* puisqu'il est précisé par $PresentationInvitee2 \subseteq Presentation$ que les présentations invitées sont un sous-ensemble des présentations. De plus, notre règle de traduction spécifiant explicitement $(\forall x : PresentationInvitee2 \bullet \exists y : PRESENTATIONINVITEE \bullet x.date = y.date \land x.heureDebut = y.heureDebut \land x.duree = y.duree \land x.media = y.media \land x.cout = y.cout)$ que tout élément de *PresentationInvitee2* correspond à un élément de *PresentationInvitee* ayant les mêmes attributs et les même contraintes, la contrainte est donc conservée pour *PresentationInvitee*.

PresentationExt

$$
\begin{array}{l}
Presentation : \mathbb{F}\, PRESENTATION \\
PresentationInvitee2 : \mathbb{F}\, PRESENTATION \\
\ldots \\
\hline
\texttt{PresentationInvitee2} \subseteq \texttt{Presentation} \\
\forall\, \texttt{x} : \texttt{PresentationInvitee2} \bullet \exists\, \texttt{y} : \texttt{PRESENTATIONINVITEE} \bullet \\
\quad \texttt{x.date} = \texttt{y.date} \land \texttt{x.heureDebut} = \texttt{y.heureDebut} \land \\
\quad \texttt{x.duree} = \texttt{y.duree} \land \texttt{x.media} = \texttt{y.media} \land \texttt{x.cout} = \texttt{y.cout} \\
\ldots \\
\forall\, p1 : Presentation \bullet \#\{p2 : Presentation \mid \\
p1 \neq p2 \land p1.date \neq p2.date \land p1.heureDebut \neq p2.heureDebut\} = 3
\end{array}
$$

PresentationInviteeExt

$$
\begin{array}{l}
PresentationExt \\
PresentationInvitee : \mathbb{F}\, PRESENTATIONINVITEE \\
\hline
\texttt{PresentationInvitee} = \\
\quad \{\texttt{x} : \texttt{PRESENTATIONINVITEE} \mid \exists\, \texttt{y} : \texttt{PresentationInvitee2} \bullet \\
\qquad \texttt{x.date} = \texttt{y.date} \land \texttt{x.heureDebut} = \texttt{y.heureDebut} \land \\
\qquad \texttt{x.duree} = \texttt{y.duree} \land \texttt{x.media} = \texttt{y.media} \land \texttt{x.cout} = \texttt{y.cout}\}
\end{array}
$$

Contraintes restreignant l'intension d'une super-classe Une sous-classe héritant des attributs de sa super-classe, il lui est possible d'en contraindre les valeurs. C'est en fait un cas particulier de contraintes sur l'intension d'une classe. Pour les présentations, une présentation invitée hérite de l'attribut "duree" de "PRESENTATION". Si nous fixons la durée d'une présentation invitée à une heure, on exprime, en OCL, dans le contexte de "PRESENTATIONINVITEE" que la durée doit être d'une heure et zéro minute.

PRESENTATIONINVITEE
self.duree.hr=1 and self.duree.min=0 ;

De façon similaire, l'expression en Z est :

$$duree.hr = 1 \land duree.min = 0$$

Cette contrainte portant sur un attribut de "PRESENTATION" dans le cadre de "PRESENTATIONINVITEE", elle s'exprime dans le schéma *PRESENTATIONINVITEE* qui regroupe les attributs d'une présentation et d'une présentation invitée :

PRESENTATIONINVITEE

$$
\begin{array}{l}
PRESENTATIONATTR \\
PRESENTATIONINVITEEATTR \\
\hline
duree.hr = 1 \land duree.min = 0
\end{array}
$$

Le schéma *PRESENTATIONINVITEE* est en fait équivalent à la fusion des déclarations de *PRESENTATIONATTR* et *PRESENTATIONINVITEEATTR* et à la conjonction de leurs prédicats :

PRESENTATIONINVITEE

$$
\begin{array}{l}
date : DATE \\
heureDebut : HEURE \\
duree : HEURE \\
media : MEDIA \\
cout : \mathbb{N} \\
\hline
duree.hr < 1 \lor (duree.hr = 1 \land duree.min = 0) \land \\
duree.hr = 1 \land duree.min = 0
\end{array}
$$

La contrainte $duree.hr < 1 \lor (duree.hr = 1 \land duree.min = 0)$ sur "PRESENTATION" est restreinte par le fait que sa conjonction avec $duree.hr = 1 \land duree.min = 0$ se réduit à $duree.hr = 1 \land duree.min = 0$.

Contraintes restreignant l'extension d'une super-classe Comme pour les contraintes restreignant l'intension d'une super-classe, il est possible d'ajouter à une sous-classe des contraintes raffinant celles sur l'extension de la super-classe. Par exemple, nous avons spécifié que le nombre de présentations simultanées ne pouvait être supérieur à 3. Il est possible de raffiner cette contrainte en précisant qu'il ne peut y avoir deux présentations invitées en même temps : deux présentations invitées doivent avoir une date ou des plages horaires différentes. L'expression OCL de cette contrainte utilisant l'opération HeureRecouvr, nous n'avons pas pu la vérifier avec le parser.

```
PRESENTATIONINVITEE
self -> forall (p1 | p1<> self implies
      p1.date <> self.date or
      not(p1.heureDebut,p1.heureDebut+p1.duree).HeureRecouvr
            (self.heureDebut,self.heureDebut+self.duree) ;
```

L'expression en Z de cette contrainte est ajoutée au schéma *PresentationInviteeExt* représentant l'extension de "PRESENTATIONINVITEE". Elle utilise une fonction *AjouterHeures* qui calcule la somme de deux heures données.

PresentationInviteeExt

$$
\begin{array}{l}
PresentationExt \\
PresentationInvitee : \mathbb{F}\, PRESENTATIONINVITEE \\
\hline
\texttt{PresentationInvitee} = \\
\quad \{\texttt{x} : \texttt{PRESENTATIONINVITEE} \mid \exists\, \texttt{y} : \texttt{PresentationInvitee2} \bullet \\
\qquad \texttt{x.date} = \texttt{y.date} \land \texttt{x.heureDebut} = \texttt{y.heureDebut} \land \\
\qquad \texttt{x.duree} = \texttt{y.duree} \land \texttt{x.media} = \texttt{y.media} \land \texttt{x.cout} = \texttt{y.cout}\} \\
\forall\, p1, p2 : PresentationInvitee \mid p1 \neq p2 \bullet p1.date \neq p2.date \lor \\
\quad ((p1.heureDebut, AjouterHeures(p1.heureDebut, p1.duree)), \\
\quad (p2.heureDebut, AjouterHeures(p2.heureDebut, p2.duree))) \\
\quad \in HeureRecouvr
\end{array}
$$

La contrainte $PresentationInvitee = \{x : PRESENTATIONINVITEE \mid \exists\, y : PresentationInvitee2 \bullet x.date = y.date \land x.heureDebut = y.heureDebut \land$

$x.duree = y.duree \wedge x.media = y.media \wedge x.cout = y.cout\}$ exprimant que toute instance de "PRESENTATIONINVITEE" correspond à une instance "PRESENTATION", la contrainte de clé de "PRESENTATIONINVITEE" est bien ajoutée aux contraintes existantes sur "PRESENTATION".

Contraintes sur les objets de la hiérarchie Ces contraintes concernent les connexions entre objets d'une super-classe et ceux de ses sous-classes ou les connexions entre objets des sous-classes. Nous avons déjà spécifié une contrainte de ce type lors de la traduction de l'héritage en traitant le cas d'une classe abstraite (cf page 81). Le caractère abstrait d'une classe s'exprime par le fait qu'une présentation est soit une présentation invitée, soit une présentation d'article :

$$Presentation = PresentationInvitee2 \cup PresentationDArticle2$$

Un autre exemple de contraintes sur les objets de la hiérarchie est la disjonction des sous-classes qui signifie qu'une instance d'une sous-classe ne peut pas être une instance d'une autre sous-classe. Ainsi une présentation invitée ne peut pas être aussi une présentation d'article (et inversement) ; ce qui signifie que les ensembles de présentations invitées et d'article sont disjoints (c'est-à-dire que leur intersection est vide *isEmpty*). Cette contrainte nécessitant de connaître à la fois la super-classe et ses sous-classes, nous ne savons pas dans quel contexte OCL la placer. Nous ne précisons donc pas son contexte. De plus, bien qu'il existe en théorie en OCL une opération "allInstances" qui renvoie l'ensemble des instances d'un type, cette opération n'est pas disponible dans le vérificateur d'expressions OCL. L'expression de la contrainte que nous donnons ci-dessous n'a donc pas pu être vérifiée.

$$(PRESENTATIONINVITEE.allInstances \Leftrightarrow > intersection(PRESENTATIONDARTICLE.allInstances)) \Leftrightarrow > isEmpty$$

En Z, *PresentationInvitee2* et *PresentationDArticle2* représentant respectivement l'ensemble des présentations qui sont aussi des présentations invitées et celui des présentations qui sont aussi des présentations d'article, l'expression équivalente est :

$$PresentationInvitee2 \cap PresentationDArticle2 = \varnothing$$

Comme nous avons spécifié la substitution d'objet au niveau des extensions, nous disposons d'un schéma *PresentationExt* qui regroupe l'ensemble des présentations *Presentation*, celui des présentations qui sont aussi des présentations invitées *PresentationInvitee2* et celui des présentations qui sont aussi des présentations d'article *PresentationDArticle2*. Il est alors possible d'ajouter à ce schéma toutes les contraintes portant sur les connexions mutuelles entre les objets de la hiérarchie, donc en particulier la disjonction des sous-classes :

PresentationExt

$Presentation : \mathbb{F}\, PRESENTATION$
$PresentationInvitee2 : \mathbb{F}\, PRESENTATION$
$PresentationDArticle2 : \mathbb{F}\, PRESENTATION$

...
$Presentation = PresentationInvitee2 \cup PresentationDArticle2$
$PresentationInvitee2 \cap PresentationDArticle2 = \varnothing$

Contraintes sur une vue de la base de données

Certaines contraintes mettent des restrictions mutuelles sur plusieurs associations. La gestion de conférences a de nombreuses contraintes portant sur plusieurs associations. Nous avons choisi l'une d'elles qui nous paraît primordiale : un auteur ne peut pas évaluer une de ses soumissions. Cette contrainte porte sur deux relations liant "CHERCHEUR" et "SOUMISSION" (Fig. 6.5). Nous considérons donc seulement la partie du modèle de gestion de conférences qui concerne les chercheurs et les soumissions, définissant ainsi une vue de ce modèle.

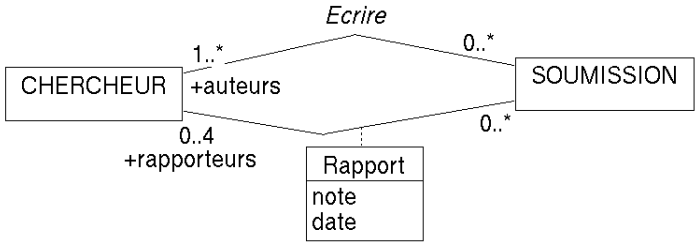


FIG. 6.5: Relations entre "CHERCHEUR" et "SOUMISSION"

Chacune de ces associations s'est traduite par des schémas Z : *RAPPORTRel* et *RapportRelExt* représentant la classe associative "RAPPORT" et *EcrireRel* décrivant l'association binaire "Ecrire". L'ensemble est ensuite groupé dans la vue *ChercheurSoumissionVue* :

RAPPORTRel

$chercheur : CHERCHEUR$
$soumission : SOUMISSION$
$note : \mathbb{N}$
$date : DATE$

...

RapportRelExt

$SoumissionExt; ChercheurExt$
$Rapport : \mathbb{F}\, RAPPORTRel$
$rapporteurs :$
$\quad SOUMISSION \pfun \mathbb{F}\, CHERCHEUR$
$soumissiondechercheur :$
$\quad CHERCHEUR \pfun \mathbb{F}\, SOUMISSION$

...

EcrireRel

SoumissionExt; *ChercheurExt*

auteurs :
$SOUMISSION \pfun \mathbb{F}\, CHERCHEUR$

soumissiondechercheur :
$CHERCHEUR \pfun \mathbb{F}\, SOUMISSION$

...

ChercheurSoumissionVue

EcrireRel; *RapportRelExt*

...

Comme dans le cas des associations, il faut choisir le contexte OCL correspondant à une contrainte globale. Pour la contrainte sur les auteurs/rapporteurs, nous choisissons de partir d'une soumission pour écrire que chacun de ses rapporteurs ne fait pas partie (*includes*) de ses auteurs.

SOUMISSION
self.rapporteurs -> forAll (r | not (self.auteurs-> includes(r))) ;

L'expression Z équivalente s'exprime au niveau du schéma de la vue *ChercheurSoumissionVue*. Elle spécifie que si un chercheur est auteur d'une soumission, il ne peut pas faire partie de ses rapporteurs :

ChercheurSoumissionVue

EcrireRel; *RapportRelExt*

$\forall\, s : \mathrm{dom}\, rapporteurs; c : Chercheur \mid c \in auteurs(s) \bullet c \notin rapporteurs(s)$

6.1.3 Bilan

Dans cette section, nous avons illustré comment les squelettes de spécifications Z nous ont permis d'identifier différents types de contraintes en fonction de leur portée sur les schémas. Le tableau 6.1 résume la classification des contraintes statiques que nous proposons et montre comment chaque type de contrainte peut être intégré aux spécifications Z.

Notre classification est proche de la distinction intra-objet, intra et inter-classes des typologies de contraintes d'intégrité statiques existantes : les contraintes sur un ou plusieurs attributs d'un objet d'une classe, d'une classe associative, d'une super-classe dans une sous-classe sont les contraintes intra-objets ; les contraintes sur les connexions mutuelles entre objets d'une même classe ou d'une même classe associative sont les contraintes d'intégrité intra-classe et toutes les autres sont les contraintes inter-classes. Notre classification des contraintes permet donc d'exprimer tous les types de contraintes statiques.

Elle est aussi suffisamment détaillée pour servir de base à un guide méthodologique pour annoter un modèle objet avec des contraintes. Elle définit le contexte le mieux approprié à l'expression de celles-ci. Elle peut être utilisée pour l'écriture de contraintes, en langue naturelle ou en OCL, sans connaissance particulière des aspects formels qui ont conduit à sa définition.

Ici, nous avons privilégié l'écriture des contraintes en OCL. Mais la sémantique et le parser supportant OCL ne sont actuellement pas suffisants pour affirmer que les contraintes que nous avons exprimé sont correctes. En leur donnant une expression

Contraintes restreignant	Schéma Z
Un attribut d'un objet d'une classe	Intension de la classe
Plusieurs attributs d'un objet d'une classe	Intension de la classe
Connexions mutuelles entre objets d'une même classe	Extension de la classe
Connexions mutuelles entre objets d'une même association	Association
Un attribut d'un objet d'une classe associative	Intension de la classe associative
Plusieurs attributs d'un objet d'une classe associative	Intension de la classe associative
Connexions mutuelles entre objets d'une même classe associative	Extension de la classe associative
L'intension d'une super-classe dans une sous-classe	Intension de la sous-classe
L'extension d'une super-classe et d'une sous-classe	Extension de la sous-classe
Connexions entre objets d'une relation d'héritage	Extension de la super-classe
Connexions mutuelles entre objets de différentes associations	Vue ou modèle global

TAB. 6.1: Classification des contraintes

équivalente en Z, d'une part, nous avons montré la forte similitude entre OCL et Z bien que certaines contraintes soient plus symétriques en Z parce qu'elles portent sur les associations ; d'autre part, nous avons fourni une expression précise et validable des contraintes.

6.2 Validation de contraintes

6.2.1 Introduction

Pour bénéficier pleinement de l'enrichissement apporté par les contraintes en détectant d'éventuelles erreurs sans attendre l'implantation dans une base de données, il faut créer une ingénierie qui les intègre réellement. Pour cela, nous utilisons les spécifications formelles Z produites à partir d'un modèle objet et de ses contraintes. Les types de contrôles proposés classiquement dans les systèmes de gestion de bases de données sont de trois types [Pha90] :

- la détection des contraintes d'intégrité préservées par une mise à jour qui vérifie qu'un ensemble de contraintes d'intégrité est satisfait après la mise-à-jour c'est-à-dire que l'ensemble de contraintes ne peut jamais être violé par la mise-à-jour ;

- le post-contrôle qui exécute la mise-à-jour, puis vérifie que l'ensemble des contraintes est satisfait ;

– le pré-contrôle qui prédit, grâce à une fonction de prédiction, si l'état final après la mise-à-jour satisfera l'ensemble des contraintes.

Suivant le principe de la détection des contraintes préservées par une mise-à-jour, nous cherchons à valider les contraintes complétant un modèle objet en essayant de savoir si une opération risque de violer l'ensemble des contraintes. Pour cela, nous utilisons le potentiel de raisonnement des spécifications formelles en validant grâce à des outils d'aide à la preuve les pré-conditions des opérations du modèle [Led98]. Nous essayons d'automatiser le plus possible ces preuves en nous basant sur leur similitude pour des contraintes semblables.

6.2.2 Validation de gardes

Les contraintes exprimées sur un modèle objet sont des propriétés invariantes qui doivent être préservées par les opérations sur la structure de données. En Z, ces contraintes sont implicitement intégrées dans la spécification des opérations manipulant un schéma de données. Contrairement à B, il n'est donc pas nécessaire de démontrer la préservation de ces invariants. Mais, l'effort de preuve est similaire car il est recommandé de calculer la précondition des opérations et de démontrer que cette précondition peut être satisfaite. Ainsi, si deux contraintes conflictuelles portent sur un attribut, la précondition d'une opération qui initialise cet attribut (ou le modifie) est *false*. Dans notre exemple, le prix d'une session ne doit pas dépasser 5000FF (cf. contrainte page 101) si bien qu'une opération qui modifie le prix d'une session a pour précondition que le nouveau prix ne dépasse pas 5000 FF. Si cette précondition est vérifiée, la postcondition est vérifiée ; sinon rien ne peut être dit sur l'état de la postcondition. En Z, seule cette notion de précondition existe alors que B utilise aussi le concept de garde qui pécise qu'en dehors de la garde, l'opération ne peut pas être activée (l'état n'est donc pas modifié). C'est dans ce sens que nous employons par la suite les termes "précondition" et "garde".

Nous proposons donc de calculer les préconditions des opérations et les utiliser lors de l'implantation comme gardes des opérations. Chaque garde est évaluée avant l'exécution de l'opération correspondante. Cependant il est parfois préférable d'employer une garde plus forte mais plus facile à calculer lors de l'exécution. Nous utilisons l'outil d'aide à la preuve Z-EVES [Saa97] pour calculer ces préconditions et valider des gardes candidates [Led98]. Z-EVES permet dans un premier temps de calculer, grâce à l'opérateur pre, la plus faible précondition d'une opération en tenant compte des contraintes exprimées en Z sur le modèle objet. Puis il faut démontrer que cette précondition est la conséquence logique d'une condition plus forte choisie comme garde, ce qui revient à prouver le théorème suivant :

$$\forall\, Etat; e? : ENTREE \mid garde(Etat, e?) \bullet \text{pre}\ Op$$

Ce théorème suppose que l'état initial *Etat* satisfait les contraintes sur son schéma et que les paramètres d'entrées *e ?* ont le bon type. Sous ces hypothèses, la garde identifiée *garde(Etat,e ?)* est une condition suffisante pour impliquer logiquement la plus faible pré-condition de l'opération *pre Op*.

Par exemple, pour valider la pré-condition de l'opération modifiant le prix d'une session *SESSIONModifierPrix*, il faut prouver que le fait que le nouveau prix ne dépasse pas 5000FF est une pré-condition de *SESSIONModifierPrix* :

theorem SESSIONModifierPrix_Pre

$$\forall\, SESSION; nvprix? : \mathbb{N} \mid nvprix? < 5000 \bullet \text{pre}\, SESSIONModifierPrix$$

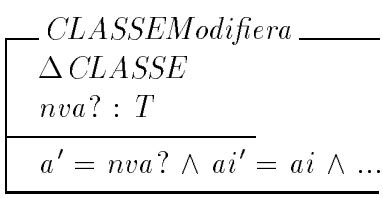
Dans ce travail, nous employons des théorèmes de cette forme afin de valider des gardes pour les opérations de base d'une classe (modification d'attribut, ajout et suppression d'objet). Nous approfondissons les propositions de [Led98] en définissant les obligations de preuves associées à ces opérations et en essayant de mettre de évidence le caractère systématique de leurs preuves suivant le type de contraintes du modèle.

6.2.3 Opérations de base et obligations de preuve

Certaines opérations associées aux classes ou aux associations sont utilisées très souvent. Elles permettent de modifier la valeur d'un attribut, de créer ou de supprimer un objet ou un lien. Leur génération automatique en fonction des cardinalités des associations a été étudiée dans [HRH96, Ngu98]. Pour éviter que les valeurs finales de certaines variables soient indéfinies après l'opération (problème du cadre), la spécification des opérations de base précise des valeurs de toutes les variables du schéma modifié. Dans ce travail, nous nous limitons aux opérations sur les classes si bien que nous ne tenons pas compte des cardinalités des associations. Ainsi un objet peut être supprimé sans que cela n'entraîne la suppression d'un objet qui lui est obligatoirement lié. Notre solution n'est pourtant pas limitative car nous considérons que les cardinalités seront prises en compte au niveau des opérations sur les associations.

Opération de modification d'un attribut

Pour toute classe "CLASSE", il est possible de modifier chacun de ses attributs "a" en lui donnant une nouvelle valeur "nva". La valeur des autres attributs "ai" reste alors identique. L'opération *CLASSEModifiera* s'exprime en Z en spécifiant que l'intension de "CLASSE" est modifiée en donnant pour valeur à *a* la valeur *nva ?* donnée en entrée. Le théorème à prouver *CLASSEModifiera_Pre* pour valider la garde de *CLASSEModifiera* suppose que l'état initial de *CLASSE* satisfait les contraintes et que la nouvelle valeur de l'attribut "nva" est bien de type T. Sous ces hypothèses, la garde identifiée qui dépend de l'état de *CLASSE* et de *nva ?* doit impliquer la plus faible pré-condition de *CLASSEModifiera*.



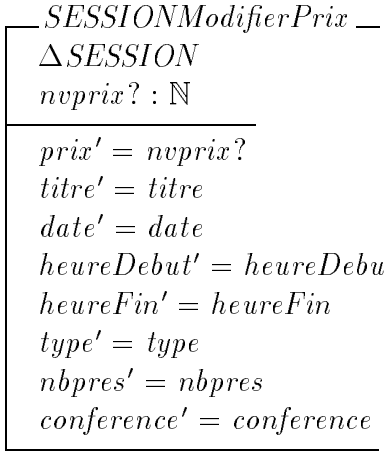
(a) Opération de modification d'un attribut

theorem CLASSEModifiera_Pre

$$\forall\, CLASSE; nva? : T \mid garde(CLASSE, nva?) \bullet \text{pre } CLASSEModifiera$$

(b) Théorème pour la validation de la garde de *CLASSEModifiera*

Ainsi l'opération de modification du prix d'une session *SESSIONModifierPrix* prend en entrée le nouveau prix d'une session *nvprix ?*. La validation de sa garde (garde(SESSION,nvprix ?)) s'effectue en prouvant le théorème *SESSIONModifierPrix_Pre*.



(c) Opération de modification de "prix"

theorem SESSIONModifierPrix_Pre

$$\forall\, SESSION; nvprix? : \mathbb{N} \mid garde(SESSION, nvprix?) \bullet \text{pre } SESSIONModifierPrix$$

(d) Théorème pour la validation de la garde de *SESSIONModifierPrix*

Comme toute opération sur l'intension d'une classe, l'opération *CLASSEModifiera* peut être promue en la couplant avec l'opération générique de promotion *ChangeClasse*. *ClasseModifiera* a donc deux paramètres d'entrées *nva ?* et *x ?* qui désigne l'objet dont l'attribut va changer. Le théorème à prouver pour valider une garde candidate a pour hypothèse que *CLASSE* et *ClasseExt* satisfont les contraintes avant l'opération et les entrées *x ?* et *nva ?* ont le bon type. La première ligne de la garde est purement technique : *x ?* doit être un objet de "CLASSE" ($x? \in Classe$) et c'est celui qui est modifié par l'opération promue ($\theta\, CLASSE = x?$).

ChangeClasse

$\Delta ClasseExt$
$\Delta CLASSE$
$x? : CLASSE$

$x? \in Classe$
$\theta CLASSE = x?$
$Classe' = (Classe \setminus \{x?\})$
$\cup \{\theta CLASSE'\}$

$ClasseModifiera == ChangeClasse \land CLASSEModifiera$

(e) Opération de modification d'un attribut

theorem ClasseModifiera_Pre

$\forall CLASSE; ClasseExt;$
$nva? : T; x? : CLASSE \mid$
$x? \in Classe \land \theta CLASSE = x?$
$\land\ garde \Leftrightarrow$
$(CLASSE, Classe, nva?, x?)$
• pre $ClasseModifiera$

(f) Théorème pour la validation de la garde de *ClasseModifiera*

Par exemple, l'opération qui modifie le prix d'une session est promue au niveau de l'extension de "SESSION" en faisant la conjonction de *SESSIONModifierPrix* et de *ChangeSession*.

ChangeSession

$\Delta SessionExt$
$\Delta SESSION$
$x? : SESSION$

$x? \in Session$
$\theta SESSION = x?$
$Session' = Session \setminus \{x?\}$
$\cup \{\theta SESSION'\}$

$SessionModifierPrix == ChangeSession \land SESSIONModifierPrix$

(g) Promotion de *SESSIONModifierPrix*

theorem SessionModifierPrix_Pre

$\forall SESSION; SessionExt$
$; nvprix? : \mathbb{N}; x? : SESSION \mid$
$x? \in Session \land$
$\theta SESSION = x? \land$
$garde(SESSION,$
$Session, nvprix?, x?))$
• pre $SessionModifierPrix$

(h) Théorème pour la validation de la garde de *SessionModifierPrix*

Opération d'ajout d'un objet

L'ajout d'une instance à une classe "CLASSE" consiste à modifier l'ensemble des objets existants de la classe *Classe* pour y inclure l'objet donné en entrée *o ?*. Pour valider une garde candidate, il faut tout d'abord qu'à l'état initial, *ClasseExt*

satisfasse les contraintes du modèle et que l'objet *o ?* à ajouter soit bien du type *CLASSE*. La garde candidate dépend alors des contraintes sur l'ensemble des objets *Classe* et de l'objet *o ?*.

AjouterClasse

$\Delta ClasseExt$
$o? : CLASSE$

$Classe' = Classe \cup \{o?\}$

(i) Opération d'ajout d'un objet

theorem AjouterClasse_Pre

$$\forall\, ClasseExt; o? : CLASSE \mid garde(Classe, o?) \bullet \text{pre } AjouterClasse$$

(j) Théorème pour la validation de la garde de *AjouterClasse*

Par exemple, l'opération ajoutant une session *AjouterSession* a pour paramètre d'entrée la session *session ?* à ajouter et modifie l'extension de *Session* en y ajoutant *session ?*. Pour valider une garde $garde(Session, session?)$ de cette opération, *SessionExt* doit satisfaire les éventuelles contraintes et l'entrée doit être une session.

AjouterSession

$\Delta SessionExt$
$session? : SESSION$

$Session' = Session \cup \{session?\}$

(k) Opération *AjouterSession*

theorem AjouterSession_Pre

$$\forall\, SessionExt; session? : SESSION \mid garde(Session, session?) \bullet \text{pre } AjouterSession$$

(l) Théorème pour la validation de la garde de *AjouterSession*

Opération de suppression d'un objet

La suppression d'un objet consiste à enlever un objet donné en entrée *o ?* de l'extension de sa classe *Classe*. Le théorème à prouver pour valider une garde candidate est similaire à celui de l'opération d'ajout d'un objet.

SupprimerClasse

$\Delta ClasseExt$
$o? : CLASSE$

$Classe' = Classe \setminus \{o?\}$

(m) Opération de suppression d'un objet

theorem SupprimerClasse_Pre

$$\forall\, ClasseExt; o? : CLASSE \mid garde(Classe, o?) \bullet \text{pre } SupprimerClasse$$

(n) Théorème pour la validation de la garde de *SupprimerClasse*

Suivant ce principe, l'opération de suppression d'une session est décrite par *SupprimerSession* et la validation d'une garde s'effectue en prouvant le théorème *SupprimerSession_Pre*.

SupprimerSession

$\Delta SessionExt$

$session? : SESSION$

$Session' = Session \setminus \{session?\}$

(o) Opération *SupprimerSession*

theorem SupprimerSession_Pre

$\forall SessionExt;$
$session? : SESSION \mid$
$garde(Session, session?)$
$\bullet$ pre $SupprimerSession$

(p) Théorème pour la validation de la garde de *SupprimerSession*

6.2.4 Preuves des opérations de base

[Led98] remarque que certaines preuves pour la validation des gardes sont très similaires en Z-EVES. Partant de ce constat, nous avons essayé d'identifier d'où provient cette similarité et d'en tirer profit pour proposer des preuves plus automatiques.

En fait, une preuve en Z-EVES est déjà en partie automatisée car Z-EVES offre un niveau intermédiaire d'automatisation entre le cas où l'utilisateur doit donner le raisonnement de sa preuve pas-à-pas et celui où il attend un résultat (vrai ou faux) immédiat. Les commandes de Z-EVES regroupent en général plusieurs actions de preuves élémentaires. Par exemple, l'instruction *prove by reduce* appelle un ensemble de règles de réécriture qui sont utilisées suivant les besoins de la preuve traitée. Ainsi les preuves sont semi-automatiques : le prouveur utilise au maximum des stratégies de preuves prédéfinies et demande une instruction à l'utilisateur quand il ne sait plus quelle stratégie appliquer. Cette particularité de Z-EVES lui fournit une certaine résistance aux évolutions des spécifications c'est-à-dire que la même preuve peut être valide pour plusieurs théorèmes de même forme. C'est cette propriété que nous utilisons pour systématiser les preuves des opérations de base suivant les types de contraintes annotant un modèle objet.

Le but de ce travail est donc d'automatiser le plus possible les preuves permettant de valider un modèle objet et ses contraintes en essayant de déterminer les preuves correspondant à la validation de gardes candidates pour les opérations de base. Pour chaque opération, nous tentons de prouver les théorèmes introduits précédemment en partant du principe que leur garde candidate correspond aux contraintes appliquées aux entrées de l'opération (*contrainte(e ?)*) ou à une garde plus forte (*c(e ?)*). Les théorèmes sont donc de la forme :

$\forall Etat; e? : ENTREE \mid garde(Etat, e?) \bullet$ pre Op
avec soit garde(Etat,e ?)=contrainte(e ?)
soit garde(Etat,e ?)=c(e ?) tel que c(e ?) $\Rightarrow$ *contrainte*$(e?)$

Ainsi pour l'opération de modification de prix d'une session, une garde candidate

correspond au fait que le nouveau prix vérifie la contrainte $prix < 5000$ du schéma *SESSION*. Une autre est que le nouveau prix est inférieur à l'ancien (qui remplissait déjà cette contrainte).

La spécification des opérations et le type de la garde étant fixés, le seul paramètre qui peut faire varier le déroulement de la preuve est le type des contraintes sur le modèle de données. Pour comprendre l'influence d'une forme de contrainte sur sa vérification, les contraintes doivent être introduites indépendamment puis incrémentalement. Pour l'instant, elles sont prises en compte une par une en laissant de côté les autres afin de mieux cerner les preuves correspondant à chacune d'entre elles. Nous avons ainsi étudié les preuves pour les opérations de base dans le cas où le modèle objet comporte :

- aucune contrainte ;
- une contrainte sur l'un des attributs d'une classe ;
- une contrainte sur plusieurs attributs d'une classe ;
- une contrainte de clé sur une classe.

Les preuves correspondant à ces différents cas ont été testées sur plusieurs exemples afin d'identifier leurs analogies. Elles ne sont pas forcément les preuves optimales pour les théorèmes de validation de gardes, elles ont néanmoins le mérite d'être systématiques. Chacune d'elle constitue ce que nous appelons un "schéma de preuve" pour une opération de base et un type de contrainte. Par exemple, nous avons identifié qu'une opération de modification d'un attribut a pour garde candidate que la nouvelle valeur de l'attribut modifié vérifie la contrainte et que sa preuve est automatique en Z-EVES. Ainsi pour l'opération modifiant le prix, la garde identifiée exprime que le nouveau prix vérifie la contrainte limitant sa valeur a 5000FF. La preuve du théorème *SESSIONModifierPrix_Pre* validant cette garde au niveau de l'intension de "SESSION" est automatique.

theorem SESSIONModifierPrix_Pre

$$\forall SESSION; nvprix? : \mathbb{N} \mid nvprix? < 5000$$
$$\bullet \text{ pre } SESSIONModifierPrix$$

Nous ne donnons pas ici les instructions qui constituent les "schémas de preuves" que nous avons définis car leur énoncé n'a d'intérêt que pour les utilisateurs du prouveur Z-EVES. Mais nous les détaillons en annexe C pour les lecteurs intéressés.

6.2.5 Bilan

Cette section présente comment valider les contraintes d'un modèle objet en vérifiant les gardes des opérations. Pour automatiser le plus possible le travail de vérification, nous avons étudié les preuves d'opérations de base pour lesquels nous avons tenté d'identifier des similitudes pour des contraintes de même type. Ainsi nous avons mieux compris la construction de preuves en constatant la ressemblance des cas pour les preuves des opérations d'un modèle ayant la même contrainte. Nous

avons aussi identifié des "schémas" de preuves correspondant à une opération et à un type de contraintes. Même si l'étude doit être approfondie pour prendre en compte plus de contraintes, elle montre qu'il est possible, grâce aux connaissances du domaine d'aider l'utilisateur dans la difficile tâche de vérification.

6.3 Raisonnements informels

Le passage d'un modèle objet à une spécification formelle propose une interprétation précise de ce modèle. Dans cette section, nous montrons comment l'utilisation de cette traduction permet de lever les ambiguités sémantiques de trois exemples. La sémantique donnée à ces exemples n'est pas forcément celle envisagée par les concepteurs des notations semi-formelles. Cependant elle a le mérite de la précision et il est possible d'adapter les propositions du chapitre 5 si elles s'avèrent incompatibles avec la sémantique attendue par la communauté. Enfin comme il nous a été difficile de trouver des exemples pour les constructions que nous voulions illustrer dans le cadre de la gestion de conférences, nous avons dû les choisir dans d'autres domaines.

6.3.1 Des cardinalités invalides

Ce premier exemple (Fig. 6.6) permet de montrer sur un exemple simple comment la traduction en Z permet de raisonner sur une modélisation. Dans cet exemple, les classes sont liées par des associations de cardinalités 1 sauf l'un des rôles qui a une cardinalité de 2. Ce cas est bien sûr caricatural mais il illustre comment notre technique aide à détecter une erreur de traduction grossière. Ce type d'erreur peut néanmoins se produire lorsque les éléments sont présentés dans des diagrammes différents.

Cet exemple (Fig. 6.6) peut modéliser une société matriarcale monogamme. a un instant donné, un homme ou une femme ne peuvent avoir qu'un seul conjoint. De plus, on impose qu'un homme doit avoir un seul enfant alors que toute femme doit avoir deux enfants au cours de deux mariages successifs.

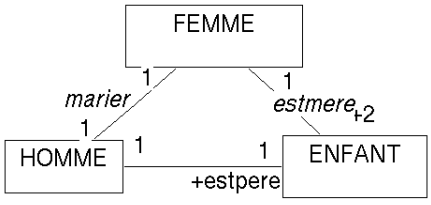


FIG. 6.6: Cardinalités invalides

La traduction en Z de ce diagramme permet d'effectuer le raisonnement suivant : d'après les cardinalités entre "HOMME" et "FEMME", tout homme est marié à une seule femme et toute femme est l'épouse d'un seul homme. Cela implique que le nombre de femmes existantes est égal au nombre d'hommes existants :

(1) $\#Femmes = \#Hommes$

où Femmes est la variable représentant l'ensemble des femmes existantes,
Hommes est la variable représentant l'ensemble des hommes existants
et # est la cardinalité d'un ensemble (c'est-à-dire son nombre d'éléments)

De même, les cardinalités entre "HOMME" et "ENFANT" permettent de déduire l'égalité (2) suivante dans laquelle *Enfants* est l'extension des enfants :

(2) $\#Hommes = \#Enfants$

Enfin l'association entre "FEMME" et "ENFANT" exprime qu'une femme a 2 enfants. Il y a donc 2 fois plus d'enfants que de femmes :

(3) $2 * \#Femmes = \#Enfants$

Or d'après (1) et (2), on a :

(4) $\#Femmes = \#Enfants$

(3) et (4) sont vraies seulement si $\#Femmes = 0$ c'est-à-dire s'il n'y a pas de femme. D'après (2) et (3), on déduit qu'il ne peut y avoir ni enfant, ni homme. Le diagramme de classes de la figure 6.6 est correct mais les classes ne peuvent avoir pas d'objets sans violer les contraintes de cardinalité. Ce diagramme spécifie donc une base de données qui ne peut être que vide ! Il existe une erreur dans les cardinalités de ce diagramme : la cardinalité entre "HOMME" et "ENFANT" doit être identique à celle entre "FEMME" et "ENFANT" ou un enfant peut ne pas être lié à un homme.

La traduction nous a aidé ici à formaliser le raisonnement sur les cardinalités pour montrer leur caractère trop contraignant. Le problème aurait aussi pu être détecté en exploitant l'outil d'aide à la preuve Z-EVES. En tentant de valider la pré-condition de l'opération d'ajout d'un homme à la base, nous aurions trouvé que cette pré-condition est *false*. Nous aurions ainsi conclu que l'opération n'est jamais réalisable et qu'il fallait reconcevoir le modèle.

6.3.2 Une sous-classe composante

La sémantique de l'héritage, de l'agrégation et de la composition est décrite dans différents documents, mais ils n'expliquent pas comment combiner ces différents concepts. En particulier, il semble autorisé d'avoir entre deux classes une relation d'héritage et une relation d'agrégation (Fig. 6.7). Par exemple, une pièce peut être spécialisée en pièce simple, mais elle peut aussi être composée de pièces simples. Mais une pièce simple peut-elle être composée d'autres pièces ?

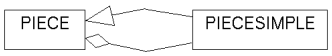


FIG. 6.7: Une sous-classe composante

D'après notre règle de traduction de l'héritage en Z, une pièce simple est une pièce.

$PIECE == PIECESIMPLE$

L'agrégation entre "PIECE" et "PIECESIMPLE" est donc une relation réflexive. Il faut préciser dans les prédicats du schéma de l'agrégation qu'une pièce ne peut pas être un agrégat d'elle-même et que cette agrégation est transitive :

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PieceAggreg}} \\
\hline
\textit{PieceExt}; \textit{PiecesimpleExt} \\
\textit{PieceDePiecesimple} : \textit{PIECESIMPLE} \pfun \textit{PIECE} \\
\textit{PiecesimpleDePiece} : \textit{PIECE} \pfun \mathbb{F}\,\textit{PIECESIMPLE} \\
\hline
\forall x1, x2 : \textit{PIECE} \mid x2 = \textit{PieceDePiecesimple}(x1) \bullet x1 \neq x2 \\
\hfill \text{[irréflexivité]} \\
\forall x1, x2, x3 : \textit{PIECE} \mid x2 = \textit{PieceDePiecesimple}(x1) \land \\
\quad x3 = \textit{PieceDePiecesimple}(x2) \bullet x3 = \textit{PieceDePiecesimple}(x1) \\
\hfill \text{[transitivité]} \\
\ldots \\
\hline
\end{array}
$$

Une pièce peut être une pièce simple et peut être composée de pièces simples mais pas d'elle-même. La réponse à notre question est donc qu'une pièce simple peut être composée d'autres pièces simples. Si cette sémantique n'est pas celle désirée, on peut introduire le concept de pièce composée. Une pièce se spécialise en pièce simple et en pièce composée et une pièce composée est composée de pièces, simples ou composées (Fig. 6.8). La détection de ce genre d'erreur conduit à affiner le diagramme pour marquer les caractéristiques structurelles de manière plus explicite.

FIG. 6.8: Une sous-classe composante - Expression alternative

6.3.3 Héritage d'association

Le diagramme de la figure 6.9 illustre le cas d'un héritage de relations de composition. Dans ce diagramme, un train est composé d'une locomotive et de voitures non motorisées. De plus, un train peut être de différents types : TGV, TER etc. Pour cette structure, on peut se demander si les éléments d'une composition sont hérités et si oui, comment traiter les restrictions qui régissent les compositions et les associations, en particulier ces restrictions changent-elles d'une super-classe à sa sous-classe.

FIG. 6.9: Diagramme de classes comportant un héritage d'association - Exemple

Question 1 : les éléments d'une composition sont-ils hérités ?
Notre traduction en Z permet de répondre oui à cette première question. En effet,

la traduction de l'héritage exprime qu'un TER et un TGV sont des trains :

$$TRAIN == TER \lor TGV$$

Si un train est composé d'une locomotive et de voitures, un TER ou un TGV ont, eux aussi en tant que trains, une locomotive et des voitures. Dans les relations de compositions entre "TRAIN" et "LOCOMOTIVE" ou "TRAIN" et "VOITURE", un train peut toujours être remplacé par un TGV ou un TER. Par exemple, la composition entre "TRAIN" et "LOCOMOTIVE" s'applique aussi entre un TER et une locomotive :

TrainLocoComp

$$TrainExt; LocomotiveExt$$
$$TrainDeLoco : LOCOMOTIVE \pfun TRAIN$$
$$LocoDeTrain : TRAIN \pfun LOCOMOTIVE$$

$$\mathrm{dom}\ TrainDeLoco = Locomotives \land \mathrm{ran}\ TrainDeLoco = Trains$$
$$LocoDeTrain = \{x : \mathrm{ran}\ TrainDeLoco; y : \mathrm{dom}\ TrainDeLoco \mid (y, x) \in TrainDeLoco \bullet x \mapsto y\}$$

Question 2 : comment traiter les restrictions des compositions et les associations ? Un objet d'une sous-classe étant un objet de sa super-classe, une sous-classe hérite de tout le contexte de sa super-classe dont ses associations. Il n'est donc pas nécessaire de re-spécifier les compositions ou les associations dans les sous-classes. Par exemple, un TER étant un train, il est composé d'une locomotive et de voitures. L'association entre une locomotive et des voitures est identique pour un train ou pour un TER.

De plus, les conditions portant sur une super-classe ne peuvent pas être affaiblies dans ses sous-classes, mais elles peuvent être renforcées. Par exemple, un TGV peut être composé de huit voitures ($8 \in 0..*$) mais pas de deux locomotives ($2 \neq 1$). On ne peut lui affecter deux locomotives qu'en modifiant le diagramme initial.

6.3.4 Bilan

Ces trois exemples montrent comment la formalisation d'un modèle objet permet de raisonner de façon informelle sur le sens. Ceci n'est réalisable que sur des cas simples comme ceux que nous avons présentés. Pour des exemples plus gros, le raisonnement informel peut difficilement être exploité pour trouver des erreurs. Néanmoins ces exemples ont une sémantique précise qui peut être utilisée par des outils conçus pour Z comme nous l'avons montré pour la validation des pré-conditions des opérations.

6.4 Conclusion

Ce chapitre illustre trois bénéfices que nous avons obtenus en traduisant les notations semi-formelles en Z. Dans un premier temps, la traduction a permis d'identifier une classification des contraintes annotant un modèle objet, définissant ainsi un guide méthodologique pour l'écriture de ces contraintes.

Nous avons ensuite essayé de systématiser la validation des contraintes par l'identification et la preuve de gardes des opérations de base. Nous avons tenté de mieux cerner les méchanismes de preuve en cherchant pour chaque forme de contrainte une preuve de base qui peut se complexifier suivant la complexité de la contrainte.

Enfin trois cas de modèles objet nous ont servi à présenter comment une meilleure compréhension de la sémantique d'un modèle semi-formel permettait d'éviter des erreurs ou des maladresses.

Chapitre 7

RoZ : un atelier pour la traduction d'UML vers Z

7.1 Présentation de RoZ

7.1.1 Support à l'approche du projet Champollion

Dans le chapitre 3, nous avons présenté l'approche du projet Champollion sous-jacente à notre travail. Elle est basée sur la traduction de spécifications semi-formelles et de contraintes en spécifications formelles. Pour faciliter ce processus, il est nécessaire de disposer d'outils supportant les différentes étapes :

1. La première étape consiste à développer une spécification semi-formelle et ses annotations à partir d'un problème initial. Afin de rester proche des habitudes de travail des concepteurs, nous proposons d'utiliser l'environnement Rational Rose [Rat96b] pour éditer les modèles semi-formels. Rose étant un éditeur de modèles en UML, nos propositions de traduction sont appliquées au diagramme de classes d'UML.

2. La spécification semi-formelle est traduite automatiquement en squelettes de spécification formelle Z grâce à notre outil, RoZ. Dans le même temps, les annotations qui expriment des contraintes sont traduites par l'utilisateur en langage formel et intégrées dans l'environnement Rose.

3. Les squelettes de spécifications formelles sont complétés par les contraintes qui ont été écrites dans Rose par l'utilisateur. Grâce à la classification des annotations, RoZ intègre chacune des contraintes au schéma Z approprié.

4. La spécification formelle est utilisée pour vérifier la cohérence de la spécification semi-formelle et de ses annotations. Comme nous l'avons vu dans le chapitre précédent, le raisonnement sur les modèles peut s'effectuer de manière informelle ou en utilisant des outils tels que le prouveur Z-EVES. Afin de préparer à l'utilisation de Z-EVES, RoZ produit des obligations de preuves pour la validation des gardes des opérations.

5. Grâce au processus de traduction, les raisonnements effectués sur la spécification formelle peuvent aider l'utilisateur à revoir sa spécification semi-formelle ou ses contraintes.

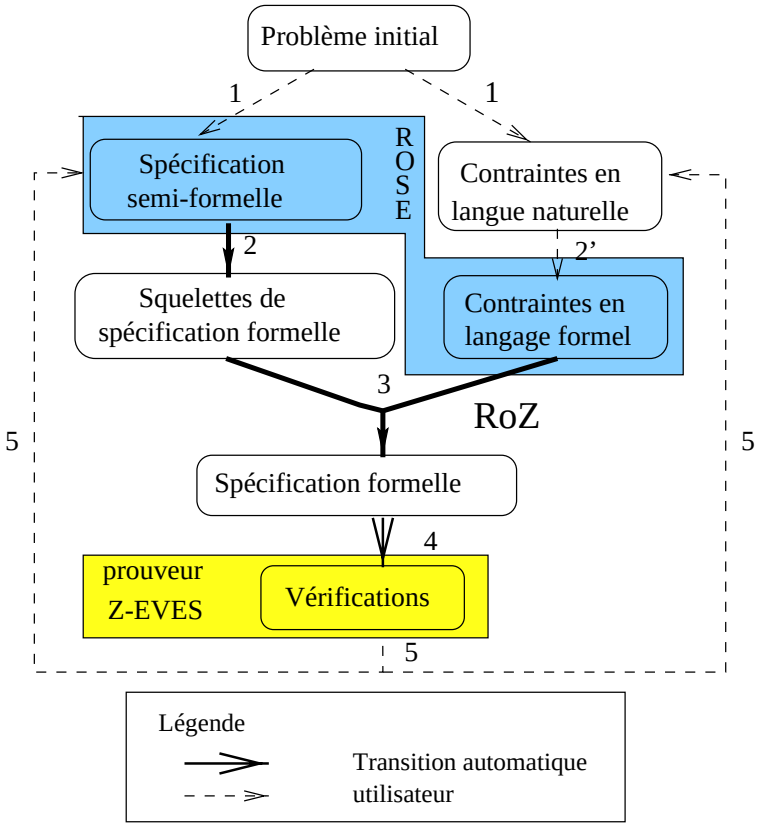


FIG. 7.1: Support outil à l'approche du projet Champollion

Nous nous intéressons ici à RoZ [DLCP00a], un outil supportant principalement les étapes 2 et 3 du processus. Mais il facilite aussi partiellement le travail de vérification des modèles en générant des obligations de preuves (étape 4).

7.1.2 Principe de RoZ

RoZ est une extension de l'environnement Rational Rose. Nous avons choisi cet environnement car Rose est actuellement l'un des outils les plus utilisés comme support à la notation UML [BJR98]. C'est aussi l'un des plus complets. Enfin, même s'il n'est pas complètement ouvert, il peut être étendu facilement. Pour l'étendre, trois composants sont disponibles [Rat96a] :

- Rose Interface d'Extensibilité
  C'est un ensemble d'interfaces utilisées par Rose Script et Rose Automatisation pour accéder à Rose.

- Rose Script
  Il sert à automatiser des fonctions manuelles de Rose, à créer des versions de

Rose spécifiques et à intégrer Rose avec d'autres applications. Pour cela, on utilise un langage de script basé sur Visual Basic. Par l'intermédiaire de fonctions et de procédures pré-définies, les scripts permettent d'accéder à l'application, aux diagrammes et à tous les éléments d'un modèle sous Rose.

- Rose Automatisation
  Il sert à connecter Rose avec d'autres applications en utilisant le serveur d'automatisation OLE.

Nous utilisons les scripts de Rose pour accéder aux éléments d'un modèle UML et réaliser l'intégration avec les spécifications formelles en Z.

L'environnement Rose est utilisé pour développer les modèles UML qui sont complétés par des annotations formelles en Z. Le point principal est que tout le travail de modélisation (modèles et annotations) est réalisé dans Rose pour être le plus proche possible de l'environnement de travail habituel des concepteurs : l'éditeur de diagrammes de Rose servant à développer les modèles et les annotations en Z sont écrites dans des formulaires. Un formulaire doit contenir toutes les informations nécessaires pour compléter les squelettes de spécifications Z produits par traduction. Ainsi chaque formulaire correspond à un type d'annotations. Il est attaché à l'élément du diagramme de classes qu'il complète. En Rose, un formulaire pré-défini est associé à chaque élément de l'application. Nous aurions aimé pouvoir modifier ces formulaires afin d'y ajouter les champs dont nous avions besoin pour le passage à des spécifications formelles. Mais cela n'étant pas possible en Rose 4.0, nous nous contentons d'utiliser les formulaires standards de Rose.

Ainsi à partir d'un diagramme de classes et de ses annotations, RoZ utilise les scripts pour :

- **générer des spécifications Z**
  A partir d'un diagrammes de classes UML et de ses annotations, une spécification Z complète est générée dans un fichier. Nous avons choisi d'exprimer la spécification Z au format Latex afin de pouvoir exploiter les outils de vérification de Z tels que Z-EVES.

- **générer les opérations de base**
  Dans le chapitre précédent (chap. 6), nous avons introduit la spécification des opérations de base sur les classes (modification d'un attribut, ajout et suppression d'un objet). RoZ peut compléter un diagramme de classes en générant automatiquement ces spécifications.

- **générer les théorèmes de validation des gardes des opérations**
  Afin de faciliter le travail de validation d'un modèle et de ses contraintes, RoZ génère, suivant le principe introduit en 6.2.2, un théorème à prouver pour valider chacune des gardes des opérations.

7.2 Visite guidée

Dans cette section, les fonctionnalités de RoZ sont illustrées sur un sous-ensemble du diagramme de classes de la gestion de conférences. Nous montrons comment

décrire un diagramme et ses annotations dans RoZ. Puis nous utilisons chacune des fonctionnalités de RoZ.

7.2.1 Construction d'un diagramme de classes dans RoZ

La partie du modèle de la gestion de conférences que nous utilisons pour illustrer l'emploi de RoZ est l'association "AvoirLieu" entre "SESSION" et "SALLE". Ces classes et leur association sont développées dans la fenêtre "Class Diagram" de Rational Rose (Fig. 7.2). On peut néanmoins noter que pour le passage à Z, le diagramme doit être complet : le type des attributs, le nom des rôles des associations et leur cardinalité doivent être définis. De plus, les types des attributs sont exprimés en Z. Par exemple, le prix d'une session est un naturel noté *nat*. Ces types (*TITRE, DATE* etc) sont spécifiés dans un fichier Z joint à la spécification.

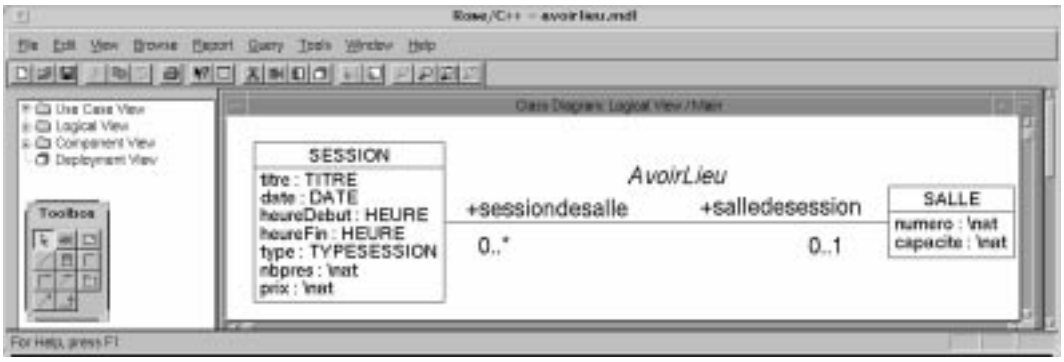


FIG. 7.2: Interface de Rose - Association entre "SESSION" et "SALLE"

Dans la section 6.1 concernant le guide méthodologique des contraintes, nous avons exprimé les quatre contraintes suivantes en complément à ce modèle :

1. Le prix d'une session en doit pas dépasser 5000 FF.
2. L'heure de début d'une session est antérieure à son heure de fin.
3. La conférence et le titre sont une clé de "SESSION".
4. Deux sessions ne peuvent pas avoir lieu en même temps dans la même salle.

Chacune de ces contraintes correspond à un type de contraintes de notre classification. Dans RoZ, chacun de ces types correspond à un type de formulaire. Ainsi chaque contrainte est exprimée (en Latex) à l'intérieur du champ "Documentation" du formulaire de l'élément sur lequel elle porte. La première contrainte restreint la valeur de l'attribut "prix" d'une session. Elle est donc écrite dans le champ "Documentation" du formulaire de "prix" (Fig. 7.3). La seconde contrainte porte sur deux attributs "heureDebut" et "heureFin". Il est donc possible de l'écrire soit dans le formulaire de "heureDebut" et soit dans celui de "heureFin". Ici nous avons choisi le formulaire de "heureDebut" (Fig. 7.3).

La troisième contrainte (la clé de "SESSION") est une comparaison entre des objets existants de la classe "SESSION", aussi nous l'exprimons dans le formulaire

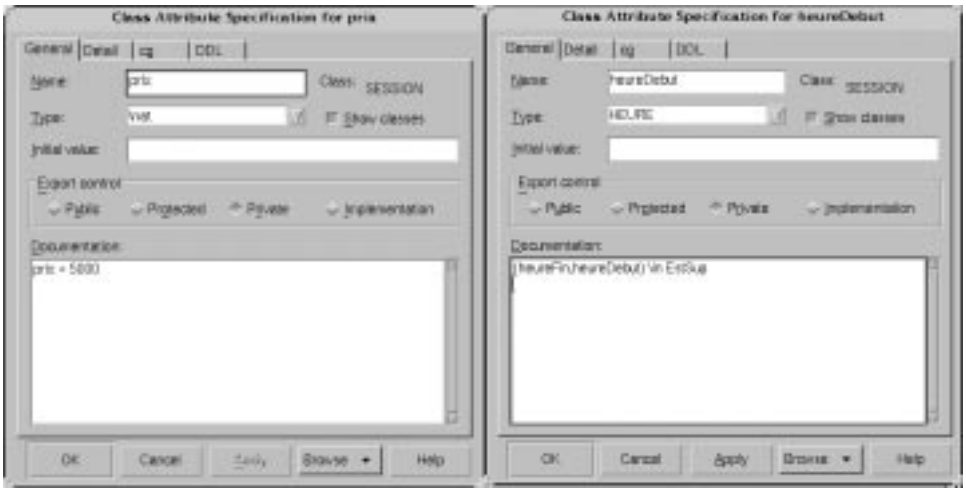


FIG. 7.3: Formulaires de "prix" et de "heureDebut"

de "SESSION" (Fig. 7.4). Enfin la dernière contrainte (deux sessions ne peuvent pas avoir lieu en même temps dans la même salle.) est relative à l'association entre "SESSION" et "SALLE". Elle s'exprime donc dans le formulaire correspondant à l'association "AvoirLieu".

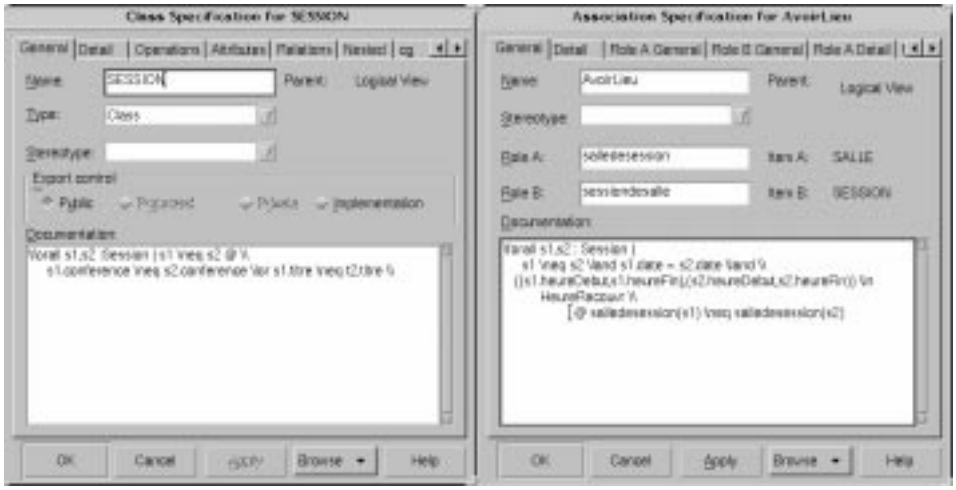


FIG. 7.4: Formulaires de "SESSION" et de "AvoirLieu"

Suivant le même principe, les formulaires sont exploités pour écrire les spécifications des opérations. Cette utilisation est illustrée dans les sections suivantes pour l'opération "ChangePrix" qui est générée automatiquement.

7.2.2 Génération des opérations de base

Nous avons vu précédemment (section 6.2.3) qu'il existe des opérations de base qui se retrouvent très souvent dans les modèles et dont la spécification peut être écrite de manière systématique. RoZ propose de générer la spécification de ces opérations sur les classes : pour chaque attribut d'une classe donnée, une opération modifiant cet

attribut est générée ; si la classe est concrète, les spécifications d'opérations d'ajout et de suppression d'un objet sont ajoutées. Ces opérations sont des "versions" anglaises des opérations de base que nous avons utilisées dans le chapitre précédent pour la validation des gardes.

Par exemple, pour la classe "SESSION", les opérations "ChangeTitre", "ChangeDate", "ChangeHeuredebut", "ChangeHeurefin", "ChangeType", ChangeNbPres", "ChangePrix", "AddSession" et "RemoveSession" sont générées automatiquement par RoZ (Fig. 7.5). La spécification de ces opérations est contenue dans leur formulaire. Prenons à titre d'exemple l'opération "ChangePrix" qui modifie le prix d'une session. Dans la partie "Operations" de "SESSION", on peut voir la signature des opérations : "ChangePrix" a pour argument "newprix" qui est le paramètre d'entrée de l'opération. Dans le champ "PostConditions" du formulaire de "ChangePrix" (Fig. 7.5), la post-condition de l'opération est écrite en Z au format Latex. Elle signifie que la nouvelle valeur de l'attribut "prix" est l'argument de l'opération *newprix ?* et que les autres attributs ne sont pas modifiés. RoZ remplit aussi le champ "Semantics" avec le mot-clé *intension operation* pour préciser que "ChangePrix" est une opération sur les attributs de la classe.

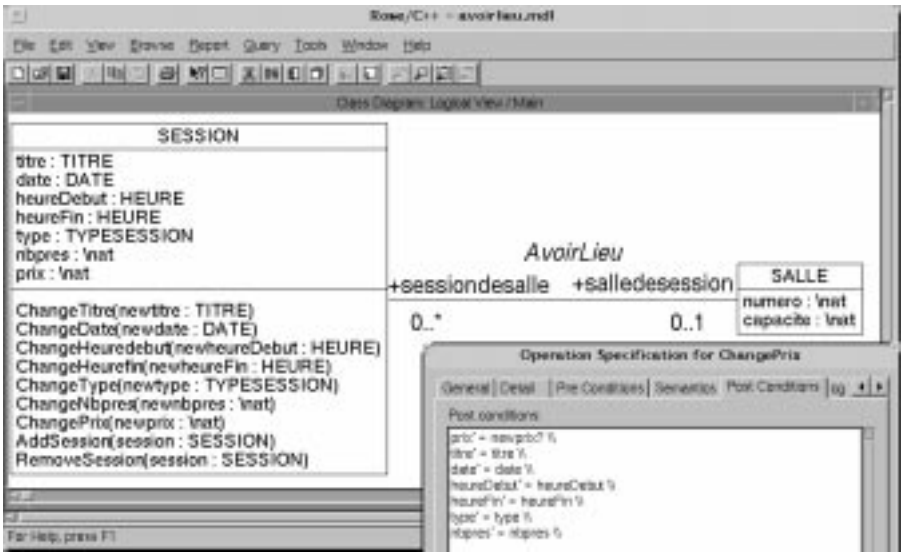


FIG. 7.5: Postcondition de "ChangePrix"

7.2.3 Génération d'une spécification Z

La génération d'une spécification formelle à partir d'un diagramme de classes est basée sur les règles de traduction du modèle objet vers Z que nous présentons au chapitre 5. L'outil peut ainsi traduire les concepts de classe, d'attribut, d'opération, d'association binaire, d'héritage. Il ne traduit pas les spécificités des agrégations et des composition et il ne supporte pas la traduction des associations n-aires (car elles ne sont pas représentées dans Rose) et des classes associatives.

Mais la spécification Z produite par traduction n'est qu'un squelette et doit être complétée. Pour cela, il faut ajouter des informations telles que la définition des types

d'attributs, les contraintes et le corps des opérations. Dans les sections précédentes, nous avons vu comment les contraintes et la spécification des opérations peuvent être incluses dans les formulaires de l'environnement Rose. Ces informations trouvent leur place dans les squelettes de spécifications Z grâce à la structuration des formulaires qui correspond à notre classification des contraintes et aux parties des opérations.

Par contre, nous n'avons pas trouvé dans Rose d'endroit global au modèle pour définir les types. Aussi la définition des types ne s'effectue pas dans l'environnement RoZ, mais dans un fichier qui est inclus dans celui des spécifications Z lors de sa génération. Ce fichier contient aussi la définition des fonctions sur les types telles que *EstSup* et *HeureRecouvr*.

Un diagramme de classes et ses informations complémentaires étant spécifiés, RoZ peut générer une spécification Z complète. Par exemple, à partir de la classe "SESSION" et de ses annotations, RoZ génère deux schémas *SESSION* et *SessionExt* pour la partie statique de la classe (Fig. 7.6), et un schéma d'opération et sa promotion éventuelle pour chaque opération. Le schéma "SESSION" contient les contraintes sur les attributs "prix", "heureDebut" et "heureFin" alors que "SessionExt" comporte les contraintes sur l'extension de la classe.

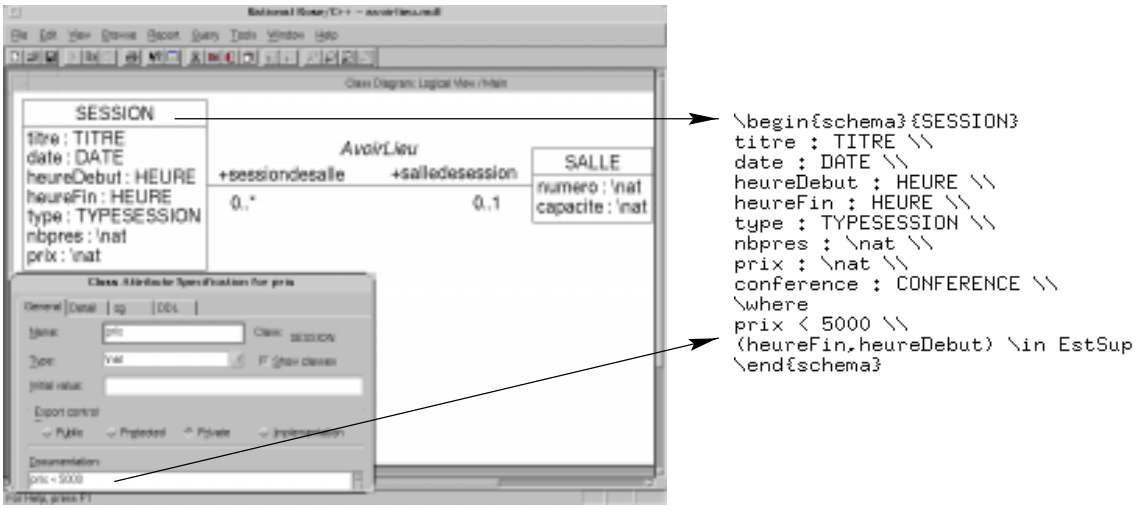


FIG. 7.6: Génération d'une spécification Z au foprmat Latex

7.2.4 Génération des théorèmes de validation des gardes

Une fois que la spécification Z a été produite, il est possible d'étudier les propriétés du modèle. Dans ce but, nous avons choisi de valider les gardes candidates des opérations [Led98] (cf. 6.2.2). Chaque opération correspond donc à un théorème à prouver afin de valider sa garde *precondition(Etat,e?)* :

theorem Op_Pre

$$\forall Etat; e? : ENTREE \mid garde(Etat, e?) \bullet \text{pre } Op$$

Ce théorème nécessite d'avoir exprimé la garde candidate de chaque opération. Suivant notre principe d'utilisation des formulaires, nous employons le champ "Pre

Conditions" du formulaire d'une opération pour y écrire sa garde. Ainsi RoZ peut générer un théorème de validation de garde pour chaque opération.

Considérons l'opération de base "ChangePrix" ("version" anglaise de l'opération *SESSIONModifierPrix*) générée précédemment :

SESSIONChangePrix

$$\Delta SESSION$$
$$newprix? : \mathbb{N}$$

$$prix' = newprix?$$
$$titre' = titre \land date' = date \land heureDebut' = heureDebut$$
$$heureFin' = heureFin \land type' = type \land nbpres' = nbpres$$

Nous avions identifié $newprix? < 5000$ comme pré-condition de *SESSIONChangePrix*. Il suffit de l'écrire dans le champ "Pre Conditions" de "ChangePrix" pour que RoZ puisse générer automatiquement le théorème servant à la valider (Fig. 7.7) :

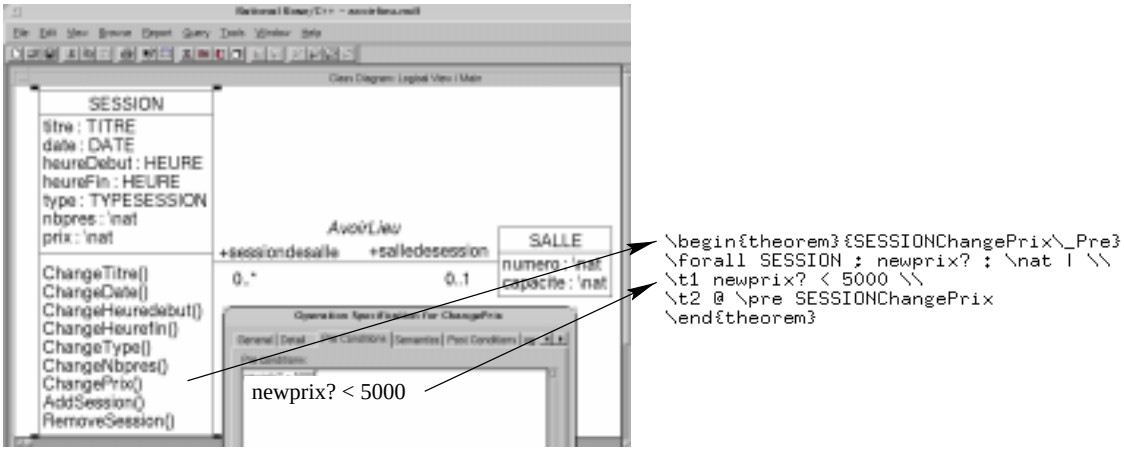


FIG. 7.7: Génération du théorème de validation de la garde de "ChangePrix"

7.3 Bilan

7.3.1 Conditions d'utilisation

RoZ utilise l'éditeur de diagrammes de classes de Rose et ses formulaires pour faire le lien avec le monde des spécifications formelles. Il permet de générer des spécifications Z correspondant aux concepts de classe, d'attribut, d'opération, d'association binaire, d'héritage, d'agrégation et de composition. Pour pouvoir générer une spécification Z complète et syntaxiquement correcte, le diagramme de classes doit contenir les informations suivantes :

- chaque attribut doit être typé ;
- chaque opération doit avoir au moins sa signature et

– les rôles des associations et leur cardinalité doivent être spécifiés.

Il faut aussi définir un fichier au format Latex contenant les types de attributs et toute autre définition non incluse dans Rose. Les autres annotations (contraintes et spécifications des opérations) sont exprimées dans les formulaires standard de Rose au format Latex de Z. RoZ tire avantage de la structure de ces formulaires pour classifier les annotations. Ainsi il exploite la structure des champs "Documentation" pour mettre en oeuvre notre classification des contraintes. Mais cette structure ne correspondant pas exactement à la classification (les contraintes sur plusieurs attributs n'ont pas de formulaire), nous avons dû l'adapter (Tab. 7.1). De plus, comme RoZ ne supporte pas la traduction des associations n-aires et des classes associatives, le support à la classification n'est pas total.

Contraintes restreignant	Formulaire de
Un attribut d'un objet d'une classe	attribut
Plusieurs attributs d'un objet d'une classe	un des attributs (au choix)
Connexions mutuelles entre objets d'une même classe	classe
Connexions mutuelles entre objets d'une même association	association
L'intension d'une super-classe	relation d'héritage
L'extension d'une super-classe et d'une sous-classe	sous-classe
Connexions entre objets d'une relation d'héritage	super-classe
Connexions mutuelles entre objets de différentes associations	modèle global

TAB. 7.1: Formulaires supportant la classification des contraintes

De façon identique à l'expression des contraintes, les formulaires des opérations servent à contenir leurs informations relatives :

– champ "Arguments"
  Il contient les paramètres d'entrées de l'opération. Le champ "Nom" contient le nom des variables et le champs "Type" leur type.

– champ "return type" ou "return class"
  Il est utilisé pour représenter le type de la sortie de l'opération. Comme il ne peut contenir qu'un seul type, toute opération Z générée ne peut avoir qu'une seule variable de sortie.

– champ "Semantics"
  Il sert à définir le type de l'opération. Il ne peut contenir que l'un des mots-clés suivants :

  – *intension operation* pour une opération sur l'intension de la classe ;
  – *extension operation* pour une opération sur l'extension de la classe ;

  - *composed operation* pour une opération composée à partir d'autres opérations ;
  - *promotion operation* pour une opération sur l'intension et l'extension de la classe ;
  - sinon le champ "Semantics" contient explicitement la déclaration des schémas accédés ou modifiés par l'opération.

- champ "Post Conditions"
  Il contient l'expression en Z de la post-condition de l'opération.

- champ "Pre Conditions"
  Il contient une garde identifiée de l'opération.

Ces conditions d'utilisation sont nécessaires pour générer une spécification formelle complète. En effet, toutes les informations ajoutées servent à remplir les "vides" des squelettes de spécifications Z produits par traduction du modèle objet seul. La structuration de ces informations grâce aux formulaires, au fichier de types ou aux détails du modèle est utilisée par RoZ pour connaître l'endroit où les intégrer dans les spécifications Z.

7.3.2 Points forts

La réalisation de l'outil RoZ nous a tout d'abord permis de préciser nos propositions de traduction. Elle démontre que ces propositions sont suffisamment détaillées pour servir de base à l'outil. En outre, elle a permis d'étudier plus rapidement des alternatives de traduction et de les tester.

Du point de vue de son utilisation, une importante caractéristique de RoZ est que tout le travail de modélisation (modèles et annotations) est réalisé dans Rose pour être aussi proche que possible de l'environnement de travail habituel. L'environnement de modélisation est utilisé comme d'habitude et des informations peuvent être ajoutées aux modèles lorsque c'est nécessaire. Il n'est donc a priori pas nécessaire d'avoir une compréhension fine de Z pour obtenir une spécification Z complète.

Mais le principal avantage de RoZ est qu'il réalise automatiquement des tâches fastidieuses. Premièrement, il génère des spécifications formelles à partir d'un diagramme annoté. Deuxièmement, il crée les opérations de base des classes. Enfin il produit des obligations de preuves. Cette automatisation limite les efforts humains pour écrire des spécifications précises et les vérifier.

De plus, RoZ offre un guide méthodologique pour savoir où exprimer quel type d'annotations. En particulier, il supporte la classification des contraintes que nous proposons. Ainsi chaque type de contraintes est écrite dans un formulaire spécifique, ce qui aide l'utilisateur à savoir où exprimer les contraintes et comment annoter un modèle.

Enfin il est important d'avoir des évolutions synchronisées des spécifications semi-formelles et formelles. Notre outil permet de toujours avoir des versions équivalentes des spécifications UML et Z. Comme tout le travail de modélisation est fait dans

l'environnement UML, une modification de la spécification s'effectue dans cet environnement et la spécification Z correspondante est générée à nouveau. La cohérence du modèle et de ses contraintes peut être revérifiée et corrigée si nécessaire.

La classification des annotations et la synchronisation des versions des spécifications UML et Z facilitent les évolutions des spécifications en fournissant un cadre clair concernant le lieu où les changements doivent avoir lieu.

7.3.3 Points faibles

Bien que nous ayons tenté de rendre l'outil RoZ le plus indépendant possible des spécifications formelles, la convivialité de RoZ est encore limitée. Il est toujours nécessaire de connaître Z et nos principes de traduction. En effet, tant que l'expression des contraintes s'effectue en Z, il faut avoir une idée de la forme des schémas Z que les contraintes complètent. Par exemple, l'écriture d'une contrainte sur une association nécessite de connaître la traduction des rôles.

De plus, l'utilisation du style Latex n'aide pas à l'écriture des annotations car ce format est loin d'être intuitif. L'interface de RoZ doit donc évoluer afin de faciliter l'expression des annotations. Il semble en particulier possible d'utiliser OCL pour être plus proche des praticiens. OCL et Z étant des formalismes proches, les annotations écrites en OCL pourraient être traduites automatiquement en Z avant leur intégration dans les squelettes de spécifications.

Comme nous n'avons pas pu adapter les formulaires à nos besoins, l'interface de RoZ pèche aussi par le fait que certaines informations telles que la définition des types doivent être exprimées dans un fichier à part. Néanmoins il semble que les versions plus récentes de Rose laissent plus de liberté quant à la personnalisation des interfaces et nous permettent ainsi de définir les formulaires que nous désirons.

Enfin RoZ pâtit des limites théoriques de notre travail. Tout d'abord, nous ne considérons actuellement que les contraintes statiques, excluant de ce fait toutes les contraintes dynamiques. Ceci est induit par l'utilisation de Z qui a pour but d'exprimer des contraintes d'invariants. Pour que RoZ puisse aussi prendre en compte les contraintes dynamiques, il faudrait étudier la traduction des modèles semi-formels, en particulier du modèle dynamique, dans d'autres langages formels plus appropriés tels que Lustre.

Les opérations de base ne concernent actuellement que les classes. Leur génération automatique pourrait aussi concerner d'autres concepts tels que les associations et tenir compte des dépendances entre les classes imposées par les cardinalités.

Enfin RoZ génère actuellement des obligations de preuve, mais il ne fournit aucune aide pour les prouver. Le travail de preuve à effectuer demeure donc important. Simultanément à la génération d'une obligation de preuve, il faudrait générer un "schéma" de preuves correspondant qui indiquerait les étapes à suivre. C'est dans ce sens que nous avons étudié la validation des gardes des opérations de base au chapitre précédent.

L'analyse des avantages et des inconvénients de RoZ est, pour l'instant, issue de notre expérience personnelle sur des exemples relativement simples. Il faut maintenant expérimenter RoZ plus largement. C'est pourquoi nous le diffusons actuellement sur Internet (http ://www-lsr.imag.fr/Les.Groupes/PFL/RoZ/index.html) avec un

exemple et une documentation [Dup99].

7.4 Comparaison avec d'autres outils

La comparaison de RoZ avec d'autres outils porte sur sa fonctionnalité principale c'est-à-dire le couplage de notations semi-formelles et formelles. Nous ne tenons donc pas compte des fonctionnalités annexes qui facilitent ou préparent l'exploitation de spécifications formelles.

Divers environnements couplant différents types de spécifications ont été proposés. Certains se basent sur des méta-outils comme GraphTalk [Fre97, Ngu98]. Un méta-outil est un outil destiné à construire d'autres outils. D'autres, comme le nôtre, proposent d'étendre des outils existants. En particulier, l'outil Rose a déjà permis de faire le lien entre UML et VDM++ [IFA] et UML et Z [Hea]. L'utilisation d'un méta-outil permet de construire exactement l'environnement désiré. Mais son développement est beaucoup plus long que l'extension d'un environnement existant et reste souvent au niveau de prototype. De plus, l'utilisation d'un environnement "standard" permet de rester proche des habitudes de travail des concepteurs de SI.

Une autre comparaison peut être faite sur la façon de coupler les types de spécifications. [Fre97] et [Hea] proposent des environnements de multi-modélisation dans lesquels les différents types de spécifications sont disponibles dans le même environnement de modélisation. On développe donc en parallèle les spécifications semi-formelles et formelles. Dans cette approche, il est nécessaire de bien connaître les différents formalismes, en particulier pour pouvoir ajouter des contraintes aux spécifications formelles. L'autre approche consiste à générer automatiquement les spécifications formelles dans un fichier. L'avantage est de disposer d'un fichier qui peut être utilisé par les outils des langages formels. C'est ce qui est proposé dans [FBLP97]. Mais cet article ne précise pas comment compléter les spécifications formelles du fichier. Notre proposition va plus loin en proposant d'effectuer tout le travail de modélisation (modèles et contraintes) dans l'environnement standard graphique d'UML. On ne voit alors des spécifications formelles que le strict minimum nécessaire à leur exploitation. Le niveau d'exploitation ultérieure de la spécification formelle (preuves etc) obtenue dépend alors du niveau de maîtrise du langage formel. En effet, il n'est pas nécessaire de connaître en détail le langage formel cible pour produire une documentation enrichie de spécifications formelles. Par contre, il faut une bonne maîtrise des spécifications formelles pour effectuer la vérification ultérieure à l'aide d'un outil d'aide à la preuve de Z par exemple.

7.5 Conclusion

Ce chapitre présente RoZ, un outil automatique pour la spécification et illustre son utilisation sur un exemple simple. RoZ a pour but d'améliorer la qualité des systèmes en ajoutant des précisions aux spécifications du système. Il exploite l'environnement Rose pour faire cohabiter des notations UML et des annotations formelles en Z : le diagramme de classes fournit la structure des spécifications Z alors que les

annotations exprimées dans des formulaires ajoutent les détails. A partir d'un diagramme annoté, RoZ offre les possibilités suivantes :

- la génération automatique de spécifications formelles Z ;
- la génération dans Rose de la spécification des opérations de base sur les classes ;
- la génération d'obligation de preuves pour valider les gardes des opérations.

Chapitre 8

Quel type de couplage, pour quels bénéfices potentiels ?

Ce chapitre présente l'approche de couplage des notations développée par l'équipe du Prof. Finkelstein à University College London (Angleterre) [EEF+99] : les différentes spécifications sont développées indépendamment, puis leur cohérence est vérifiée suivant des règles pré-définies. Dans le cadre de cette approche, nous avons défini des règles de cohérence entre des notations semi-formelles et Z qui nous permettent d'établir la comparaison avec notre approche de traduction.

8.1 Vérification de la cohérence a posteriori par méta-modélisation

8.1.1 Approche

L'approche présentée dans [EEF+99] a pour but d'établir des liens de cohérence entre des documents qui ont été développés indépendamment. En effet, de nombreux documents sont généralement écrits par différents acteurs tels que le concepteur du système ou le futur utilisateur. Certains d'entre eux ont des "objets" communs qui définissent des zones de recouvrement entre documents. Les contenus se recouvrant donnent lieu à des relations de cohérence entre les documents ou leurs éléments. Il faut donc assurer l'intégrité des documents à certains moments du processus de développement, en tolérant dans le même temps des incohérences passagères.

L'approche que nous étudions pour gérer la cohérence de spécifications contenues dans des documents se base sur la définition de relations de cohérence entre différents types de documents et leur vérification (Fig. 8.1) :

1. La première étape consiste à identifier les types de documents utilisés dans le processus de développement.

2. La structure de chaque type de document est ensuite définie. Cela revient à spécifier le méta-modèle des documents.

3. A partir de la structure des documents, il est possible d'exprimer des relations

de cohérence entre les concepts des différents documents. Ces relations sont décrites à travers des règles de cohérence, une règle spécifiant une forme de relation ou un fait qui doit être vérifié. Puis il est possible d'exécuter les vérifications de cohérence définies à partir des règles, de générer les liens entre les documents ou d'identifier les incohérences.

4. Lors de la quatrième étape, les résultats de l'étape précédente sont visualisés : les liens de cohérence créés complètent les documents et permettent de naviguer entre leurs éléments communs ; sinon les incohérences sont notifiées.

5. Après avoir établi des liens de cohérence pour un ensemble de documents, il faut aussi gérer les modifications éventuelles des documents. Si un changement se produit, les vérifications de cohérence doivent être effectuées à nouveau. En fait, les documents sont comparés régulièrement (toutes les x minutes ou après avoir reçu une notification de modification) avec leur version antérieure. Leur cohérence est à nouveau vérifiée et leurs liens sont recréés.

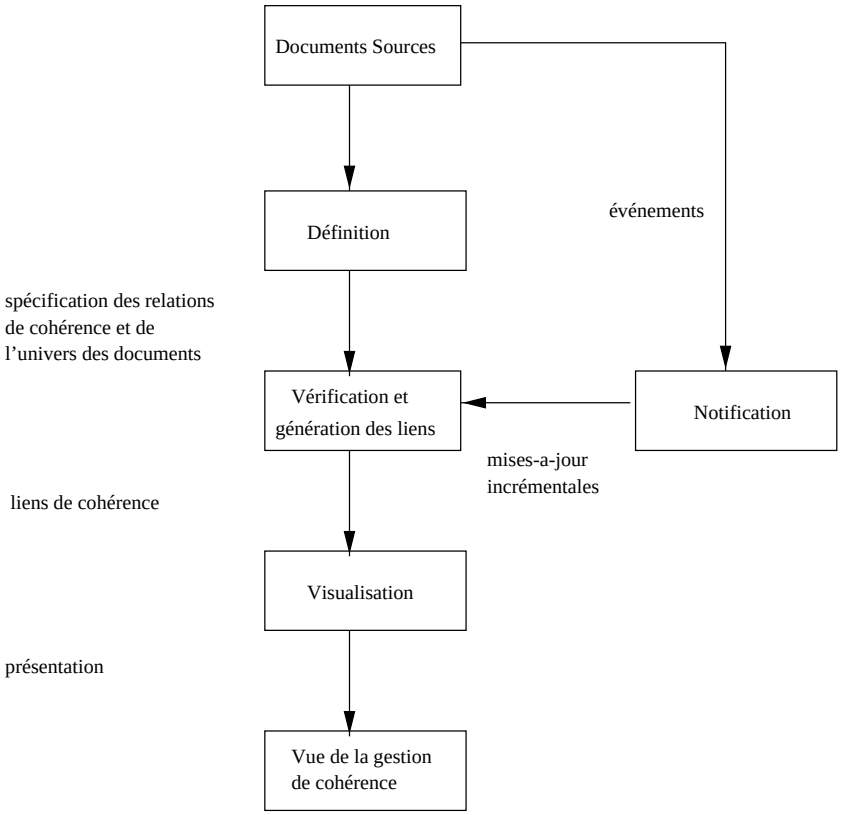


FIG. 8.1: Approche de gestion de cohérence a posteriori [EEF+99]

8.1.2 Outil de support

L'approche proposée par [EEF+99] est supportée par un outil qui gère la cohérence des documents grâce à l'eXtensible Markup Language (XML [BPSM98]) et à

ses technologies associées. XML est un langage de description de données qui est un sous-ensemble du Standard Generalized Markup Language (SGML). XML permet aux utilisateurs de définir les balises relatives au contenu de leurs documents. Il maintient la séparation entre les détails de présentation et la structure des données, permettant ainsi l'intégration de données à partir de diverses sources. Il fournit un ensemble de types d'éléments qui servent à définir les types des documents appelés Définitions de Type de Documents (DTDs). Une DTD contient la définition de la syntaxe d'un document XML en présentant les relations structurelles entre ses éléments. La figure 8.2 montre un document contenant un diagramme de classes UML avec un fragment de sa représentation en XML basée sur UXF [SY98], une DTD pour baliser UML. Les deux premières lignes de la représentation XML servent à définir le contexte du document c'est-à-dire la nature du document (ici un document XML de version 1.0) et le type de DTD utilisé (ici "uml_id.dtd"). Vient ensuite la description du modèle dont le titre est "Conference Class Diagram". Ce modèle fait partie du paquetage "Vue Soumission" qui est constitué d'un diagramme de classes. Entre les balises "<ClassDiagram>" et "</ClassDiagram>", le diagramme est spécifié. Ici nous avons extrait à titre d'exemples le déclaration de la classe "SOUMISSION", de son attribut "titre" et de son association avec "CONFERENCE".


```
<?xml version="1.0"?>
<!DOCTYPE  Model SYSTEM "uml_id.dtd">
<Model>
<TaggedValue>
        <Tag>Title<Value>Conference Class Diagram</Value></Tag>
</TaggedValue>
<Package NAME="Vue Soumission">
<ClassDiagram>

<Class NAME="SOUMISSION"
       ID="class1"
       VISIBILITY="public">
<Attribute NAME="titre"
           ID="attribute1"
           VISIBILITY="public"
           TYPE="TITRE">
</Attribute>
...
<Association PEER="CONFERENCE"
             ID="association1"
             NAME="soumettre">
        <AssocRole MULTIPLICITY="1..1" ID="assocrole1"/>
        <PeerAssocRole MULTIPLICITY="0..*" ID="peerassocrole1"/>
</Association>
...
</ClassDiagram>
</Package>
</Model>
```

FIG. 8.2: Représentation XML d'un modèle objet

XML et ses technologies associées sont utilisées pour mettre en oeuvre l'approche de gestion de cohérence entre des documents. On considère que les documents à gérer sont fournis en XML, leur structure étant alors définie dans des DTDs. Les règles de

cohérence sont spécifiées suivant la structure (i.e. le DTD) des documents à gérer. Suivant ces règles, l'outil qui a été développé permet de créer les liens hypertextes entre les éléments des différents documents : pour chaque règle de cohérence, le vérificateur de cohérence crée un lien entre les éléments s'ils vérifient la règle, sinon un lien d'incohérence est généré.

8.1.3 Règles de cohérence

Pour pouvoir créer les liens entre les documents, le vérificateur de cohérence se base sur un format pré-défini de règles. Elles doivent être de la forme *Source/Destination/Condition*. *Source* et *Destination* représentent les éléments potentiels à lier et *Condition* décrit la condition à vérifier pour créer un lien entre les éléments source et cible. Suivant ce format, la règle "pour toute classe, il doit exister un schéma Z de même nom dans un document Z" a pour source les classes des différents documents contenant des diagrammes de classes. Les destinations potentielles sont les schémas Z des documents Z. Une classe et un schéma sont liés s'ils ont le même nom (condition).

De plus, le vérificateur de cohérence définit trois types de règles de cohérence : *CT, CF* et *IF*. Le premier argument (*C* ou *I*) détermine si deux éléments sont liés parce qu'ils sont respectivement dans un état cohérent ou incohérent. *I* sert en fait à exprimer les règles négatives du type "toute super-classe d'une classe C ne peut pas être sous-classe de C dans un autre modèle". Dans le cas d'une règle de ce type, il existe une incohérence si la forme affirmative de la règle est vérifiée. Le second argument (*T* ou *F*) précise si la règle de cohérence est obligatoire ou non. Si la valeur est *T*, chaque élément source doit correspondre à un élément cible sinon il existe une incohérence. Si la valeur est *F*, s'il existe une destination correspondant à l'élément source, un lien est créé entre la source et la destination ; sinon (s'il n'y a pas de destination), cela ne signifie pas forcément qu'il y a une incohérence. Cela permet de préciser que la vérification d'une règle n'est pas obligatoire pour tolérer, par exemple, des incohérences momentanées.

Par la suite, nous définissons nos règles de cohérence pour qu'elles puissent être intégrées dans le vérificateur de cohérence c'est-à-dire en nous basant sur le format *Source/Destination/Condition*. Néanmoins nous ne précisons pas le type de chaque règle qui dépend de l'étape du processus de développement dans lequel nous nous trouvons.

8.2 Règles de cohérence entre le modèle objet et Z

Cette section présente les règles pour vérifier la cohérence entre des notations semi-formelles objet et Z. Deux ensembles de règles sont définis : le premier décrit des recouvrements entre les documents alors que le deuxième correspond à nos propositions de traduction du chapitre 5. Nous ne donnons ici que les règles de recouvrement concernant le modèle objet. D'autres règles traitant d'autres modèles semi-formels dont le modèle dynamique, sont énumérées dans l'annexe D.

8.2.1 Structure des spécifications

Les règles de cohérence sont décrites suivant la structure des documents contenant des spécifications semi-formelles et formelles. Le vérificateur de cohérence utilisant les DTDs de XML pour définir la structure des documents, nous devons employer une DTD pour chacune des notations de notre travail.

Pour le modèle objet, nous utilisons la DTD existante pour le diagramme de classes d'UML défini dans UXF [SY98]. Pour Z, nous avons dû créer notre propre DTD en nous basant sur le Z Interchange Format défini dans le standard de Z (ZIF [Z95]). Le ZIF est un langage à balises basé sur SGML qui spécifie les balises pour les plus hauts niveau de la hiérarchie de la syntaxe Z. Pour pouvoir l'utiliser avec le vérificateur de cohérence, il a été adapté à XML et complété afin de décrire des détails de plus bas niveau de structuration d'une spécification Z. Ces DTDs complets sont présentés dans l'annexe D.

8.2.2 Règles de recouvrement

Nous appelons règles de recouvrement les règles qui définissent des éléments communs entre des documents contenant des spécifications semi-formelles et Z. Pour chaque concept d'un modèle, nous essayons d'identifier le ou les éléments qui lui correspondent dans la notation cible. Le but recherché est ici de donner une idée à l'utilisateur des éléments qui pourraient être communs. C'est ensuite à lui de décider si deux éléments sont vraiment cohérents ou non. L'écriture des règles de recouvrement n'a donc pas pour but de rechercher dans la spécification destination l'expression de toutes les particularités d'un concept, mais de donner des pistes sur les éléments qui ont de fortes probabilités d'être liés.

Comme toute règle a un sens (de la source vers la destination), nous identifions des règles de recouvrement allant des spécifications semi-formelles vers Z et des spécifications Z vers le semi-formel.

Du modèle objet vers Z

Comme pour les propositions de traduction du chapitre 5, nous définissons des règles de recouvrement pour les principaux concepts du modèle objet (classe, attribut, opération, association et héritage).

De manière intuitive, le concept de classes est proche de celui de schéma Z. La règle de recouvrement pour les classes spécifie donc que toute classe doit correspondre à un schéma de même nom. L'expression de cette règle pour le vérificateur de cohérence doit spécifier sa source, sa destination et sa condition. La source est issue d'un document contenant des ensembles de modèles, les paquetages (*(all,package)*) pour lesquels nous examinons tous les diagrammes de classes (*(all,classdiagram)*). Parmi ces diagrammes, nous étudions toutes les classes (*(all,class)*). Suivant le même principe d'accès aux éléments d'un document, la destination est spécifiée en considérant tous les schémas de tous les paragraphes écrits en Z, de toutes les sections du document. Pour chaque couple (classe,schéma), les noms de la classe et du schéma sont comparés (class.name=schema.name). De plus, le schéma Z correspondant à

une classe doit être un schéma d'état et non un schéma d'opération (*schema .purpose = state*). Si un couple (classe,schéma) vérifie ces deux conditions, le schéma et la classe sont liés par un lien de cohérence.

Règle de cohérence 1 : Classe
Pour toute classe d'un modèle objet, il doit exister un schéma Z de même nom dans le document Z.
Source: (all,package) ;(all,classDiagram) ;(all,class)
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : class.name=schema.name and schema .purpose = state

Si une classe correspond à un schéma Z, l'équivalent Z d'un attribut est une variable. La source de la règle de recouvrement pour les attributs est donc l'ensemble des attributs des classes alors que la destination est l'ensemble des variables des schémas. Pour faire le lien entre un attribut et une variable, il faut non seulement qu'ils aient le même nom, mais aussi que la classe et le schéma auxquels ils appartiennent aient aussi le même nom.

Règle de cohérence 2 : Attribut
Pour tout attribut d'une classe d'un modèle objet, il doit exister une variable de même nom dans le schéma Z correspondant à la classe.
Source: (all,package) ;(all,classDiagram) ;(all,class) ;(all,attribute)
Destination : (all,section) ;(all,Zparas) ;(all,schema)(all,variable)
Condition : variable.name=attribute.name and
ancestror(class).name=ancestror(schema).name and schema.purpose = state

Bien que le principe d'encapsulation d'opération n'existe pas en Z, un schéma d'opération correspond au concept d'opération du modèle objet. Ainsi toute opération doit être liée à un schéma d'opération dans la spécification Z.

Règle de cohérence 3 : Opération
Pour toute opération d'une classe d'un modèle objet, il doit exister un schéma d'opération de même nom dans le document Z.
Source: (all,package) ;(all,classDiagram) ;(all,class) ;(all,method)
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : method.name=schema.name and schema.purpose=operation

Le concept d'association du modèle objet ne trouve pas d'équivalence directe en Z. En fait, plusieurs concepts de Z nous semblent pouvoir l'exprimer : il peut être représenté par un schéma, un produit cartésien, une relation ou une fonction. La destination de la règle de recouvrement peut donc être soit un schéma, soit une variable de schéma de type produit cartésien, relation ou fonction. Si l'association est décrite dans un schéma Z, ils doivent avoir le même nom. Si l'association correspond à une variable, cette variable doit définir un produit cartésien, une relation ou une fonction.

Pour des raisons de simplicité, la règle ne précise pas que les types liés par ces relations doivent avoir le même nom que les classes liées par l'association. De même, nous ne cherchons pas dans une spécification Z les particularités d'une agrégation ou d'une composition en la traitant comme une association simple.

Règle de cohérence 4 : Association/Agrégation/Composition
Pour toute association (ou agrégation ou composition) d'un modèle objet, il doit exister un schéma Z, une relation, une fonction ou un produit cartésien de même nom dans le document Z.
Source: (all,package) ;(all,classDiagram) ;(all,class) ;(all,association)
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : association.name=schema.name
OR
Destination : (all,section) ;(all,Zparas) ;(all,schema) ;(all,variable)
Condition : association.name = variable.name and
(variable.(all,rel).kind='fct' or variable(all,rel).kind='relation' or variable(all,rel).kind='cartesianproduct'))

La règle de recouvrement pour les associations est assez imprécise. Elle montre déjà les limites entre la volonté d'exprimer de façon simple des liens entre spécifications et le besoin de précision qui assure que les éléments liés sont bien cohérents.

Pour l'héritage, il est difficile de trouver des concepts équivalents en Z. La seule notion qui possède des similarités en Z est l'inclusion de schéma. Pour ne pas entrer dans la complexité de représentation de l'héritage en Z, nous nous contentons de faire correspondre à l'héritage l'inclusion de schéma, même si celle-ci ne représente que l'héritage d'attributs. La source de la règle est donc l'ensemble des relations d'héritage des diagrammes de classes. Dans le DTD d'UML, l'héritage s'exprime en définissant dans une classe sa super-classe *(all,Generalization)*. La destination de la règle est l'ensemble des schémas inclus (*(all, formals)*) dans les schémas des spécifications Z. La condition spécifie qu'une sous-classe (c'est-à-dire l'élément à partir duquel part la relation de généralisation (*ancestror(class)*) et le schéma incluant un autre schéma ont des noms identiques et que le nom de la super-classe (*Generalization.from*) est le même que celui du schéma inclus (*formals.name*).

Règle de cohérence 5 : Héritage
Pour toute sous-classe d'un modèle objet, il doit exister un schéma Z de même nom incluant un schéma correspondant à la super-classe.
Source: (all,package) ;(all,classDiagram) ;(all,class) ;(all,Generalization)
Destination : (all,section) ;(all,Zparas) ;(all,schema) ;(all,formals)
Condition : ancestror(class).name=ancestror(schema).name and
Generalization.from=formals.name

La règle de recouvrement pour l'héritage met en évidence que s'il n'existe pas de concepts proche de celui considéré, la règle peut se limiter à chercher dans le document destination une représentation très partielle.

De Z vers le modèle objet

Les règles de recouvrement de Z vers le modèle objet sont symétriques à celles allant de modèle objet vers Z. Nous étudions donc les concepts que nous avons évoqués précédemment (schéma, inclusion de schéma, produit cartésien, relation et fonction).

Lors de la définition des règles allant du modèle objet vers Z, nous avons traité différemment les schémas d'état et les schémas d'opérations. Nous retrouvons cette distinction puisque le type du schéma influe sur le concept du modèle objet correspondant. Si un schéma est d'état, il peut correspondre à une classe alors que s'il est d'opération, il existe une opération équivalente dans le modèle objet et un événement, une action dans le modèle dynamique. Il est donc nécessaire de filtrer la source de la règle pour savoir si un schéma est d'état ou d'opération. Nous étendons le format *Source/Destination/Condition* pour exprimer une condition sur la source. Par exemple, la règle concernant les schémas d'états n'a de sens que si les schémas examinés sont des schémas d'état. Sinon la règle ne s'applique pas.

Règle de cohérence 6 : Schema d'état
Pour tout schéma d'état, il peut exister une classe de même nom dans le modèle objet.
Source: (all,section) ;(all,Zparas) ;(all,schema)
Filtre :schema.purpose='data type'
Destination : (all,package) ;(all,classDiagram) ;(all,class)
Condition : schema.name=class.name

La règle pour les schémas d'opération a aussi la particularité de nécessiter des opérateurs entre les destinations. Un schéma d'opération correspond à la fois à une opération du modèle objet et à un autre concept dans le modèle dynamique. Cela revient à faire la conjonction de deux règles de recouvrement ayant les mêmes sources, mais des destinations et des conditions différentes.

De plus, dans le modèle dynamique, les concepts d'événement ou d'action peuvent correspondre à un schéma d'opération. La destination de la règle peut donc varier : les destinations potentielles sont liées par des OU.

Règle de cohérence 7 : Schéma d'opération
Pour tout schéma Z d'opération, il peut exister une opération de même nom dans le modèle objet et un événement ou une action dans le modèle dynamique.
Source: (all,section) ;(all,Zparas) ;(all,schema)
*Filtre :*schema.purpose='operation'
Destination : (all,package) ;(all,classDiagram) ;(all,class) ;(all,method)
Condition : schema.name=method.name
AND
Destination : (all,package) ;(all,stateDiagram) ;(all,event)
Condition : event.name= schema.name
OR
Destination : (all,package) ;(all,stateDiagram) ;(all,action)
Condition : action.name= schema.name

Dans la section précédente, nous avons parlé des concepts de produit cartésien, de relation et de fonction pour évoquer la notion d'association du modèle objet. Ces concepts doivent donc correspondre à une association dans un modèle objet.

Règle de cohérence 8 : Relation/Produit cartésien
Pour toute relation ou pour tout produit cartésien, il doit exister une association de même nom dans le modèle objet.
Source: (all,section) ;(all,Zparas) ;(all,schema) ;(all,variable)
Destination : (all,package) ;(all,classDiagram) ;(all,class) ;(all,association)
Condition : variable.name=association.name and
(variable.(all ;rel).kind='relation' or
variable.(all,rel).kind='cartesianproduct')

De plus, une fonction ayant un sens, elle peut aussi correspondre à un rôle d'une association.

Règle de cohérence 9 : Fonction
Pour toute fonction, il doit exister une association ou un rôle de même nom dans le modèle objet.
Source: (all,section) ;(all,Zparas) ;(all,schema) ;(all,variable)
Destination : (all,package) ;(all,classDiagram) ;(all,class) ;(all,association)
Condition : (variable.name=association.name or
variable.name=association.(all,role).name)
and variable.(all,rel).kind='function'

Le dernier concept Z que nous étudions est l'inclusion de schéma. Toute inclusion de schéma d'une spécification Z peut correspondre à une relation d'héritage du modèle objet.

Règle de cohérence 10 : Inclusion de schéma
Pour tout schéma B incluant un schéma A, il doit exister une classe B héritant d'une classe A dans le modèle objet.
Source: (all,section) ;(all,Zparas) ;(all,schema) ;(all,formals)
Destination : (all,package) ;(all,classDiagram) ;(all,class) ;(all,Generalization)
Condition : ancestror(schema).name=ancestror(class).name and
formals.name=Generalization.from

Conclusion sur les règles de recouvrement

La définition des règles de recouvrement laisse une totale liberté à celui qui les écrit : c'est lui qui décide de leur pertinence et de la confiance qui peut leur être accordée. Les liens de cohérence ou d'incohérence ne sont qu'une aide de départ pour étudier de manière approfondie les relations entre divers documents. Ils doivent servir à mettre en évidence des concepts qui semblent communs ou des incohérences potentielles afin de provoquer un dialogue entre les différents protagonistes. Les discussions servent à mieux comprendre les documents et à en lever d'éventuelles ambiguïtés. Tout ce processus favorise l'intervention des utilisateurs et dépend fortement d'eux.

La cohérence entre deux documents dépend fortement de la vision du concepteur si bien qu'il est parfois difficile de connaître la nature de cette cohérence. Le problème vient du fait que le notion de recouvrement n'est pas clairement définie. Si les concepts des différents types de documents sont proches, la relation de cohérence à définir est naturelle. Sinon faut-il choisir parmi les solutions possibles (cf. association), ne faire une correspondance que partielle (cf. héritage), écrire une règle totalement subjective ou ne pas écrire de règle du tout ?

Dans ce travail, nous sommes partie du principe qu'une règle de recouvrement pouvait exprimer les différents choix de traduction possibles en assouplissant l'expression des traductions pour permettre une écriture aisée des règles. Mais jusqu'à quel degré l'assouplissement des traductions garantit-il la cohérence entre les concepts et peut-on réellement couvrir toutes les traductions possibles ? Il est donc très difficile de savoir si une règle de recouvrement décrit réellement une relation de cohérence et de quelle cohérence il s'agit.

Les règles de recouvrement font de la sémantique statique entre les différentes spécifications. Pour des spécifications orthogonales telles que les modèles statique et dynamique, où les correspondances sont peu nombreuses, elles sont suffisantes pour établir une certaine cohérence. Par contre pour des spécifications complémentaires c'est-à-dire des vues représentant les mêmes aspects d'un système avec des formalismes différents, elles ne sont pas assez précises pour garantir une cohérence suffisante.

8.2.3 Règles de traduction

Une autre façon d'exprimer la cohérence entre des spécifications est de définir des règles de traduction. Une règle de traduction doit correspondre à une équivalence sémantique entre les concepts des notations. Dans le contexte de la vérification de cohérence a posteriori, elle sert à vérifier si des spécifications déve-

loppées indépendamment, ont suivi le processus de traduction. Nous décrivons ici nos propositions de traduction du modèle objet vers Z sous la forme du format Source/Destination/Condition.

Du modèle objet vers Z

Suivant notre règle de traduction 1, une classe correspond en Z à deux schémas d'état : le premier ayant le même nom que la classe et le second ayant pour variable l'ensemble (*schema.(all,variable).(all,type).set = 'fset'*) des éléments de la classe. Pour une classe source, il existe donc deux schémas destinations (règle 11).

Règle de cohérence 11 : Classe
Chaque **classe "CCC"** donne lieu à :

- **un schéma-type** de nom *CCC* contenant la déclaration des attributs,
- **un schéma** *CccExt* définissant l'ensemble des objets de la classe Il comprend une seule variable *Ccc* définissant l'ensemble des objets existants ($Ccc : \mathbb{F}\ CCC$).

Source: (all,package) ;(all,classDiagram) ;(all,class)
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : class.name=schema.name and schema.purpose = 'state'
AND
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : class.name-'Ext' = schema.name and schema.purpose = 'state' and schema.(all,variable).(all,type).set='fset' and schema.(all,variable).(all,type).name=class.name

Nous ne détaillons pas ici l'expression des règles pour les attributs et les opérations car elle n'illustre aucun cas particulier, ni aucune originalité par rapport aux autres règles.

La règle de traduction des associations (règle 12) est divisée en deux parties afin de la rendre plus lisible. La première partie représente l'association dans son ensemble alors que la deuxième se concentre sur l'expression de ses rôles. Comme une classe, une association est décrite dans deux schémas Z décrivant son intension et son extension. La première règle de traduction est donc similaire à celle pour les classes : elle distingue deux destinations correspondant aux deux sens d'une association.

De plus, elle précise que les variables du schéma d'intension correspondent aux classes liées par l'association. Dans la DTD décrivant les diagrammes de classes, les associations sont l'un des éléments des classes. Pour toute association, il faut ensuite définir quelle est sa classe cible. Dans la DTD, il y a donc une classe source à partir de laquelle l'association est définie et une classe cible qui est un attribut de l'association. Pour accéder aux classes liées par une association, il faut accéder d'une part à sa classe source (*ancestror(class).name*) et d'autre part à sa classe cible (*association.peer*).

Règle de cohérence 12 : Association
Chaque **association "A"** entre les classes "CCC_1, .., CCC_n" donne lieu à :

- **un schéma Z** *ARel* décrivant les liens entre les objets potentiels
  Pour chaque classe CCC_i impliquée dans l'association, il existe une variable ccc_i de type CCC_i (ccc_i : CCC_i).

- **un schéma Z** *ARelExt* définissant l'extension de l'association.

Source: (all,classDiagram) ;(all,class) ;(all,association)
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : association.name-'Rel'=schema.name and
schema.(all,variable).name =ancestror(class).name and
schema.(all,variable).name = association.peer *AND*
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : association.name-'RelExt' = schema.name and schema.purpose = 'state'

Les rôles d'une association sont décrits en Z à la fois par la définition de fonctions et par des prédicats spécifiant leur cardinalité minimale et leur réciprocité. L'expression en Z des cardinalités minimales et de la réciprocité des fonctions nécessite de détailler la structure d'une spécification Z afin de décrire toutes les formes possibles de prédicats. Cela revient à écrire au niveau de la DTD de Z sa grammaire. Compte-tenu du niveau de détail demandé, nous nous contentons d'exprimer la cardinalité maximale des rôles.

Les rôles illustrent bien les limites d'une vérification basée sur la structure des documents et sur des règles de cohérence. En effet, la précision des propositions de traduction nécessite à la fois des structures de documents très détaillées et l'expression de conditions complexes.

Ces limites se retrouvent dans le cas de l'héritage. Bien qu'il n'existe a priori pas d'obstacle théorique à l'écriture de la règle de traduction concernant l'héritage, elle nécessite une précision très approfondie des DTD décrivant les structures des modèles et un niveau de détail qui la rendrait illisible. Sa complexité nous amène à ne pas la décrire sous forme de Source/Destination/Condition.

Conclusion sur les règles de traduction

Les règles de traduction sont des règles de cohérence qui décrivent une équivalence sémantique entre les concepts des notations utilisées. La cohérence est donc définie comme étant la vision de l'équivalence sémantique du concepteur. Elles doivent être précises afin de garantir cette équivalence et d'assurer un haut niveau de confiance dans les liens de cohérence créés.

Mais l'équivalence sémantique n'est réalisable que pour des spécifications complémentaires pour lesquels les langages utilisés sont proches. Dans ce cas, la précision des règles les rend plus difficiles et parfois même presque impossibles à écrire dans le cadre Source/Destination/Condition. Le degré de complexité demandé peut décourager plus d'un conecpteur.

Règle de cohérence 13 : Rôle
Chaque **rôle "R"** de l'association allant des classes "$CCC_1, ..., CCC_n$" vers une classe "CCC_b" donne lieu à **une fonction** R. Les cardinalités sont spécifiées de la façon suivante :

- **la cardinalité minimale** s'exprime par des prédicats dans *ARelExt*.

  Si la cardinalité minimale est **1**, alors on ajoute un prédicat qui précise que tout nuplet dont les éléments sont de type $(CCC_1, ..., CCC_n)$, il existe un élément de type CCC_b. Si le rôle est monovalué, le prédicat est de la forme : $\forall\, c1 : Ccc1 ; ... ; cn : Cccn \bullet \exists\, cb : Cccb \bullet cb = R(c1, ..., cn)$

  Si le rôle est multi-valué, le prédicat est de la forme : $\forall\, c1 : Ccc1 ; ... ; cn : Cccn \bullet \exists\, cb : Cccb \bullet cb \in R(c1, ..., cn)$

- **la cardinalité maximale** s'exprime par les fonctions représentant les rôles.

  Si la cardinalité maximale de "R" est **égale à 1**, $R : CCC_1 \times ... \times CCC_n \pfun CCC_b$. Sinon (cardinalité maximale **supérieure à 1**) $R : CCC_1 \times ... \times CCC_n \pfun \mathbb{F}\ CCC_b$.

Source: (all,package) ;(all,classDiagram) ;(all,class) ;(all,association) ;(all,role)
Destination : (all,section) ;(all,Zparas) ;(all,schema) ;(all,variable)
Condition : ancestror(association).name-'Rel'=ancestror(schema).name and
role.name=variable.name and
variable ;(all,source).name=role ;(all,source).name and
variable ;(all,des).name=role ;(all,des).name and variable ;(all,rel)='parfct'

De plus, chaque règle de traduction n'est que l'une des solutions envisageables pour traduire un concept. Dans une approche de développement des spécifications indépendamment, elle peut seulement servir à vérifier que ce développement a suivi une approche de traduction donnée. En effet, il paraît peu probable qu'a priori les spécifications vérifient des règles de traduction à moins que celles-ci n'aient été imposées dès le début du développement.

8.3 Comparaison des approches

8.3.1 Comparaison des règles

Les règles de recouvrement et de traduction définissent toutes les deux des règles de cohérence entre différentes spécifications puisqu'elles expriment des liens entre ces spécifications. Tout dépend du type de cohérence souhaitée à savoir une cohérence forte garantissant une certaine équivalence sémantique ou une cohérence plus restreinte mettant simplement en évidence les points communs des spécifications.

Les règles de traduction nécessitent d'avoir des sémantiques identiques entre deux spécifications, leur garantissant ainsi une équivalence forte. Elles doivent ainsi être précises même si cela impose qu'elles soient contraignantes. Les contraintes se justifient par le fait qu'une fois l'équivalence entre deux spécifications établie, il est

possible d'utiliser indifféremment l'une ou l'autre puisque tout raisonnement ou toute vérification effectué sur l'une d'elles est valide pour l'autre. Chaque concepteur peut choisir de travailler avec le formalisme qui lui convient le mieux en sachant qu'une représentation quasiment équivalente est disponible pour un autre concepteur.

Néanmoins les règles de traduction ne décrivent qu'une façon de dériver un formalisme dans un autre et d'autres règles expriment aussi la même équivalence sémantique. Elles reflètent la vision de l'équivalence sémantique de leur concepteur si bien qu'il est difficile de les utiliser a posteriori pour vérifier si deux spécifications sont identiques. Enfin leur précision les rend difficilement exprimables dans l'outil de vérification de cohérence basé sur les règles Source/Destination/Condition.

Les règles de recouvrement identifient des éléments communs entre deux spécifications. Elles sont moins contraignantes que les règles de traduction puisqu'elles ne se limitent pas à une seule façon d'interpréter un formalisme. Elles peuvent considérer les diverses manières de lier deux éléments.

Mais en gagnant de la liberté pour exprimer des liens de cohérence, elles perdent en précision. Il est difficile de savoir quand une règle de recouvrement définit vraiment une relation de cohérence et ce que cohérence signifie.

Compte tenu de leurs caractéristiques respectives, on peut dire que les règles de recouvrement et celles de traduction ne s'appliquent pas dans le même cadre : les règles de recouvrement conviennent pour la gestion de la cohérence entre des spécifications orthogonales alors que les règles de traduction sont bien adaptées pour vérifier la cohérence entre des spécifications complémentaires.

8.3.2 Comparaison des démarches

Bien que les règles de recouvrement et de traduction aient des caractéristiques intrinsèques qui induisent des avantages et des inconvénients, leurs bénéfices potentiels dépendent aussi de leur utilisation.

Dans notre travail, nous avons vu comment les propositions de **traduction** peuvent être utilisées dans le cadre de deux approches. Elles peuvent servir soit à **générer des spécifications formelles**, soit à **vérifier la cohérence de spécifications a posteriori par méta-modélisation**.

Compte tenu de leur précision et de leur aspect contraignant, il nous semble qu'elles ne sont pas adaptées à une approche basée sur la vérification de règles a posteriori par méta-modélisation.

Elles correspondent mieux à une approche générative c'est-à-dire à une approche qui utilise des règles de traduction pour produire à partir d'une spécification une autre spécification dans un formalisme différent. Dans le contexte de la génération d'une spécification formelle, cette approche peut alors avoir deux buts : la vérification de propriétés ou la vérification de la cohérence a posteriori.

Une approche générative permet de produire de manière quasi automatique une spécification à partir d'une autre. C'est le principe que nous employons pour développer une spécification formelle Z à partir de modèles semi-formels. Dans le cadre du couplage de spécifications semi-formelles et formelles, il a pour avantage d'aider le concepteur dans la difficile tâche de l'écriture de spécifications formelles. Les spécifi-

cations ainsi produites sont des vues équivalentes qui sont utilisées pour leurs avantages respectifs. Les spécifications semi-formelles sont relativement faciles à écrire et offrent un bon vecteur de communication alors que la précision des spécifications formelles sert à raisonner sur les modèles. L'équivalence sémantique entre les deux types de spécifications assure que tout ce qui est valide pour l'un des modèles est valide aussi pour l'autre.

L'approche générative peut aussi servir à vérifier la cohérence a posteriori : les différentes spécifications sont développées en parallèle puis traduites dans un langage commun. Une fois qu'elles sont exprimées dans le même formalisme, elles peuvent être comparées aisément pour établir leur cohérence. On peut par exemple déterminer si une spécification est un raffinement d'une autre.

Néanmoins il n'est pas toujours aisé de conserver l'équivalence sémantique. La traduction nécessite de coupler des formalismes suffisamment proches pour pouvoir exprimer les concepts de l'un dans l'autre. Cela implique qu'il peut être nécessaire d'utiliser plusieurs langages formels cibles ou de n'effectuer qu'une traduction partielle qui serait suffisante pour vérifier les propriétés voulues.

De plus, si on veut utiliser cette démarche pour vérifier la cohérence a posteriori, il faut développer des règles de traduction de chacun des langages de spécifications utilisés en un langage commun. Le travail demandé est alors considérable même s'il paraît nécessaire pour établir de façon plus automatique de réels liens de cohérence. Aussi l'approche générative ne nous semble pas la plus adaptée dans le cadre de la vérification de cohérence a posteriori qui nécessite et doit encourager des discussions entre les interlocuteurs quelle que soit l'approche choisie.

Les règles de recouvrement n'ont de sens que dans le cadre d'une **vérification de la cohérence a posteriori**. Chaque concepteur développe ses spécifications dans son formalisme préféré et on essaie ensuite d'établir des liens de cohérence avec les spécifications des autres concepteurs. Cette approche laisse une grande liberté aux concepteurs qui choisissent leur formalisme et leur méthode de développement. Les spécifications d'un système sont donc développées dans le modèle qui correspond le mieux aux besoins ou aux connaissances des concepteurs. On obtient ainsi un ensemble de spécifications qui peuvent représenter des vues complémentaires ou orthogonales.

La vérification a posteriori par méta-modélisation a pour avantages d'être indépendante des formalismes utilisés et de gérer des spécifications représentant des vues soit complémentaires soit orthogonales d'un système (bien qu'elle soit plus adaptée à la gestion de vues othogonales).

Mais cette liberté dans le développement des spécifications rend plus difficile la vérification de la cohérence si bien que l'on peut se demander ce que signifie "cohérence". Les concepteurs doivent donc être fortement impliqués dans la vérification afin d'obtenir des spécifications cohérentes.

8.3.3 Bilan

Les approches présentées dans ce chapitre ont des avantages très différents pour le couplage de spécifications semi-formels et formels. L'utilisation de l'une ou l'autre dépend des bénéfices attendus du couplage et du processus de développement utilisé.

L'approche générative est plutôt adaptée pour coupler des spécifications complémentaires écrites dans des notations proches comme celles du modèle objet et de Z. Dans notre contexte, elle permet valider des spécifications semi-formelles grâce à l'utilisation de spécifications formelles. Elle s'intègre donc dans un processus où le développement des spécifications formelles suit celui des spécifications semi-formelles.

La vérification a posteriori par méta-modélisation aide à obtenir des modèles divers mais relativement cohérents entre eux. Elle fonctionne plus particulièrement pour des spécifications orthogonales telles que les différents diagrammes d'UML, qui sont développées en parallèle.

8.4 Conclusion

Ce chapitre décrit une autre approche de couplage de notations, la vérification de la cohérence a posteriori par méta-modélisation. Après avoir décrit cette approche, nous étudions deux types de règles de cohérence, les règles de recouvrement et de traduction, entre le modèle objet et Z.

Ce travail nous permet de comparer ces différents types de règles afin d'en déduire leur cadre d'utilisation le plus approprié. Ainsi les règles de traduction correspondent plutôt à une approche générative de spécifications alors que celles de recouvrement sont adaptées à la vérification a posteriori. Partant de ce constat, nous comparons les approches générative et de vérification de cohérence a posteriori par méta-modélisation pour le couplage de notations semi-formelles et formelles. L'approche générative a pour principal avantage d'utiliser leur complémentarité pour tirer profit des bénéfices de chacune d'elles. La vérification a posteriori par méta-modélisation laisse une plus grande liberté aux concepteurs et permet de coupler plus aisément des modèles variés.

Chapitre 9

Conclusion

9.1 Contribution

Le travail que nous venons de présenter s'inscrit dans le cadre de l'utilisation conjointe de notations semi-formelles et formelles pour la spécification des SI. Il a pour objet d'enrichir mutuellement ces deux types de notations en définissant un cadre structurant et exploitable de couplage. Ainsi nous avons choisi une stratégie basée sur la traduction de spécifications semi-formelles en spécifications formelles que nous avons tenté de rendre facilement utilisable.

Vers une traduction formelle exploitable

Tout d'abord, nous avons adopté une approche générative de spécifications formelles qui apportent une aide à leur écriture et permet de les produire plus rapidement. Nos propositions de traduction des modèles objet et dynamique en squelettes de spécifications Z et Object-Z se distinguent des autres travaux existants par la multitude des concepts traités et, parfois, par leur sémantique. Nous avons en particulier précisé la sémantique des associations n-aires, de l'agrégation, de la composition et de l'héritage. Nous sommes consciente que comparativement aux spécifications formelles produites cognitivement par une personne, une traduction peut fournir des spécifications moins concises, moins claires, moins élégantes [FLP95, FKV91]. Nous avons essayé d'éviter de tomber dans ce travers pour que les squelettes de spécifications formelles obtenus soient effectivement exploitables. Ils doivent donc être lisibles, faciles à compléter et la correspondance entre le modèle objet initial et sa traduction doit permettre la traçabilité.

Pour faciliter la lisibilité et la traçabilité, nous avons attaché une importance particulière à ce que les squelettes de spécifications Z conservent le plus possible la structure du modèle objet. Cette rigueur dans la correspondance des représentations a permis de définir un guide méthodologique pour l'expression des contraintes annotant un modèle objet. D'une part, ce guide apporte une aide à l'écriture ou la modification des contraintes ; d'autre part, il décrit comment les squelettes de Z peuvent être complétés pour produire une spécification formelle achevée.

Toujours dans le but d'illustrer l'utilisabilité des spécifications formelles obtenues par traduction, nous avons montré deux types de raisonnement possibles grâce

à la précision sémantique apportée par la formalisation. Le premier concerne une réflexion informelle sur le sens des spécifications qui n'est possible que sur des cas simples comme ceux que nous avons développés.

Le deuxième exploite la sémantique grâce à un outil d'aide à la preuve conçu pour Z, Z-EVES. Partant du travail [Led98] qui montre comment utiliser cet outil pour valider des gardes d'opérations, nous avons tenté d'exploiter des connaissances du domaine pour proposer des "schémas" de preuves réutilisables. Ainsi nous avons mené une expérience avec Z-EVES qui s'est concentrée sur des preuves d'opérations de base pour des types de contraintes bien identifiées. Elle a mis en évidence qu'il existe effectivement des similitudes dans les preuves de contraintes de même type et qu'il est possible de faciliter le travail de vérification d'un modèle en identifiant des "schémas" de preuves. Pour l'instant, de tels schémas ont été définis seulement pour un nombre restreint de contraintes prises indépendamment.

Enfin la dernière activité de mise en valeur de notre travail de traduction a consisté à développer un outil de support, baptisé RoZ qui fait cohabiter les notations UML et Z. Il réalise automatiquement les tâches fastidieuses de génération de Z, des opérations de base et d'obligations de preuve. Son originalité réside dans l'idée que tout le travail de modélisation (modèle et annotations en notation formelles) doit être réalisé dans l'environnement de modélisation habituel. Cela est rendu possible grâce à l'exploitation du guide méthodologique pour l'expression des annotations.

Etude des approches de couplage

Outre la mise en valeur de la génération de spécifications formelles, nous avons tenté de mieux comprendre les approches de couplage de notations. Dans un premier temps, nous avons cherché à mieux cerner les implications du choix du langage formel cible dans notre stratégie de traduction. En particulier, nous voulions savoir si un langage formel orienté objet était obligatoire pour décrire le modèle objet. Nous avons proposé des règles de traduction des modèles objet et dynamique (donné en annexe) en Z et Object-Z. Même si elle est complexe, la traduction du modèle objet en Z est possible en faisant l'approximation de quelques simulations, pour l'encapsulation d'opérations par exemple. Il n'est donc pas nécessaire de disposer d'un langage objet pour représenter des concepts objet.

Mais si la traduction du modèle objet en Z a mis partiellement en évidence la complexité dûe à un certain éloignement sémantique entre les notations semi-formelles et formelles, celle du modèle dynamique a clairement montré l'impossibilité de représenter certains concepts dans une notation inadaptée. Néanmoins bien que l'utilisation d'un langage formel cible proche de la notation traduite (comme c'est le cas pour le modèle objet et Object-Z) facilite et simplifie la traduction, elle ne doit pas diminuer les possibilités d'exploitation des spécifications formelles produites. Ainsi nous avons préféré travailler avec Z pour valider certaines idées plutôt que d'être limités par le manque de sémantique et d'outils d'Object-Z.

Dans un deuxième temps, nous avons étudié une autre approche de couplage, la vérification de cohérence par méta-modélisation. Sa comparaison avec la traduction a montré les avantages et les limites potentiels de ces approches. Si la vérification de cohérence par méta-modélisation est adaptée pour vérifier la cohérence entre des

spécifications orthogonales, elle est limitée pour les spécifications complémentaires comme celles que nous avons étudiées et elle n'offre aucun support au développement ou à l'exploitation de spécifications formelles. Ce n'est donc pas l'approche adéquate pour le travail que nous désirions mener. Néanmoins la traduction n'a pas la souplesse de la vérification de cohérence par méta-modélisation et ne permet pas de traiter aisément de modèles très divers.

En résumé, nous pensons que ce travail constitue des avancées pour les points suivants :

- nos propositions de traduction sont complètes dans le sens où elles couvrent les principaux concepts d'un modèle objet [DLCP97, DLCP98, DLCP00b]. La formalisation de ces concepts a permis en particulier de préciser la sémantique des associations n-aires, de l'agrégation, de la composition et de l'héritage.

- nous avons présenté un guide méthodologique [Dup00] qui offre une aide à l'écriture des contraintes annotant un modèle objet.

- nous avons essayé de faciliter le travail de validation d'un modèle et de ses contraintes en développant la réutilisation des preuves grâce à des "schémas" de preuves.

- enfin l'environnement RoZ [DLCP00a] offre un support outil original à notre approche de couplage de notations semi-formelles et formelles.

9.2 Discussion et perspectives

Difficulté liée à la formalisation

L'approche que nous avons choisie dans ce travail se base sur une traduction totale de tous les concepts des modèles étudiés qui doit permettre d'en produire une formalisation exploitable. La diversité des concepts impose de prendre en compte des problèmes dont les solutions peuvent être contradictoires.

Il faut tout d'abord coller à la sémantique des modèles même si elle est imprécise, ambiguë voire incohérente. Par exemple, la sémantique de l'agrégation qui est définie de diverses manières a nécessité de préciser dans un premier temps le sens choisi. Puis il a fallu la formaliser dans toute sa complexité.

La représentation formelle d'un concept est aussi complexifiée par le fait que tous les autres concepts d'un modèle doivent être considérés pour proposer une traduction. Toutes les constructions d'un modèle, y compris celles qui sont peu employées, doivent être formalisées même si elles rendent le reste incohérent. C'est par exemple le cas de l'héritage multiple que nous n'avons pas pu formalisé en Z car sa prise en compte empêche la représentation de l'héritage suivant la sémantique que nous avions définie. Sa formalisation nécessite sans doute de proposer une alternative de traduction totalement différente de celle que nous avons choisie.

Bien qu'elle doive représenter la sémantique des concepts semi-formels de façon satisfaisante, toute proposition de traduction ne doit pas négliger l'utilisation

ultérieure des spécifications formelles produites. Il faut donc trouver une formalisation qui soit correcte mais aussi élégante et utilisable. Par exemple, dans ce travail, nous avons favorisé une traduction qui soit traçable et qui facilite l'expression des contraintes annotant un modèle objet. Mais elle rend plus complexe l'expression des opérations à cause du mécanisme de promotion.

Enfin l'usage de la formalisation ne doit se limiter à un seul contexte. Idéalement, une formalisation devrait permettre à la fois une expression facile des contraintes, des opérations, des preuves ou de la génération de code...

La formalisation est donc une activité contraignante dont certaines des difficultés citées ci-dessus peuvent remettre en cause la nécessité. Toutefois, nous pensons que même en assouplissant ces contraintes, une traduction en langage formel est bénéfique et que toute formalisation même partielle peut être exploitée au mieux.

Exploiter la précision sémantique

Certains pourraient être tentés de ne pas contraindre la sémantique des modèles pour en faciliter la formalisation. D'aucuns prétendent que l'une des forces des modèles semi-formels est leur absence de sémantique précise. Ainsi, il est possible d'interpréter ces notations en fonction d'une sémantique "maison" ou d'une méthode particulière. Par ailleurs, d'autres encouragent l'adoption d'une sémantique normalisée, qui permettrait la réalisation d'outils cohérents par les éditeurs de logiciels. Notre démarche ne tranche pas cette discussion et nous pensons qu'elle peut s'adapter aux deux points de vue. Des règles de traduction alternatives sont capables d'exprimer des sémantiques différentes, même s'il nous semble plus rationnel de ne faire le travail qu'une seule fois pour décrire une sémantique normalisée.

Les sémantiques alternatives peuvent permettre aux concepteurs de préciser leur vision d'un concept dans leur contexte. Lors de la formalisation, le concepteur n'aurait qu'à choisir la traduction correspondant au sens voulu. Ainsi l'interprétation d'un modèle dépendrait de choix sémantiques clairs et explicites qui amélioreraient la compréhension des modèles.

Cette approche est particluièrement bien adaptée dans le cadre de **la formalisation de patrons d'analyse et de conception** ([Coa92, Fow97, GHJV94]) qui définissent des solutions ré-utilisables à des problèmes similaires. Ces solutions devant être employées dans divers contextes, il semble intéressant de proposer aux concepteurs les différentes interprétations possibles de la solution qu'ils envisagent. Le choix d'un patron et de son sens seraient réellement dictés par la compréhension du concepteur de son problème et de sa solution. De plus, les solutions des patrons étant des modèles semi-formels de petite taille, elles semblent particulièrement bien adaptées pour une vérification exploitant leur formalisation. Elles seraient ainsi vérifiées une fois pour toutes et assureraient que chaque nouveau modèle les instanciant sont corrects. Une perspective intéressante de notre travail de traduction est donc de formaliser et de valider les solutions réutilisables que sont les patrons d'analyse et de conception. D'ailleurs un travail de traduction a déjà été utilisé pour formaliser une des sémantique du patron composite [MMLS00] et nous étudions actuellement dans notre équipe une autre approche de formalisation basée sur la méta-modélisation [Has00].

Vérifier des propriétés des modèles

Si nous désirons représenter la sémantique des modèles semi-formels de façon la plus complète possible, il semble obligatoire de s'interroger sur le choix du langage formel cible. Or il n'existe pas de langage formel "universel" capable d'exprimer tous les aspects des diagrammes semi-formels. Le choix d'un langage dépend donc de l'application souhaitée et des modèles à traduire. VDM et B sont des alternatives raisonnables à Z si on s'intéresse respectivement à leurs outils de prototypage et de raffinement. Des langages d'expression des aspects réactifs seraient certainement plus appropriés à l'analyse des modèles dynamiques. Aussi dans le cadre du concours concernant les outils d'exploitation des langages formels de la conférence FM'99, nous avons proposé une approche où les aspects statiques d'une modélisation étaient traduits en Z avec RoZ et validés avec Z-EVES et où les aspects dynamiques donnaient lieu à des spécifications en Lustre afin de les valider grâce à un outil de génération de tests [dBD00]. On peut donc envisager que les modèles semi-formels donnent lieu à **des traductions dans divers langages dont le choix dépendrait des propriétés à vérifier**. Dans ce cas, nous ferions beaucoup de formalisation très partielles qui suffiraient aux vérifications souhaitées.

Mais si on désire explorer divers modèles pour en vérifier la cohérence, il semble plus approprié de ne disposer que d'une seule formalisation. Ainsi, en traduisant des aspects des modèles objet et dynamique vers le même langage cible, on peut tenter d'établir la cohérence de la description formelle et valider des éléments de cohérence entre les deux modèles. C'est dans ce but que nous avons proposé des règles de traduction pour les modèles objet et dynamique. Ce travail a montré qu'il n'est pas toujours possible de représenter dans un seul langage formel la sémantique de plusieurs modèles. Aussi il peut être intéressant de ne disposer que d'une traduction partielle de ces modèles qui permette de vérifier les liens souhaités entre eux. Une suite naturelle de notre travail dont une première étude a été réalisée dans [Mau00] est donc d'**approfondir les liens de cohérence à établir et à vérifier entre les modèles statiques et dynamiques** en assouplissant les obligations de la formalisation.

Exploiter la connaissance du domaine des SI

Nous avons déjà développé un guide méthodologique qui permet de savoir où écrire une contrainte sur un modèle objet. Ce guide nous sert aussi à compléter les squelettes de schémas Z obtenus par traduction. Nous obtenons ainsi des spécifications formelles complètes pour lesquelles nous pouvons utiliser les outils de raisonnement de Z. Il est alors souhaitable de développer une connaissance de domaine qui facilite l'exploitation des spécifications formelles. Nous avons déjà évoqué cette possibilité pour proposer **des "schémas" de preuves pour la validation des contraintes**. Ce travail n'est qu'une première ébauche et doit être approfondi. Dans ce sens, notre outil RoZ pourrait proposer des contraintes pré-définies dont l'utilisateur aurait juste à préciser le domaine d'application. Par exemple, chaque classe proposerait une contrainte de clé pour laquelle l'utilisateur aurait à spécifier le ou les attributs clés. On aurait ainsi dans un même environnement des contraintes pré-définies et les "schémas" de preuves pour les valider.

Un modèle et ses contraintes étant validés grâce aux spécifications formelles, il serait intéressant de **générer automatiquement la base de données** correspondant à ces modèles. Nous avons évoqué dans les difficultés liées à la formalisation, le besoin d'obtenir des spécifications formelles utilisables dans plusieurs contextes. Or de même qu'il n'existe pas de langage formel "universel", il n'existe pas non plus de traduction qui favorise à la fois l'expression des contraintes, des opérations, les preuves etc. Ainsi si nos propositions s'orientent vers une expression aisée des contraintes, elles ne facilitent pas leur validation par les preuves. En effet d'après nos expériences de traduction, il semble que d'autres propositions qui rendent l'écriture des contraintes plus complexe, permettent de produire des spécifications Z pour lesquelles les preuves sont plus simples. Il faudrait en fait pouvoir disposer de plusieurs règles de traduction représentant la même sémantique du modèle semi-formel et favorisant des exploitations différentes des spécifications produites. Nous obtiendrions ainsi des spécifications formelles équivalentes (dont les liens sont à étudier) qui seraient utilisées suivant leur type d'exploitation le plus approprié. Par exemple, nos propositions actuelles de traduction pourraient servir à la lecture des spécifications Z et à l'expression des contraintes alors qu'une solution équivalente serait employée pour la génération de la base de données correspondant au modèle.

Une démarche d'avenir ?

Nous espérons que la présentation de ce travail a donné au lecteur quelques pistes de réflexion sur l'utilité du couplage des notations semi-formelles et formelles, et qu'elle contribuera modestement à faire du développement des SI une véritable discipline de l'ingénieur, précise et rigoureuse.

Bibliographie

[ABH+95] D.J. Andrews, H. Bruun, B.S. Hansen, P.G. Larsen, N. Plat et al. *Information Technology — Programming Languages, their environments and system software interfaces — Vienna Development Method-Specification Language Part 1 : Base language.* ISO, 1995.

[Abr77] J.-R. Abrial. Manuel du langage Z (Z/13). Rapport de recherche, Electricité de France, 1977.

[Abr96] J.R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[ABW+89] M. Atkinson, F. Bancilhon, D. De Witt, K. Dittrich, D. Maier et S. Zdonik. The Object-Oriented Database System Manifesto. Dans *1st Int. Conf. on Deductive and Object-Oriented Database -DOOD'89*, Kyoto, Japan, 1989.

[AF94] Y. Amghar et A. Flory. Contraintes d'intégrité dans les bases de données orientées objet. *Ingénierie des Systèmes d'Information*, 2(5), 1994.

[AG91] A.J. Alencar et J.A. Goguen. OOZE : An Object-Oriented Z Environment. Dans P. America, éditeur, *ECOOP'91*, pages 180–199, Genève, Suisse, Juillet 1991. Springer-Verlag.

[And95] P. André. *Méthodes formelles et à objet pour le développement du logiciel : études et propositions.* PhD thesis, Université de Rennes, Juillet 1995.

[AS97] K. Achatz et W. Schulte. A formal OO method inspired by Fusion and Object-Z. Dans J. P. Bowen, M. G. Hinchey et D. Till, éditeurs, *ZUM'97 : The Z Formal Specification Notation, 10th International Conference of Z Users*, volume 1212 of *Lecture Notes in Computer Science*, pages 92–111, Reading, UK, April 1997. Springer-Verlag.

[BC95] R. Bourdeau et B. Cheng. A Formal Semantics of Object Models. *IEEE Transactions on Software Engineering*, 21(10) :799–821, Octobre 1995.

[BDS96] H. Bowman, J. Derrick et M. Steen. Cross-viewpoint consistency in open distributed processing. *Software Engineering Journal*, 11(1) :44–57, Janvier 1996.

[BGV94] M. Bouzeghoub, G. Gardarin et P. Valduriez. *Objets : de C++ à Merise.* Eyrolles, 1994.

[BH95] J. Bowen et M. Hinchey. Seven more myths of formals methods. *IEEE Software*, pages 34–41, Juillet 1995.

[BJR98] G. Booch, I. Jacobson et J. Rumbaugh. *The Unified Modeling Language-User Guide*. Addison-Wesley, 1998.

[Bla93] M. Blaha. Aggregation of parts of parts of parts. *Journal of Object-Oriented Programminng*, 6(5) :14–20, 1993.

[BM95] T. Bouaziz et M. Meziane. Héritage de contraintes d'intégrité dans les bases de données orientées objet. Dans *Actes du 13^{me} congrès INFORSID*, pages 205–222, Grenoble, Juin 1995. AFCET-AFIA.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cumings, 1994.

[Bou97] M. Bouzeghoub. Unification des méthodes objets : fusion ou confusion. Dans *Actes du 15^{me} congrès INFORSID*, pages 53–65, Toulouse, Juin 1997. AFCET-AFIA.

[BP83] F. Bodart et Y. Pigneur. *Conception assictée des applications informatiques : étude d'opportunité et analyse conceptuelle*. Masson, Paris, 1983.

[BPSM98] T. Bray, J. Paoli et C.M. Sperberg-McQueen. Extensible Markup Language. Rapport de recherche, World Wide Web Consortium, 1998. Recommendation : http ://www.w3.org/TR/1998/REC-xml-19980210.

[Bru98] J. Brunet. An enhanced definition of composition and its use for abstraction. Dans C. Rolland et G. Grosz, éditeurs, *5th International Conference on Object Oriented Information System - OOIS'98*, Paris, France, 1998.

[BW96] R. Büssow et M. Weber. A steam-boiler control specification with Statecharts and Z. Dans J.-R. Abrial, E. Börger et H. Langmaack, éditeurs, *Formal Methods for Industrial Applications*, volume 1165 of *Lecture Notes in Computer Science*, pages 109–128. Springer-Verlag, 1996.

[CAB+96] D. Coleman, P. Arnold, B. Bodoff, C. Dollin, H. Gilchrist, F. Hayes et P. Jeremaes. *Fusion : La méthode orientée-objet de la 2^{e} génération*. Masson, 1996.

[CD94] S. Cook et J. Daniels. *Designing Object Systems - Object-Oriented Modelling with Syntropy*. Prentice-Hall, 1994.

[CGR95] D. Craigen, S. Gerhart et T. Ralston. Formal methods reality check : Industrial use. *IEEE Transactions on Software Engineering*, 21(2), Février 1995.

[Che76] P. Chen. The entity-relationship model : Toward a unifying view of data. *ACM Transactions on Database Systems*, pages 9–36, Mars 1976.

[Che99] E. Cheminot. *Formalisation de spécifications de logiciels : traitement d'annotations en langue naturelle contrôlée*. PhD thesis, Institut National Polytechnique de Grenoble, Décembre 1999.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud et J. Plaice. LUSTRE, a declarative language for programming synchronous systems. Dans *14th Symposium on Principles of Programming Languages (POPL 87), Munich*, pages 178–188. ACM, 1987.

[Civ93] F. Civello. Roles for composite objects in object-oriented analysis and design. Dans *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28 of *ACM SIGPLAN Notices*, pages 376–393. ACM Press, octobre 5–9 1993.

[Coa92] P. Coad. Object-Oriented Patterns. *Communications of the ACM*, 35(9), Septembre 1992.

[CR92] E. Cusack et G.H.B. Rafsanjani. ZEST. Dans S.Stepney, R.Barden et D.Cooper, éditeurs, *Object-Orientation in Z*, pages 113–126. Springer-Verlag, 1992.

[CW96] E. Clarke et J. Wing. Formal methods : state of the art and future directions. Rapport de recherche CMU-CS-9-178, Carnegie Mellon University, Aout 1996.

[Dür94] E. Dürr. VDM++ reference manual. Rapport de recherche afro/cg/lrm/v9, Cap Gemini - Ultrecht University, 1994. Esprit III - AFRODITE Project.

[DA82] C. Delobel et M. Adiba. *Base de données et systèmes relationnels*. Dunod Informatique, 1982.

[Dat95] C. Date. *An Introduction to Database Systems*. Addison-Wesley, sixth edition édition, 1995.

[dBD00] L. du Bousquet et S. Dupuy. A Multi-Formalism Approach for the validation of UML models. *Formal Aspects of Computing*, 2000. Résumé de deux pages, a paraitre.

[Del92] H. Delugach. Specifying Multiple-Viewed Software Requirements with Conceptual Graphs. *Journal of System Engineering*, 19 :207–224, 1992.

[DKRS91] R. Duke, P. King, G. Rose et G. Smith. The Object-Z Specification Langage : Version 1. Rapport de recherche 91–1, Department of Computer Science, University of Queensland, Australia, Software Verification Research Centre, April 1991.

[DL91] J. Dick et J. Loubersac. Integrating Structured and Formal Methods : a visual approach to VDM. Dans *3rd International Conference ESEC'91*, volume 550 of *Lecture Notes in Computer Science*, pages 37–59, Milan, Octobre 1991. Springer-Verlag.

[DLCP97] S. Dupuy, Y. Ledru et M. Chabre-Peccoud. Integrating OMT and Object-Z. Dans A. Evans et K. Lano, éditeurs, *Proceedings of BCS FACS/EROS ROOM Workshop, technical report GR/K67311-2, Dept. of Computing, Imperial College, 180 Queens Gate*, London, UK, June 1997.

[DLCP98] S. Dupuy, Y. Ledru et M. Chabre-Peccoud. Translating the OMT dynamic model into Object-Z. Dans *11th Int. Conf. of Z Users-ZUM'98*, volume 1493 of *Lecture Notes in Computer Science*, Berlin, Germany, 1998. Springer-Verlag.

[DLCP00a] S. Dupuy, Y. Ledru et M. Chabre-Peccoud. An Overview of RoZ : a Tool for Integrating UML and Z Specifications. Dans *12th Conference on Advanced information Systems Engineering-CAiSE'2000*, volume 1789 of *Lecture Notes in Computer Science*, Stockholm, Suède, 2000. Springer-Verlag.

[DLCP00b] S. Dupuy, Y. Ledru et M. Chabre-Peccoud. Vers une intégration utile de notations semi-formelles et formelles : une expérience en UML et Z. *L'Objet, numéro thématique Méthodes formelles pour les objets*, 6(1), 2000.

[DRS94] R. Duke, G. Rose et G. Smith. Object-Z : a Specification Langage Advocated for Description of Standards. Rapport de recherche 94–45, Departement of Computer Science, University of Queensland, Australia, Software Verification Research Centre, Decembre 1994.

[DS96] L. Dunkley et A. Smith. Improving Access of the Commercial Developer to Formal Methods : Integrating MERISE with Z. Dans A.Bryant et L.Semmens, éditeurs, *Method Integration Workshop*, Electronic Workshop in Computing, Leeds, Mars 1996. Springer-Verlag.

[Dup99] S. Dupuy. RoZ version 0.3 : an environment for the integration of UML and Z. http ://www-lsr.imag.fr/Les.Groupes/PFL/RoZ/index.html, 1999.

[Dup00] S. Dupuy. Vers une prise en compte des contraintes en UML grâce à Z. Dans M. Leonard, éditeur, *INFORSID'2000*, Lyon, France, 2000.

[ECM+99] A. Evans, S. Cook, S. Mellor, J. Warmer et A. Wills. Advanced Methods and Tools for a Precise UML - Panel. Dans *2nd International Conference on the Unified Modeling Language - "UML"'99*, volume 1723 of *Lecture Notes in Computer Science*, Fort Collins, USA, Octobre 1999. Springer.

[EEF+99] E. Ellmer, W. Emmerich, A. Finkelstein, D. Smolko et A. Zisman. Consistency Management of Distributed Documents using XML and Related Technologies. Rapport de recherche 99-94, UCL-CS Research Note, Department of Computer Science, University College London, 1999.

[EFLR98] A. Evans, R.B. France, K. Lano et B. Rumpe. Developing the UML as a Formal Modelling Notation. Dans J. Bezivin et P-.A. Muller, éditeurs, *UML'98 - Beyond the Notation*, Lecture Notes in Computer Science, Mulhouse, France, Juin 1998. Springer.

[FB97] R. France et J.-M. Bruel. Using Formal Techniques to Strengthen Informal Object-Oriented Modeling Techniques : The FuZE Experience. Rapport de recherche, Florida Atlantic University, Boca Raton, USA, Departement of Computer Science and Engineering, 1997.

[FBLP97] R. France, J.-M. Bruel et M. Larrondo-Petri. An Integrated Object-Oriented and Formal Modeling Environment. *Journal of Object Oriented Programming*, pages 25–34, Novembre/Décembre 1997.

[FBLPS97] R. France, JM. Bruel, M. Larrondo-Petrie et M. Shroff. Exploring the Semantics of UML type structures with Z. Dans H. Bowman et J. Derrick, éditeurs, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 247–260, Canterbury, UK, 1997. Chapman and Hall, London.

[FELR98] R. France, A. Evans, K. Lano et B. Rumpe. Developing UML as a formal modeling notation. *Computer Standards and Interfaces*, 19 :325–334, 1998.

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein et M. Goedicke. ViewPoints : A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1) :31–58, 1992.

[FKV91] M.D. Fraser, K. Kumar et V. Vaishnavi. Informal and Formal Requirements Specification Languages : Bridging the Gap. *IEEE Transactions on Software Engineering*, 17(5) :454–465, Mai 1991.

[FKV94] M.D. Fraser, K. Kumar et V. Vaishnavi. Strategies for incorporating formal specifications in software development. *Communications of the ACM*, 37(10) :74–84, octobre 1994.

[FL95] P. Facon et R. Laleau. Des spécifications informelles aux spécifications formelles : compilation ou interprétation ? Dans *Actes du 13^{me} congrès INFORSID*, pages 47–62, Grenoble, Juin 1995. AFCET-AFIA.

[FL96] P. Facon et R. Laleau. Des modèles d'objets aux spécifications formelles ensemblistes. *Ingénierie des systèmes d'information*, 4(2) :239–276, 1996.

[FLN96a] P. Facon, R. Laleau et H.P. Nguyen. Dérivation de spécifications formelles B à partir de spécifications semi-formelles de systèmes d'information. Dans H. Habrias, éditeur, *First conference on the B method*, Nantes, Novembre 1996.

[FLN96b] P. Facon, R. Laleau et H.P. Nguyen. Mapping Object Diagrams into B Specifications. Dans A.Bryant et L.Semmens, éditeurs, *Method Integration Workshop*, Electronic Workshop in Computing, Leeds, March 1996. Springer-Verlag.

[FLP95] R.B. France et M.M. Larrondo-Petrie. A Two-Dimensional View of Integrated Formal and Informal Specification Techniques. Dans *Proc. of the ZUM'95 : The Z formal Specification Notation*, pages 434–448, Limerick, Septembre 1995. Springer-Verlag.

[Fow97] M. Fowler. *Analysis Patterns : reusable object models*. Addison-Wesley, Reading MA, 1997.

[Fre97] J.C. Freire. *Ingénierie des Systèmes d'Informations : Une Approche de Multi-Modélisation et de Méta-Modélisation*. PhD thesis, Université Joseph Fourier, Grenoble, 1997.

[GHJV94] E. Gamma, R. Helm, R. Johnson et J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Compagny, 1994.

[GHW85] J. Guttag, J. Horning et J. Wing. Larch in five easy pieces. Rapport de recherche 5, DEC Systems Research Center, 1985.

[Gir95] J.P. Giraudin. Evolution de la modélisation des systèmes d'information. Dans EC2 et Cie, éditeur, *8^{me} Journées Internationales on Software Engineering and its applications*, Paris, Novembre 1995.

[GR98] M. Gogolla et M. Richters. On Combining Semi-Formal and Formal Object Specification Techniques. Dans F. Parisi-Presicce, éditeur, *12^{th} International Workshop on Abstract Data Types - WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 238–252. Springer-Verlag, 1998.

[Gri95] A. Griffiths. An Extended Semantic Foundation for Object-Z. Rapport de recherche 95-39, Software Verification Research Centre, Department of Computer Science, The University of Queensland, Australia, Octobre 1995.

[GS97] A. Galloway et B. Stoddart. Integrated formal methods. Dans *Actes du 15^{me} congrès INFORSID*, pages 549–576, Toulouse, Juin 1997. AFCET-AFIA.

[Hab95] H. Habrias. Les spécifications formelles pour les systèmes d'information : Quoi ? Pourquoi ? Comment ? *Ingénierie des systèmes d'information*, 3(2) :205–253, 1995.

[Hal90a] A. Hall. Seven myths of formals methods. *IEEE Software*, Septembre 1990.

[Hal90b] A. Hall. Using Z as Specification Calculus for Object-Oriented Systems. Dans G.Goos et J.Hartmanis, éditeurs, *VDM'90 : VDM and Z-Formal Methods in Software Development*, pages 290–318, Kiev, April 1990. Springer-Verlag.

[Hal94] A. Hall. Specifying and Interpreting Class Hierarchies in Z. Dans J.P.Bowen et J.A.Hall, éditeurs, *Proc. of the 8th Z User Meeting*, Workshop in Computing, pages 120–138, Cambridge, Juin 1994. Springer-Verlag.

[Ham94] J. Hammond. Producing Z Specifications From Object-Oriented Analysis. Dans J.P.Bowen et J.A.Hall, éditeurs, *Proc. of the 8th Z User Meeting*, Workshop in Computing, pages 120–138, Cambridge, Juin 1994. Springer-Verlag.

[Har87] D. Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, 1987.

[Has00] I. Hassine. Formalisation, Instantiation et composition de patterns. Rapport de stage de DEA MATIS, Grenoble, Septembre 2000.

[Hea] Headway Software. The RoZeLink 1.0. http ://www.calgary.shaw.wave.ca/headway/index.htm.

[HG97] H. Habrias et B. Griech. Formal Specification of Dynamic Constraints with the B method. Dans Michael G. Hinchey et Shaoying Liu, éditeurs, *Proc. of the 1st IEEE Int. Conf. on Formal Engineering Methods*, Hiroshima, Japan, 1997. IEEE Computer Society Press.

[HRH96] N. Hadj-Rabia et H. Habrias. Formal specification from NIAM model : a bottom-up approach. Dans *Proc. of the 11th Int. Symposium on Computer and Information Science*, Antalya, Turquie, 1996.

[HRW93] J. Hagelstein, D. Roelants et P. Wodon. Formal requirements made practical. Dans Ian Sommerville et Manfred Paul, éditeurs, *Proceedings of the Fourth European Software Engineering Conference*, volume 717 of *Lecture Notes in Computer Science*, pages 127–144. Springer-Verlag, septembre 1993.

[HS99] M. Heisel et J. Souquières. A method for requirements elicitation and formal specification. Dans J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau et E. Métais, éditeurs, *Proceedings of the 18^{t}h International Conference on Conceptual Modeling - ER'99*, pages 309–324, Paris, France, Novembre 1999. Springer-Verlag.

[IBM] IBM. OCL Parser-version 0.3. http ://www-4.ibm.com/software/ad/standards/ocl.html.

[IEE90] IEEE. Standard glossary of software engineering terminology. Rapport de recherche 610.12-1990, IEEE, 1990.

[IFA] IFAD. The Rose-VDM++ Link. http ://www.ifad.dk/Products/rose-vdmpp.htm.

[Jac83] M.A. Jackson. *System Development.* Prentice-Hall, 1983.

[Jac99a] D. Jackson. Alloy : A lightweight object modelling notation. http ://sdg.lcs.mit.edu/publications.html, Juillet 1999.

[Jac99b] M. Jackson. Specializing in software engineering. *IEEE Software*, pages 119–121, Novembre/Décembre 1999.

[JCO92] I. Jacobson, M. Christerson et G. Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley, 1992.

[Joh96] W. Johnston. A Type Checker for Object-Z. Rapport de recherche 96–24, Departement of Computer Science, University of Queensland, Australia, Software Verification Research Centre, Septembre 1996.

[KC99] S.-K. Kim et D. Carrington. Formalizing the UML Class Diagram Using Object-Z. Dans *2nd International Conference on the Unified Modeling Language - "UML"'99*, volume 1723 of *Lecture Notes in Computer Science*, Fort Collins, USA, Octobre 1999. Springer.

[Ken97] Stuart Kent. Constraint diagrams : Visualizing assertions in object-oriented models. Dans *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 327–341, New York, octobre 5–9 1997. ACM Press.

[KGR99] S. Kent, S. Gaito et N. Ross. A meta-model semantics for structural constraints in UML. Dans H. Kilov, B. Rumpe et I. Simmonds, éditeurs, *Behavioral specifications for businesses and systems*, chapitre 9, pages 123–141. Kluwer Academic Publishers, Norwell, MA, September 1999.

[KH99] S. Kent et J. Howse. Mixing Visual and Textual Constraint Languages. Dans *2nd International Conference on the Unified Modeling Language - "UML"'99*, volume 1723 of *Lecture Notes in Computer Science*, Fort Collins, USA, Octobre 1999. Springer.

[Kob99] C. Kobryn. Uml 2001 : a standardization odyssey. *Communications of the ACM*, 42(10) :29–37, Octobre 1999.

[KR94] H. Kilov et J. Ross. *Information modeling - An Object-Oriented Approach.* Prentice Hall, 1994.

[Lan95] K. Lano. *Formal object-oriented development.* Springer, 1995.

[LB98] K. Lano et J. Bicarregui. Semantics and Transformations for UML Models. Dans J. Bezivin et P-.A. Muller, éditeurs, *UML'98 - Beyond the Notation*, Lecture Notes in Computer Science, Mulhouse, France, Juin 1998. Springer.

[Led98] Y. Ledru. Identifying pre-conditions with the Z/EVES theorem prover. Dans *Proc. of the 13th Int. Conf. on Automated Software Engineering.* IEEE, 1998.

[LFB96] P. Larsen, J. Fitzgerald et T. Brookes. Applying formal specification in industry. *IEEE Software*, pages 48–56, Mai 1996.

[LG96] K. Lano et S. Goldsack. Intregrated Formal and Object-Oriented Methods : The VDM++ Approach. Dans A.Bryant et L.Semmens, éditeurs, *Method Integration Workshop*, Electronic Workshop in Computing, Leeds, Mars 1996. Springer-Verlag.

[LH94] K. Lano et H. Haughton. The Z++ manual. Rapport de recherche, Imperial College, Londres, 1994. ftp ://theory.doc.ic.ac.uk/theory/papers/Lano/z++.ps.

[LHR95] R. Laleau et N. Hadj-Rabia. Génération automatique de spécifications VDM à partir d'un schéma conceptuel de données. Dans *Actes du 13^{me} congrès INFORSID*, pages 63–78, Grenoble, Juin 1995. AFCET-AFIA.

[LHW96] K. Lano, H. Houghton et P. Wheeler. Integrating Formal and Structured Methods in Object-Oriented System Development. Dans *Formal Methods and Object technology*, chapitre 7. Springer, 1996.

[Mar94] J. Marciniak, éditeur. *Encyclopedia of Software Engineering.* Wiley-Interscience publication, 1994.

[Mau00] O. Maury. Vérification de la cohérence des diagrammes de classes et d'états-transitions dans UML. Rapport de stage de DEA ISC, Grenoble, Juin 2000.

[MC92] S.L. Meira et A.L.C. Cavalcanti. The MooZ Specification Langage - version0.4. Rapport de recherche ES/1.92, Universidade Federal de Pernambuco, 1992.

[MC94] A. Moreira et R. Clark. Complex objects : Aggregates. Rapport de recherche CM-123, Department of Computing Science and Mathematics, University of Stirling, Scotland, Août 1994.

[MM88] D. Marca et C. McGowan. *SADT : Structured Analysis and Design Technique.* McGraw-Hill, New-York, 1988.

[MM97] V. Misić et S. Moser. Formal Approach to Metamodeling : Object-Oriented Perspective. Dans J. Bosch et S. Mitchell, éditeurs, *ECOOP'97*, Lecture Notes in Computer Science, Finlande, Juin 1997. Springer-Verlag.

[MMLS00] R. Marcano, E. Meyer, N. Levy et J. Souquières. Utilisation de patterns dans la construction de spécifications en UML et B. Dans Y. Ledru, éditeur, *Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2000*, pages 66–80, Grenoble, France, 2000.

[MS99] E. Meyer et J. Souquières. A systematic approach to transform OMT diagrams to a B specification. Dans J. Wing, J. Woodcock et J. Davies, éditeurs, *World Congress on Formal Methods in the Development of Computing Systems - FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 875–896, Toulouse, France, 1999. Springer-Verlag.

[NECH92] D. Nanci, B. Espinasse, B. Cohen et H. Heckenroth. *Ingénierie des systèmes d'information avec Merise, Vers une deuxième génération.* Sibex, 1992.

[Ngu98] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles.* PhD thesis, Conservatoire National des Arts et Métiers, 1998.

[NR69] P. Naur et B. Randel. Software Engineering : A Report on a Conference sponsored by NATO science comitee. Rapport de recherche, NATO, 1969.

[NR94] N. Nagui-Raïss. A Formal Software Specification Tool Using The Entity-Relationship Model. Dans *13^{th} International Conference on the Entity/Relationship approach*, volume 881 of *Lecture Notes in Computer Science*, Manchester, Décembre 1994. Springer-Verlag.

[Obj95] Groupe Technologie Objet. La technologie à objets - Domaines et utilisations. *Ingénierie des Systèmes d'Information*, 3(6) :739–776, 1995.

[Ou98] Y. Ou. On using UML class diagram for object-oriented database design-Specification of integrity constraints. Dans J. Bezivin et P-.A. Muller, éditeurs, *UML'98 - Beyond the Notation*, Lecture Notes in Computer Science, Mulhouse, France, Juin 1998. Springer.

[Ous97] C. Oussalah et alii. *Ingénierie objet - Concepts et techniques.* InterEditions, 1997.

[PdRHP94] C. Petersohn, W.-P. de Roever, C. Huizing et J. Peleska. Formal Semantics for Ward and Mellors's Transformation. Dans D. Till, éditeur, *6^{th} Refinement Workshop*, Workshops in Computing, pages 14–41. Springer-Verlag, 1994.

[Pet77] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3) :223–252, 1977.

[Pha90] N. H. Le Pham. *Optimisation globale des contrôles d'intégrité dans une base de données.* PhD thesis, Université de Genève, 1990.

[PKP91] A. Plat, J. Van Katwijk et K. Pronk. A Case for Structured Analysis/Formal Design. Dans *Fourth International Conference VDM'91*, volume 551 of *Lecture Notes in Computer Science*, pages 81–105, Noordwijkerhout, Hollande, Octobre 1991. Springer-Verlag.

[Pre94] Roger Pressman. *Software Engineering - A Practitioner Approach.* Mac Graw-Hill Book Company Europe, 1994. third edition.

[PST96] B. Potter, J. Sinclair et D. Till. *An introduction to formal specification and Z.* Prentice-Hall International, 1996.

[PWM93] F. Polack, M. Whiston et K. Mander. The SAZ Project : Integrating SSADM and Z. Dans *International Symposium Formal Methods Europe*, volume 670 of *Lecture Notes in Computer Science*, Odense, Danemark, Avril 1993. Springer.

[Ran90] G.P. Randell. Translating data flow diagrams into Z (and vice versa). Rapport de recherche 90019, Royal Signals and Radar Establishement, Malvern, Octobre 1990.

[Rat96a] Rational Software Corporation. *Rational Rose - Extensibility Guide*, 1996.

[Rat96b] Rational Software Corporation. *Rational Rose - Using Rational Rose 4.0*, 1996.

[RBP+91] J. Rumbaugh, M. Blaha, M. Premerlani, F. Eddy et W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall International, 1991.

[RFB88] C. Rolland, O. Foucault et F. Benci. *Conception des systèmes d'information - La méthode REMORA.* Eyrolles, Paris, 1988.

[Rum95] J. Rumbaugh. What is a method ? *Journal of Object Oriented Programming*, pages 10–16 and 26, 1995.

[Saa97] M. Saaltink. The Z/EVES system. Dans J. Bowen, M. Hinchey et D. Till, éditeurs, *Proc. 10th Int. Conf. on the Z Formal Method (ZUM)*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–88, Reading, UK, april 1997. Springer-Verlag, Berlin.

[Sai96] H. Saiedian. An Invitation to Formal Methods. *IEEE Computer*, 24(4) :16–30, Avril 1996.

[SC95] G. Spanoudakis et P. Constantopoulos. Integrating specification : a similarity reasoning approach. *Automated Software Engineering Journal*, 2(4) :311–342, 1995.

[Sek98] E. Sekerinski. Graphical design of reactive systems. Dans *2nd International Conference on B Conference, B'98 : Recent Advances in the Development of the B method*, volume 1393 of *Lecture Notes in Computer Science*, Montpellier, France, Avril 1998. Springer-Verlag.

[SF97] M. Shroff et R.B. France. Towards a formalization of UML class structures in Z. Dans *21st International Computer Software and Applications Conference - COMPSAC'97*, Washington, Etats-Unis, 1997. IEEE Computer Society.

[SFLP98] M. Saksena, R. France et M. Larrondo-Petrie. A characterization of aggregation. Dans C. Rolland et G. Grosz, éditeurs, *5th International Conference on Object Oriented Information System - OOIS'98*, Paris, France, 1998.

[SG99] A. Simons et I. Graham. 30 things that go wrong in Object Modelling with UML1.3. Dans *Behavioral specifications of buisinesses and systems*, chapitre 16, pages 221–242. Kluwer Academic Publishers, 1999.

[SGO91] L. Logrippo S. Gallouzi et A. Obaid. Le LOTOS : théorie, outils, applications. Dans O. Rafiq, éditeur, *CFIP'91*, pages 385–404, 1991.

[Som92] I. Sommerville. *Software Engineering.* Addison Wesley, 1992. 4th edition.

[Spi92] J.M. Spivey. *The Z notation.* Prentice-Hall International, 1992.

[SS77] J.M. Smith et D.C.P. Smith. Database Abstractions : Aggregation and Generalization. *ACM TODS*, 2(2), Juin 1977.

[SY98] J. Suzuki et Y. Yamamoto. Making UML models exchangeable over the Internet with XML : the UXF Approach . Dans J. Bezivin et P-.A. Muller, éditeurs, *UML'98 - Beyond the Notation*, Lecture Notes in Computer Science, Mulhouse, France, Juin 1998. Springer.

[Tha99] T. Thanitsukkarn. *Multiperspective Development Environment for Configurable Distributed Applications.* PhD thesis, Imperial College of Science, Technology and Medecine, University of London, Février 1999.

[TRR83] H. Tardieu, A. Rochfeld et C. Rolland. *La méthode Merise : principes et outils.* Editions d'Organisation, Paris, 1983.

[UML97] UML Partners. Unified Modeling Language version 1.1. Rapport de recherche ad/97-08-11, OMG document, 1997. http ://www.rational.com.

[UML99] UML Revision Task Force. OMG Unified Modeling Language Specification version 1.3. Rapport de recherche ad/99-06-08, Object Management Group, 1999. http ://www.omg.org/uml.

[Voa99] J. Voas. Software quality's eight greatest myths. *IEEE Software*, pages 118–120, Septembre/Octobre 1999.

[WDH97] R. Wieringa, E. Dubois et S. Huyts. Integrating semi-formal and formal requirements. Dans A. Olivé et J. Pastor, éditeurs, *9th International Conference on Advanced Information Systems Engineering - CAiSE'97*, volume 1250 of *Lecture Notes in Computer Science*, Barcelone, Espagne, Juin 1997. Springer-Verlag.

[Win90] J. Wing. A specifier's introduction to formal methods. *IEEE Computer*, pages 8–24, Septembre 1990.

[WK98] J. Warmer et A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1998.

[WRC97] E. Wang, H. Richter et B. Cheng. Formalizing and Integrating the Dynamic Model within OMT. Dans *19th International Conference on Software Engineering*, Boston, USA, Mai 1997. IEEE CS Press.

[YC79] E. Yourdon et L.L. Constantine. *Structured Design*. Prentice-Hall, 1979.

[Z95] Z Standards Panel. Z Notation version 1.2. Rapport de recherche, Septembre 1995. http ://www.comlab.ox.ac.uk/oucl/groups/zstandards/.

[Zha97] X. Zhang. *A Rigorous Approach to Comparison of Representational Properties of Object-Oriented Analysis and Design Methods*. PhD thesis, Queen's University, Kingston, Canada, 1997.

[ZJ93] P. Zave et M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4) :379–411, 1993.

[ZJ97] P. Zave et M. Jackson. Four dark corners for requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1) :1–30, 1997.

ANNEXE

Annexe A

Z : la boîte à outils mathématiques

Les schémas et la définition d'ensembles nécessitent d'écrire des expressions sur les nombres, les ensembles, les relations et les prédicats. Une partie importante de Z est donc constituée d'une boîte à outils mathématiques qui permet de décrire de façon simple de nombreuses structures. On dispose en particulier des notations mathématiques classiques pour les expressions sur les nombres(+,-,=,$\leqslant$ etc...). Pour les autres types d'expressions, les opérations vont être décrites plus précisément. Pour plus de détails sur les différentes notations Z, vous pouvez consulter [Spi92].

A.1 Ensembles

Z permet de manipuler les opérations de base de l'algèbre ensembliste. On utilisera principalement :

Symbole	Nom	Signification
$\mathbb{P}$,$\mathbb{F}$	ensembles des parties (finis)	Si S est un ensemble, $\mathbb{P}$ S est l'ensemble de tous les sous-ensembles de S. $\mathbb{F}$ S est l'ensemble de tous les sous-ensembles finis de S.
=, $\neq$	égalité, inégalité	
$\in$,$\notin$	appartenance et non appartenance	L'élément e appartient à l'ensemble E(e $\in$ E)
$\varnothing$	ensemble vide	ensemble sans élément
$\subseteq$,$\subset$	sous-ensemble, sous-ensemble propre	Un ensemble S est sous-ensemble de l'ensemble T (S $\subseteq$ T) si tout élément de S est aussi élément de T. S est propre (S $\subset$ T) si en plus S est différent de T.
$\cup$	union de deux ensembles	Les éléments de S $\cup$ T appartiennent soit à S, soit à T, soit aux deux.

Symbole	Nom	Signification
$\cap$	intersection de deux ensembles	Les éléments de S $\cap$ T appartiennent à la fois à S et à T.
$\setminus$	différence de deux ensembles	Les éléments de S $\setminus$ T sont des éléments de S mais pas de T.
#	cardinalité d'un ensemble fini	nombre d'éléments de cet ensemble

A.2 Relations et Fonctions

Symbole	Nom	Signification
$\times$	produit cartésien	Si E1,...En sont des ensembles, alors E1$\times$...$\times$En est l'ensemble de tous les n-uplets de la forme (x1,...,xn) où xi$\in$ Ei pour tout i, $1 \leqslant i \leqslant n$.
$\leftrightarrow$	relation binaire	Si X et Y sont des ensembles, alors X$\leftrightarrow$ Y est l'ensemble des relations binaires entre X et Y. Chacune de ces relations est un sous-ensemble de X$\times$ Y.
$\mapsto$	correspondance binaire	Elle représente une paire ordonnée i.e. un couple d'une relation binaire.
dom, ran	domaine et codomaine d'une relation	Si R est une relation binaire entre X et Y, alors le domaine de R est l'ensemble des éléments de X qui participent à R. Le codomaine de R est l'ensemble des éléments de Y pour lesquels il existe au moins un élément de X relié par R.

Les relations qui associent à chaque élément de leur domaine au plus une image sont appelées des fonctions. Soit f une relation entre les ensembles A et B (A $\leftrightarrow$ B) :

- f est une fonction si tout élément de A a au plus une image par la relation.
- f est une fonction totale si tout élément de A a une et une seule image.
- f est une fonction partielle si son domaine est inclus dans A. (Les fonctions totales osnt donc un sous-ensemble des fonctions partielles.)
- f est injective si tout élément de B est au plus l'image d'un élément de A par la relation c'est-à-dire si la relation inverse est une fonction.
- Une fonction est dite finie si son domaine est fini.

– f est surjective si tout élément de B est au moins l'image d'un élément de A par la relation.

– une bijection est une fonction totale, injective et surjective.

Le tableau ci-dessous résume les propriétés des différentes fonctions :

Fonctions		Contraintes		
Nom	Symbole	Dom f	UnUn	Ran f
Fonction Partielle	$\nrightarrow$	$\subseteq A$		$\subseteq B$
Fonction Totale	$\rightarrow$	$= A$		$\subseteq B$
Injection Totale	$\rightarrowtail$	$= A$	Oui	$\subseteq B$
Injection Partielle	$\nrightarrowtail$	$\subseteq A$	Oui	$\subseteq B$
Surjection Totale	$\twoheadrightarrow$	$= A$		$= B$
Surjection Partielle	$\nrightarrow\!\!\!\!\twoheadrightarrow$	$\subseteq A$		$= B$
Bijection	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$= A$	Oui	$= B$
Fonction Partielle Finie	$\nrightarrow\!\!\!\!\!\!\rightarrow$	$\subseteq A$		$\subseteq B$
Injection Partielle Finie	$\nrightarrowtail\!\!\!\!\!\!\rightarrow$	$\subseteq A$	Oui	$\subseteq B$

A.3 Prédicats

La logique du premier ordre est basée sur les connecteurs suivants qui permettent de contruire des prédicats complexes :

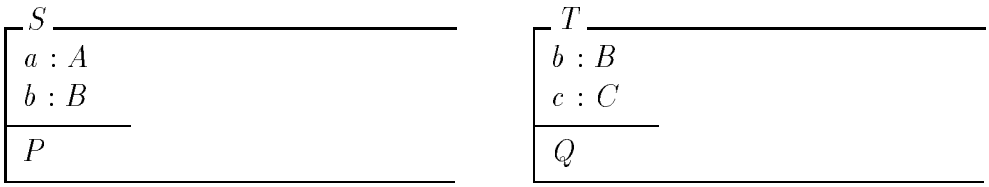
Symbole	Nom	Signification
true, false	vrai, faux	
$\neg$	négation	$\neg$ P signifie que P n'est pas vrai.
$\vee$	disjonction	P1$\vee$P2 : soit P1, soit P2, soit tous les deux sont vrais.
$\wedge$	conjonction	P1$\wedge$P2 : P1 et P2 sont vrais
$\Rightarrow$	implication	Soit P1 est faux et P2 est quelconque, soit P1 et P2 sont vrais.
$\Leftrightarrow$	équivalence	P1 et P2 sont tous les deux vrais ou ils sont tous les deux faux.
$\exists$	quantificateur existentiel	$\exists$S$\bullet$P est vrai s'il existe au moins une façon d'attribuer des valeurs aux variables introduites par S telle que à la fois la propriété S et le prédicat P soient vrais.
$\exists_1$	quantificateur d'existence unique	Il n'existe qu'une seule façon d'attribuer des valeurs aux variables de S.
$\forall$	quantificateur universel	$\forall$S$\bullet$P est vrai, quelles que soient les valeurs des variables de S, P est vrai.

A.4 Combinaison de schémas

Les schémas Z peuvent être combinés de différentes manières. Nous introsuisons ici les opérateurs sur les schémas : la conjonction, la disjonction, la négation, la quantification et la composition.

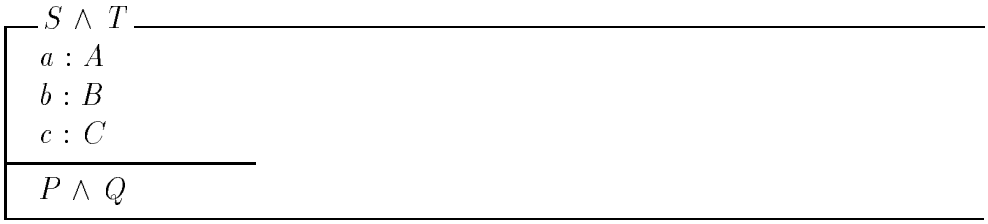
A.4.1 Conjonction de schémas

Soient S et T deux schémas introduits par :



où P et Q sont des prédicats sur les variables des schémas.

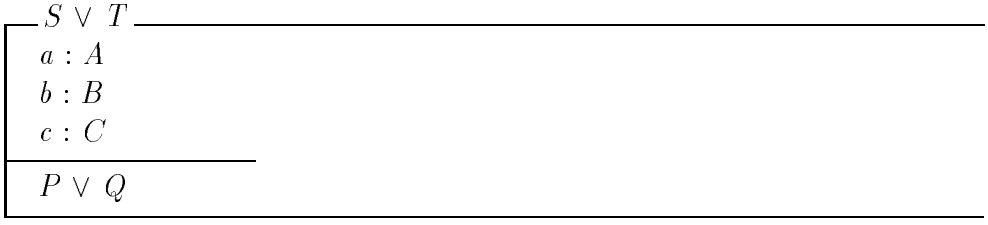
S $\wedge$ T dénote la conjonction des schémas S et T : un nouveau schéma est formé en fusionnant les parties déclaratives de S et T et en faisant la conjonction de leur prédicat :



Si la même variable est déclarée dans les deux schémas, comme b dans notre exemple, leurs types doivent être correspondre ou S $\wedge$ T est indéfini.

A.4.2 Disjonction de schémas

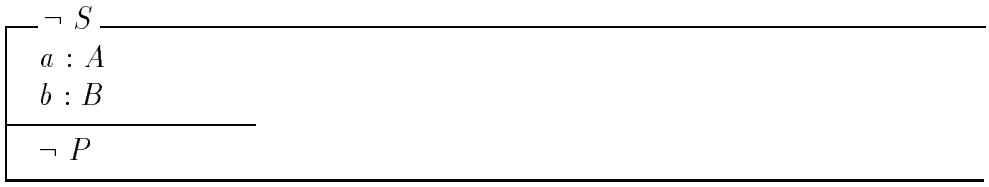
Si on considère les schémas S et T introduits dans la section précédente, la disjonction S $\vee$ T correspond au schéma suivant :



Comme pour la conjonction, les déclarations sont jointes. Mais cette fois les prédicats sont disjoints.

A.4.3 Négation de schéma

La négation d'un schéma ¬ S est obtenue en effectuant la négation des prédicats de S :



A.4.4 Quantification de schéma

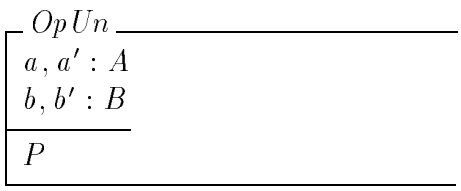
Certains composants d'un schéma peuvent être quantifiés alors que la déclarartion des autres ne change pas. Par exemple, si on considère toujours le schéma S, ∀ b : B • S correspond au schéma :

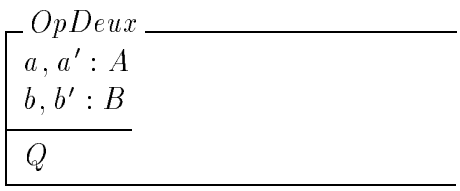
Les composants quantifiés sont supprimés de la partie déclaration, mais ils sont quantifiés dans les prédicats.

A.4.5 Composition de schémas

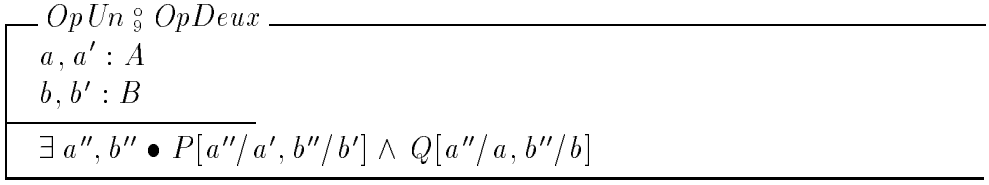
La composition de schémas sert à combiner des schémas d'opérations qui se réfèrent au même état. Si OpUn et OpDeux sont des schémas d'opérations, chacun incluant des copies primées et non primées de schéma d'état S, alors leur composition OpUn ⨾ OpDeux décrit un changement dans l'état qui correspond à l'opération OpUn suivie par OpDeux. Il existe donc un état intérédiaire S" entre les opérations OpUn et OpDeux.

Soient OpUn et OpDeux les schémas d'opération suivants sur S :





La composition OpUn ⨾ OpDeux est équivalente au schéma suivant dans lequel l'état intermédiaire a" et b" est caché :



Annexe B

Modèle dynamique

Cette annexe présente nos propositions de traduction concernant un modèle dynamique basé sur les diagrammes d'états-transitions de Harel [Har87]. Elle décrit les concepts de ce modèle, puis les travaux existants de traduction en langages formels orientés modèles ou objets. Enfin nous développons des règles de traduction pour Z et Object-Z.

B.1 Description du modèle dynamique

Le modèle dynamique que nous avons choisi d'étudier est basé sur les diagrammes d'états d'UML. Ces diagrammes spécifient le comportement des objets d'une classe par la séquence des états par lesquels passent ces objets au cours de leur cycle de vie en réponse à des événements. Comme le modèle objet, ils peuvent être structurés par généralisation ou composition. Nous allons en présenter les concepts en commençant par ceux de base (état, événement, condition etc), puis en présentant des versions plus évoluées telle que la hiérarchisation et la concurrence d'agrégat.

B.1.1 Etat et transition

Un état est une situation pendant la vie d'un objet durant laquelle l'objet satisfait une condition, attend un stimulus ou effectue une action. Une fois dans un état, il peut se produire des stimuli qui déclenchent le passage vers un autre état. On appelle **transition** cette relation indiquant qu'un objet change d'état et **événement** l'occurrence d'un stimulus qui déclenche une transition. Selon la version 1.3 d'UML [UML99], un événement peut être de quatre types :

1. la réception d'un appel d'opération ;

2. une durée écoulée après un événement donné notée par le mot-clé *après* ;

3. un événement de changement noté par *quand* qui se produit quand une expression booléenne explicite devient vraie en résultat d'une modification de valeurs d'un ou plusieurs attributs ou associations ;

4. la réception d'un signal, un signal étant la spécification d'un stimulus asynchrone communiqué entre les instances.

Par exemple (Fig. B.1), une soumission à une conférence passe de l'ètat "AEvaluer" à l'état "Acceptee" quand elle reçoit le signal "ReponsePositive".

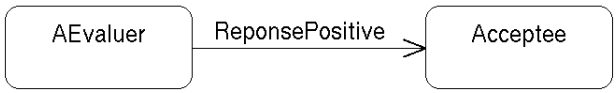


FIG. B.1: Transition entre les états "AEvaluer" et "Acceptee"

Une transition qui ne possède pas d'événement associé est appelée transition automatique.

B.1.2 Eléments complémentaires

Condition Une transition peut être accompagnée d'une condition de déclenchement (garde). **Cette condition** est une expression booléenne qui est évaluée pour franchir la transition. Une transition est franchie seulement si la condition est satisfaite.

Un exemple de transition guardée (Fig. B.2) est le cas où une soumission passe de l'état "Soumise" à l'état "AEvaluer" si la date de réception de la soumission est antérieure à la date limite de soumission.

FIG. B.2: Transition guardée entre les états "Soumise" et "AEvaluer"

Action et activité **Une action** est une opération atomique qui peut être effectuée à l'entrée (mot-clé *entrée*), à la sortie (mot-clé *sortie*) d'un état ou lors du franchissement d'une transition. L'utilisation d'actions dans un état est équivalente à l'écriture de ces actions sur chaque transition entrant ou sortant d'un état. Cela évite simplement de les ajouter sur toutes les transitions entrantes ou sortantes.

Ainsi lorsqu'une soumission passe de l'état "AEvaluer" à "Acceptee" lors de la réception de l'événement "ReponsePositive", l'opération "ModifierStatut", qui modifie le statut final d'une soumission, est appelée avec le paramètre "accepte" (Fig. B.3).

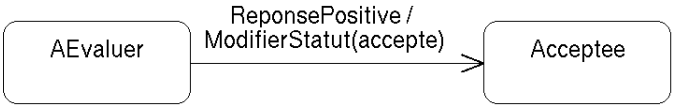


FIG. B.3: Transition entre les états "AEvaluer" et "Acceptee"

Une activité est une opération non-atomique (séquence d'actions) qui dure jusqu'à ce qu'elle soit terminée ou interrompue par un événement. Le lancement de l'activité d'un objet dans un certain état est représenté par le mot-clé *faire*.

B.1.3 Echange d'événements

Les objets d'un système interagissent en échangeant des événements lors du franchissement des transitions. Graphiquement, l'envoi d'un événement est représenté par une flèche pointillée qui pointe vers la classe de l'objet récepteur et qui est étiquetée par le nom de l'événement et ses paramètres éventuels.

Par exemple, lors que l'état d'une soumission passe de "AEvaluer" à "Acceptee", l'événement "DonnerReponse" est envoyé aux auteurs.

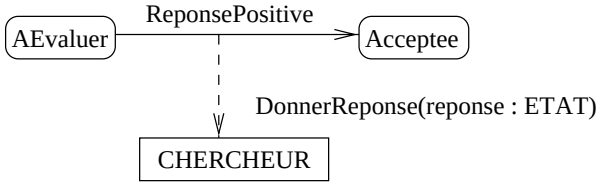


FIG. B.4: Envoi d'événement à la classe "CHERCHEUR"

B.1.4 Diagrammes d'états hiérarchiques

La hiérarchie d'états correspond au concept généralisation/spécialisation dans le modèle objet. Les sous-états héritent des transitions de leurs super-états de la même façon que les sous-classes héritent des propriétés de leur super-classe. Chaque transition ou action appliquée à un état est donc appliquée à ses sous-états. Il s'agit donc d'une décomposition OU des états dont les sous-états sont exclusifs entre eux (on ne peut donc être que dans l'un des sous-états).

Dans le cas du modèle dynamique d'une soumission, l'état "AEvaluer" peut être spécialisé (Fig. B.5). Quand une soumission doit être évaluée, elle est initialement en attente d'affectation de ses rapporteurs (état "EnAttente"), puis en relecture (état "Affectee") et enfin évaluée partiellement (état "EvalueePartiellement") quand les rapporteurs ont rendu leur avis, mais que le comité de programme n'a pas encore pris de décision. Quand la réponse du comité de programme est connue, la soumission passe alors du sous-état "EvalueePartiellement" de l'état "AEvaluer" à "Acceptee" ou "Refusee".

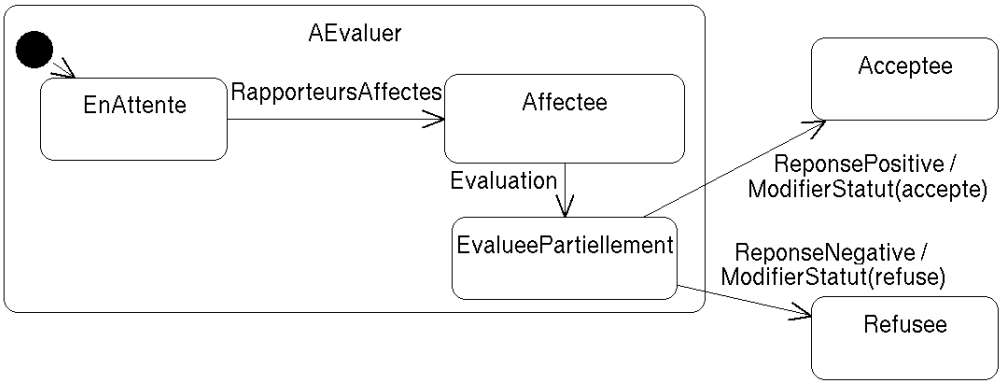


FIG. B.5: Spécialisation de l'état "AEvaluer"

B.1.5 Concurrence d'agrégat

La concurrence d'agrégat est définie dans OMT [RBP+91] pour aider à décrire la dynamique d'un ensemble de classes liées par des agrégations. L'état de l'agrégat correspond aux états combinés des diagrammes d'états de ses composants.

Par exemple, une session est composée de plusieurs présentations (Fig. B.6).

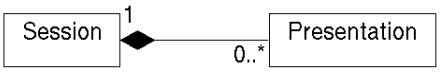


FIG. B.6: Composition entre "Session" et "Presentation"

Le comportement d'une session dépend de celui de ses présentations : pour planifier une session, il faut connaître toutes les présentations qui vont la constituer ; elle peut être annulée si toutes les présentations sont annulées ; elle est en cours si une des présentations est en cours et elle se termine quand toutes les présentations sont finies.

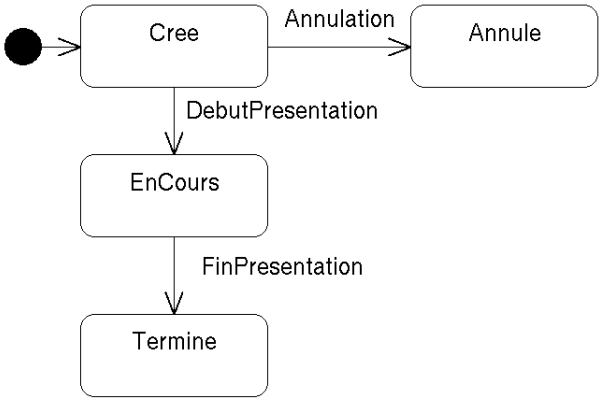


FIG. B.7: Modèle dynamique de "Presentation"

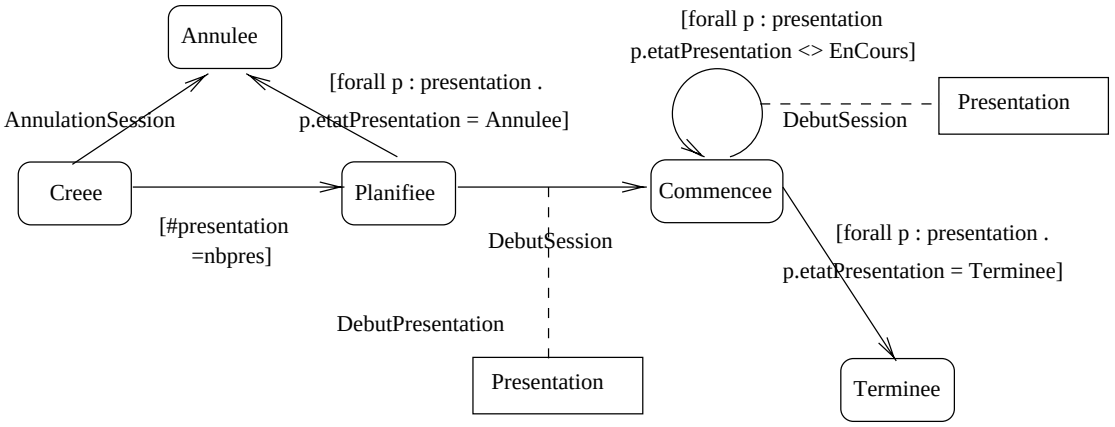


FIG. B.8: Modèle dynamique de "Session"

B.2 Travaux de traduction existants

Comme nous nous intéressons au modèle dynamique complétant un modèle objet, nous limitons la présentation des travaux concernant la dynamique à ceux dont nous avons précédemment décrit la traduction de la partie statique :

- [FLN96a, Ngu98, MS99] traduisent le modèle dynamique d'OMT en des spécifications B ;
- [Lan95, LG96, LHW96] dérivent des spécifications B, Z++ ou VDM++ à partir de diagrammes d'états-transitions.

Nous nous intéressons aussi à [Sek98] qui traduit les diagrammes d'état en B bien qu'il n'intègre pas de modèle objet.

Pour décrire ces travaux, nous nous basons sur l'extrait du diagramme d'états suivant qui décrit le comportement de la classe "Classe" :

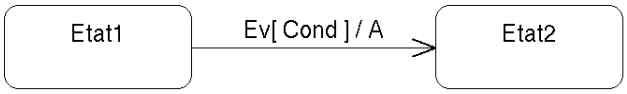


FIG. B.9: Diagramme d'états

B.2.1 Etat

Représentation sous forme de variable de type énuméré Dans la plupart des propositions [Lan95, LHW96, MS99, Sek98], **un état** est représenté par **une variable qui est associée à un type énuméré**. Les valeurs de ce type correspondent aux différents états de la classe "Classe" qui est représenté par le diagramme. Par exemple, en B, [LHW96, Sek98, MS99] définissent un ensemble (clause **Sets**) dans la machine représentant la classe.

Machine Classe
Sets EtatClasse = { Etat1, Etat2,..., Etatn}
...

Comme pour les attributs, un état peut être décrit par une variable d'un certain type ([Lan95]) ou sous forme de fonction ([LHW96, MS99]). Dans le premier cas, le langage utilisé est soit VDM++, soit Z++ ; l'état est alors décrit en ajoutant à la classe qui formalise la classe dont le comportement est représenté par le diagramme d'états, une variable *étatClasse* de type *EtatClasse*.

class CLASSE
types $EtatClasse = < etat1 > | < etat2 > | \ldots | < etatn >$
Instance variables étatClasse : EtatClasse
...

Dans le deuxième cas [LHW96, MS99], la variable *étatClasse* est une fonction entre l'ensemble des (identités d')objets existants *classes* et le type *EtatClasse* :

Machine Classe
Sets IDClasse
EtatCLasse = {Etat1, Etat2,..., Etatn}
Variables classes, ..., etatClasse
Invariant etatClasse $\in$ classes $\rightarrow$ EtatClasse ...

Représentation sous forme d'ensemble d'objets Dans [LHW96], la représentation sous forme de variable de type énuméré n'est qu'une démarche alternative. En effet, si un diagramme d'états comporte moins de 8 à 10 états, **une variable** est ajoutée pour chaque état. Elle représente **l'ensemble des objets dans cet état** et elle doit respecter l'invariant $etati \subseteq classes$ qui signifie que les objets dans l'état i sont des objets de la classe. De plus, les ensembles d'objets dans les états sont disjoints deux à deux ($etat1 \cap etat2 = \varnothing$) et l'union de ces ensembles est égale à l'ensemble des objets de la classe ($etat1 \cup .. \cup etatn = classes$). L'état n'est donc plus une caractéristique raffinant la spécification d'un objet.
Machine Classe
Sets IDClasse
Variables classes, ..., etat1, etat2,...,etatn
Invariant ...
$etat1 \subseteq classes \wedge etat1 \cup .. \cup etatn = classes \wedge etat1 \cap etat2 = \varnothing \wedge ...$

Représentation par un prédicat La troisième proposition pour représenter un état ([FLN96a, Ngu98]) consiste à spécifier **un prédicat** qui porte sur les variables et les ensembles décrivant cet état. Elle distingue deux cas : celui où un état est exprimable à partir des attributs ou des associations de la classe et celui où il ne l'est pas. Dans le premier cas, le prédicat porte sur les variables correspondant aux attributs et/ou aux liens des objets et évite ainsi de spécifier des fonctions redondantes. Par exemple, si l'état correspond à un attribut "a" de la classe ($a \in classes \rightarrow \{val1, val2, ..., valn\}$) , il existe un prédicat de la forme suivante par état :

Etat1(o) $\widehat{=}$ o $\in$ $a^{-1}[\{etat1\}]$

Sinon, si un état n'est pas définissable à partir d'autres données, il est proposé de créer une nouvelle variable. Cette variable est une fonction totale de l'ensemble des instances existantes vers l'ensemble des états. Le prédicat est alors décrit à l'aide de cette variable.
Machine Classe
Sets IDClasse
Variables classes, ..., etatClasse
Definitions Etat1(o) $\widehat{=}$ o $\in$ $etatClasse^{-1}[\{etat1\}]$ ...
Invariant etatClasse $\in$ classes $\rightarrow$ { etat1 , etat2,..., etatn} ...

Cette solution a pour avantages de ne pas rajouter inutilement des informations déjà connues et de faire lien entre le modèle objet et le modèle dynamique en spécifiant un état en fonction des attributs d'une classe.

Bilan La représentation par un prédicat semble être la proposition la plus appropriée. D'une part, l'état reste une caractéristique des objets ; d'autre part, elle introduit une variable d'état seulement en cas de nécessité, sinon un lien est spécifié entre l'état et les attributs de la classe.

B.2.2 Transition

Confusion entre transition et événement Les formalisations des concepts d'événement et de transition sont souvent confondues, **une transition** se traduisant par **une opération portant le nom de l'événement** [FLN96a, Ngu98, Sek98, Lan95, LHW96]. Cette opération spécifie le changement d'un état vers un autre. Dans ce cadre, **une condition** sur une transition se traduit par **une précondition de l'opération de transition** et **une action ou une activité** est décrite par **une opération**. Des exemples typiques de cette démarche sont [LHW96, FLN96a, Ngu98]. Dans [LHW96], une transition est modélisée par une opération *Ev* dans la machine abstraite B représentant la classe. La précondition de *Ev* est que l'objet soit bien dans l'état source de la transition et que l'éventuelle condition sur la transition soit respectée. Sa post-condition spécifie le changement d'état en supprimant l'objet de l'ensemble des objets dans l'état source et en l'ajoutant à celui des objets dans l'état cible. Si une action est appelée sur la transition, l'opération qui lui est associée est appelée dans la post-condition de *Ev* grâce à l'opération $\|$ qui correspond à la conjonction de schémas en Z ($\wedge$).

Machine Classe
Sets IDClasse
Variables classes, ..., etat1, etat2,...,etatn
Invariant ...
$etat1 \subseteq classes \wedge etat1 \cup .. \cup etatn = classes \wedge etat1 \cap etat2 = \varnothing \wedge ...$
Operations
Ev(o) =
PRE o : etat1 $\wedge$ cond
THEN etat1 := etat1 $\setminus$ {o} $\|$ etat2 := etat2 $\cup$ {o} $\|$ A
END

[FLN96a, Ngu98] supposent qu'il existe une classe d'événements ayant comme paramètres (a1,a2,...,an). Un événement se traduit par une opération *Ev* qui a pour paramètre l'objet sur lequel porte l'opération et les valeurs correspondant aux paramètres. Une condition est une précondition de cette opération.

Machine Classe
Sets IDClasse
Variables classes, ..., etatClasse
Definitions Etat1(o) $\widehat{=}$ o $\in$ etatClasse^{-1}[{etat1}] ...
Invariant etatClasse $\in$ classes $\rightarrow$ { etat1 , etat2,..., etatn}
Operations
Ev(o,va1,va2...,van) =
PRE o $\in$ classes $\wedge$ va1 $\in$ Typea1 $\wedge$ ... $\wedge$ cond
THEN ... [prédicat décrivant l'etat cible]
END

Dans [FLN96a], toute action ou activité est une opération B, mais rien n'explique comment ces opérations sont appelées à partir des transitions. [Ngu98] est beaucoup plus précis en décrivant les préconditions des opérations représentant les événements suivant des scenarios d'éxécution. De plus, dans ce travail, une action n'est pas représentée par une opération à part, mais constitue le corps de la substitution de l'opération de transition. L'action n'a donc plus d'existence propre, empêchant ainsi de faire lien avec une opération du modèle objet. De plus, cela implique que si elle est appelée sur plusieurs transitions, sa substitution correspondante est répétée inutilement plusieurs fois.

Machine Classe
Sets IDClasse
Variables classes, ..., etatClasse
Definitions Etat1(o) $\widehat{=}$ o $\in$ etatClasse^{-1}[{etat1}] ...
Invariant etatClasse $\in$ classes $\rightarrow$ { etat1 , etat2,..., etatn} ...
Operations
Ev(o,va1,va2...,van) =
PRE va1 $\in$ Typea1 $\wedge$ ... $\wedge$ cond $\wedge$ [prédicat décrivant l'état source]
THEN S [substitution correspondant à l'effet de l'action A]
END

[Sek98] représente aussi un événement par une opération, mais qui est spécifiée par une expression conditionnelle.

Machine Classe
Sets EtatClasse = { Etat1, Etat2, ..., Etatn }
Variables etatClasse
Invariant etatClasse $\in$ EtatClasse ...
Operations
Ev = IF etatClasse = Etat1
THEN etatClasse := Etat2 $\parallel$ A
END

Dans [Lan95], une transition est aussi représentée par une opération qui modifie l'état d'un objet. L'auteur utilise la clause **sync** de VDM++ qui stipule que l'invocation d'une opération peut commencer son éxécution seulement si la condition spécifiée est vérifiée. Ce concept se distingue de celui de précondition qui indique seulement sous quelles conditions l'opération peut être effectuée. **sync** est donc utilisée pour préciser l'état dans lequel doit être un objet avant d'éxécuter l'opération de transition et les conditions à réaliser.

class Classe
types *EtatClasse* =< *etat*1 >|< *etat*2 >| ... |< *etatn* >
Instance variables etatClasse : EtatClasse
methods
ev() ==
[ext wr etatClasse
post etatClasse = etat2] ;
...
sync
per ev $\Rightarrow$ cond $\wedge$ etatClasse = etat1 ;

Ici encore, une action sur une transition est décrite par une opération, mais son invocation sur une transition est exprimée dans **la description du modèle d'éxécution**. Pour cela, Lano utilise la clause **Thread** de VDM++ qui permet de décrire le comportement des objets actifs en spécifiant suivant les messages reçus quels sont les chemins d'éxécution à suivre. Dans le cas de la formalisation d'un diagramme d'états, les expressions du **Thread** signifient que si l'objet est dans l'état source et si la condition sur la transition est vérifiée, alors on attend la demande d'éxécution de l'opération de transition *ev* et on exécute l'opération *A*.

Thread
while true do
sel (etatClasse = etat1 ∧ cond)**answer** ev -> A ,
...

Distinction entre transition et événement Le seul travail qui distingue événement et transition est [MS99]. **Une transition** est alors représentée par **une opération de changement d'état dans la machine abstraite de la classe** et **un événement** est **une opération dans la machine du modèle d'éxécution**. L'opération de transition a pour précondition que l'objet courant doit faire partie des instances existantes de la classe et que son état est bien l'état source de la transition. Sa postcondition spécifie le nouvel état et appelle si besoin une opération *A* correspondant à une action qui modifie les attributs de la classe. Une opération d'une machine ne pouvant pas être appelée par une autre opération de la même machine, une définition intermédiaire *defact* est utilisée à la place de l'opération *A*.

Machine Classe
Sets IDClasse
EtatCLasse = {Etat1, Etat2,..., Etatn}
Variables classes, ..., etatClasse
Invariant ...
etatClasse ∈ classes → EtatClasse
Definitions
defact1 == ...
Operations
A = defact1 ;
TransitionEt1Et2(o) =
PRE
o ∈ classes ∧ etatClasse(o) = Etats1
THEN
etatClasse(o) = Etats2 || defact1

Le modèle d'éxécution donne lieu à une machine spécifique qui inclut les machines des classes. L'intérêt est que la dynamique du système est représentée plus globalement, permettant ainsi d'exprimer aisément les éléments (conditions, envoi d'événements ...) portant sur d'autres classes. Dans cette nouvelle machine, les opérations *Ev* représentant les événements sont spécifiées. Chaque opération est définie par une substitution SELECT qui a autant de cas que le nombre de transitions où

l'événement peut se produire et qui, pour chaque cas, appelle l'opération de transition correspondante. Ainsi un même événement se trouvant sur plusieurs transitions ne donne lieu qu'à une seule opération *Ev*. Il n'y a pas de multiplication inutile des opérations.

De plus, une distinction entre événements interne et externe est introduite. Un événement interne *Ev'* est spécifié par une définition *DefEv'* et n'a pas d'opération spécifique alors qu'un événement externe correspond à une opération dans laquelle un événement interne peut être appelé.

Machine Controle
Includes Classe1, Classe2
Invariant ...
Definitions
DefEv' == ... ;
Operations
Ev(o1,o2,a1,...,an) =
  PRE
    o1 ∈ classes1 ∧ o2 ∈ classes2 ∧ va1 ∈ Typea1 ∧ ...
  THEN
    SELECT etatClasse(o1) = Etat1 ∧ cond1
      THEN TransitionEt1Et2(o1)
    WHEN etatClasse(o1) = Etat3 ∧ cond2
      THEN TransitionEt3Et2(o1) —— DefEv'
    ELSE skip END
  END

Bilan Seul [MS99] introduit une différence de représentation entre les concepts de transition et d'événement. Cela a pour avantage d'éviter d'introduire plusieurs opérations correspondant à un même événement si celui-ci se trouve sur plusieurs transitions.

Néanmoins tous les travaux existants se situent dans un cadre où tous les constituants d'un modèle dynamique sont des opérations ou des conditions sur ces opérations. Un seul type d'événement est donc considéré : les appels d'opérations. Ces propositions sont limitatives par rapport aux différents types (appel d'opération, signal, durée écoulée ou changement) que nous avons introduits dans la section B.1.1 (page 9), ce qui simplifie la représentation formelle du modèle dynamique.

B.2.3 Envoi d'événements

Soit la figure B.10 représentant une partie du diagramme d'états de la classe "Classe1". Sur la transition passant de "Etat1" à "Etat2" un événement "A" est envoyé à la classe "Classe2".

L'envoi d'événements d'une classe "Classe1" vers une classe "Classe2" n'est rien d'autre que **l'appel d'une opération** de "Classe2". Lorsque les opérations ont été spécifiées comme des opérations B, VDM++ ou Z++ [Lan95, Sek98, MS99], la formalisation de cet envoi correspond à **l'appel de l'opération** *A*. Dans [Lan95], une association se traduisant dans la classe *Classe1* par une variable *c2* faisant

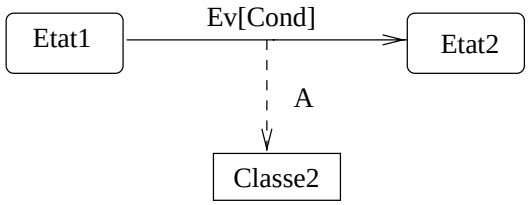


FIG. B.10: Envoi d'un événement à la classe "Classe2"

référence à *Classe2*, il suffit pour envoyer un événement d'appeler l'opération *A* à partir de *c2*.

class Classe1
types *EtatCvclasse* =< *etat*1 >|< *etat*2 >| ... |< *etatn* >
Instance variables c2 : @Classe2
etatClasse : EtatClasse
methods ...
Thread
  while true do
    sel (etatClasse = etat1 ∧ cond)**answer** ev -> c2.A ,
    ...

Dans [MS99], l'envoi d'un événement à une autre classe s'effectue dans la machine *Controle* représentant le modèle d'éxécution. Dans l'opération de description d'un événement, l'opération correspondant à l'appel est appelée simultanément à l'opération de transition.

Machine Controle
Includes Classe1, Classe2
Invariant ...
Operations
Ev(o1,o2,a1,...,an) =
  PRE
    o1 ∈ classes1 ∧ o2 ∈ classes2 ∧ va1 ∈ Typea1 ∧ ...
  THEN
    SELECT etatClasse(o1) = Etat1 ∧ cond1
      THEN TransitionEt1Et2(o1) ∧ A
    WHEN etatClasse(o1) = Etat3 ∧ cond2
      THEN TransitionEt3Et2(o1)
    ELSE skip END
  END

[Sek98] ne représentant pas le modèle objet, l'envoi d'un événement correspond à l'appel de l'opération pour cet événement sans tenir compte des liens entre objets.

Machine Classe
Sets EtatClasse = { Etat1, Etat2, ..., Etatn }
Variables etatClasse
Invariant etatClasse ∈ EtatClasse ...

Operations
Ev = IF etatClasse = Etat1
THEN etatClasse := Etat2 || A
END

Par contre, [Ngu98] ne représentant pas explicitement une action par une opération, **l'envoi d'un événement** ne correspond pas à un appel d'opération mais à l'inclusion dans l'opération représentant la transition d'**une substitution correspondant à l'action**.

Machine Classe
...
Operations
Ev(o,va1,va2...,van) =
PRE va1 ∈ Typea1 ∧ ... ∧ cond1 ∧ prédicat décrivant l'état source
THEN IF cond2 ∧ prédicat décrivant l'état source de "Classe2"
S1 [substitution correspondant à l'effet de l'action A1 et A2]
ELSE S2 [substitution correspondant à l'effet de l'action A1]
END

Bilan Dans tous ces travaux, l'envoi d'événements se traduit de façon assez naturelle par l'appel d'une opération correspondante ou par l'éxécution de sa substitution correspondante. Mais la diversité des types d'événements n'est pas prise en compte.

B.2.4 Diagramme d'états hiérarchiques

Soit le diagramme d'états hiérarchique suivant :

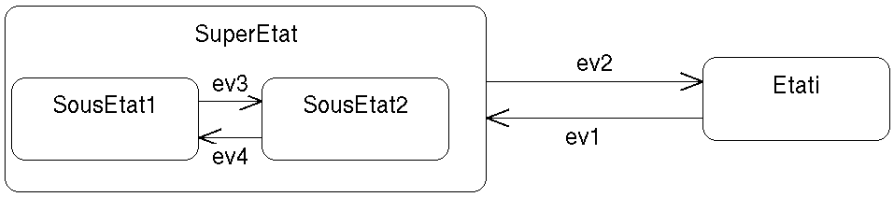


FIG. B.11: Diagramme d'états hiérarchique

La formalisation des diagrammes hiérarchiques est mentionnée seulement dans [Lan95, LHW96, Sek98]. De plus, [LHW96] ne présente pas vraiment une manière de représenter la structuration des diagrammes, puisque la traduction s'effectue comme si les diagrammes étaient à plat : le super-état n'a pas de valeur dans le type énuméré des états (on y trouve à la place le nom de ses sous-états) et les opérations représentant les transitions vont et aboutissent aux sous-états.

Dans [Lan95], le principe de traduction suivant est donné. Comme pour les états, **les sous-états** donnent lieu à **une variable de type énuméré** <SousEtat1>, <SousEtat2>,...,<SousEtatn>. L'appartenance des sous-états au super-état doit être explicite bien que la représentation des sous-états <SousEtat1>, <SousEtat2>,...,<SousEtatn> ait seulement du sens aux points où la variable du super-état a la valeur <SuperEtat>. Ainsi le lien entre super et sous-états est mis

en évidence, mais tout test sur la valeur du sous-état nécessite aussi d'inclure un test sur la valeur du super-état.

Une transition se traduit toujours par **une opération dont la post-condition spécifie l'état cible** et le modèle d'éxécution devient alors :

Thread
while true do
sel etatClasse = Etati **answer** ev1 ->
while etatClasse = SuperEtat do
sel
answer ev2,
sousetatClasse = <SousEtat1> answer ev3,
sousetatClasse = <SousEtat2> answer ev4 ;

Une transition entrante dans le super-état met la variable d'états *etatClasse* à *SuperEtat* et la variable de sous-état dans le sous-état initial (*SousEtat1* par exemple). Une transition sortante d'un super-état donne simplement la nouvelle valeur de l'état et les transitions internes aux super-états ne modifie que la variable *sousetatClasse*.

Bien que la solution [Lan95] nécessite de spécifier explicitement le sous-état source ou cible d'une transition, elle permet de représenter de manière uniforme et au même niveau (*Thread*) toutes les transitions qu'elles concernent le super-état, les sous-états ou les deux.

[Sek98] modélise aussi les sous-états par des variables supplémentaires pour chaque super-état. Néanmoins, il ne précise pas le lien existant entre les sous-états et leur super-état.

Machine Classe
Sets EtatClasse = { SuperEtat, Etat1, Etat2, ..., Etatn }
SousEtatClasse = { SousEtat1, SousEtat2, ..., SousEtatn }
Variables etatClasse , sousetatclasse
Invariant etatClasse ∈ EtatClasse ∧ sousetatclasse ∈ SousEtatClasse...

Quand on entre/sort d'un super-état, il faut spécifier l'état dans lequel on entre/sort. Par exemple, si en entrant dans le super-état, on entre aussi dans *SousEtat1*, [Sek98] propose :

Machine Classe
Sets EtatClasse = { SuperEtat, Etat1, Etat2, ..., Etatn }
SousEtatClasse = { SousEtat1, SousEtat2, ..., SousEtatn }
Variables etatClasse , sousetatclasse
Invariant etatClasse ∈ EtatClasse ∧ sousetatclasse ∈ SousEtatClasse...
Operations
Ev = IF etatClasse = Etati
THEN etatClasse := SuperEtat || sousetatclasse := SousEtat1
END

B.2.5 Récapitulatif

Le tableau B.1 résume l'apport de chacun des travaux présentés précédemment.

Concepts Articles	Etat	Evénement	Condition	Action	envoi d'événements	DE hiérachiques	DE concurrents
[LHW96]	T	T	T	T		A	A
[Ngu98]	T	T	T	T	T^-		
[MS99]	T	T	T	T	T		
[Lan95]	T	T	T	T	T	T	T^-

TAB. B.1: Récapitulatif des travaux sur le modèle dynamique

Remarque :
Nous n'avons pas présenté les travaux de [Lan95] concernant les diagrammes d'états concurrents car nous n'étudierons pas ce concept par la suite. De plus, il n'est pas totalement traité par l'auteur.

B.3 Traduction du modèle dynamique en Z et Object-Z

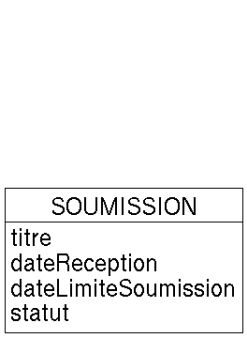
Le point de départ de la traduction du modèle dynamique est les spécifications formelles obtenues à partir du modèle objet. Ces spécifications sont complétées par les données et les opérations dérivées du modèle dynamique.

Dans ce travail, nous limitons le modèle dynamique présenté précédemement. Nous ne traitons que deux types d'événements (les appels d'opération et les signaux) et nous faisons l'hypothèse simplificatrice que tous les appels d'opérations, toutes les actions et toutes les conditions du modèle dynamique à traduire portent sur l'intension de la classe concernée par ce modèle. Nous ne considérons donc pas les cas où une opération ou une condition porte sur l'extension de la classe ou sur ses liens avec d'autres classes. Bien que restrictive, cette hypothèse permet néanmoins de traiter la plupart des transitions des modèles dynamiques. Nous présentons donc tout d'abord comment les concepts du modèle dynamique peuvent être représentés en Z ou en Object-Z [DLCP98] en tenant compte de cette hypothèse. Puis nous discutons de la généralisation de la traduction du modèle dynamique à tous les cas de transitions.

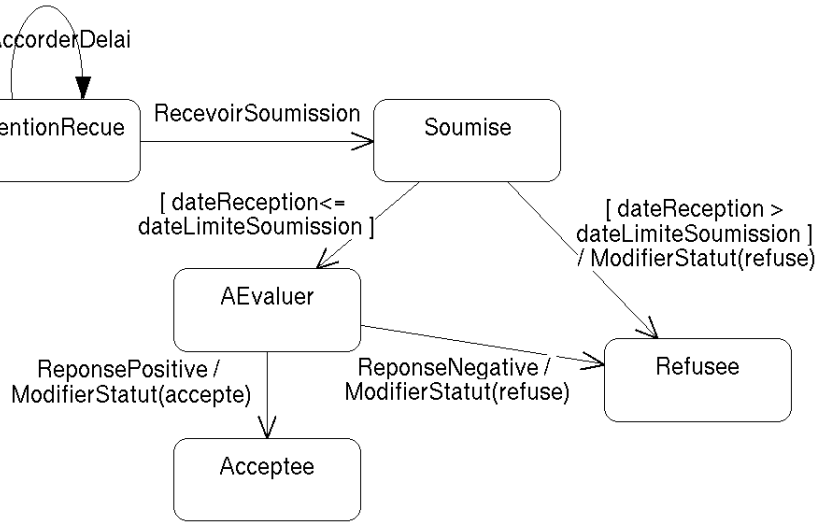
B.3.1 Traduction du modèle dynamique en Z

Exemple B.1 *Dans cette section, nous reprenons le modèle dynamique de la classe "SOUMISSION" (Fig. B.12). Une soumission existe à partir du moment où une intention de soumission est reçue (état "IntentionRecue"). Il est possible dans cet état d'accorder un délai pour la soumission ("AccorderDelai") c'est-à-dire de reculer la date limite de soumission. Quand la soumission proprement dite est reçue, elle passe dans l'état "Soumise". Si elle est reçue après la date limite, elle est automatiquement refusée ; sinon elle suit le processus d'évaluation (état "AEvaluer"). Suivant le résultat de ce processus, la soumission est soit "Acceptee", soit "Refusee". Quand une soumission passe dans l'état "Acceptee" (resp. "Refusee"), l'attribut "statut"*

qui représente le statut final de la soumission prend pour valeur "accepte" (resp. "refuse").



(a) Classe "SOUMISSION"



(b) Modèle dynamique de "SOUMISSION"

FIG. B.12: Classe "SOUMISSION" et son modèle dynamique

Traduction du concept d'état

Les états d'un modèle dynamique correspondent aux différentes situations dans lesquelles peut se trouver un objet. Ils peuvent être représentés formellement comme un ajout d'information sur les données. Ils complètent par une nouvelle variable le schéma Z décrivant l'intension de la classe décrit par le modèle dynamique. Cette nouvelle variable décrivant l'état d'un objet a pour type un type énuméré ayant pour valeurs les différents états possibles.

Le fait d'introduire systématiquement une nouvelle variable pour l'état est redondant si l'état peut être défini à partir des attributs de la classe. Mais dans ce cas, il est possible d'écrire un prédicat qui explicite le lien entre les valeurs de l'état et des attributs. C'est alors un gain d'information qui précise la corrélation entre les modèles objet et dynamique.

Exemple B.2

Les états du diagramme de "SOUMISSION" sont décrits par un type EtatSoumission qui énumère les différentes valeurs d'états possibles. La variable etatSoumission est ajoutée au schéma SOUMISSION pour permettre de préciser l'état dans lequel se trouve chaque soumission. De plus, il est possible de faire un lien entre l'attribut "statut" et la variable d'état : si le statut est "accepte" (resp. refuse), etatSoumission doit être "Acceptee" (resp. "Refusee").

$$STATUT ::= inconnu \mid accepte \mid refuse$$
$$EtatSoumission ::= IntentionRecue \mid Soumise \mid AEvaluer \mid Refusee \mid Acceptee$$

Proposition 19 : Etat
Les états donnent lieu à **une variable** *etatClasse* dans le schéma d'intension de la classe *CLASSE*. L'ensemble des états donne lieu à **un type énuméré** *EtatClasse* dont les valeurs correspondent aux états.

$$EtatClasse ::= etat1 \mid etat2 \mid ... \mid etatn$$

CLASSE

... [Attributs de "CLASSE"]
$etatClasse : EtatClasse$

SOUMISSION

$titre : TITRE$
$dateReception : DATE$
$dateLimiteSoumission : DATE$
$statut : STATUT$
$etatSoumission : EtatSoumission$

$statut = accepte \Rightarrow etatSoumission = Acceptee$
$statut = refuse \Rightarrow etatSoumission = Refusee$

Traduction du concept de transition

Evénement Dans la section B.1.1, nous avons distingué quatre sortes d'événements : les appels d'opérations, les durées écoulées, les changements de valeurs d'un objet et les signaux. Nous ne traitons ici que les deux principaux c'est-à-dire les opérations et les signaux.

Si un événement est **un appel d'opération**, il doit correspondre à une opération de la classe. Ces opérations ont été exprimées en Z lors de la traduction du modèle objet. Les squelettes produits seront complétés lors de la traduction du concept de transition pour exprimer le changement d'état provoqué par l'appel de l'opération.

Si un événement est de type **signal**, il est caractérisé par l'instant auquel il se produit. Les différents signaux ont tous en commun cette propriété, aussi nous considérons qu'il existe un signal de base *SIGNAL* qui se spécialise en différents types de signaux *SIGNALi*. Contrairement à notre proposition concernant l'héritage en Z, nous n'introduisons pas de type $SIGNAL1 \vee SIGNAL2 \vee ... \vee SIGNALn$ car il n'est pas ensuite possible de spécifier que tout signal est de ce type et de distinguer les différents types de signaux dans les opérations correspondant aux transitions c'est-à-dire que ces opérations ne peuvent pas avoir d'entrées de type $SIGNAL1 \vee SIGNAL2 \vee ... \vee SIGNALn$ dont on précise dans la partie prédicative

le type (*SIGNAL*1 ou *SIGNAL*2 ou *SIGNALn*.

SIGNAL

$instant : INSTANT$

SIGNALi

SIGNAL

... [attributs de SIGNALi]

Pour les transitions automatiques c'est-à-dire n'ayant pas d'événement, nous considérons qu'elles sont en fait déclenchées par un signal interne. Un signal interne est de type *SIGNAL* car a priori, il n'a pas de caractéristique spécifique.

Pour un événement, la traduction en Z ne peut donc pas être automatique puisqu'elle nécessite de préciser le type de chaque événement. Mais c'est un gain d'information par rapport au modèle dynamique puisque'il faut donner explicitement le type de l'événement.

Proposition 20 : Evénement

Tout événement de type appel d'opération correspond à **une opération** du modèle objet dont la traduction a été précédemment effectuée en Z.

Les événements de type signal donne lieu à **un schéma** Z *SIGNAL* ayant pour variable l'instant auquel se produit un signal. Chaque signal *SIGNALi* est traduit par **un schéma** Z incluant *SIGNAL*.

SIGNAL

$instant : INSTANT$

SIGNALi

SIGNAL

... [attributs de SIGNALi]

Exemple B.3 *Dans la figure 12(b), "RecevoirSoumission" est un appel à l'opération "RecevoirSoumission" qui donne une valeur à la date de réception d'une soumission alors que "ReponsePositive" est un signal.*

D'après notre règle de traduction, "RecevoirSoumission" se traduit donc par un schéma d'opération SOUMISSIONRecevoirSoumission dont la spécification ne change pas pour l'instant par rapport à la traduction du modèle objet.

"ReponsePositive" est un signal caractérisé par sa variable réponse qui doit avoir la valeur "positive".

$REPONSE ::= positive \mid negative$

ReponsePositive

SIGNAL

$reponse : REPONSE$

$reponse = positive$

Condition Une condition est une expression booléenne portant sur les valeurs des attributs et/ou des liens d'un objet. Nous avons fait l'hypothèse simplificatrice qu'une condition portait nécessairement sur l'intension de la classe c'est-à-dire sur ses attributs. Une condition se traduit donc naturellement par un prédicat sur le schéma d'intension de la classe *CLASSE*.

Proposition 21 : Condition
Toute condition "Cond" portant sur les attributs de "CLASSE" donne lieu à **un prédicat** *Cond* dans un schéma Z *XCond* incluant le schéma d'intension *CLASSE* de la classe de l'objet.

XCond

CLASSE	[inclusion du schéma sur lequel porte la condition]
Cond	[expression de la condition]

Exemple B.4 *Dans l'exemple du modèle dynamique de "SOUMISSION", la transition entre "Soumise" et "Refusee" a pour condition que l'attribut "dateReception" d'une soumission ait une valeur supérieure à la date limite de soumission. dateReception et dateLimiteSoumission étant du type de base DATE, il n'est pas possible d'appliquer l'opérateur de comparaison >. On suppose qu'une relation d'ordre total non-réflexive DateInf a été introduite précédemment : si deux dates d1 et d2 ne sont pas égales,* $(d1, d2) \in DateInf$ *signifie que d1 est avant d2. La condition "dateReception > dateLimiteSoumission" s'écrit alors :*

EvaluationCond

SOUMISSION

$(dateLimiteSoumission, dateReception) \in DateInf$

Action Toute action doit correspondre à une opération du modèle objet. Elle doit donc avoir été traduite en Z précédemment. Mais avec la traduction du modèle dynamique, une variable d'état *etatClasse* a été ajoutée au schéma d'intension de la classe. Donc si l'opération n'est pas aussi un événement d'une autre transition, il faut préciser qu'elle ne modifie pas la variable d'état.

Exemple B.5 *Sur la transition entre "Soumise" et "Refusee", l'action "ModifierStatut" est appelée. Cette action modifie la variable statut en fonction d'une valeur donnée en entrée. Dans le cas de cette transition, cette valeur doit être "refuse". Les autres variables de SOUMISSION ne changent pas, en particulier etatSoumission.*

Proposition 22 : Action
Toute action "A" (portant sur l'intension de sa classe "CLASSE") a donné lieu à **un schéma d'opération** *CLASSEA* lors de la traduction du modèle objet. Si l'action n'est pas aussi un événement du modèle dynamique, la spécification de *CLASSEA* est complétée pour préciser que la variable d'état *etatClasse* n'est pas modifiée. Dans le cas contraire, *etatClasse* est aussi changé.

CLASSEA

$\Delta CLASSE$
...

...
$etatClasse' = etatClasse$

SOUMISSIONModifierStatut

$\Delta SOUMISSION$
$s? : STATUT$

$statut' = s? \land titre' = titre \land dateReception' = dateReception \land$
$dateLimiteSoumission' = dateLimiteSoumission \land$
$etatSoumission' = etatSoumission$

Transition Une transition correspond au passage d'un état à un autre. Ce changement d'état correspond en Z à la modification de la valeur de la variable représentant l'état *etatClasse*. Nous rappelons que nous ne nous intéressons ici qu'aux transitions dont les éléments portent sur l'intension de la classe.

Si l'événement déclenchant la transition est un appel d'opération "Op", les squelettes de spécifications produits lors de la traduction du modèle objet sont complétés en spécifiant que la variable d'état doit être dans l'état source *etati* avant l'opération et dans l'état cible *etatj* après l'opération.

CLASSEOp

$\Delta CLASSE$
...

...
$etatClasse = etati \land etatClasse' = etatj$

Si l'événement est un signal, il faut créer un nouveau schéma d'opération qui prend en entrée le signal déclenchant la transition et qui modifie la valeur de la variable d'état.

ModifierEtatiEtatj

$\Delta CLASSE$
$s? : SIGNALi$

$etatClasse = etati \wedge etatClasse' = etatj$

Si la transition est automatique c'est-à-dire si elle n'a pas d'événement, on suppose qu'un signal interne a été envoyé et la transition se traduit comme dans le cas de la réception d'un signal. Nous n'étudions pas le moteur d'exécution qui gère les signaux et plus généralement, les envois d'événements entre les objets.

Une transition comportant une condition ou une action donne aussi lieu à **la combinaison des différents schémas** la composant. L'opération qui modifie l'état (*CLASSEOp* ou *ModfierEtat*) et la condition sont liés par une conjonction, ce qui aboutit au fait que la condition est considérée comme une pré-condition de l'opération de modification d'état. S'il existe une action sur la transition, cette conjonction est composée (opérateur $\fatsemi$) avec l'opération représentant l'action. L'opérateur $\fatsemi$ permet de définir une nouvelle opération comme étant la composition de deux autres : la nouvelle opération modifie l'état du schéma de E1 à E3, la première opération modifiant l'état du schéma de E1 à E2 et la deuxième modifiant l'état du schéma de E2 à E3. Cette solution suppose donc qu'il existe un état intermédiaire pendant la transition : l'action est effectuée sur la transition seulement après l'opération de modification d'état. Pour pouvoir représenter la simultanéité des opérations sur une transition, il aurait fallu faire la conjonction des différentes opérations. Mais cela créerait des incompatibilités dûes à des spécifications d'opérations divergentes : par exemple, l'opération de modification d'état change *etatClasse* alors que l'action ne la modifie pas.

Cette solution n'est pas totalement satisfaisante dans la mesure où elle ne reflète pas exactement la sémantique des transitions. En effet, les opérations d'une transition sont effectuées en séquence. Les transitions ne sont pas atomiques du fait de l'existence d'un état intermédiaire lors de leur franchissement.

De plus, cela conduit à un affaiblissement de l'invariant qui spécifie le lien entre l'état et les attributs. Par exemple, nous avons exprimé que la valeur du statut d'une soumission détermine celle de son état ($statut = accepte \Rightarrow etatSoumission = Acceptee$). En fait, il existe une équivalence entre les valeurs du statut et de l'état qui ne peut pas être spécifiée car elle n'est pas vérifiée dans l'état intermédiaire de la transition entre "AEvaluer" et "Accepetee".

Enfin, cette solution nécessite d'être approfondie pour pouvoir considérer les échanges d'événements entre objets : les opérations regroupant les éléments d'une transition représentent bien le comportement d'un objet lors du franchissement de cette transition, mais ces opérations ne sont jamais appelées directement car c'est seulement leur événement déclenchant qui est envoyé à l'objet.

Exemple B.6 *La transition entre "Soumise" et "Refusee" est guardée par la condition que la date de reception d'une soumission soit supérieure à la date limite de soumission et appelle l'action "ModifierStatut". Elle n'a pas d'événement explicite et se traduit donc comme s'il existait un signal interne. Elle donne lieu à une opé-*

Proposition 23 : Transition

Le changement d'état effectué lors d'une transition "T" se traduit différemment suivant le type de l'événement déclenchant "T".
Si l'événement est **un appel d'opération**, le schéma d'opération produit lors de la traduction du modèle objet est complété en spécifiant que avant l'opération, la variable d'état *etatClasse* a pour valeur la valeur correspondant à l'état source "etati" et que après l'opération, elle a pour valeur celle correspondant à l'etat cible *etatj*.

CLASSEOp

$\Delta CLASSE$
...

$(etatClasse = etati \land etatClasse' = etatj)$
...

Si l'événement est **un signal**, **un schéma d'opération** *ModifierEtatij* qui a pour variable d'entrée le signal et qui modifie la valeur de la variable d'état est créé.

ModifierEtatij

$\Delta CLASSE$
$s? : SIGNALi$

$etatClasse = etati \land etatClasse' = etatj$
...

Si **la transition est automatique**, on suppose qu'un signal interne a été envoyé et la transition se traduit suivant la proposition précédente pour les signaux.
S'il existe une condition "Cond" ou une action "A" sur une transition, les différents schémas la composant sont combinés. Si l'événement est un appel d'opération "Op", la transition se traduit par :

$$TransitionEtatij \mathrel{\widehat=} (CLASSEOp \land XCond) \fatsemi CLASSEA$$

Si l'événement est un signal, la transition se traduit par :

$$TransitionEtatij \mathrel{\widehat=} (ModifierEtatij \land XCond) \fatsemi CLASSEA$$

ration ModifierSoumiseRefusee qui modifie seulement la variable etatSoumission.

ModifierSoumiseRefusee

$\Delta SOUMISSION$
$s? : SIGNALINTERNE$

$etatSoumission = Soumise \land etatSoumission' = Refusee$
$titre' = titre \land dateReception' = dateReception \land$
$dateLimiteSoumission' = dateLimiteSoumission \land statut' = statut$

Pour représenter la transition, l'opération ModifierSoumiseRefusee de modification d'etatSoumission, le schéma de la condition EvaluationCond (exemple B.4) et l'opération SOUMISSIONModifierStatut (exemple B.5) sont combinées :

$$TransitionSoumiseRefusee == (ModifierSoumiseRefusee \wedge EvaluationCond) \fatsemi SOUMISSIONModifierStatut$$

Traduction des diagrammes d'états hiérarchiques

Les exemples illustrant nos propositions de traduction pour les diagrammes d'états hiérarchiques concernent la spécialisation de l'état "AEvaluer" d'une soumission présentée dans la section B.1.4 (page 11).

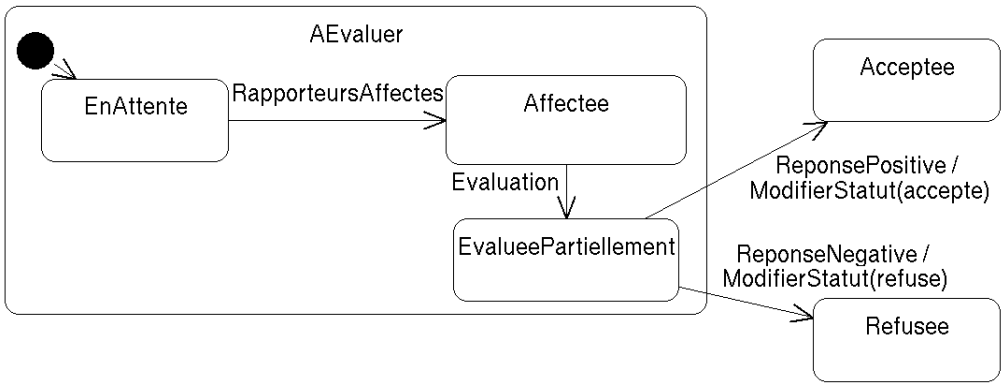


FIG. B.13: Spécialisation de l'état "AEvaluer"

Sous-état Un sous-état étant un état particulier, le principe de traduction d'un état peut s'appliquer à un sous-état. Un sous-état donne donc lieu à une variable dont le type énumère les différents sous-états existants. De plus, ce type a un élément *SuperetatVide* qui permet d'exprimer le lien entre un super-état et ses sous-états. Si l'objet est dans le super-état, il doit être dans l'un de ses sous-états ($etatClasse = SuperEtat \Leftrightarrow sousetatSuper \in \{sousetat1, sousetat2, ..., sousetatn\}$). Cette solution a donc pour avantage de mettre en évidence la structuration des états. Cependant si les niveaux hiérarchiques se multiplient, le nombre de variables de sous-états et de contraintes se multiplient, faisant exploser la taille du schéma.

Exemple B.7 *L'état "AEvaluer" se décompose en trois sous-états "EnAttente", "Affectee" et "EvalueePartiellement". Pour ces sous-états, le type SousetatAEvaluer est créé et énumère les valeurs des différents sous-états et AEvaluerVide. Comme pour les états, une variable sousetatAEvaluer est introduite dans le schéma SOUMISSION. Dans les prédicats, on précise que si etatSoumission a pour valeur AEvaluer, alors sousetatAEvaluer doit avoir l'une des valeurs correspondant à un sous-état ; sinon sousetatAEvaluer a pour valeur AEvaluerVide.*

Proposition 24 : Sous-état
Pour chaque état hiérarchique "SuperEtat", **les sous-états** donnent lieu à **une variable** *sousetatSuper* dans le schéma d'intension de la classe *CLASSE*. L'ensemble des sous-états donne lieu à **un type énuméré** *SousetatSuper* dont les valeurs correspondent aux sous-états et à la valeur *SuperetatVide*.

$$EtatClasse ::= SuperEtat \mid etat2 \mid ... \mid etati$$
$$SousetatSuper ::= sousetat1 \mid sousetat2 \mid ... \mid sousetatn \mid SuperetatVide$$

CLASSE

```
...                                              [Attributs de "CLASSE"]
etatClasse : EtatClasse
sousetatSuper : SousetatSuper
-----------------------------------------------------------------------
etatClasse = SuperEtat ⇔
    sousetatSuper ∈ {sousetat1, sousetat2, ..., sousetatn}
```

$$EtatSoumission ::= IntentionRecue \mid Soumise \mid AEvaluer \mid Refusee \mid Acceptee$$
$$SousetatAEvaluer ::= EnAttente \mid Affectee \mid EvalueePartiellement \mid AEvaluerVide$$

SOUMISSION

```
titre : TITRE
dateReception : DATE
dateLimiteSoumission : DATE
statut : STATUT
etatSoumission : EtatSoumission
sousetatAEvaluer : SousetatAEvaluer
-----------------------------------------------------------------------
etatSoumission = AEvaluer ⇔
    sousetatAEvaluer ∈ {EnAttente, Affectee, EvalueePartiellement}
```

Transition **Une transition entre deux sous-états** est une transition comme une autre, mise à part que c'est la variable de sous-état qui est modifiée.

Le second cas de transition possible est celui **d'une transition entre le super-état et un autre état**. Si la transition sort du super-état, elle est décrite comme les autres transitions car les contraintes liant le super-état et ses sous-états font implicitement passer la variable *sousetatSuper* à la valeur *SuperetatVide*. Par contre, s'il faut être dans un sous-état particulier pour pouvoir effectuer la transition, l'opération de changement d'état doit aussi préciser la valeur initiale du sous-état.

ModifierSousEtatEtat

$\Delta CLASSE$
...

...
$etatClasse = SuperEtat \wedge sousetatSuper = sousEtati \wedge etatClasse' = Etat$

Si la transition entre dans le super-état, cela signifie qu'elle entre aussi dans le sous-diagramme. Il faut donc aussi donner une valeur à la variable de sous-état dans l'opération de transition. Si le sous-état dans lequel arrive la transition n'est pas spécifié explicitement, sa valeur correspond au sous-état initial du sous-diagramme. Soient, par exemple, "Etat" l'état source de la transition entrant dans "SuperEtat" et "SousEtati" le sous-état initial du sous-diagramme d'états. Si la transition entrant dans le super-état est déclenchée par un signal "s", l'opération représentant le changement d'état entre "Etat" et "SuperEtat" donne la valeur *SousEtati* à la variable de sous-état :

ModifierEtatSuperEtat

$\Delta CLASSE$
$s? : SIGNALi$

$etatClasse = Etat \wedge etatClasse' = SuperEtat \wedge sousetatSuper' = sousEtati$

Comme pour les autres transitions, ces opérations de changement d'états peuvent être combinés avec des schémas représentant les conditions ou les actions.

En fait, cette proposition est simplement une extension de notre règle de traduction des transitions. Le sous-diagramme est donc bien vu comme un raffinement de son super-diagramme. Néanmoins elle ne propose pas une vision totalement hiérarchique des transitions puisqu'elle ne distingue pas les différents niveaux de transitions. Si un sous-diagramme est défini, toutes les transitions y compris celles du diagramme principal doivent préciser la valeur du sous-état.

Exemple B.8 *Dans la spécialisation de l'état "AEvaluer", les transitions entre "EnAttente" et "Affectee" et entre "Affectee" et "EvalueePartiellement" se traduisent comme les transitions sur le diagramme de "SOUMISSION" mise à part que c'est la variable sousetatAEvaluer qui est modifiée.*

La transition entre "EvalueePartiellement" et "Refusee" précise la transition entre "AEvaluer" et "Refusee" en contraignant un objet dans "AEvaluer" à être dans "EvalueePartiellement" pour pouvoir passer dans "Refusee". L'opération ModifierAEvaluerRefusee est Donc raffinée pour spécifier le sous-état dans lesquel doit se trouver "AEvaluer" au départ de la transition.

Proposition 25 : Transition

Les transitions entre deux sous-états se traduisent suivant la proposition 23 mais en spécifiant dans le schéma d'opération de changement d'état (événement de type appel d'opération ou signal) que c'est la variable de sous-état qui est modifiée.

$$\Delta CLASSE$$
$$\ldots$$
$$(sousetatSuper = sousetati \land sousetatSuper' = sousetatj)$$
$$\ldots$$

Si la possibilité de franchissement **d'une transition sortant dans le super-état** *SuperEtat* ne dépend pas du sous-état, alors la transition se traduit suivant la proposition 23 ; sinon le schéma de l'opération de changement d'état précise la valeur du sous-état *sousetati* avant l'opération :

$$\Delta CLASSE$$
$$\ldots$$
$$(etatClasse = SuperEtat \land sousetatSuper = sousetati \land etatClasse' = Etat2)$$
$$\ldots$$

Pour **une transition entrant dans le super-état** *SuperEtat*, il faut ajouter à l'opération qui modifie l'état une post-condition qui spécifie le sous-état *SousEtati* dans lequel la transition entre.

$$\Delta CLASSE$$
$$\ldots$$
$$\ldots$$
$$(etatClasse = Etat \land etatClasse' = SuperEtat \land sousetatSuper' = SousEtati)$$

ModifierAEvaluerAcceptee

$$\Delta SOUMISSION$$
$$s? : ReponsePositive$$
$$(etatSoumission = AEvaluer \land sousetatAEvaluer = EvalueePartiellement \land etatSoumission' = Acceptee)$$
$$titre' = titre \land dateReception' = dateReception \land dateLimiteSoumission' = dateLimiteSoumission \land statut' = statut$$

La transition déclenchant l'action "ModifierStatut", il faut composer l'opération de

changement d'état avec l'opération correspondant à "ModifierStatut" :

$$TransitionAEvaluerAcceptee == ModifierAEvaluerAcceptee \fatsemi SOUMISSIONModifierStatut$$

Le dernier cas de transition à étudier est celui des transitions entrant dans "AEvaluer" qui doivent spécifier le sous-état dans lequel elle entre. L'opération ModifierSoumiseAEvaluee est modifiée pour préciser que si elle se trouve dans le cas où etatSoumission prend pour valeur AEvaluer, sousetatAEvaluer prend pour valeur EnAttente.

ModifierSoumiseAEvaluer

$$\Delta SOUMISSION$$
$$s? : SIGNAL$$

$$etatSoumission = Soumise \land$$
$$\quad etatSoumission' = AEvaluer \land sousetatAEvaluer' = EnAttente$$
$$titre' = titre \land dateReception' = dateReception \land$$
$$dateLimiteSoumission' = dateLimiteSoumission \land statut' = statut$$

L'opération de transition TransitionRecueRefusee s'exprime comme précédemment.

$$TransitionSoumiseRefusee \mathrel{\widehat=} (ModifierSoumiseAEvaluer \land EvaluationCond) \fatsemi SOUMISSIONModifierStatut$$

B.3.2 Traduction du modèle dynamique en Object-Z

Comme pour les propositions concernant le modèle objet, les règles traitant de la traduction du modèle dynamique en Object-Z suivent les mêmes principes que celles ayant Z comme langage cible. Nous ne revenons pas ici sur ces points, mais nous détaillons les différences de traduction dûes à l'utilisation d'un langage formel orienté objet. Il faut néanmoins noter que nous n'utilisons pas les invariants d'histoire pour représenter la dynamique d'une classe. En effet, ils ne font actuellement pas partie de la sémantique d'Object-Z et ne sont pas supportés par le vérificateur de type. La définition de leur syntaxe et leur sémantique n'est donc actuellement pas stable.

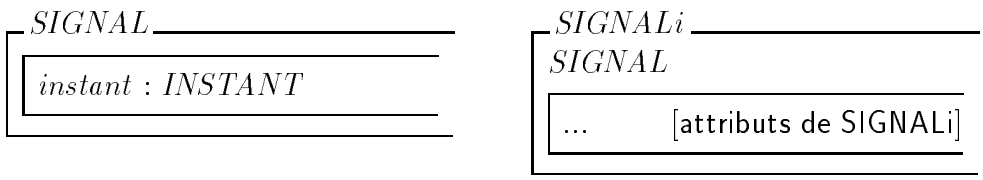
Traduction du concept de transition

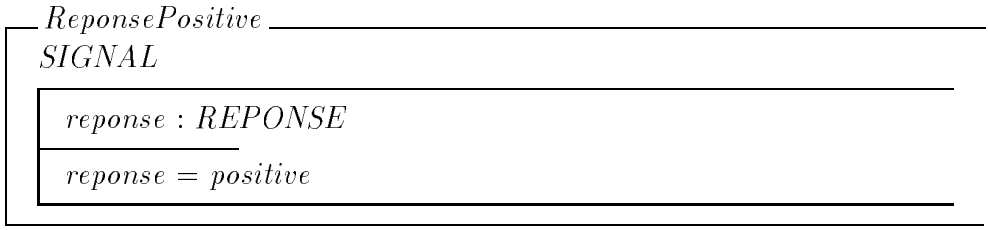
Evénement Pour les événements de type signal, nous utilisons l'héritage d'Object-Z pour décrire la spécialisation entre la classe des signaux en général et des types de signaux ayant des caractéristiques spécifiques. Il existe donc une classe *SIGNAL* dont héritent tous les signaux *SIGNALi*. Ceci ajoute une propriété que nous n'avons pas en Z : tous les *SIGNALi* sont des *SIGNAL*.

Exemple B.9 *Le signal "ReponsePositive" donne lieu à une classe ReponsePositive qui hérite de la classe SIGNAL.*

$$REPONSE ::= positive \mid negative$$

Proposition 26 : Evénement-Signal
Les événements de type signal donne lieu à **une classe** Object-Z *SIGNAL* représentant les caractéristiques communes à tous les signaux. **Chaque signal** "SIGNALi" est traduit par **une autre classe** *SIGNALi* héritant de *SIGNAL*.





Condition Une condition est toujours considérée comme une pré-condition de l'opération qui modifie l'état d'un objet. Mais en Object-Z, il est possible de définir textuellement une opération lors de la composition de schémas d'opérations. Une condition peut donc être considérée comme une opération qui ne modifie pas de variable, mais qui impose certaines valeurs initiales lors de la conjonction. Par exemple, une condition spécifiant que l'attribut "x" d'une classe "Classe" doit avoir la valeur "x1" s'exprime par *[x = x1]* et peut être combinée avec des opérations leur ajoutant ainsi une pré-condition. Contrairement à la proposition de traduction en Z, cette solution a pour avantage de ne pas multiplier les schémas.

Proposition 27 : Condition
Toute condition "Cond" portant sur les attributs de "CLASSE" donne lieu à **un prédicat** *[Cond]* à ajouter dans la représentation de la transition sur laquelle elle porte.

Exemple B.10 *Comme en Z, la condition sur la transition entre "Soumise" et "Refusee" s'exprime par* $(dateLimiteSoumission, dateReception) \in DateInf$. *Pour pouvoir l'inclure dans la traduction de la transition, il faut la noter entre crochets.*

Transition Le changement d'état sur les transitions se traduit comme un Z dans **une opération qui modifie la variable représentant l'état**. Mais l'écriture de cette opération change par rapport à Z car en Object-Z, la liste des éléments modifiés par l'opération (Δ) comprend les variables à changer et non l'ensemble d'un schéma.

Si l'événement sur la transition est un appel d'opération, il faut ajouter à la liste de ses variables modifiées la variable d'état.

CLASSE

...

Op

$\Delta etatClasse$

...

$etatClasse = Etati \land etatClasse' = Etatj$

...

Si l'événement est un signal, une opération de changement d'état est créée. Elle a dans la liste de Δ la variable d'état. De plus, pour pouvoir regrouper ultérieurement les opérations de transitions, elle prend en entrée un objet de *SIGNAL* ou de ses sous-classes, la classe à laquelle il appartient étant précisée dans les prédicats ($s? \in SIGNALi$).

CLASSE

...

ModifierEtatEtatiEtatj

$\Delta etatClasse$

$s? : \downarrow SIGNAL$

$s? \in SIGNALi$

$etatClasse = Etati \land etatClasse' = Etatj$

Si nécessaire, la transition est toujours **la combinaison de ses différents éléments (événement, condition et action)**. Les éléments modifiés par une opération étant en Object-Z seulement des variables, il n'y a pas de risque d'incompatibilité entre deux schémas d'opérations si bien que l'opérateur de conjonction $\land$ peut être employé. Contrairement à Z, cette traduction exprime donc que les transitions sont atomiques. Il n'est donc pas nécessaire d'assouplir les invariants liant un état et les attributs de la classe. Ainsi le statut et l'état d'une soumission sont équivalents pour accepté et refusé :
$statut = accepte \Leftrightarrow etatSoumission = Acceptee$

Pour un événement de type appel d'opération, l'opération de transition est la conjonction de l'opération "Op", de la condition "Cond" et de l'action "A". De plus, l'opération de transition prend le nom de l'événement suffixé par ses états *OpEtatij* et l'appel d'opération correspondant est renommé *CLASSEOp*. Si l'appel d'événement n'apparaît qu'une seule fois dans le modèle dynamique, elle prend directement le nom de l'événement afin de pouvoir être appelée à partir d'un autre modèle dynamique.

$OpEtatij \mathrel{\widehat=} CLASSEOp \land [\ cond\ ] \land A$

Pour un signal, la transition s'exprime par :

$TransitionEtatij \mathrel{\widehat=} ModifierEtatij \land [\ cond\ ] \land A$

Enfin, pour pouvoir simuler l'envoi d'événements entre objets, nous regroupons les opérations de transitions dans une opération de choix de transitions et nous les rendons invisibles. Seules les opérations du modèles objet et l'opération de choix

restent accessibles aux autres classes. Ceci n'a pas été fait en Z puisque Z ne permet pas de préciser la visibilité d'une opération.

Regrouper les opérations de transitions correspond à créer **une nouvelle opération qui choisit la transition à exécuter**. Ce choix correspond à vouloir effectuer une disjonction entre les opérations de transitions. Mais en Object-Z, l'opérateur $\vee$ ne peut pas être utilisé entre des schémas d'opérations ; il existe seulement un opérateur de choix indéterministe [] . L'utilisation de [] pour exprimer le choix entre différentes transitions permet de décrire des modèles dynamiques indéterministes, mais il impose que les opérations aient toutes les mêmes entrées/sorties.

Dans le cas des transitions déclenchées par des signaux, le choix doit s'effectuer entre ces transitions. Or les opérations de transition sont constituées des opérations de changement d'état qui ont toutes la même signature, de conditions qui n'ont ni d'entrée, ni sortie et d'opérations représentant des actions. Les divergences de signatures entre les opérations de transition peuvent donc seulement apparaître à cause des actions. Pour ces actions, les valeurs des entrées doivent être données lors de l'appel de l'opération ; il est ensuite possible de cacher ces entrées. A priori, les seules sorties possibles représentent des informations exploitables par l'un des objets du système ; ce ne peut donc être que des événements à envoyer à d'autres objets. Pour que toutes les actions aient les mêmes sorties, nous proposons de ne pas considérer ces événements comme des sorties d'opérations, mais de supposer qu'ils sont mis dans une file d'attente d'événements qui est modifiée lors de l'action. Dans ce cadre, il est possible d'utiliser l'opérateur de choix indéterministe entre les opérations de transition :

$ChoixTransition \mathrel{\widehat=} TransitionEtat12$ [] $TransitionEtat23$ [] ... [] $TransitionEtatij$

De même si un appel d'opération "Op" correspond à plusieurs transitions, l'opération de choix s'effectue entre les différentes transitions possibles :

$Op \mathrel{\widehat=} OpEtat12$ [] $OpEtat23$ [] ... [] $OpEtatij$

Cette solution est satisfaisante du point de vue de la sémantique des transitions. De plus, les opérations de choix des transition permettent de préparer l'échange d'événements entre objets. En effet, l'envoi d'un événement à un objet correspond simplement à l'appel de son opération de choix de transition. Mais ces opérations de choix de transitions nécessitent que toutes les opérations de transition aient la même signature, ce qui complexifie la traduction.

Exemple B.11 *Pour le modèle dynamique de "SOUMISSION", la transition entre "IntentionRecue" et "Soumise" comporte seulement un événement "RecevoirSoumission" qui est une opération modifiant la date de réception, le titre et l'état d'une soumission.*

La transition entre "Soumise" et "Refusee" donne lieu à une opération de changement d'état ModifierSoumiseRefusee, une condition $\left[(dateLimiteSoumission, dateReception) \in DateInf\right]$ *et à une opération ModifierStatut correspondant à l'action de même nom. La conjonction de ces trois éléments représente la transition TransitionSoumiseRefusee. Lors de l'appel de l'action, la valeur d'entrée de s ? doit être "refusee". L'entrée doit aussi être cachée pour que toutes les transitions déclenchées par un signal aient la même*

Proposition 28 : Transition
Une transition entre les états "Etati" et "Etatj" comportant un événement "ev", une condition "Cond" et une action "A" se traduit dans la classe d'intension *CLASSE* par :

- **une opération effectuant le changement d'état** ;

CLASSEEv

$\Delta etatClasse$
...

$etatClasse = Etati \wedge$
$\quad etatClasse' = Etatj$
...

ModifierEtatij

$\Delta etatClasse$
$ev? : SIGNAL$

$ev? \in SIGNALi$
$etatClasse = Etati \wedge$
$\quad etatClasse' = Etatj$

- **la conjonction des opérations composant la transition** ;
  - pour une transition dont l'événement est un appel d'opération, l'opération de transition porte le nom de cette opération et les entrées éventuelles d'une action doivent être cachées.
    $OpEtatij \mathrel{\widehat=} CLASSEEv \wedge [\ cond\ ] \wedge (A \setminus (entrees\ ?))$
  - pour une transition déclenchée par un signal, les entrées éventuelles d'une action doivent être cachées :
    $TransitionEtatij \mathrel{\widehat=} ModifierEtatij \wedge [\ cond\ ] \wedge (A \setminus (entrees\ ?))$

Toutes les transitions déclenchées par des signaux sont regroupées dans **une opération de choix de la transition** à effectuer.
$ChoixTransition \mathrel{\widehat=} TransitionEtat12\ []\ TransitionEtat23\ []\ ...\ []\ TransitionEtatij$
Si un appel d'opération déclenche plusieurs transitions, une opération de choix portant le nom de l'opération est aussi créée :
$Op \mathrel{\widehat=} OpEtat12\ []\ OpEtat23\ []\ ...\ []\ OpEtatij$
La visibilité des propriétés de la classe est restreinte à l'opération de choix de transitions et aux opérations du modèle objet.
$\lceil(ChoixTransition, Op1, .., Opn)$

signature. Il est alors possible d'écrire l'opération ChoixTransition qui laisse un choix indéterministe entre les transitions déclenchées par un signal.

Enfin la classe SOUMISSION ne laisse visible à l'extérieur que les opérations de la classe telle que RecevoirSoumission ou AccorderDelai et l'opération de choix de transition ($\lceil(AccorderDelai; RecevoirSoumission; ChoixTransition)$).

$EtatSoumission ::=$
$\quad IntentionRecue \mid Soumise \mid AEvaluer \mid Refusee \mid Acceptee$

SOUMISSION

$\lceil (AccorderDelai; RecevoirSoumission; ChoixTransition)$

$titre : TITRE$
$dateReception : DATE$
$dateLimiteSoumission : DATE$
$statut : STATUT$
$etatSoumission : EtatSoumission$

ModifierStatut

$\Delta(statut)$
$s? : STATUT$

$statut' = s?$

RecevoirSoumission

$\Delta(dateReception, titre, etatSoumission)$
$d? : DATE$
$t? : TITRE$

$titre' = t? \land dateReception' = d?$
$etatSoumission = IntentionRecue \land etatSoumission' = Soumise$

ModifierSoumiseRefusee

$\Delta(etatSoumission)$
$s? : \downarrow SIGNAL$

$s? \in SIGNAL$
$etatSoumission = Soumise \land etatSoumission' = Refusee$

$TransitionSoumiseRefusee \mathrel{\widehat=} ModifierSoumiseRefusee \land$
$\quad [(dateLimiteSoumission, dateReception) \in DateInf\,] \land$
$\quad ([\, s? : STATUT \mid s? = refuse\,] \land ModifierStatut) \setminus (s?)$
$ChoixTransition \mathrel{\widehat=} TransitionSoumiseAEvaluer \;[]\; TransitionSoumiseRefusee \;[]$
$\quad TransitionAEvaluerAcceptee \;[]\; TransitionAEvaluerRefusee$

Traduction de la concurrence d'agrégat

La concurrence d'agrégat est une façon de combiner des modèles dynamiques : l'état de l'agrégat correspond aux états combinés des modèles dynamiques des composants et l'agrégat interagit avec et en fonction de ses composants. C'est, en fait, un cas particulier de conditions sur les liens d'associations et d'envois d'événements. Alors que nous avons restreint nos propositions aux conditions et actions portant sur l'intension d'une classe et que nous n'avons pas abordé les échanges d'événements, cette particularité peut être traitée du fait de notre règle de traduction de l'agrégation qui inclut une variable représentant les composants dans la classe agrégat. La concurrence d'agrégat s'exprime alors au niveau de l'intension de la classe agrégat.

Certains états de l'agrégat dépendent des valeurs de leurs composants. Il est

alors possible de préciser le lien entre l'état de l'agrégat et ses composants par un prédicat :

$etatAgregat = etat1 \Leftrightarrow etatComposant1 = etatC1 \wedge ... \wedge etatComposantn = etatCn$

Le fait que l'agrégat dépende de ses composants signifie aussi qu'il peut leur envoyer des événements pour modifier leur comportement et que les conditions sur les transitions peuvent porter sur des composants. L'agrégation se traduisant en Object-Z par l'inclusion d'une référence à chacun des composants dans l'intension de la classe agrégat, **une condition portant sur des composants** s'exprime comme une condition sur un attribut et **l'envoi d'un événement à un composant** se traduit par l'appel d'une opération *Ev* représentant un signal ou un appel d'opération. Pour préciser quel est le composant pour lequel l'événement est envoyé, il faut le spécifier par une expression de la forme $[\ c : COMPOSANT \mid ...] \bullet c.Ev$. L'opérateur $\bullet$ est sémantiquement équivalent à la conjonction d'opération, mais il permet d'utiliser les variables de la première opération dans la deuxième (ici, la sélection de c ou le choix du signal).

Si l'événement envoyé est un appel d'opération, l'opération correspondante fait partie des opérations composant la transition.

$TransitionEtatij \mathrel{\widehat=} ChangementEtat \wedge [\ cond\ ] \wedge (A \setminus (entrees\ ?)) \wedge [\ c : COMPOSANT \mid ...] \bullet c.Op$

Si l'agrégat envoie un signal à l'un de ces composants, il appelle l'opération de choix d'une transition du composant en précisant le type de signal à envoyer :

$TransitionEtatij \mathrel{\widehat=} ChangementEtat \wedge [\ cond\ ] \wedge (A \setminus (entrees\ ?)) \wedge ([\ s\ ? :\downarrow SIGNAL \mid s\ ? \in SIGNAL1\ ] \wedge [\ c : COMPOSANT \mid ...] \bullet c.ChoixTransition)$

La concurrence d'agrégat se traduit de façon assez simple en Object-Z grâce à la traduction de l'agrégation et des transitions. Mais ce n'est qu'un cas particulier de conditions sur les associations et les envois d'événements qui tire profit de la nature de l'agrégation.

Exemple B.12 *Reprenons le cas du comportement d'une session qui est constituée de présentations (Fig. B.14).*

Le comportement d'une session (Fig. B.15) dépend de celui de ses présentations composantes.

Dans cet exemple, l'état "Annulee" d'une session correspond à l'annulation de toutes les présentations prévues pour cette session.

$etatSession = Annulee \Leftrightarrow (\forall p : presentation \bullet p.etatPresentation = Annule)$

Pour la traduction des transitions, un exemple de condition portant sur un des composants est celle entre "Planifiee" et "Annulee" qui spécifie que si toutes les présentations sont annulées alors la session passe dans l'état "Annulee". La transition s'exprime en combinant ses différents composants : l'opération de changement d'état ModifierPlanifieeAnnulee et la condition :

$TransitionPlanifieeAnnulee \mathrel{\widehat=} ModifierPlanifieeAnnulee \wedge [\forall p : PRESENTATION \mid p \in presentation \bullet p.etatPresentation = Annule\ ]$

Enfin pour envoyer l'événement "DebutPresentation" à la classe "Presentation", on appelle l'opération ChoixTransition de PRESENTATION avec le signal Debut-

Proposition 29 : Concurrence d'agrégat

Soient *etatAgregat* l'état de l'agrégat, *etat1* une des valeurs de l'état de l'agrégat, *etatComposant1, etatComposantn* les états de deux composants quelconques de l'agrégat, *etatC1* une des valeurs de *etatComposant1* et *etatCn* une des valeurs de *etatComposantn*. Si **l'état agrégé** *etat1* est constitué d'**un état de chacun des modèles dynamiques des composants**, il faut exprimer un prédicat de la forme suivante dans *CLASSE* :

$etatAgregat = etat1 \Leftrightarrow$

$etatComposant1 = etatC1 \land ... \land etatComposantn = etatCn$

Une condition sur un des composants se traduit comme **une condition sur un attribut** (proposition 27).

Soient *TransitionEtatij* une transition du modèle dynamique de l'agrégat allant de l'état i à l'état j, *ChangementEtat* l'opération représentant le changement d'état de l'agrégat, *cond* la condition sur *TransitionEtatij*, *A* l'opération correspondant à l'action appelée sur la transition et *c* la variable représentant un composant de l'agrégat.

L'envoi d'un événement à un composant se traduit par l'appel d'une opération. Si l'événement envoyé est **un appel d'opération**, l'opération correspondante *Op* fait partie des opérations composant la transition.

$TransitionEtatij \mathrel{\widehat=} ChangementEtat \land [\ cond\ ] \land (A \setminus (entrees\ ?)) \land$
$[\ c : COMPOSANT \mid ...] \bullet (c.Op \setminus (entrees\ ?))$

Si l'agrégat envoie **un signal** à l'un de ces composants, il appelle **l'opération de choix d'une transition du composant** *ChoixTransition* en précisant le type de signal *SIGNALi* à envoyer :

$TransitionEtatij \mathrel{\widehat=} ChangementEtat \land [\ cond\ ] \land (A \setminus (entrees\ ?)) \land$
$([\ s? : \downarrow SIGNAL \mid s? \in SIGNALi\ ] \land [\ c : COMPOSANT \mid ...] \bullet$
$c.ChoixTransition)$

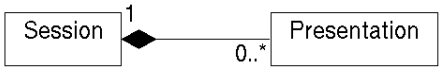


FIG. B.14: Composition de "Session"

Presentation. Sur cette transition, l'événement doit être envoyé à la première présentation c'est-à-dire celle dont l'heure de début est l'heure de début de la session :

$TransitionPlanifieeCommencee \mathrel{\widehat=} ModifierPlanifieeCommencee \land$
$[p : PRESENTATION; s? : \downarrow SIGNAL \mid p \in presentation \land$
$p.heureDebut = heureDebut \land s? \in DebutPresentation\ ] \bullet p.ChoixTransition$

Ces informations sont des compléments à la classe SESSION :

$EtatSession : : = Creee \mid Planifiee \mid Commencee \mid Terminee \mid Annulee$

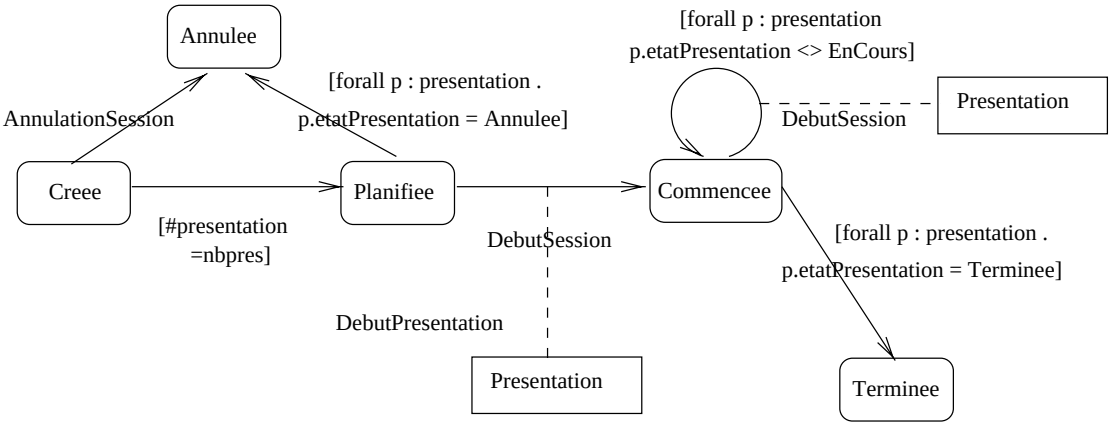


FIG. B.15: Modèle dynamique de "Session"

SESSION

$\uparrow(ChoixTransition)$

$titre : TITRE$
$heureDebut : HEURE$
$heureFin : HEURE$
$nbpres : \mathbb{N}$
$presentation : \mathbb{F}\, PRESENTATION$
$etatSession : EtatSession$

$etatSession = Annulee \Leftrightarrow$
$\quad (\forall\, p : presentation \bullet p.etatPresentation = Annule)$

ModifierPlanifieeAnnulee

$\Delta(etatSession)$
$s? : \downarrow SIGNAL$

$s? \in SignalInterne$
$(etatSession = Planifiee \land etatSession' = Annulee)$

ModifierPlanifieeCommencee

$\Delta(etatSession)$
$s? : \downarrow SIGNAL$

$s? \in DebutSession$
$(etatSession = Planifiee \land etatSession' = Commencee)$

$$
\begin{array}{l}
TransitionPlanifieeAnnulee \mathrel{\widehat=} ModifierPlanifieeAnnulee \land \\
\quad [\forall p : PRESENTATION \mid p \in presentation \\
\quad \bullet\ p.etatPresentation = Annule\ ] \\
TransitionPlanifieeCommencee \mathrel{\widehat=} ModifierPlanifieeCommencee \land \\
\quad [p : PRESENTATION; s? : \downarrow SIGNAL \mid p \in presentation \land \\
\quad p.heureDebut = heureDebut \land s? \in DebutPresentation\ ] \\
\quad \bullet\ p.ChoixTransition \\
ChoixTransition \mathrel{\widehat=} TransitionCreeePlanifiee\ []\ TransitionPlanifieeAnnulee \\
\quad []\ TransitionPlanifieeCommencee\ []\ TransitionCommencee\ [] \\
\quad TransitionCommenceeTerminee
\end{array}
$$

B.3.3 Généralisation

Extension des propositions

Dans les sections précédentes, nous avons présenté des propositions de traduction du modèle dynamique en Z et Object-Z sous l'hypothèse simplificatrice que toutes les actions ou conditions portaient sur l'intension de la classe dont le comportement est décrit par le modèle dynamique. Etendre ces propositions à tout type d'action ou de condition revient à considérer les cas d'une action ou d'une condition sur l'extension de la classe ou sur l'une de ses associations. Comme nous avons proposé de représenter les transitions par une opération qui correspond à la combinaison des différents éléments (événement, condition et action), la généralisation de nos propositions est un problème de promotion des opérations : une transition ne doit plus être vue seulement au niveau de l'intension de la classe, mais elle doit se situer au niveau des associations. Par exemple, si une transition doit déclencher une action portant un lien avec une autre classe, la transition doit se situer au niveau de l'association correspondant à ce lien. Ainsi lors que l'évaluation d'une soumission est connue, il faut déclencher une action qui prévient ses auteurs.

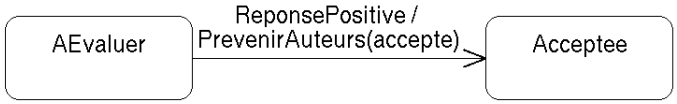


FIG. B.16: Exemple d'action sur un lien

Une transition étant composée de l'opération représentant le changement d'état, du schéma correspondant à la condition et du schéma d'opération de l'action, il faut promouvoir l'opération de changement d'état au niveau de l'action pour pouvoir appliquer l'opérateur de composition de schémas. Cette solution est réalisable en étudiant plus précisément les problèmes liés à la promotion des opérations, mais elle serait très lourde à réaliser compte tenu de la complexité de la promotion des opérations.

De plus, en Z, les transitions ne sont pas exprimées de façon satisfaisante puisque elles ne sont pas instantanées du fait de l'utilisation de l'opérateur de composition entre les opérations de changement d'état et d'action. Aussi nous avons pensé adopter

une solution totalement différente qui suggère une vision nouvelle de la complémentarité entre modèle objet et modèle dynamique.

Une autre solution

Contrairement aux propositions existantes qui voient le modèle dynamique comme un complément d'information au modèle objet, nous proposons de considérer les modèles objet et dynamique comme deux points de vue différents sur la modélisation d'un système (Fig B.17). Le modèle objet représente la structure statique du système tandis que le modèle dynamique ne modélise que les changements d'états. La spécification du système doit alors mettre en commun ces deux points de vue. Les modèles objet et dynamique sont donc traduits séparément en spécifications formelles, puis leur lien est aussi traduit dans un modèle global.

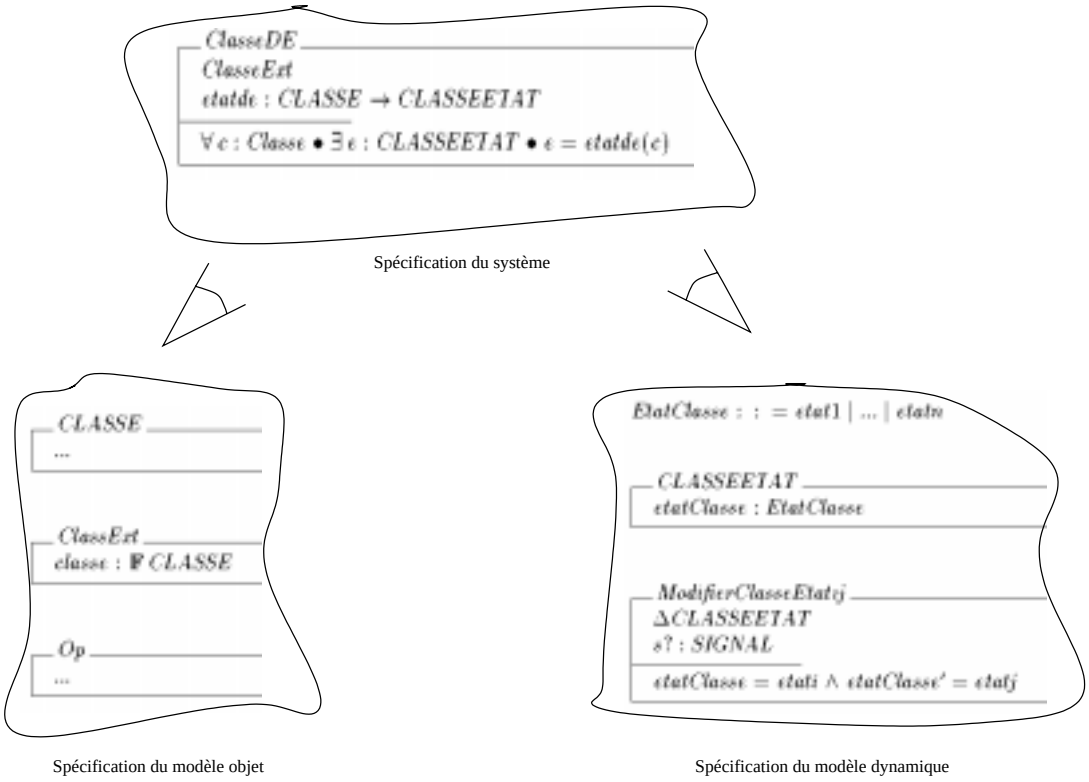
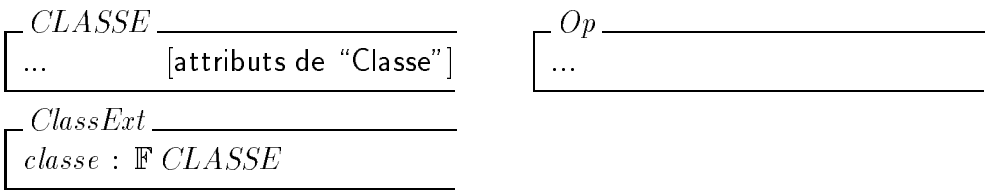


FIG. B.17: Vision de la répartition entre modèle objet et modèle dynamique

La traduction du modèle objet s'effectue suivant les règles que nous avons présenté dans la section 5.1. Pour une classe "Classe", on a donc des schémas pour son intension, son extension et ses opérations :



Pour ne pas avoir d'élément commun avec le modèle objet, la traduction du modèle dynamique ne comporte que des schémas représentant les états de chaque classe et les opérations de modification d'états. Par exemple, le modèle dynamique d'une classe "Classe" se traduirait en des spécifications Z suivantes :

$$EtatClasse ::= etat1 \mid ... \mid etatn$$

$$\begin{array}{|l} CLASSEETAT \\ \hline etatClasse : EtatClasse \\ \hline \end{array}$$

$$\begin{array}{|l} ModifierClasseEtatij \\ \hline \Delta CLASSEETAT \\ s? : SIGNAL \\ \hline etatClasse = etati \land etatClasse' = etatj \\ \hline \end{array}$$

Pour faire le lien entre les deux traductions, il suffit d'écrire que tout objet existant de la classe est lié à un état.

$$\begin{array}{|l} ClasseDE \\ \hline ClasseExt \\ etatde : CLASSE \pfun CLASSEETAT \\ \hline \forall c : Classe \bullet \exists e : CLASSEETAT \bullet e = etatde(c) \\ \hline \end{array}$$

Grâce à ce schéma, les spécifications formelles du modèle objet et du modèle dynamique sont liées : le schéma d'état *CLASSEETAT* est lié à l'extension de la classe *ClasseExt* qui est lui même lié aux schémas d'associations etc. Il est donc dorénavant possible de coupler des schémas d'opérations venant des deux modèles. Il est en particulier possible de représenter les transitions en faisant la conjonction d'une opération de changement d'état, d'une condition et d'une action. Il suffit alors de promouvoir tous les schémas d'une transition au niveau des associations.

Cette proposition n'est qu'à l'état d'ébauche, mais elle semble intéressante car elle offre la possibilité de bien séparer les deux points (statique et dynamique) d'un système et d'en spécifier explicitement les liens.

B.3.4 Comparaison

Comparaison entre les traductions en Z et Object-Z

Les traductions du modèle dynamique en Z et Object-Z suivent les mêmes principes. Néanmoins les propositions pour Object-Z sont plus complètes que celles pour Z. Nous ne pouvons donc pas les comparer en comptant le nombre de structures ou de lignes générées pour chacune des traductions. La comparaison porte plutôt sur les spécificités des traductions.

Etat Le concept d'état est représenté de façon totalement équivalente en Z et en Object-Z. Il n'y a donc pas différence à utiliser Z ou Object-Z.

Evénement En Object-Z comme en Z, nous avons distingué deux types d'événements : les appels d'opération et les signaux. Or si la représentation des appels d'opération est équivalente dans les deux langages, Object-Z offre une meilleure représentation du concept de signal en offrant une réelle notion de spécialisation entre les signaux.

Transition Bien que le principe de traduction des transitions soit le même en Object-Z qu'en Z, Object-Z complète cette traduction grâce à ses concepts spécifiques sur les opérations.

Tout d'abord pour représenter une transition, il est possible de faire la conjonction de deux opérations sans risque d'incompatibilité. Cela permet d'exprimer que les transitions sont atomiques et d'exprimer sans restrictions les liens entre l'état et les attributs.

Ensuite Object-Z dispose de deux opérateurs qui permettent une meilleure représentation sémantique du modèle dynamique. L'opérateur de choix indéterministe permet de spécifier des modèles dynamiques non déterministes. L'opérateur de visibilité $\restriction$ permet de ne pas rendre accessibles toutes les opérations intermédiaires pour les transitions. L'utilisation de ces deux opérateurs permet de spécifier un moteur de choix de transitions. Il est alors possible de décrire les échanges d'événements soit en appelant l'opération correspondante, soit en envoyant un signal à l'opération de choix de transitions.

Diagrammes d'états hiérarchiques La structuration hiérarchique d'un modèle dynamique n'introduit pas de différences notables entre Z et Object-Z. Les avantages à l'utilisation d'Object-Z sont ceux cités précédemment pour la traduction des transitions.

Concurrence d'agrégat La concurrence d'agrégat est un concept qui n'est pas exprimable de façon simple en Z, mais qui l'est en Object-Z. Cette particularité ne provient pas d'une limite particulière de Z pour représenter un des concepts de la concurrence d'agrégat, mais vient de la traduction de l'agrégation en Z et en Object-Z. En effet, l'agrégation s'exprime en Object-Z en introduisant une nouvelle variable représentant un composant dans l'intension de la classe de l'agrégat. Cela rend simple la définition de l'état de l'agrégat en fonction de ses composants dans la classe agrégée, l'envoi d'événement à un composant ou une condition portant sur un composant. La traduction de ces concepts est possible en Z, mais elle nécessite que nous supprimions l'hypothèse simplificatrice sur laquelle nous avons basé nos propositions.

Bilan Z et Object-Z sont équivalents pour représenter les états ou leur structuration des modèles dynamiques. Par contre Object-Z a un net avantage sur Z pour décrire des concepts tels que les transitions ou le non-déterminisme d'un modèle qui ont plutôt attrait à la dynamique car il est plus développé que Z en matière d'opérations et de leur combinaison.

Comparaison avec les travaux similaires

Pour comparer les concepts traités par rapport aux travaux similaires, nous reprenons le tableau récapitulant les propositions de traduction des diagrammes d'états-transitions en ajoutant ce travail.

Le travail B.2 montre que nous nous sommes attachés à exprimer la plupart des concepts du modèle dynamique dont des concepts complexes concernant la structu-

Concepts Articles	Etat	Evénement	Condition	Action	envoi d'événements	DE hiérachiques	DE concurrents
[LHW96]	T	T	T	T		A	A
[Ngu98]	T	T	T	T	T^-		
[MS99]	T	T	T	T	T		
[Lan95]	T	T	T	T	T	T	T^-
[DUP00]	T	T	T^-	T	A	T	

TAB. B.2: Synthèse des travaux sur le modèle objet

ration. Ces résultats peuvent être nuancés par l'hypothèse simplificatrice que nous avons fait au début de ce travail. Mais nous avons exposé comment nous pourrions étendre notre solution.

La principale originalité des propositions de traduction en Z et Object-Z est le fait de prendre en compte explicitement plusieurs types d'événements alors que les autres travaux se limitent aux appels d'opérations.

B.3.5 Conclusion

Cette annexe contient nos propositions de traduction du modèle dynamique en Z et Object-Z. Ces propositions sont limitées par le fait que nous ne considérons que les transitions dont les éléments portent sur l'intension de la classe. Nous avons évoqué comment elles pourraient être étendues pour supprimer cette hypothèse.

Mais compte tenu de la complexité de la solution prévue, nous avons envisagé une approche de traduction totalement différente qui n'est pour l'instant qu'à l'état d'ébauche. Son intérêt réside dans la représentation les modèles objet et dynamique comme deux points de vue différents d'un système dont le lien est défini formellement.

Comme pour le modèle objet, nous avons comparé Z et Object-Z comme langage formel cible de la traduction du modèle dynamique. Là encore nous avons conclu à une meilleure adaptation d'Object-Z grâce à sa gestion des opérations et de leur combinaison.

Enfin nous avons positionné ce travail par rapport aux travaux existants.

Annexe C

Preuves des opérations de base

C.1 Preuves sans contrainte

S'il n'existe aucune contrainte sur une classe, il n'y a, a priori, pas de gardes aux opérations de base. Les preuves de ces opérations sont alors automatiques. Par exemple, pour une opération d'ajout d'un élément, le théorème validant une garde correspond au théorème introduit dans la section 6.2. Mais dans le cas où il n'existe pas de contrainte, il n'y a pas de garde identifiée.

theorem ClasseOp_Pre

$$\forall\, ClasseExt; x? : CLASSE \bullet \text{pre } ClasseOp$$

Preuve de ClasseOp_Pre :
`prove by reduce ;`
true

Ainsi en considérant que "SESSION" n'a pas de contrainte, le théorème pour valider la garde de l'opération d'ajout d'une session *AjouterSession* se prouve automatiquement en Z-EVES.

theorem AjouterSession_Pre

$$\forall\, SessionExt; session? : SESSION \bullet \text{pre } AjouterSession$$

Preuve de AjouterSession_Pre :
`prove by reduce ;`
true

C.2 Preuves avec une contrainte sur l'intension de la classe

Dans cette section, nous étudions les preuves des opérations de base d'une classe lorsqu'il existe des contraintes sur un ou plusieurs attributs de cette classe. Nous supposons qu'il n'y a pas d'autres contraintes sur l'extension de la classe.

C.2.1 Opération de modification d'un attribut

Suivant notre démarche qui veut que la garde candidate à valider corresponde aux contraintes appliquées aux entrées de l'opération, il existe deux cas : soit l'opération modifie un attribut non contraint, soit elle change un attribut contraint. **Si**

l'opération modifie un attribut non contraint, il n'y a pas de garde candidate. On se trouve alors dans un cas équivalent à celui des preuves des opérations sans contrainte : le théorème de l'opération *CLASSEModifiera* modifiant l'attribut "a" au niveau de l'intension de "CLASSE" n'a pas de garde candidate et est prouvé automatiquement par Z-EVES :

theorem CLASSEModifiera_Pre

$$\forall\, CLASSE;\ e? : ENTREE \bullet \text{pre}\ CLASSEModifiera$$

Preuve de CLASSEModifiera_Pre :
`prove by reduce ;`
true

Le théorème correspondant à l'opération promue *ClasseModifiera* n'a lui non plus pas de garde candidate puisque nous avons supposé qu'il n'existait pas de contrainte sur l'extension de "CLASSE" :

theorem ClasseModifiera_Pre

$$\forall\, CLASSE;\ ClasseExt;\ x? : CLASSE;\ e? : ENTREE \mid x? \in Classe \land \theta CLASSE = x? \bullet \text{pre}\ ClasseModifiera$$

Preuve de ClasseModifiera_Pre :
`prove by reduce ;`
true

Par exemple, la classe "SESSION" a deux contraintes sur son intension. La première limite le prix d'une session à 5000 FF et la deuxième spécifie que le début de la session a lieu avant sa fin. L'opération qui modifie le titre d'une session n'est donc pas concernée par les contraintes. Etant donné que nous ne tenons pour l'instant pas compte des contraintes sur l'extension, les preuves au niveau de l'intension et de l'extension de "SESSION" sont automatiques. Au niveau de l'intension, le théorème à prouver *SESSIONModifierTitre_Pre* spécifie que *nvtitre ?* doit être de type *TYPE*, mais il n'y a pas de garde particulière sur le nouveau titre :

theorem SESSIONModifierTitre_Pre

$$\forall\, SESSION;\ nvtitre? : TITRE \bullet \text{pre}\ SESSIONModifierTitre$$

Preuve de SESSIONModifierTitre :
`prove by reduce ;`
true

Le théorème correspondant à l'opération promue *SessionModifierTitre* tient aussi compte de la session donnée en entrée. Mais comme nous ne contraignons pas l'extension de "SESSION", le théorème de *SessionModifierTitre* n'a pas non plus de pré-condition identifiée :

theorem SessionModifierTitre_Pre

$$\forall\, SESSION;\ SessionExt;\ nvtitre? : TITRE;\ x? : SESSION \mid x? \in Session \land \theta SESSION = x? \bullet \text{pre}\ SessionModifierTitre$$

Preuve de SessionModifierTitre_Pre :
`prove by reduce ;`
true

Dans le cas où l'opération modifie un attribut contraint, le théorème correspondant à l'opération a souvent pour garde identifiée que la nouvelle valeur de l'attribut vérifie la contrainte :

theorem CLASSEModifiera_Pre

$$\forall\, CLASSE; e? : ENTREE \mid contrainte(e?) \bullet \text{pre } CLASSEModifiera$$

theorem ClasseModifiera_Pre

$$\forall\, CLASSE; ClasseExt; x? : CLASSE; e? : ENTREE \mid x? \in Classe \land \theta CLASSE = x? \land contrainte(e?) \bullet \text{pre } ClasseModifiera$$

La preuve du théorème dépend alors de la forme de la contrainte. Si la contrainte ne contient pas d'opérateur logique ou si ses parties sont liées par un "ET", la preuve est automatique. Sinon s'il existe un "OU", une implication ou une équivalence entre les parties d'une contrainte, la preuve doit considérer chacun des cas contenus dans la contrainte.

Le premier cas est illustré par la contrainte sur le prix d'une session. Pour l'opération modifiant le prix, la garde identifiée exprime que le nouveau prix vérifie la contrainte limitant sa valeur à 5000FF. La preuve du théorème *SESSIONModifierPrix_Pre* validant cette garde au niveau de l'intension de "SESSION" est automatique.

theorem SESSIONModifierPrix_Pre

$$\forall\, SESSION; nvprix? : \mathbb{N} \mid nvprix? < 5000 \bullet \text{pre } SESSIONModifierPrix$$

Preuve de SESSIONModifierPrix :
`prove by reduce;`
true

Comme il n'existe pas de contrainte sur l'extension de "SESSION", la preuve de l'opération *SESSIONModifierPrix_Pre* promue au niveau de l'extension est aussi automatique :

theorem SessionModifierPrix_Pre

$$\forall\, SESSION; SessionExt; nvprix? : \mathbb{N}; x? : SESSION \mid x? \in Session \land \theta SESSION = x? \land nvprix? < 5000 \bullet \text{pre } SessionModifierPrix$$

Preuve de SessionModifierPrix_Pre :
`prove by reduce;`
true

Pour "SESSION", la contrainte $heureFin.hr > heureDebut.hr \lor (heureFin.hr = heureDebut.hr \land heureFin.min > heureDebut.min)$ illustre le cas d'une contrainte complexe dont les parties sont liées par une disjonction. Elle va influencer les théorèmes des opérations de modification des heures de début et de fin. Ainsi l'opération *SESSIONModifierHeureDebut* qui modifie l'heure de début d'une session a pour garde identifiée que la nouvelle heure vérifie la contrainte sur les heures dont l'expression $(heureFin, nvheureDebut?) \in EstSup)$ correspond à $heureFin.hr > nvheureDebut?.hr \lor (heureFin.hr = nvheureDebut?.hr \land heureFin.min > nvheureDebut?.min)$:

theorem SESSIONModifierHeuredebut_Pre

$$\forall SESSION; nvheureDebut? : HEURE \mid$$
$$heureFin.hr > nvheureDebut?.hr \lor$$
$$(heureFin.hr = nvheureDebut?.hr \land heureFin.min > nvheureDebut?.min)$$
$$\bullet \text{ pre } SESSIONModifierHeuredebut$$

En Z-EVES, la preuve de *SESSIONModifierHeuredebut_Pre* comprend 16 étapes qui correspondent aux combinaisons possibles des heures de début et de fin :

```
try lemma SESSIONModifierHeuredebut_Pre ;
prove by reduce ;
split heureFin.hr > heureDebut.hr ;
cases ; – cas 1 : avant l'opération, l'heure de l'heure de début est inférieure à l'heuree de l'heure de fin
prove by reduce ;
split heureFin.hr > nvheureDebut ?.hr ;
cases ; – cas 1.1 : l'heure de la nouvelle heure de début est inférieure à l'heure de l'heure de fin
prove by reduce ;
          true
next ; – cas 1.2 : l'heure de la nouvelle heure de début n'est pas inférieure à l'heure de l'heure de fin
prove by reduce ;
next ; – cas 2 : l'heure de l'heure de début n'est pas inférieure à l'heuree de l'heure de fin
prove by reduce ;
split heureFin.hr=nvheureDebut?.hr ∧ heureFin.min>nvheureDebut ?.min ;
cases ;  – cas 2.1 : l'heure de l'heure de fin est égale à la nouvelle heure de l'heure de début, mais ses minutes sont supérieures
prove by reduce ;
          true
next ; – cas 2.2 : l'heure de fin n'est pas égale à la nouvelle heure de début ou ses minutes ne sont pas supérieures
prove by reduce ;
          true
next ;
          tous les cas ont été traités.
```

Le théorème *ClasseModifiera_Pre* correspondant à l'opération de modification promue au niveau de l'extension *ClasseModifiera* a la même garde candidate que celle de l'opération sur l'intension puisque nous supposons qu'il n'existe pas de contrainte sur l'extension de "CLASSE". En Z-EVES, la preuve de *ClasseModifiera_Pre* est alors identique à celle de l'opération sur l'intension.

theorem ClasseModifiera_Pre

$$\forall CLASSE; ClasseExt; x? : CLASSE; e? : ENTREE \mid$$
$$x? \in Classe \land \theta CLASSE = x? \land contrainte(e?) \bullet \text{ pre } ClasseModifiera$$

Par exemple, l'opération *SESSIONModifierHeureDebut* promue au niveau de l'extension de "SESSION" contient aussi la pré-condition $(heureFin.hr > nvheureDebut?.hr \lor (heureFin.hr = nvheureDebut?.hr \land heureFin.min > nvheureDebut?.min)$. Sa preuve est identique à celle de *SESSIONModifierHeureDebut_Pre* car il n'y a pas de contrainte sur l'extension de "SESSION".

theorem SessionModifierHeureDebut_Pre

$$\forall SESSION; SessionExt$$
$$; nvheureDebut? : HEURE; x? : SESSION \mid$$
$$x? \in Session \land \theta SESSION = x? \land (heureFin.hr > nvheureDebut?.hr \lor$$
$$(heureFin.hr = nvheureDebut?.hr \land heureFin.min > nvheureDebut?.min))$$
$$\bullet \text{ pre } SessionModifierHeuredebut$$

C.2.2 Opération d'ajout ou de suppression d'un objet

Comme nous suppposons qu'il n'existe pas de contrainte sur l'extension de la classe considérée, les opérations d'ajout et de suppression d'un objet n'ont pas de garde identifiée. Leur théorème de validation d'une garde s'écrit comme s'il n'y avait pas de contrainte et leur preuve est automatique :

theorem ClasseOp_Pre

$$\forall ClasseExt; x? : CLASSE \bullet$$
$$\text{pre } ClasseOp$$

Preuve de ClasseOp_Pre :
`prove by reduce ;`
true

Ainsi le théorème pour valider la pré-condition de l'opération d'ajout d'une session *AjouterSession* n'est pas modifié par rapport au cas des opérations sans contrainte :

theorem AjouterSession_Pre

$$\forall SessionExt; session? : SESSION \bullet$$
$$\text{pre } AjouterSession$$

Preuve de AjouterSession_Pre :
`prove by reduce ;`
true

C.3 Preuves avec contrainte sur l'extension de la classe

Nous étudions maintenant les preuves des théorèmes de validation de gardes des opérations de base dans le cas où il existe une contrainte de clé sur l'extension de la classe. Comme précédemment, nous supposons que le modèle ne comporte pas d'autre contrainte, en particulier sur l'intension de la classe.

C.3.1 Opération de modification d'un attribut

La contrainte de clé portant sur l'extension d'une classe, elle n'influence pas le théorème de validation d'une opération de modification d'un attribut sur l'intension *CLASSEModifiera*. La preuve de *CLASSEModifiera_Pre* est toujours automatique.

theorem CLASSEModifiera_Pre

$$\forall\, CLASSE; e? : ENTREE \bullet \text{pre}\ CLASSEModifiera$$

Preuve de CLASSEModifiera_Pre :
`prove by reduce ;`
true

Ainsi l'opération modifiant le prix d'une session *SESSIONModifierPrix* n'a pas de garde identifiée et la preuve du théorème *SESSIONModifierPrix_Pre* est automatique.

theorem SESSIONModifierPrix_Pre

$$\forall\, SESSION; nvprix? : \mathbb{N} \bullet \text{pre}\ SESSIONModifierPrix$$

Preuve de SESSIONModifierPrix :
prove by reduce ;
true

Lors de la promotion au niveau de l'extension de la classe des opérations de modification des attributs, il faut tenir compte de la contrainte de clé qui exprime une propriété quantifiée sur c1 et c2. Si l'attribut à modifier ne fait pas partie de la clé, il n'y a pas de garde identifiée pour l'opération :

theorem ClasseOp_Pre

$$\forall\, CLASSE; ClasseExt; x? : CLASSE; e? : ENTREE \mid x? \in Classe \land \theta CLASSE = x? \bullet \text{pre}\ ClasseOp$$

La preuve de *ClasseOp_Pre* est néanmoins influencée par la contrainte de clé. Z-EVES distinguant deux éléments *c1* et *c2*, la preuve tient compte du fait que l'un ou l'autre est l'élément modifié. Elle s'effectue en 16 pas de preuves qui se décomposent en trois cas : `prove by reduce ;`

`split c1 =` $\theta CLASSE[nva?/a]$;
`cases ;` – cas 1 : c1 est l'élément modifié
`instantiate` $c1\_0 == \theta CLASSE, c2\_0 == c2$;
`prove by reduce ;`
true
`next ;` – cas 2 : c1 n'est pas l'élément modifié
`split c2 =` $\theta CLASSE[nva?/a]$;
`cases ;` – cas 2.1 : c2 est l'élément modifié
`simplify ;`
`instantiate` $c1\_0 == c1, c2\_0 == \theta CLASSE$;
`prove by reduce ;`
true
`next ;` – cas 2.2 : c2 n'est pas l'élément modifié
`prove by reduce ;`
`instantiate` $c1\_0 == c1, c2\_0 == c2$;
`prove by reduce ;`
true
`next ;`
tous les cas ont été traités.

Par exemple, l'opération de modification du prix d'une session promue *SessionModifierPrix* a pour théorème :

theorem SessionModifierPrix_Pre

$$\forall SESSION; SessionExt; nvprix? : \mathbb{N}; x? : SESSION \mid$$
$$x? \in Session \land \theta SESSION = x? \bullet \text{pre } SessionModifierPrix$$

La preuve de *SessionModifierPrix_Pre* suit le modèle de *ClasseOp_Pre* pour lequel les cas correspondent au fait que l'un des éléments traités est l'élément pour lequel on change le prix.

```
prove by reduce ;
split s1 = θSESSION[nvprix?/prix] ;
cases ; – cas 1 : s1 est l'élément modifié
instantiate s1__0 == θSESSION, s2__0 == s2 ;
prove by reduce ;
          true
next ; – cas 2 : s1 n'est pas l'élément modifié
split s2 = θSESSION[nvprix?/prix] ;
cases ; – cas 2.1 : s2 est l'élément modifié
instantiate s1__0 == s1, s2__0 == θSESSION ;
prove by reduce ;
          true
next ; – cas 2.2 : s2 n'est pas l'élément modifié
prove by reduce ;
instantiate s1__0 == s1, s2__0 == s2 ;
prove by reduce ;
          true
next ;
          tous les cas ont été traités.
```

Si l'attribut modifié par l'opération fait partie de la clé, l'opération *ClasseModifiera* a une garde identifiée du type $\forall y : Classe \bullet y.a \neq nva? \lor y.ai \neq x?.ai \lor ....$ Si la clé a un seul élément "a", le théorème *ClasseModifiera_Pre* à prouver est :

theorem ClasseModifiera_Pre

$$\forall CLASSE; ClasseExt; nva? : T; x? : PERSON \mid$$
$$x? \in Classe \land \theta CLASSE = x? \land (\forall y : Classe \bullet y.a \neq nva?)$$
$$\bullet \text{ pre } ClasseModifiera$$

Comme la contrainte de clé porte sur deux éléments quelconques, le prouveur va différencier deux éléments *c1* et *c2* de "CLASSE". Si l'un de ces éléments est l'élément dont on modifie la clé, la garde sert à prouver que la nouvelle valeur de l'attribut clé est bien unique. La preuve est alors légèrement différente de celle identifiée précédemment, mais ses cas sont les mêmes.

```
prove by reduce ;
split c1 = θCLASSE[nva?/a] ;
cases ;  – cas 1 : c1 est l'élément modifié
instantiate c == c2 ;
prove by reduce ;
```

true
`next ;` – cas 2 : c1 n'est pas l'élément modifié
`split c2 =` $\theta CLASSE[nva?/a]$;
`cases ;` – cas 2.1 : c2 est l'élément modifié
`instantiate c == c1 ;`
`prove by reduce ;`
true
`next ;` – cas 2.2 : c1 et c2 ne sont pas l'élément modifié
`prove by reduce ;`
`instantiate` $c1\_0 == c1, c2\_0 == c2$;
`prove by reduce ;`
true
`next ;`
tous les cas ont été traités

Par exemple, si on considère que la clé de "SESSION" est son titre, la garde candidate spécifie que toutes les sessions existantes doivent avoir un titre différent du nouveau titre ($\forall s : Session \bullet s.titre \neq nvtitre?$).

theorem SessionModifierTitre_Pre

$$\forall SESSION; SessionExt; nvtitre? : TITRE; x? : SESSION \mid x? \in Session \land \theta SESSION = x? \land (\forall s : Session \bullet s.titre \neq nvtitre?) \bullet \text{pre } SessionModifierTitre$$

Sa preuve s'effectue suivant le fait que l'une des deux sessions est celle dont la clé est modifiée. En appliquant la preuve de *ClasseModifiera_Pre* à *SessionModifierTitre_Pre*, on obtient la preuve suivante :

`prove by reduce ;`
`split s1 =` $\theta SESSION[nvtitre?/titre]$;
`cases ;` – cas 1 : s1 est l'élément modifié
`instantiate s == s2 ;`
`prove by reduce ;`
true
`next ;` – cas 2 : s1 n'est pas l'élément modifié
`split s2 =` $\theta SESSION[nvtitre?/titre]$;
`cases ;` – cas 2.1 : s2 est l'élément modifié
`instantiate s == s1 ;`
`prove by reduce ;`
true
`next ;` – cas 2.2 : s1 et s2 ne sont pas l'élément modifié
`prove by reduce ;`
`instantiate` $s1\_0 == s1, s2\_0 == s2$;
`prove by reduce ;`
true
`next ;`
tous les cas ont été traités

Si la clé est composée de plusieurs attributs (comme c'est le cas normalement

pour "SESSION"), la garde identifiée spécifie que la valeur de l'un des composants clé de l'élément modifié doit le différencier des autres objets. Par exemple, la clé de "SESSION" est constituée des attributs "conference" et "titre". Pour l'opération qui modifie le titre *SessionModifierTitre*, l'élément modifié se distingue des autres soit par sa conférence, soit par son nouveau titre.

theorem SessionModifierTitre_Pre

$$\forall SESSION; SessionExt; nvtitre? : TITRE; x? : SESSION \mid$$
$$x? \in Session \land \theta SESSION = x? \land$$
$$(\forall s : Session \bullet s.conference \neq x?.conference \lor s.titre \neq nvtitre?)$$
$$\bullet \text{ pre } SessionModifierTitre$$

La preuve se complexifie (20 pas) pour prendre en compte les différents cas. Il faut donc considérer les cas où deux objets *s1* et *s2* se différencient par leur conférence et celui où ils ont des titres différents :

cas 2.1 : s2 est l'élément dont le titre est changé.

`prove by reduce ;`
`split s1 =` $\theta SESSION[nvtitre?/titre]$;
`cases ;` – cas 1 : s1 est l'élément dont le titre est changé
`instantiate s == s2 ;`
`prove by reduce ;`
rq : l'analyse par cas doit s'effectuer suivant le premier élément de la clé
`split s2.conference = conference ;`
`cases ;` – cas 1.1 : la conference de l'élément modifié est la même que celle de s2
simplify ;
true
`next ;` – cas 1.2 : la conference de l'élément modifié est differente que celle de s2
`simplify ;`
true
`next ;` – cas 2 : s1 n'est pas l'élément dont le titre est changé
`split s2 =` $\theta SESSION[nvtitre?/titre]$;
`cases ;` – cas 2.1 : s2 est l'élément dont le titre est changé
`instantiate s == s1 ;`
`prove by reduce ;`
true
`next ;` – cas 2.2 : s1 et s2 ne sont pas l'élément dont le titre est changé
`prove by reduce ;`
`instantiate` $s1\_0 == s1, s2\_0 == s2$;
`prove by reduce ;`
true
`next ;`
tous les cas ont été traités

C.3.2 Opération d'ajout d'un objet

Pour ajouter un objet *o ?* à une classe, cet objet doit respecter les contraintes sur l'ensemble des objets de la classe. Ainsi dans le cas d'une contrainte de clé, *o ?*

doit se différencier des autres objets par la valeur d'au moins l'un des attributs clés $ac1, ac2, ..., acn$.

theorem AjouterClasse_Pre

$$\forall\, ClasseExt; o? : CLASSE \mid (\forall\, x : Classe \bullet o?.ac1 \neq x.ac1 \lor o?.ac2 \neq x.ac2 \lor ... \lor o?.acn \neq x.acn) \bullet \text{pre } AjouterClasse$$

La preuve "de base" (clé ayant un seul composant) de *AjouterClasse_Pre* s'effectue en 15 pas qui correspondent à trois cas suivant que *c1* et *c2* sont ou non des objets de la classe :

`prove by reduce ;`
`instantiate` $c1\_\_0 == c1, c2\_\_0 == c2$`;`
`prove by reduce ;`
`split` $c1 \in Classe \land c2 \in Classe$ `;`
`cases ;` – cas 1 : c1 et c2 sont des objets de Classe
`prove by reduce ;`
true
`next ;` – cas 2 : c1 et c2 ne sont pas des objets de Classe
`split c1` $\in$ `Classe ;`
`cases ;` – cas 2.1 : c1 est un objet de Classe alors que c2 n'en est pas un
`instantiate x == c1 ;`
`with enabled` $(CLASSE\$member)$ `prove by reduce ;`
true
`next ;` – cas 2.2 : c1 n'est pas un objet de Classe
`instantiate` $x\_\_0 == c2$ `;`
`with enabled` $(CLASSE\$member)$ `prove by reduce ;`
true
`next ;`
tous les cas ont été traités.

Ainsi si on considère que la clé de "SESSION" est son titre, la garde identifiée est $\forall\, s : Session \bullet session?.titre \neq s.titre$. Sa preuve suit exactement le modèle de *AjouterClasse*.

theorem AjouterSession_Pre

$$\forall\, SessionExt; session? : SESSION \mid \forall\, s : Session \bullet session?.titre \neq s.titre \bullet \text{pre } AjouterSession$$

La preuve de *AjouterSession_Pre* suit exactement le schéma décrit précédemment pour AjouterClasse_Pre :

`prove by reduce ;`
`instantiate` $s1\_\_0 == s1, s2\_\_0 == s2$ `;`
`prove by reduce ;`
`split s1` $\in$ `Session` $\land$ `s2` $\in$ `Session ;`
`cases ;` – cas 1 : s1 et s2 sont des objets de Session
`prove by reduce ;`

true

`next ;` – cas 2 : c1 et c2 ne sont pas des objets de Classe

`split s1 ∈ Session ;`

`cases ;` – cas 2.1 : s1 est une session de la base alors que s2 n'en est pas une

`instantiate s == s1 ;`

`with enabled` $(SESSION\$member)$ `prove by reduce ;`

true

`next ;` – cas 2.2 : s1 n'est pas une session de la base

`instantiate` s_0 `== s2 ;`

`with enabled` $(SESSION\$member)$ `prove by reduce ;`

true

`next ;`

tous les cas ont été traités.

Si la clé est constituée de plusieurs attributs, la preuve se complexifie en ajoutant des cas, chaque cas correspondant à la différenciation de l'objet à ajouter par un attribut. Par exemple, pour ajouter un objet de session, dans le cas où sa clé est constituée de "conference" et de "titre", il faut distinguer les cas où l'objet à ajouter se distingue par sa conférence ou par son titre :

`prove by reduce ;`

`instantiate` $s1\_0 == s1, s2\_0 == s2$ `;`

`prove by reduce ;`

`split s1 ∈ Session ∧ s2 ∈ Session ;`

`cases ;` – cas 1 : s1 et s2 sont des sessions de la base

`prove by reduce ;`

true

`next ;` – cas 2 : s1 et s2 ne sont pas des sessions de la base

`simplify ;`

`split s1 ∈ Session ;`

`cases ;` – cas 2.1 : s1 appartient à la base

`instantiate s == s1 ;`

`prove by reduce ;`

`rq : l'analyse par cas doit s'effectuer suivant le premier élément cité dans la clé.`

`split s1.conference = session?.conference ;`

`cases ;` – cas 2.1.1 : s1 appartient à la base et s1 et l'élément à modifier session ? ont des dates identiques

`simplify ;`

true

`next ;` – cas 2.1.2 : s1 appartient à la base et s1 et l'élément à modifier session ? ont des dates différentes.

`with enabled` $(SESSION\$member)$ `prove by reduce ;`

true

`next ;` – cas 2.2 : s1 et s2 n'appartiennent pas à la base

`instantiate` $s\_0 == s2$ `;`

`prove by reduce ;`

true

```
next ;
```

tous les cas ont été traités

C.3.3 Opération de suppression d'un objet

Etant donné que nous ne tenons pas compte des cardinalités des associations, il n'existe pas de contrainte particulière pour pouvoir supprimer un objet. Il n'y a donc pas de garde identifiée à l'opération *SupprimerClasse_Pre*. Sa preuve s'effectue en 3 pas et est identique quelque soit la clé de "CLASSE".

theorem SupprimerClasse_Pre

$\forall$ *ClasseExt*; *o*? : *CLASSE* $\bullet$
pre *SupprimerClasse*

Preuve de *SupprimerClasse_Pre* :

```
prove by reduce ;
instantiate
   c1__0 == c1 , c2__0== c2 ;
simplify ;
```

true

Ainsi l'opération de suppression d'une session a pour théorème :

theorem SupprimerSession_Pre

$\forall$ *SessionExt*; *session*? : *SESSION*
$\bullet$ pre *SupprimerSession*

Preuve de *SupprimerSession_Pre* :

```
prove by reduce ;
instantiate
   s1__0 == s1 , s2__0== s2 ;
simplify ;
```

true

Annexe D

Vérification de cohérence par méta-modélisation

Cette annexe décrit les DTDs des modèles pour lesquels nous avons défini des règles de cohérence. Elle présente ensuite des règles de recouvrement pour des concepts des divers diagrammes UML autres que les diagrammes de classes.

D.1 Structure des modèles

D.1.1 Structure des modèles semi-formels

La structure des modèles semi-formels correspond en fait aux DTD qui ont été défini par [SY98] pour les diagrammes d'UML. Le DTD suivant décrit les élements communs à différents diagrammes et fait référence aux DTDs de chacun d'eux. Ainsi un modèle a un nom et est consitué d'un certain nombre de paquetages (< !ELEMENT Model (TaggedValue ?, Package*)>).

< !– Junichi Suzuki
suzuki@yy.cs.keio.ac.jp
http ://www.yy.cs.keio.ac.jp/ suzuki/project/uxf/
Id : *uml.dtd*1.91998/05/2013 : 04 : 36*junExpjun* –>
< !– extended with id attribute, Ernst Ellmer, 23.11.1998 –>
< !ENTITY % id "ID CDATA #REQUIRED">
< !ELEMENT Model (TaggedValue ?, Package*)>
< !ELEMENT TaggedValue (Tag*)>
< !ELEMENT Tag (#PCDATA—Value)*>
< !ELEMENT Value (#PCDATA)>
< !ELEMENT Note (id, CLink*)>
< !ELEMENT id (#PCDATA)>
< !ELEMENT CLink (Obj-href, Rule-href)>
< !ELEMENT Obj-href (#PCDATA)>
< !ELEMENT Rule-href (#PCDATA)>
< !ELEMENT Package (TaggedValue ?, Note*, Dependency*, ClassDiagram ?, CollaborationDiagram ?, StatechartDiagram ?, ClassDescription ?)>

<!ATTLIST Package
NAME CDATA #REQUIRED>
<!ENTITY % ObjectElements "(TaggedValue?, (Attribute —Operation —Generalization —Association —Dependency —Note)*)">
<!ENTITY % ClassDiagram SYSTEM "class_diagram.dtd">
%ClassDiagram;
<!ENTITY % CollaborationDiagram SYSTEM "collaboration_diagram.dtd">
%CollaborationDiagram;
<!ENTITY % StatechartDiagram SYSTEM "statechart_diagram.dtd">
%StatechartDiagram;
<!ENTITY % ClassDescription SYSTEM "Class_description.dtd">
%ClassDescription;

Structure du modèle objet

La structure du modèle objet correspond au DTD qui a été défini pour le diagramme de classes d'UML.

<!– Junichi Suzuki
suzuki@yy.cs.keio.ac.jp
http ://www.yy.cs.keio.ac.jp/ suzuki/project/uxf/
$Id : class_diagram.dtd 1.3 1998/05/20 13 : 05 : 05 jun Exp jun$ –>
<!ELEMENT ClassDiagram (TaggedValue?, (Class —Interface —Note)*)>
<!ELEMENT Class %ObjectElements;>
<!ELEMENT Interface %ObjectElements;>
<!ATTLIST Class
%id;
NAME CDATA #REQUIRED
ABSTRACT (true—false) "false"
VISIBILITY (public—private) #REQUIRED
ACTIVE (true—false) #IMPLIED>
<!ELEMENT Attribute (Note*)>
<!ATTLIST Attribute
%id;
VISIBILITY (public—protected—private) #REQUIRED
TYPE CDATA #REQUIRED
NAME CDATA #REQUIRED
INITVAL CDATA #IMPLIED
CONSTRAINT CDATA #IMPLIED
DERIVATION (true—false) "false"
CLASSSCOPE (true—false) "false">
<!ELEMENT Operation ((Parameter—Exception—Note)*)>
<!ATTLIST Operation
%id;
VISIBILITY (public—protected—private) #REQUIRED
NAME CDATA #REQUIRED

RETURN CDATA #REQUIRED
CLASSSCOPE (true—false) "false"
CONCURRENCY (sequential—guarded—concurrent) "sequential"
EXCEPTION CDATA #IMPLIED>
<!ELEMENT Parameter EMPTY>
<!ATTLIST Parameter
%id;
NAME CDATA #REQUIRED
TYPE CDATA #IMPLIED
DEFAULTVAL CDATA #IMPLIED
DIRECTION (in—out—inout) #IMPLIED>
<!ELEMENT Exception EMPTY>
<!ATTLIST Exception
%id;
NAME CDATA #REQUIRED
BODY CDATA #IMPLIED>
<!ELEMENT Generalization EMPTY>
<!ATTLIST Generalization
%id;
FROM CDATA #REQUIRED
TYPE (public—private—protected) "public">
<!ELEMENT Association ((AssocRole, PeerAssocRole)— Note*)>
<!ATTLIST Association
%id;
PEER CDATA #REQUIRED
NAME CDATA #IMPLIED>
<!ELEMENT AssocRole EMPTY>
<!ATTLIST AssocRole
%id;
MULTIPLICITY CDATA #IMPLIED
ORDERING (ordered—unordered) #IMPLIED
QUALIFIER CDATA #IMPLIED
ROLENAME CDATA #IMPLIED
NAVIGABILITY (true—false) "false"
CHANGEABILITY (true—frozen—addOnly) "true"
ASSOCCLASS CDATA #IMPLIED
AGGREGATION (none—aggregate—composite) "none"
AGGRKIND (unShared—shared) "unShared">
<!ELEMENT PeerAssocRole EMPTY>
<!ATTLIST PeerAssocRole
%id;
MULTIPLICITY CDATA #IMPLIED
ORDERING (ordered—unordered) #IMPLIED
ROLENAME CDATA #IMPLIED>
<!ELEMENT Dependency (Note*)>
<y!ATTLIST Dependency

%id ;
PEER CDATA #REQUIRED
NAME CDATA #IMPLIED
DESCRIPTION CDATA #IMPLIED
DEPKIND (refine—bind) #IMPLIED>

D.2 Structure de Z

Le DTD de Z est une extension du Z Interchange format.

<!– Z Interchange Format adapted to XML by S.Dupuy 20-06-1999 –>
<!ENTITY % id "ID ID #REQUIRED">
<!ENTITY % name "NAME CDATA #IMPLIED">
<!ELEMENT Zsection (parent* , (Zparas — informalZ)+) >
<!ELEMENT Zparas (fixity — givendef — axdef — constraint — schemadef — gendef — abbrevdef — goal — freetypedef)* >
<!ELEMENT informalZ (Zsection — Zparas — declaration — name+) >
<!ELEMENT parent (#PCDATA — sub — mixedname)*>
<!ELEMENT goal (#PCDATA — sub — mixedname)*>
<!ELEMENT constraint (#PCDATA — sub — mixedname)*>
<!ELEMENT declaration (variable,type,(rel,type)*)>
<!ELEMENT body (#PCDATA — sub — mixedname)* >
<!ELEMENT predicate EMPTY>
<!ELEMENT exp (#PCDATA — sub — mixedname)* >
<!ELEMENT givendef (givenset—enumeratedtype—interval—comprehensionset)+ >
<!ELEMENT givenset EMPTY>
<!ELEMENT enumeratedtype (value)+>
<!ELEMENT value EMPTY>
<!ELEMENT interval EMPTY>
<!ELEMENT comprehensionset EMPTY>
<!ELEMENT formals EMPTY>
<!ELEMENT sub (#PCDATA — sub — mixedname)* >
<!ELEMENT namearg (#PCDATA — sub — mixedname)* >
<!ELEMENT axdef (decpart,axpart ?) >
<!ELEMENT schemadef ((formals)*,decpart ?,axpart ?) >
<!ELEMENT gendef (formals ?, decpart, axpart ?) >
<!ELEMENT abbrevdef (formals ? , body) >
<!ELEMENT freetypedef (branch+) >
<!ELEMENT branch (exp ?) >
<!ELEMENT axpart (predicate+) >
<!ELEMENT decpart (declaration+) >
<!ELEMENT fixity ((namearg — exparg)*) >
<!ELEMENT exparg (exp , exp, exp, name+) >
<!ELEMENT mixedname (#PCDATA) > <!ELEMENT variable EMPTY>

```
<!ELEMENT rel EMPTY>
<!ELEMENT type EMPTY>

    <!– ************* Element Attributes ************* –>

    <!ATTLIST Zsection
%id;
%name;>
<!ATTLIST Zparas
%id;>
<!ATTLIST givendef
group NMTOKEN #IMPLIED >
<!ATTLIST enumeratedtype
%id;
%name;>
<!ATTLIST value
%id;
%name;>
<!ATTLIST interval
%id;
%name;
sup CDATA #REQUIRED
inf CDATA #REQUIRED
>
<!ATTLIST givenset
%name;>
<!ATTLIST comprehensionset
%id;
%name;
def CDATA #REQUIRED>
<!ATTLIST axdef
%id;
group NMTOKEN #IMPLIED >
<!ATTLIST constraint
%id;
group NMTOKEN #IMPLIED >
<!ATTLIST gendef
%id;
group NMTOKEN #IMPLIED >
<!ATTLIST goal
%id;
group NMTOKEN #IMPLIED >
<!ATTLIST abbrevdef
%id;
%name;
group NMTOKEN #IMPLIED >
```

<!ATTLIST freetypedef
%id ;
%name ;
group NMTOKEN #IMPLIED >
<!ATTLIST branch
%id ;
%name ; >
<!ATTLIST schemadef
%id ;
%name ;
group NMTOKEN #IMPLIED
style (vert — horiz) "vert"
purpose (state — operation — datatype) #IMPLIED >
<!ATTLIST formals
%id ;
%name ; >
<!ATTLIST predicate
%id ;
logexp CDATA #REQUIRED
label CDATA #IMPLIED >
<!ATTLIST fixity
%id ; category (leftfun — rightfun — rel) #REQUIRED
prec CDATA #IMPLIED
firstarg (normal — type) #IMPLIED
lastarg (normal — type) #IMPLIED >
<!ATTLIST namearg
%id ;
midarg (normal — type) #REQUIRED >

<!– Attributes for variable and type if mantory to chack the consistency with UML attributes –>
<!ATTLIST variable
%id ;
%name ; >
<!ATTLIST type
%name ;
set (fset—pset) #IMPLIED >
<!ATTLIST rel
kind (fct—relation—cartesianproduct) #REQUIRED >

D.3 Règles de recouvrement

D.3.1 D'UML vers Z

Du modèle dynamique vers Z

L'étude des règles de recouvrement du modèle dynamique vers Z est intéressante parce que, contrairement au modèle objet, il n'existe a priori pas de fortes similitudes entre les deux types de spécifications.

Ainsi le concept d'état d'un objet qui n'est représenté qu'implictement en Z n'a pas d'équivalence directe. La règle de recouvrement pour les états dépend donc fortement de la vision de celui qui l'écrit. Ici nous nous référons à ce que nous avons exprimé pour la proposition de traduction des états pour spécifier qu'un état doit correspondre à la valeur d'un type énuméré.

Règle de cohérence 14 : Etat
Un état d'un modèle dynamique peut être représenté par une valeur de même nom dans un type énuméré dans le document Z.
Source: (all,package) ;(all,stateDiagram) ;(all,state)
Destination : (all,section) ;(all,Zparas) ;(all,enumeratedtype) ;(all,value)
Condition : state.name=value.name

Dans le cas des états, la règle de recouvrement est totalement subjective puisqu'il n'y a pas de concept correspondant en Z. D'ailleurs, on peut se demander si cette règle doit vraiment exister. Là encore, c'est à la personne qui écrit les règles de décider de leur bien-fondé.

Tous les types d'événements (appel d'opération, signal, etc) n'ont pas d'équivalence directe en Z. Néanmoins si l'événement correspond à un appel d'opération, il doit lui correspondre en Z un schéma d'opération. La règle de recouvrement ne décrit alors que l'un des cas possibles de cohérence. C'est un exemple typique où elle doit être de type *CF* : si la correspondance existe entre un événement et un schéma d'opération, le lien entre les deux doit être créé ; sinon il n'y a pas forcément d'incohérence.

Règle de cohérence 15 : Evénement
Un événement d'un modèle dynamique peut être représenté par un schéma d'opération de même nom dans le document Z.
Source: (all,package) ;(all,stateDiagram) ;(all,event)
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : event.name=schema.name and schema.purpose='operation'

Contrairement aux deux concepts précédents, il paraît naturel que le concept de condition corresponde à celui de prédicat Z. Toute condition d'un diagramme d'état doit donc être identique à un des prédicats des schémas des spécifications Z.

De même le concept d'action est lié à celui d'opération du modèle objet. Sa règle de recouvrement est similaire à celle concernant les opérations : toute action doit correspondre à un schéma d'opération.

Règle de cohérence 16 : Condition
Pour toute condition sur une transition d'un modèle dynamique, il doit exister un prédicat identique dans le document Z.
Source: (all,package) ;(all,stateDiagram) ;(all,condition)
Destination : (all,section) ;(all,Zparas) ;(all,schema) ;(all,axpart) ;
(all,predicate)
Condition : condition.expression=predicate.expression

Règle de cohérence 17 : Action
Pour toute action dans un modèle dynamique, il doit exister un schéma d'opération Z de même nom.
Source: (all,package) ;(all,stateDiagram) ;(all,action)
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : action.name=schema.name and schema.purpose='operation'

Diagramme d'objets

Règle de cohérence 18 : Objet
Pour tout objet d'un diagramme d'objets, il doit exister une variable de même nom dans un schéma Z.
Source: (all,package) ;(all,objectDiagram) ;(all,object)
Destination : (all,section) ;(all,Zparas) ;(all,schema) ;(all,variable)
Condition : object.name=variable.name

Diagrammes de collaboration ou de séquence

Règle de cohérence 19 : Objet
Pour tout objet d'un diagramme de collaboration ou de séquence, il doit exister une variable de même nom dans un schéma Z.
Source: (all,package) ;(all,collaborationDiagram) ;(all,object)
Destination : (all,section) ;(all,Zparas) ;(all,schema) ;(all,variable)
Condition : object.name=variable.name

Règle de cohérence 20 : Message
Pour tout message dans un diagramme de collaboration ou de séquence, il doit exister un schéma d'opération de même nom.
Source: (all,package) ;(all,collaborationDiagram) ;(all,message)
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : message.name=schema.name and schema.purpose='operation'

Diagramme d'activité

Règle de cohérence 21 : Activité
Pour toute activité d'un diagramme d'activités, il doit exister un schéma d'opération Z de même nom.
Source: (all,package) ;(all,activityDiagram) ;(all,activity)
Destination : (all,section) ;(all,Zparas) ;(all,schema)
Condition : activity.name=schema.name and schema.purpose='operation'

D.3.2 De Z vers UML

Règle de cohérence 22 : section
Pour toute section Z, il doit exister un paquetage de même nom dans un document UML.
Source: (all,section)
Destination : (all,package) ;
Condition : section.name=package.name

Règle de cohérence 23 : Pipes de calcul de schémas
Pour tout pipes de calcul de schémas entre 2 opérations (A>>B), s'il existe 2 opérations de même nom dans un diagramme de classes UML, A doit être un message juste avant B dans un diagramme de séquence ou A doit être une activité précédant B dans un diagramme d'activité.
Source: (all,section) ;(all(Zparas) ;(all,schemadef)
Destination : (all,package) ;(all,sequenceDiagram) ;(all,message)
Condition : schemadef.kind='schemacalculuspipe' and
schemadef ;(all,firstop).name=message.name and
schemadef.(all,secondop).name=message.(all,next).name
OR
Destination :(all,package) ;(all,activityDiagram) ;(all,activity)
Condition : schemadef.kind='schemacalculuspipe' and
schemadef.(all,firstop).name=activity.name and
schemadef.(all,secondop).name=activity.(all,next).name

Résumé :

Les notations semi-formelles et formelles semblant complémentaires, leur couplage semble un cadre intéressant pour pouvoir bénéficier de leurs avantages respectifs tout en diminuant leurs points faibles. En effet, d'une part, les notations semi-formelles qui pêchent par leur précision sont de bons vecteurs de communication dont le coût de formation est raisonnable ; d'autre part, les langages formels apportent la précision et le potentiel de raisonnement manquant aux notations semi-formelles. Dans ce travail, nous proposons une approche de traduction de modèles semi-formels objet en des spécifications formelles en Z ou en Object-Z afin de fournir un couplage bénéfique de ces deux types de spécifications.

Nous cherchons à rendre nos propositions les plus utilisables possible en en montrant trois bénéfices avérés : un guide méthodologique pour l'expression des contraintes annotant un modèle objet, une aide à la vérification des modèles et de leurs contraintes et des raisonnements informels sur la sémantique de modèles simples. Nous avons aussi développé un outil de support à notre approche, RoZ qui permet de faire cohabiter les notations semi-formelles et formelles.

Enfin nous étudions une autre approche de couplage, la vérification de cohérence par méta-modélisation pour laquelle nous proposons des règles de cohérence entre le modèle objet et Z. Ce travail nous permet de comparer cette approche avec notre stratégie de traduction afin de mieux comprendre leurs avantages et leurs inconvénients.

Mots clés : couplage de notations, spécifications semi-formelles, spécifications formelles, Z, Object-Z, contraintes.

Abstract :

Semi-formal and formal notations being complementary, their joint use could define an interesting framework in order to take advantage of their good points by reducing their drawbacks. On the one hand, semi-formal notations which are imprecise are good communication vectors with affordable training cost ; on the other hand, formal languages bring precision and their reasoning abilities which miss to semi-formal notations. In this work, we propose a translation approach from semi-formal object models to formal specifications in Z and Object-Z so as to offer a powerful integration of these two kinds of specifications.

We want to make our proposals the most useful possible by showing three established advantages : a methodological guidance to express constraints annotating an object model ; an help to check the models and their constraints and informal reasoning about the semantics of simple models. We also have developed a tool, RoZ that supports our approach by making semi-formal and formal notations live together.

Finally, we study another integration approach, the consistency checking by meta-modelling for which we propose consistency rules between the object model and Z. This work enables us to compare this approach with our translation strategy to understand their advantages and drawbacks.

Key words : notation integration, semi-formal specifications, formal specifications, Z, Object-Z, constraints.