Université de Montréal

Cadre conceptuel pour la composition des objets et la spécification du comportement

par

Dunia Ramazani

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophæ Doctor (Ph.D.)
en informatique

Septembre, 1999

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée:

Cadre conceptuel pour la composition des objets et la spécification du comportement

présentée par:

Dunia Ramazani

a été évaluée par un jury composé des personnes suivantes:

Rachida Dssouli	présidente du jury
Gregor v. Bochmann	directeur de recherche
Jean Vaucher	membre du jury
Hafedh Mili	examinateur externe
.....	représentant du doyen

Thèse acceptée le:.....

Sommaire

Cette thèse porte sur la définition d'un cadre conceptuel permettant la spécification du comportement des objets composés. Nous notons dans les approches existentes une carence de mécanismes et de techniques permettant de décrire adéquatement le comportement d'un objet composé en tenant compte de ses composants et des mécanismes de composition qui sont mis en jeu. Sont négligés les aspects reliés aux interactions entre les composants, à leur contribution dans le comportement de l'objet composé, et à l'identification des objets composés tant dans le problème que dans sa solution. Cela nuit aux méthodes d'analyse et de conception orientée objet basées sur le concept de raffinement successif telle que l'analyse et conception orientée objet de type descendante [deCh91] et la méthode HOOD [Robi92], à la description des cadres d'application [Wirf90, John88, Coad92], et à la vérification des propriétés multi-objets.

Pour pallier à ce problème, nous proposons une extension du paradigme objet resumée par les points suivants:

- Un objet composé est avant tout un objet qui dispose d'une structure.
- La structure d'un objet composé est faite de la structure hiérarchique qui spécifie le lien entre les composants et l'objet composé, et de la structure horizontale qui indique le type des composants ainsi que leurs contraintes structurelles et comportementales dans le contexte de l'objet composé.
- Les propriétés d'un objet sont de deux ordres: les propriétés implicites et les propriétés explicites. Parmi les propriétés implicites, nous distinguons les propriétés inhérentes et les agrégats de propriétés.
- Les objets sont composés à travers leurs interactions.
- Les interactions entre objets ont lieu uniquement dans le contexte des associations qui lient ces objets.
- Dans chaque interaction, chacun des objets précise les hypothèses qu'il assume sur les autres objets participant à cette interaction.

Cette extension du paradigme objet qui forme notre cadre conceptuel pour la composition des objets repose sur le lien entre la structure et le comportement des objets composés. Ce lien est établi par l'identification de la contribution de chaque composant au comportement de l'objet composé. Il est réalisé par la classification des propriétés des objets en propriétés inhérentes et en agrégats de propriétés.

La mise en oeuvre du cadre conceptuel dans une méthode de développement orienté objet requiert une formulation précise et rigoureuse de la sémantique des concepts, principes et heuristiques sous-jacents au cadre conceptuel. Dans cette optique, nous avons étendu la méthode OMT [Rumb91]. La méthode ainsi produite est dénommée Extended Object Modeling Technique (XOMT) [Rama96b]. Cela nous a permis de constater que les concepts d'objet composé et de composition d'objets reposent sur des concepts non encore maîtrisés du paradigme objet, à savoir, l'association, l'interaction des objets, les interfaces supportées par un objet, et la notion de raffinement des spécifications.

Les techniques et méthodes formelles fussent-elles orienté objet ne sont pas encore complètement arrimées au paradigme objet. Afin de permettre la spécification des applications à l'aide de XOMT, nous avons défini une notation formelle orienté objet dénommée Composition Specification Language (CSL).

La théorie de composition proposée par CSL ressemble à bien des égards aux théories proposées par Fiadeiro et Maibaum [Fiad92], Abadi et Lamport [Abad95], et Lam et Shankar [Lam94]. Dans cette théorie de composition, la preuve des propriétés de la composition tire profit de la structure de l'objet composé ainsi que du lien entre la structure et le comportement de cet objet. Notre théorie de composition et celle proposée par Abadi et Lamport [Abad95] reposent sur les mêmes principes de base. Cependant, dans Abadi et Lamport [Abad95], la composition est faite au détriment de l'encapsulation des objets. Dans notre théorie, l'encapsulation est respectée. Cela est réalisé en imposant que la communication entre les objets se fassent uniquement par les paramètres des actions impliquées lors des interactions.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

CAD	Computer Aided Design
CAM	Computer Aided Manufacturing
CASE	Computer Aided Software Engineering
CLOS	Common Lisp Object System
CSL	Composition Specification Language
Disco	Distributed Cooperation
E/R	Entity/Relationship
HOOD	Hierarchical Object-Oriented Design
IGLOO	Ingénierie du Logiciel Orienté-Objet
ISO	International Organization for Standardization
LOTOS	Language of Temporal Ordering Specification
NM	Networks Management
ODP	Open Distributed Computing
OLE	Object Linking and Embedding
OMG	Object Management Group
OMT	Object Modeling Technique
OOA	Object-Oriented Analysis
OOADM	Object-Oriented Analysis and Design Methods
OODA	Object-Oriented Design and Analysis
OORAM	Object-Oriented Role Modeling and Analysis
OSI	Open System Interconnection
RM-ODP	Reference Model for Open Distributed Computing
SAP	Service Access Point
SD	Sense of Direction
SDL	System Description Language
TLA	Temporal Logic of Actions
UML	Unified Modeling Language
XOMT	Extended Object Modeling Technique

Remerciements

En mon nom personnel ainsi qu'au nom de la famille Ramazani, je remercie tous ceux et celles qui ont contribués de près ou de loin et qui m'ont encouragé dans ce travail.

Je dois reconnaître que la patience et les efforts personnels du Professeur Gregor v. Bochmann ont joués un rôle prépondérant dans la réalisation complète de ces travaux. Il faut aussi mentionner les nombreux encouragements et les interventions de la Professeure Rachida Dssouli. J'aimerai aussi remercier Lucie Lévesque qui durant ces années n'a cessée de m'apporter un soutien non-négligeable. Sans oublier, mon ami Petre Dini avec lequel j'ai eu de nombreuses discussions.

Finalement, je remercie les membres de la famille Ramazani qui ont sans cesse exigés la fin de ces travaux et y ont contribué tant financièrement que moralement.

Introduction

Cette thèse porte sur la définition d'un cadre conceptuel permettant la spécification du comportement des objets composés. Avant de décrire cette problématique, il est à noter que la composition des objets est une vaste problématique englobant de nombreuses considérations [Dami92]. De ce fait, il convient de définir le contexte dans lequel ces travaux se sont déroulés, dans le but de circonscrire les attendus et limites de nos travaux. Ces derniers ont été effectués dans le cadre du projet *Ingénierie du Logiciel Orienté-Objet* (IGLOO) [Boch92]. Ensuite, nous donnons un aperçu de la problématique de la composition des objets. Finalement, nous décrivons la structure de cette thèse.

1.1. Contexte de la recherche

Réalisé de 1993 à 1996, IGLOO a pour origines les prétendus avantages qu'apporte l'approche par objet au développement d'applications et à leur maintenance. Le projet IGLOO est né du constat de l'incontournabilité du logiciel orienté objet en tant que palliatif à la crise du génie logiciel. Par crise du génie logiciel, nous entendons les faits suivants [Press93]: l'incapacité à faire face à la demande sans cesse croissante en logiciels, la difficulté d'entretenir le logiciel existant, et la difficulté de réaliser des logiciels de plus en plus complexes.

Certaines caractéristiques du logiciel orienté objet indiquent que celui-ci s'avère être un instrument adéquat pour pallier à cette crise. La capacité de répondre à une demande sans cesse croissante de logiciels est améliorée par l'adoption de techniques de réalisation de logiciel qui font usage de la réutilisation. Ainsi, des logiciels sont réalisés à partir de ceux qui sont disponibles. Le mécanisme d'encapsulation mis en oeuvre dans le logiciel orienté objet accroît la localité ainsi que la cohésion du code et facilite l'entretien du logiciel. Le paradigme orienté objet modélisant naturellement le domaine d'application, il permet la

réalisation de logiciels dont la complexité se résume à celle déjà présente dans ledit domaine [Broo87].

Le projet IGLOO comptait quatre axes principaux, nommément méthodologie, applications, outils et réutilisation, regroupant des aspects reliés au logiciel orienté objet. Nos travaux ont été effectués dans le cadre de la partie méthodologie. Cet axe s'intéressait à la spécification du comportement des objets. Ce qui justifie la spécification des objets composés comme étant un des aspects de cet axe. En particulier, nous retenons que:

(1) Le comportement des objets composés est souvent influencé par celui de leurs composants et par les mécanismes de composition qui ont été mis en oeuvre pour les constituer.
(2) La spécification du comportement des objets composés revêt une importance pour les applications dans les domaines des systèmes distribués et de la gestion des réseaux.

Dans les domaines d'application visés par le projet IGLOO, certains objets résultent de l'assemblage complexe d'autres objets. Les concepteurs ont souvent recours à de savants artifices pour représenter les dépendances qui existent entre un objet et ses composants. Ces artifices ne sont pas uniformes [Rumb88]. L'un des objectifs de notre recherche a consisté à uniformiser la modélisation des objets composés. Cet objectif a été atteint par la proposition d'un cadre conceptuel permettant la description des objets composés, l'emphase étant mise sur la spécification du comportement de ces objets. Nous avons aussi cerné les mécanismes de composition des objets qui sont des mécanismes de réutilisation.

1.2. Composition des objets

1.2.1. Objets composés dans le dévéloppement objet

D'emblée, un constat s'impose: dans les méthodes d'analyse et de conception orienté objet, les approches pour modéliser les objets composés sont

nombreuses. Par contre, nous notons une carence de mécanismes et de techniques permettant de décrire adéquatement le comportement d'un objet composé en tenant compte de ses composants et des mécanismes de composition qui sont mis en jeu. De plus, les aspects relatifs à la composition sont perdus à travers les différentes étapes du développement ou transformés en des aspects connexes.

Par ailleurs, à l'implantation, la composition est traduite par des références aux composants par le biais d'attributs et à cela s'ajoute l'encapsulation des aspects dynamiques de la composition dans les méthodes de l'objet composé qui font usage des composants. Cela nuit au retraçage des aspects reliés à la composition des objets d'une étape à l'autre du processus de développement. Or, l'un des avantages de l'approche par objet est d'offrir un modèle de représentation de l'application qui est commun pour toutes les étapes, donc de permettre le retraçage à travers les différentes étapes du développement orienté objet. Quand il est malaisé de retracer les aspects reliés à la composition, c'est une importante caractéristique du développement orienté objet qui n'est pas respectée ce qui cause la perte des avantages inhérents à cette caractéristique tels qu'énumérés plus haut.

Nous avons souligné que d'une méthode de développement à l'autre le traitement de la composition n'est pas uniforme. Il est médiocre dans la plupart des méthodes. Plus spécifiquement, sont négligés les aspects reliés aux interactions entre les composants, à leur contribution dans le comportement de l'objet composé, à l'identification des objets composés tant dans le problème que dans sa solution.

1.2.2. Améliorations possibles

Dans le but de favoriser le retraçage des aspects relatifs à la composition d'une étape à l'autre et pour éviter la multitude d'interprétations ou du moins permettre le passage d'une interprétation à une autre, nous affirmons que c'est le concept même d'objet composé qui doit être clarifié. Il doit être enrichi pour mieux décrire les objets composés et surtout leur comportement. C'est dans ces termes que nous caractérisons nos travaux.

Nous avons aussi voulu intégrer entièrement le concept d'objet composé ainsi que la description de son comportement dans une des méthodes de développement orienté objet existantes. A cet effet, il faut noter la distinction entre:

- d'une part, la composition à l'étape d'analyse qui représente des aspects ayant une contrepartie dans la situation réelle ou hypothétique (du problème à résoudre);
- et, d'autre part, la composition à la conception qui englobe à la fois les aspects de l'analyse et ceux qui sont issus des décisions de conception.

Ces derniers concernent la réalisation des applications par le biais de programmes et n'ont pas nécessairement de contrepartie dans la situation concernée, qu'elle soit réelle ou hypothétique. Il faut de plus veiller à ce que les langages de programmation orienté objet soient dotés de constructions leur permettant de mieux représenter les aspects relatifs à la composition que l'on retrouve à l'étape de conception.

Nous estimons que la description des objets composés telle que proposée dans la littérature est inadéquate pour :

- primo, les méthodes d'analyse et de conception orientée objet basées sur le concept de raffinement successif telles que l'analyse et conception orientée objet de type descendante [deCh91] et la méthode HOOD [Robi92];
- secundo, la documentation des cadres d'application [Wirf90, John88, Coad92] ;
- tertio, la vérification des propriétés multi-objets.

Il existe un lien entre ces trois points et la spécification des objets composés.

1.2.3. Analyse orienté objet descendante

L'analyse orientée objet de type descendante consiste à imaginer une application comme étant formée de plusieurs niveaux d'abstraction. À chaque niveau se trouvent des objets qui représentent la structure de l'application et à travers leurs interactions décrivent le fonctionnement de l'application. Dans un tel contexte, il convient d'établir des relations entre les différents niveaux d'abstraction, entre les objets et leurs représentations aux niveaux

correspondants, ainsi que la représentation de leurs interactions à ces niveaux. Un objet unitaire au plus haut niveau d'abstraction peut être représenté par un ensemble d'objets à un niveau inférieur. Cela nous amène à considérer cet objet unitaire comme étant un assemblage d'autres objets, un objet composé.

La distribution des fonctions (services) de l'objet composé à travers ses composants est cruciale pour la validation du passage d'un niveau d'abstraction à un autre et de la représentation du domaine modélisé. Le problème se complique lorsque l'on désire valider les interactions entre des objets situés aux différents niveaux d'abstraction.

1.2.4. Cadres d'application

Un cadre d'application est caractérisé par la fonctionnalité globale qu'il offre et par la distribution des fonctions qu'il impose aux objets. Ces deux éléments doivent être documentés explicitement et de manière appropriée pour faciliter l'utilisation des cadres d'application dans la réalisation de logiciels orientés objets. Ce problème est analogue à celui de la représentation à un niveau d'abstraction donné des interactions entre des objets et la mise en relation avec les interactions avec un niveau d'abstraction supérieur.

Le cadre d'application, par les fonctions globales qu'il offre, peut être considéré comme étant une unité. Son fonctionnement interne à travers les objets qui le constituent est analogue à l'activité au sein d'un objet composé. Ce faisant, des mécanismes similaires peuvent être utilisés autant pour documenter les cadres d'application que pour spécifier les objets composés.

1.2.5. Propriétés multi-objets

La vérification de propriétés multi-objets requiert avant tout la représentation de ces propriétés. Par une propriété multi-objets, nous entendons celle qui implique plusieurs objets. Une telle propriété caractérise globalement ces objets. La représentation de ce type de propriété est analogue à la représentation de celles des objets composés. Vérifier des propriétés multi-objets nous oblige aussi à tenir compte des mécanismes de composition des objets, car à travers ces mécanismes se retrouvent les mécanismes et

techniques d'établissement des propriétés multi-objets. Cela nous amène à considérer comment les techniques de spécification formelle tiennent compte de la composition, sans oublier que la description du comportement d'un objet est souvent faite par le biais de ces techniques.

1.2.6. Apports de cette thèse

Cette thèse offre un cadre conceptuel qui pourrait servir de fondement sémantique à des méthodes de développement orienté objet ainsi qu'à des langages de programmation orienté objet. Elle démontre que les concepts d'objet et de composition d'objets reposent sur des concepts non encore maîtrisés du paradigme objet, à savoir, l'association, l'interaction entre objets, les interfaces supportées par un objet, et la notion de raffinement de la spécification des objets.

Le modélisateur dispose de plusieurs possibilités lorsque différentes interprétations de ces concepts sont utilisées. Néanmoins le pragmatisme du concepteur et le soucis de réutilisation du logiciel orienté objet conçu restreignent les interprétations de ces concepts. En particulier, l'établissement d'un lien entre la structure et le comportement de l'objet composé fait ressortir la contribution de la structure dans la description de l'objet composé. Cet aspect fondamental se retrouve aussi dans les systèmes distribués où l'existence de différents sites et la répartition entre eux des composants du système influent sur le comportement du système. Cela nous a conduit à considérer le Modèle de Référence ODP comme cadre de description des objets composés [Rama96a]. Cette analogie renforce la stratégie de modélisation qui consiste à voir une application comme étant elle-même un objet composé.

D'autre part, les techniques et méthodes formelles fussent-elles orientées objet ne sont pas encore complètement arrimées au paradigme objet. Ainsi, à la description de la structure des objets composés, il est apparu que le concept d'association entre les objets est interprété de diverses manières. Cela nous a conduit à proposer plusieurs approches de description des associations [Rama97]. De plus, il est apparu que la modélisation de l'environnement d'un objet est nécessaire lorsque l'on veut composer les objets.

Nous avons aussi expérimenté la spécification formelle des objets composés avec Object-Z [Duke94]. Ici, bien qu'elle soit, du point de vue de la notation, adéquate pour décrire la structure de l'objet composé en termes d'associations, elle ne l'est pas pour décrire les interactions entre les composants. Dans Object-Z, le fait que le langage ne supporte pas la concurrence de manière explicite nous a conduit à envisager une autre approche de formalisation.

En plus des concepts et principes du cadre conceptuel, nous appliquons les principes suivants:

- Les interactions entre objets se font dans le contexte des associations qui lient ces objets;
- Les objets sont composés à travers leurs méthodes;
- Les hypothèses sur l'environnement de chacun des objets sont explicites dans la définition des méthodes que les objets supportent;
- Les interactions sont explicitement définies en utilisant des constructions spécifiques pour les représenter.

Ces principes permettent de représenter adéquatement la structure des objets composés. Lorsque nous ajoutons à cela la possibilité de raffiner la spécification d'une structure, l'ensemble de ces principes constituent une théorie de composition. La théorie de composition ainsi établie ressemble aux théories proposées par Fiadeiro et Maibaum [Fiad92], Abadi et Lamport [Abad95], et Lam et Shankar [Lam94]. Dans cette théorie de composition, la preuve des propriétés de la composition tire profit de la structure de l'objet composé et du lien entre la structure et le comportement de cet objet.

La problématique du nommage des objets est associée à celle de la description de leurs interactions. De nouvelles exigences sur le nommage des objets s'imposent dans le contexte des objets composés. Le concept d'identité qui est employé dans le paradigme objet pour identifier les objets est inadéquat pour satisfaire à ces exigences. Afin de pallier à ce problème, nous utilisons des noms contextuels pour identifier les objets indépendamment de leur identité. Le

concept de *sense of direction* a été utilisé pour produire des mécanismes de nommage contextuel qui répondent aux caractéristiques d'un mécanisme d'identification d'objets. Les identités des objets sont remplacées par des noms sans nuire à la sémantique de l'application.

Afin de faciliter l'utilisation du cadre conceptuel dans des cas réels, il fallait l'intégrer à une méthode orienté objet. Cela fut fait avec OMT à travers son extension appellée Extended Object Modeling Technique (XOMT) [Rama96b]. Dans le chapitre 7, nous décrivons la relation entre XOMT et UML [Booc96].

1.3. Démarche méthodologique

La démarche méthodologique entreprise pour résoudre les problèmes abordés dans cette thèse consistait à définir un cadre conceptuel. Ici, la notion de cadre conceptuel est réduite à un ensemble de concepts, principes et heuristiques permettant d'effectuer l'analyse d'un domaine, c'est-à-dire décrire des situations et des phénomènes dans un domaine particulier, par exemple la description des applications en gestion des réseaux. Ce cadre conceptuel tient compte des approches de modélisation proposées dans la littérature et de notre expérience dans la spécification des applications ayant des objets composés. Il tient aussi compte des critères permettant de réconcilier les approches existantes et de relier la structure et le comportement des objets composés.

Par la suite, le cadre conceptuel a été formalisé. Pour ce faire, il fallait choisir la technique de formalisation visant à capturer la sémantique des concepts et principes énoncés dans le cadre conceptuel. Afin de démontrer l'utilité pratique du cadre conceptuel, ce dernier a été intégré dans une méthode de développement orienté objet. Ces activités principales furent la source de nombreuses autres activités de recherche.

1.4. Organisation de la thèse

La problématique de la composition et décomposition des objets peut être abordée selon différentes perspectives. Dans cette thèse, différents aspects de

cette problématique sont étudiés. Nous avons mis l'accent sur les aspects pertinents à la définition du cadre conceptuel et à sa mise en oeuvre pratique. Ce qui justifie l'apparence d'hétérogénéité dans les aspects étudiés.

Nous avons retenu le format de la thèse par articles. Le chapitre 2 présente une revue de la littérature afférente à la composition des objets. Elle est guidée par le processus de développement orienté objet. Le chapitre motive les concepts et principes sous-jacents au cadre conceptuel, avant de le présenter. Le chapitre 3 montre l'intégration du cadre conceptuel dans la méthode OMT. Y est démontré la différence entre la sémantique des objets composés figurant dans notre cadre conceptuel et celle proposée par OMT [Rama96b]. De plus, quelques défauts de la méthode OMT sont relevés.

Le chapitre 4 se penche sur spécification du comportement des objets composés. Il présente aussi notre approche de spécification formelle des compositions ainsi que le langage CSL utilisé à cet effet. Le chapitre 5 porte sur l'intégration de notre cadre conceptuel dans les langages orienté objet. Une seule problématique y est abordée, celle du nommage des objets composés.

Le chapitre 6 relate une étude de cas dans la composition des objets. Ce chapitre illustre la théorie de composition. L'objectif est de montrer comment l'ensemble de ces travaux peuvent être utilisés comme un tout cohérent lors de la spécification d'applications comportant des objets composés.

Le chapitre 7 résume les contributions de ces travaux. Il critique les résultats de chacun des articles et présente les problèmes qui n'ont pas été résolus. En conclusion, nous mentionnons notre contribution au savoir et les perspectives de recherches qui en découlent. Les références bibliographiques ont été extraites des différents articles. Elles sont présentées en une seule compilation pour faciliter la lecture de la thèse.

Chapitre 2

État de l'art et approche proposée

La composition des objets est un champs d'étude englobant presque tous les aspects du développement orienté objet. Toute contribution dans ce domaine a un impact immédiat sur l'ensemble des concepts reliés au paradigme objet.

Avant de décrire cette problématique, nous présentons dans ce chapitre le processus général de développement des applications suivant l'approche par objet. Nous couvrons les concepts et principes sous-jacents au développement orienté objet, les méthodes, méthodologies et outils de développement, ainsi que l'apport des spécifications formelles. Cette présentation, bien que sommaire, vise à fournir au lecteur[1] un cadre d'interprétation des problèmes et des aspects relatifs à la composition des objets.

Par la suite, nous revoyons les différentes recherches entreprises dans ce domaine. En utilisant le processus de développement orienté objet comme fil conducteur nous mettons en relation les différents aspects de la composition des objets, allant de l'analyse à l'implantation. En fin de chapitre, le lecteur trouvera une synthèse de cette revue.

2.1. Le développement orienté objet

L'approche de développement dite orientée objet est un processus de réalisation des applications dont les différentes étapes (l'analyse, la conception et l'implantation) font usage des concepts sous-jacents à l'approche orientée objet [Aski92]. Elle utilise les mêmes concepts et principes allant de l'étape d'analyse à celle de l'implantation. Cela permet un développement par des raffinements successifs des modèles obtenus à chaque étape.

---

[1]Notez que le masculin est utilisé uniquement pour alléger le texte.

En effet, les différents modèles produits à chaque étape du développement se basent sur les concepts d'objet et d'interaction d'objets pour décrire les aspects statiques et dynamiques de l'application à réaliser. Cela allant du problème à sa solution. Du fait de l'homogénéité des concepts utilisés tout le long du processus de développement, le développement orienté objet procède par le raffinement successif des modèles obtenus dans les différentes étapes du processus de développement [Hend90]. Dans ce qui suit, nous examinons comment, d'un problème à sa solution, procède le développement orienté objet.

Le problème à résoudre consiste généralement en une situation réelle ou hypothétique que l'on désire traduire dans un modèle à objet. Le développement par objet se base sur le fait que l'on peut modéliser le monde (réel ou hypothétique) en utilisant des entités et des phénomènes impliquant ces entités. On reconnaît l'existence d'un état du monde à modéliser à un instant donné. Cet état est modifié par les phénomènes qui s'y produisent. Les entités du monde sont décrites par des objets, tandis que les phénomènes sont décrits statiquement par des relations entre objets et dynamiquement par le biais d'interactions d'objets.

2.1.1. Analyse orienté objet

L'étape d'analyse consiste à modéliser le domaine du problème et à identifier les frontières (environnement) du système. Une fois ces frontières cernées, les interactions du système et de son environnement constituent les services offerts par l'application à modéliser. Le domaine du problème est capturé par un diagramme d'objets sur lequel figurent les objets du domaine et leurs relations. Les interactions entre les objets du système et l'environnement peuvent être décrites soit formellement en utilisant par exemple des *statecharts* [Hare88], soit informellement par le biais de notations ad hoc. La description des interactions du système et de son environnement est déclarative. Ce type de description spécifie le "quoi ?" et non le "comment ?" de ces interactions[2]. La

[2]Ici, le "quoi" réfère aux conditions nécessaires pour la réalisation de l'interaction ainsi qu'aux résultats attendus de cette interaction. Tandis que le "comment" porte sur la manière dont est réalisée l'interaction. Ces éléments sont causalement reliés comme suit: les conditions de réalisation précèdent l'élaboration de la manière de réalisation . Cette dernière permet d'obtenir les résultats attendus.

modélisation du domaine du problème ainsi que la description des interactions entre le système et son environnement constituent les modèles d'analyse.

2.1.2. Conception orienté objet

Lors de la seconde étape, celle de conception, sont recherchés les moyens permettant de préciser comment se réalisent les services offerts par le système, et ce, en termes d'interactions entre composantes du système. À cette occasion, des décisions dites de conception sont prises. Elles peuvent ajouter des objets et des relations dans le système, enrichir les objets et relations et finalement définir des interactions entre objets. D'autres aspects sont aussi pris en considération, tels que l'optimisation des interactions, la réutilisation des descriptions d'objets par le biais de l'héritage, *et cetera* [Cole94]. Ici, les modèles dits de conception englobent les modèles d'analyse. L'architecture du système est aussi prise en considération. S'y ajoute la description des interactions entre objets, la structure finale des objets en termes d'attributs et de méthodes, les graphes d'héritage, etc.

2.1.3. Implantation orienté objet

Finalement, à l'étape d'implantation, les éléments définis à la conception sont traduits dans un langage d'implantation. Une correspondance est établie entre les concepts utilisés lors de la conception et les construits du langage d'implantation sélectionné. Cette correspondance n'est pas souvent bijective. En effet, elle requiert une bonne connaissance du langage d'implantation, car certains concepts utilisés lors de la conception n'ont pas d'équivalent dans le langage d'implantation.

2.1.4. Avantages du développement orienté objet

Les avantages du développement par objet sont nombreux et le lecteur en trouvera une meilleure présentation dans [Hend90]. Néanmoins, parmi ceux-ci nous nous attardons sur la facilité du développement. Cela est possible étant donné que:

(1) les concepts d'objet, de relation entre objets et d'interactions entre des objets, concepts utilisés durant l'analyse et la conception, sont intuitifs et

familiers aux usagers potentiels et ils correspondent à ce que l'on retrouve dans le domaine du problème;

(2) les mêmes concepts étant à la base des modèles de chaque étape, le passage d'une étape à l'autre est aisé. En particulier, le passage du problème à la solution est sans accroc[3];

(3) le raffinement successif des modèles allant du problème à la solution permet de mieux appréhender le processus de développement sans pour autant en diminuer la complexité;

(4) la réutilisation des descriptions d'objets, des relations entre objets ainsi que des descriptions de leurs interactions est possible.

Le développement orienté objet repose sur un ensemble de concepts et de principes qui sont utilisés durant les différentes étapes du processus. Ces concepts et principes forment le modèle objet supporté par le développement orienté objet. Ce modèle objet décrit essentiellement ce qu'est un objet, comment est-ce qu'il est relié aux autres objets et comment est-ce qu'il interagit avec ceux-ci. Une méthode ou technique orienté objet est un ensemble d'étapes et d'heuristiques indiquant comment spécifier, enrichir et réaliser des applications suivant un modèle objet donné. Pour ce faire, les méthodes orienté objet proposent des notations permettant de capturer les divers aspects d'une application, notamment l'aspect structurel où sont décrits les interconnexions des multiples objets formant l'application, et l'aspect dynamique qui montre comment les objets interagissent ou plus particulièrement comment un objet répond en présence de différents phénomènes. Compte tenu du volume d'informations inhérent au développement orienté objet et surtout du fait que les spécifications sont réutilisables, il existe de nombreux outils supportant certaines parties du développement orienté objet. Ces outils ne diffèrent pas beaucoup d'autres outils connus en génie logiciel, excepté qu'ils supportent un ou plusieurs modèles objets.

---

[3]En effet, comme les modèles produits à la conception englobent ceux de l'analyse, les modèles de la conception constituent une extension de ceux de l'analyse.

2.1.5. Formalisme dans le développement orienté objet

Le soucis de précision et de minimalité a amené plusieurs chercheurs à entrevoir la possibilité d'utiliser du formalisme dans le développement orienté objet. L'apport des spécifications formelles dans le processus de développement orienté objet se situe au niveau de la rigueur de la description des applications qui permet la validation et la vérification des spécifications. Il devient possible d'automatiser la génération des implantations des applications à partir de leur spécification formelle, possibilité à laquelle s'ajoute celle de prouver des propriétés de la spécification de l'application. Les spécifications formelles sont un complément nécessaire aux méthodes et techniques orienté objet, car elles permettent une application rigoureuse des principes et concepts sous-jacents aux méthodes et techniques orienté objet. Par ailleurs, on peut réutiliser et comparer des spécifications formelles. Ce qui n'est pas le cas des spécifications informelles. De plus, la formalisation des concepts et principes sous-jacents à une méthode orienté objet offre un cadre permettant d'aborder les questions de réalisation des différentes caractéristiques des applications à l'aide des langages de programmation orienté objet, puisqu'il devient possible de vérifier la consistance et la cohérence des descriptions qui sont faites. Dans certains cas, la vérification peut être faite automatiquement.

2.1.6. Limites de l'approche par objet

Par ailleurs, il faut noter que l'approche par objet ne résout pas tous les problèmes. Elle est limitée, en particulier au niveau de son potentiel de modélisation [Jack95]. C'est cela qui a en partie motivé notre recherche: on note l'existence d'objets composés dans les problèmes inhérents aux systèmes répartis, à la modélisation des entreprises et à la gestion des réseaux téléinformatiques. Dans ces domaines, les objets du problème sont souvent décomposables en d'autres objets qui jouent un rôle prépondérant dans le comportement de ceux auxquels ils appartiennent.

La question sur laquelle nous nous sommes penchés est de savoir dans quelle mesure le développement orienté objet décrit adéquatement ces situations. En particulier comment est-ce qu'il tient compte du comportement des objets

composés et ce, sans égard à l'étape de développement prise en considération. Mais avant de nous pencher sur cette question, il convient de revoir les recherches précédentes reliées à la composition des objets.

2.2. Recherches précédentes dans le domaine

2.2.1. Civello

L'examen du traitement de la composition des objets dans les méthodes d'analyse et /ou de conception orienté objet a déjà été réalisé par Frank Civello [Cive93]. Il s'intéresse à l'utilisation de la composition durant les étapes d'analyse et de conception lors du développement d'une application. Le fait qu'au courant de la conception, les relations d'agrégation peuvent être raffinées, modifiées et que de nouvelles agrégations peuvent être ajoutées, il devient primordial de garder la trace de ces différents changements. En particulier, lors de la conception, comme l'ensemble des changements relatifs à l'agrégation reflète des décisions de conception, il faut les enregistrer afin de faciliter l'évolution et l'entretien de l'application.

Par ailleurs, le modèle de l'application résultant de la conception doit toujours respecter les exigences établies à l'analyse. L'examen des changements ainsi que leur impact sur le modèle de l'application permettent de valider le modèle de l'application vis-à-vis de l'agrégation. Civello propose de faire la distinction entre les caractéristiques des relations d'agrégation à l'étape d'analyse et celles relatives à l'étape de conception. À l'analyse, les relations d'agrégation ont une contre partie dans le domaine de l'application. Tandis qu'à la conception, en plus des caractéristiques physiques et fonctionnelles des objets, on s'intéresse à la visibilité, l'encapsulation, le partage, l'invariance, l'immutabilité et au contrôle des composants.

L'article de Civello inclut une revue sommaire des méthodes suivantes: *OOA* de Coad et Yourdon [Coad92], *OOAD* de Booch [Booc94], *OMT* de Rumbaugh et al. [Rumb91], *CRC* de Wirfs-Brock et al.[Wirf90], *Objectory* de Jacobson [Jaco92] et *HOOD* [Robi92]. Cette revue met l'emphase sur les aspects ci-après de l'agrégation:

1) la notation qui est utilisée pour représenter l'agrégation dans les diagrammes d'objet;
2) la terminologie utilisée dans la méthode pour désigner l'agrégation et ses caractéristiques;
3) la sémantique de l'agrégation par rapport au domaine de modélisation. Il s'agit du pouvoir de représentation des agrégats présents dans le domaine du problème;
4) les différentes formes d'agrégation;
5) les propriétés des relations d'agrégation;
6) la sémantique de l'agrégation lors de la conception et cela en termes de principes de génie logiciel;
7) l'apport de l'agrégation dans la méthode;
8) l'objectif sous-jacent à la présence de l'agrégation.

De cette revue, il ressort que les méthodes examinées ne permettent pas de représenter toutes les formes d'agrégation que l'on peut retrouver dans le domaine du problème. Elles semblent favoriser une seule forme de composition. Elles ne font pas la distinction entre l'agrégation à l'étape d'analyse et l'agrégation lors de la conception. On peut classifier ces méthodes en deux classes, celles qui utilisent l'agrégation lors de l'analyse comme OMT et OOA et celles qui l'utilisent à la conception telles que OODA et HOOD. A part OMT, aucune des méthodes ne couvrent l'agrégation de l'analyse à l'implantation en passant par la conception. Ces méthodes ne font pas de distinction significative entre l'agrégation et les associations diverses entre les objets. Finalement, elles n'offrent pas de recommandations quant à l'impact des décisions de conception sur les agrégats présents dans le domaine du problème. En somme, l'agrégation demeure une problématique d'actualité. Mais le fait qu'elle couvre tout le processus de développement orienté objet rend les contributions difficiles.

2.2.2. La sémantique des hiérarchies d'agrégats

En s'inspirant des travaux de Bunge [Bung77, Bung79], Liu [Liu92] propose d'intégrer le concept de propriétés résultantes[4] dans la définition des schémas de bases de données. Selon Liu, lorsque vous définissez un objet comme étant formé d'autres objets, automatiquement, l'interface (ensemble des propriétés observables) des composants est disponible au niveau de l'objet composé. Par exemple, si on crée un objet de type terminal et que l'on mentionne qu'un objet de type clavier est une partie du terminal alors les opérations disponibles au niveau du clavier le deviennent automatiquement au niveau du terminal.

Liu affirme que cela engendre une économie dans la formulation des requêtes dans le cas d'une base de données d'assemblages physiques. La raison avancée est que pour effectuer une recherche sur des objets, nous souhaitons connaître les propriétés de leurs composants. Le fait de rendre disponible à l'interface du composé les interfaces de ses composants permet de faire abstraction de la structure du composé et de rendre transparent tout changement interne à cette structure.

Il s'agit d'un pas significatif dans l'établissement du lien entre la structure des objets composés et leur comportement. Néanmoins, cette technique a des limites. Par exemple, dans le cas d'un objet dont certaines composantes sont du même type, il est difficile d'associer à une composante précise une propriété visible au niveau du composé. Liu propose de réélaborer les propriétés des composants pour permettre leur distinction. Cela pourrait avoir pour conséquence de ramener les problèmes que l'on tente de solutionner, à savoir, faire abstraction de la structure interne de l'objet composé.

2.2.3. BellCore

Dans le but d'élaborer une bibliothèque de composants génériques pour la création d'applications en télécommunications et dans les systèmes d'information, des chercheurs de Bell Communications Research ont déterminé les caractéristiques des objets composés dans ces domaines [Bell92].

---

[4]Nous allons revoir plus loin ce que représentent des propriétés résultantes. Cela lors d'un examen du modèle de Mario Bunge.

En plus des caractéristiques que nous avons déjà mentionnées, ils imposent que les propriétés structurelles de l'objet composé ne soient pas héritées par les composants, vu que les composants définissent le composé et non l'inverse. Certaines propriétés de l'objet composé dépendent de celles des composants. Elles sont soit résultantes soit émergentes, leur nombre étant supérieur ou égal à un. Il doit exister au moins une propriété résultante. Les propriétés émergentes constituent l'identité de l'objet composé et sont indépendantes des composants. De plus, la création d'un objet composé n'altère pas les propriétés de ses composants.
L'équipe de Bell Communication Research insiste sur l'existence d'une grande variété de contraintes d'intégrité, contraintes reliées à la composition. Par exemple, l'ordonnancement des composants, la variabilité de la structure du composé et le partage des composants.

Quatre formes de composition sont proposées: l'assemblage, la subordination, l'encapsulation et la liste.
Dans l'*assemblage*, l'existence de l'objet composé implique celle d'au moins un de ses composants. C'est le cas d'un véhicule .
Dans la *subordination*, l'existence du composant implique celle d'au moins un composé. Considérons un document. Le document peut exister indépendamment de ses composants. Mais une fois que le contenu du document est créé, celui-ci est toujours associé à un document.
Pour l'*encapsulation* un regroupement d'objets dépend fortement de la situation modélisée. Il s'agit du lien contenant - contenu.
Finalement, dans une *liste* l'existence d'un composant implique celle du composé tandis que l'existence du composé implique celle d'au moins un composant.

À ces formes de composition sont adjointes les propriétés d'ordonnancement des composants, de la variabilité de structure du composé et de partage des composants pour donner lieu à des compositions plus élaborées et complexes. En guise d'illustration, la subordination associée au partage de composants, sans tenir compte de l'ordonnancement et de la variabilité, aboutit à la forme

que les auteurs appellent "contenance" et qui est très utilisée dans les systèmes de télécommunications. Ainsi, il existe des formes primitives de composition.

2.2.4. Composition des spécifications

Nous examinons dans ce paragraphe les recherches sur la composition et décomposition des spécifications. Il s'agit d'une problématique connexe à la composition des objets, car ultimément nous allons composer des spécifications d'objets. Par ailleurs, au niveau architectural, certains principes de composition des spécifications influencent la composition des objets.

Trois approches ont été sélectionnées: *Principle of Composition, Theorem of Composition* et la technique de Martin Feather. Il est à noter que les travaux de Zave et Jackson [Zave93] quoique cités dans le domaine de la composition des spécifications, n'ont pas d'application directe dans le cadre de la spécification des objets composés.

Principle of Composition

Abadi et Lamport font une étude exhaustive du rapport entre la décomposition et la composition des systèmes [Abad95]. Ils proposent un principe de composition qui permet de déduire les propriétés d'un système en appliquant un raisonnement à ses composants. Les propriétés des composants doivent être connues. D'où une connaissance de l'environnement dans lequel évolue un composant, celui-ci n'affichant pas le comportement désiré en présence d'un environnement hostile. En outre, le raisonnement sur un système à partir de ses composants dépend du mode de formation du système.

Le raisonnement servant à décomposer un système diffère de celui qui mène à sa composition. Les auteurs énoncent deux théorèmes:

a) Le théorème de la décomposition d'un système

L'environnement de chaque composant est formé par les composants restants et est implicite. Pour chaque composant, des hypothèses sont émises sur son environnement et sur le comportement qu'il manifeste dans cet environnement. La vérification de ces hypothèses est étendue à tous les composants. C'est

pendant cette vérification qu'est validée la contribution de chaque composant au comportement du système. Chaque composant est vérifié séparément. Le fait de se concentrer sur un composant à la fois facilite la méthode de preuve. Une décomposition est justifiée en ce qu'elle fournit une spécification de composant, laquelle peut être raffinée sans tenir compte des spécifications du reste des composants. Ce théorème permet de prouver que la composition du raffinement de la spécification de ce composant avec les spécifications des autres composants réalise toujours la spécification du système. Par exemple, soit M, un système. Il est décomposé en M_1 et M_2 tels que

$M_1 \wedge M_2 \rightarrow M$,

le $\wedge$ désignant la conjonction et $\rightarrow$ la conséquence. Un raffinement séparé de M_1 et M_2 aboutit respectivement à M_1' et M_2'. Il faut maintenant vérifier si $M_1' \wedge M_2' \rightarrow M_1 \wedge M_2$.

Par le théorème de la décomposition, ce problème est réduit à un autre, en l'occurrence, prouver que $M_1' \wedge E_1 \rightarrow M_1$ et $M_2' \wedge E_2 \rightarrow M_2$,

E_1, E_2 étant les environnements respectifs de M_1 et M_2.

b) Le théorème de la composition

Des hypothèses sur l'environnement de chaque composant sont émises et les garanties quant au comportement individuel des composants sont décrites. Ensuite, il faut prouver que la composition des composants réalise la spécification du système en termes d'hypothèses sur l'environnement et de garanties offertes par le comportement du système. Le problème est que les composants ont pour environnement d'autres composants. Pour garantir le comportement d'un composant, il faut raisonner de manière circulaire en recourrant à ce même comportement. Cet écueil est surmonté grâce au théorème de la composition qui permet de raisonner sur les propriétés de sûreté en présence d'interdépendances. En guise d'illustration, soit deux composants M_c et M_d. Leur composition nécessite qu'ils interagissent : l'un servira d'environnement à l'autre. Autrement dit, chacun des deux garantira son comportement en présence de l'autre schématiquement,

$M_d \rightarrow M_c$ et $M_c \rightarrow M_d$

Ensuite, il faut prouver que

$[(M_d \rightarrow M_c) \wedge (M_c \rightarrow M_d)] \rightarrow M_c \wedge M_d$.

Ce résultat est vérifié à l'aide du théorème de la composition sous deux conditions : restreindre les hypothèses sur l'environnement et s'assurer que le comportement d'un composant englobe des propriétés dites de sureté et de vivacité.

Abadi et Lamport soulignent le caractère général des principes utilisés et leur applicabilité dans un contexte orienté objet. En effet, le théorème de la décomposition est applicable dans le développement orienté objet lors de la substitution des composants par des composants spécialisés. Le théorème permet de faire la preuve de consistence de la nouvelle composition obtenue. Quant à lui, le théorème de la composition sert lors du raffinement d'un objet en plusieurs autres objets. Il permet de valider le passage d'un niveau d'abstraction à un autre.

Par ailleurs, les auteurs établissent une distinction entre les interfaces des composants. Les mécanismes de composition et de décomposition que nous venons de voir sont implantés dans le langage de spécification TLA [Lamp94], langage dont la sémantique fait usage des deux théorèmes mentionnés. TLA a été utilisé pour formaliser d'autres langages et vérifier des systèmes hybrides.

Theory of Interfaces

Lam et Shankar adoptent une représentation graphique de la structure et des interactions au sein d'un système [Lam94]. Un système est un graphe acyclique dirigé dont les noeuds représentent des modules et les arcs des interactions entre modules. L'origine d'un arc est un fournisseur de services; sa destination, un utilisateur de services. Les hypothèses sous-jacentes à ce modèle sont les suivantes : les modules capables d'interagir sont les seuls à être composés; un module offre ses services à travers ses interfaces; une interface contient toute l'information nécessaire pour concevoir et implanter séparément chaque module.

L'interface nous renseigne sur les événements contrôlés par le module et ceux qui dépendent de l'environnement du module. Ainsi, nous sommes en mesure de définir les propriétés (de sûreté et de vivacité) que le module doit satisfaire et nous pouvons assumer que l'environnement du module satisfait aussi certaines

propriétés. Muni de ces informations, il devient possible de prouver qu'un module satisfait ses interfaces sans examiner l'implantation de son environnement.

Lam et Shankar optent pour une approche de conception par décomposition, vu que le domaine d'application envisagé est celui des télécommunications. Dans ce domaine, les systèmes sont bâtis en couches successives, les inférieures offrant leurs services aux supérieures. Ce contexte permet de modéliser le système par un graphe dirigé, dans lequel l'origine d'un arc représente la couche qui fournit des services, tandis que la destination indique la couche qui utilise des services. Une telle représentation facilite la décomposition d'un système.

Une fois le système décomposé, dans approche par décomposition, nous devons valider la décomposition obtenue. Cette validation est faite en démontrant que la composition des modules obtenus réalise le système. À ce stade, les graphes étant dirigés et acycliques, Lam et Shankar se réfèrent aux travaux de Misra et Chandy [Misr81], qui ont prouvé qu'il est possible de composer des systèmes (décrits par leurs propriétés de sûreté) sans impliquer un raisonnement circulaire. Toutefois, une considération est à retenir: dans une interface, certains événements sont contrôlés par le module, alors que les autres le sont par l'environnement du module. Par la suite, les propriétés de vivacité sont composées grâce à une définition moins contraignante de la *satisfaction d'une interface par un module.*

Lam et Shankar estiment que le théorème de la composition s'applique dans un contexte orienté objet, car pour les objets, seules les propriétés de sûreté sont mises en jeu. Ils ajoutent que leur théorie fournit une base sémantique pour les méthodes de preuve associées aux langages de spécification. Dernier commentaire, une notation relationnelle supportant la spécification à l'aide du théorème de la composition a été produite et l'expérimentation se poursuit jusqu'à présent.

Les travaux de Martin Feather

La décomposition des systèmes est une problématique qui va de pair avec la composition des systèmes, car la validation d'une décomposition requiert la composition des sous-systèmes issus de cette décomposition. Martin Feather [Feat87] propose une approche de réalisation de système reposant sur une technique de décomposition des systèmes.

En effet, un système peut être décomposé en une collection de composants qui interagissent. Le processus de décomposition proposé par Martin Feather procède comme suit. Avant tout, la spécification du comportement du système doit être disponible. De cette spécification s'ensuit une élaboration de la décomposition du système; élaboration intuitive mais guidée par des éléments de la spécification. Cette décomposition indique les responsabilités et propriétés de chaque composant. Elle impose des contraintes aux interactions des composants. Cette information nous aide à spécifier complètement chaque composant. Finalement, il faut vérifier si la composition des spécifications des composants réalise la spécification initiale du système tout en satisfaisant aux contraintes énoncées lors de la décomposition du système.

Les avantages de cette technique sont:
1) l'existence d'une description explicite de l'ensemble du système;
2) la description explicite de la décomposition du système;
3) la dérivation du comportement de chaque composant à partir de la spécification du système et de sa décomposition.

Il est possible d'appliquer la méthode de Feather en utilisant le langage *Gist*, langage supporté par un outil d'édition et de vérification des spécifications.

2.2.5. Synthèse des précédentes recherches

Nous retenons des recherches précédentes que la terminologie relative à la composition des objets est abondante. On y retrouve différents termes qui n'ont pas toujours la même interprétation d'un auteur à un autre. En particulier, la sémantique de la relation d'agrégation varie d'un auteur à un autre. Son utilisation dans le dévéloppement orienté objet est quelque peu arbitraire.

Civello note que les méthodes orientées objet ne font pas une distinction entre la sémantique de l'agrégation durant l'étape d'analyse où elle est sensée représenter des phénomènes réels ou hypothétiques et de celle de la conception où elle représente essentiellement une relation entre des objets informatiques. Les travaux chez BellCore concluent à l'existence de formes canoniques de composition. Ces formes peuvent servir de base à des opérateurs de composition ou encore à une algèbre d'objets. Toujours dans ce même contexte, Liu va de l'avant avec l'établissement d'une algèbre des objets calquée sur la sémantique de l'héritage multiple. Au niveau de la composition des spécifications, la notion d'environnement joue un rôle clé dans la composition.

2.3. Concepts et principes de la composition des objets

Comme nous venons de le voir, la terminologie reliée à la composition des objets est très riche. La plupart des auteurs ne s'entendent même pas sur une interprétation commune et uniforme des termes utilisés. Afin de clarifier cette terminologie, nous recourrons au concept d'objet composé tel que proposé par Mario Bunge [Bung77, Bung79]. Soulignons que les travaux de Bunge ont servi de base à la formalisation du paradigme objet par Wand [Wand89]. Nous donnons les caractéristiques principales du modèle de Bunge, caractéristiques que nous utiliserons ensuite pour établir notre approche de spécification des objets composés.

Bien que de nombreux travaux sur la composition des objets aient été réalisés dans les sciences cognitives [Leje67, Leon40, Tvers84], leur apport en génie logiciel, en particulier au niveau de la spécification du comportement des objets composés, reste à démontrer.

2.3.1. Modèle de Bunge

Selon Bunge, un objet composé est une collection d'objets interconnectés par des propriétés et disposant d'une *intégrité sémantique unitaire*; en d'autres termes, la collection peut être traitée comme un seul objet. L'environnement de cet objet est l'ensemble d'objets du monde auxquels il est connecté. Il dispose

d'une *structure* qui est l'ensemble des relations entre ses composantes. Le processus par lequel un objet composé est formé à partir d'une collection d'objet est appelé *composition*. Pour Bunge, un objet composé dispose de propriétés qui sont indépendantes de ses composants. Ces propriétés sont dites *émergentes*. Certaines propriétés perceptibles des objets qui ont été composés peuvent devenir non-observables. Tandis que celles qui demeurent observables sont appelées propriétés *résultantes* ou *inhérentes* de l'objet composé.

Bunge signale l'existence d'une différence conceptuelle entre l'agrégation des objets et leur inclusion dans un ensemble, d'une part et, d'autre part, entre l'agrégation des objets et la relation d'appartenance à un ensemble. Cette distinction évite les écueils de la modélisation des situations avec la relation d'agrégation. Finalement, cet auteur établit une distinction entre deux formes de composition, l'*additive* et la *multiplicative*. Il nomme la première *juxtaposition* et la seconde *superposition*. La différence entre les deux est que, dans la juxtaposition, les composants demeurent accessibles après la formation de l'objet composé tandis que dans la superposition, il est impossible d'accéder aux composants de quelque manière que ce soit sans détruire l'objet composé.

Le modèle de Bunge fixe la terminologie reliée à la composition des objets. Mais comme nous nous intéressons à la spécification du comportement des objets composés, nous devons définir la notion de comportement d'un objet. Plus tard, sera établie la relation entre le modèle de Bunge et la notion de comportement d'un objet composé.

2.3.2. Modèle sémantique du comportement d'un objet

Il y a plusieurs interprétations de ce que représente le comportement d'un objet. Certaines méthodes orienté objet se limitent à la définition des signatures des opérations supportées par l'objet. La signature d'une opération est formée du nom de l'opération, des types de ses arguments et du type du résultat retourné par l'opération. Dans certains cas, cette signature est complétée d'une définition opérationnelle de la sémantique de l'opération, par exemple du code ou du pseudo-code.

D'autres méthodes orienté objet proposent une description déclarative de la sémantique des opérations sous la forme d'invariants, de pré- et post-conditions reliées aux opérations ou encore sous la forme de contraintes comportementales diverses. Les invariants doivent être préservés par toutes les opérations supportées par un objet et sont décrits en termes de relations mathématiques entre les attributs d'un objet. Les pré- et post-conditions s'expriment aussi en termes de prédicats logiques qui, pour un état donné de l'objet, sont vrais ou faux. Les autres formes de contraintes peuvent aussi être exprimées mathématiquement.

Une notion générale de comportement a été proposée par Lamport et celle-ci est applicable aux objets [Lamp94]. Elle est aussi analogue à celle proposée par Hoffman et Snodgrass [Hoff88]. Cette notion consiste à définir un comportement comme étant un ensemble de séquences finies ou infinies d'exécutions d'opérations d'un objet (ou d'états de celui-ci). Un état est défini par l'ensemble des séquences d'opérations qui peuvent mener à cet état et par l'ensemble des opérations qui peuvent être exécutées (acceptées) ou refusées une fois arrivé dans cet état. Une telle interprétation de la notion d'état est abstraite : elle est indépendante de l'implantation de l'objet. Cette vision du comportement d'un objet suppose que :
(1) la sémantique des opérations est définie, (2) l'ordonnancement temporel entre les opérations est aussi défini, (3) l'acceptation ou le refus d'une opération peut être déterminé en fonction de l'état dans lequel l'objet se trouve et (4) lorsque plus d'une opération peut être exécutée ou refusée dans un état donné, l'objet peut décider de l'opération qu'il accepte. Cette notion de comportement englobe celle généralement utilisée dans les méthodes orienté objet où l'on se limite à la définition des signatures des opérations.

Dans le contexte des objets composés, l'état de l'objet composé est fonction de l'état de ses composants. De plus, différents aspects du comportement d'un objet doivent être pris en compte. Ce sont entre autres le non-déterminisme, les différentes relations entre les comportements, le parallélisme et la concurrence, et l'acceptation ou le refus d'une opération. Le fait d'accepter ou de refuser une opération dépend de l'état et de l'environnement de l'objet. Par ailleurs, si nous

nous intéressons à la manière selon laquelle l'objet influence son environnement, alors nous devons considérer le lien de cause à effet entre les requêtes reçues par l'objet et celles qu'il génère. Ces aspects sont à conjuguer avec la possibilité d'avoir des objets passifs et des objets actifs. Un objet passif ne génère aucun stimulus sui generis. Il ne fait que répondre à des requêtes ou envoyer des requêtes qui sont conséquentes à la réception d'autres requêtes. Pour de tels objets, il faut connaître la sémantique de chaque requête et l'ordre dans lequel ces requêtes peuvent être invoquées.

Différents modèles mathématiques sont disponibles pour représenter la notion de comportement et d'interaction des objets. Chaque modèle a ses avantages et ses inconvénients. Nous revoyons dans ce qui suit les principaux modèles sémantiques utilisés pour représenter le comportement et l'interaction des objets.

Modèle à base de processus

Ici, un objet est représenté par un ou plusieurs processus tel que définit dans le calcul des processus [Miln89] dont un exemple est le langage de spécification formelle LOTOS [ISO89a].

Dans LOTOS, à la base de toute spécification se trouve la notion de processus communiquant avec son environnement par des portes (*gates*). Le processus est défini par ses portes et son expression de comportement. Cette dernière définit les événements pouvant se produire aux différentes portes et leur ordonnancement temporel. Les processus peuvent évoluer de façon indépendante (c'est le parallélisme) ou en synchronisme avec d'autres par l'intermédiaire de différents rendez-vous. Un rendez-vous est une interaction impliquant des événements communs à des processus. Tous les processus doivent être prêts et aucun d'eux ne joue le rôle principal d'enclencheur du rendez-vous. Par ailleurs, certains événements peuvent dépendre d'un seul processus: ce sont des événements internes. Un processus composé est un processus formé par d'autres processus. La communication entre ces derniers processus peut être cachée.

Lorsque nous modélisons un objet par un processus LOTOS, la composition des objets devient analogue à celle des processus. Un objet composé est représentable par un processus composé (structuré). Les interactions des différents composants peuvent être cachées aux objets ne faisant pas partie de cette composition: elles constituent alors des actions internes à l'objet composé. Notons qu'un processus LOTOS ne peut être engagé dans plus d'une interaction à la fois. Les différentes formes de composition de processus sont des formes de composition des objets. Ce sont la composition parallèle, l'entrelacement, la synchronisation et la séquence.

Le calcul de processus permet de décrire des activités concurrentes à l'intérieur d'un processus et entre les processus. Cette capacité à représenter la concurrence a poussé plusieurs chercheurs à utiliser ce calcul pour décrire formellement les applications orienté objet concurrentes et la notion d'objet actif. Les représentations utilisant LOTOS diffèrent de celles utilisant π-calculus [Miln93] du fait qu'avec ce dernier, l'interconnexion dynamique des objets est modélisable.

Modèle à base d'automates à états finis communicants

Ce modèle est le plus répandu en dépit de ses limitations, cela parce que la plupart des ingénieurs/concepteurs sont familiers avec les concepts d'état et de transition d'états, parce qu'ils sont aussi habitués à un raisonnement opérationnel reposant sur les principes de transition d'états. Par exemple, la notation dite *statechart* [Hare88] repose sur la modélisation du comportement d'un objet par une machine à états communicante. Dans OMT [Rumb95a], le modèle de base d'un objet est celui d'une machine à états communicante munie de différentes files, pour la réception et l'envoi de messages. La machine à états définit le traitement effectué à chacun des messages reçus et envoie eventuellement des messages. Ce traitement est défini en termes de transitions qui représentent le passage d'un état à un autre. Ces états ne sont pas nécessairement distincts. La réception et l'envoi de message sont possibles dans la même transition ou dans plusieurs transitions.

Pour composer des objets dans ce modèle, nous exprimons le fait que la ou les files d'envoi (de sortie) d'un objet correspondent aux files de réception d'un ou de plusieurs autres objets. Dans ce contexte, la taille des files et le traitement réservé aux messages qui ne sont pas attendus dans l'état où ils sont reçus ont un impact sur le pouvoir de modélisation du modèle. Pour plus de détails, nous renvoyons le lecteur aux publications relatives aux techniques de description formelle SDL [ITU92] et Estelle [ISO89b].

Les modèles à base de processus et de machines à états communicantes sont les principaux modèles dans lesquelles la composition des objets est discutable. En outre, les réseaux de Petri [Pete81] sont considérés comme une extension des machines à états afin de tenir compte de la concurrence. Un objet modélisé à l'aide d'un réseau de Petri peut avoir des actions (opérations) concurrentes (concurrence intra-objet). Ces dernières sont synchronisables avec des opérations d'autres objets. Différentes représentations des objets à l'aide des réseaux de Petri ont été proposées dans la littérature [Kapp91]. Il faut noter que pour composer des réseaux de Petri, ils sont amenés à partager une ou plusieurs places.

Le modèle dit de traces est aussi basé sur les automates finis, excepté les ajouts suivants [Wang93] : la trace représente la séquence des opérations exécutées par un objet depuis sa création; la légalité d'une trace permet de définir la consistance et la complétude d'une spécification d'objet. Une tentative a été faite pour utiliser la synchronisation des traces comme mécanisme de communication entre objets [Cole92].

2.4. Composition des objets à travers le développement orienté objet

2.4.1. Analyse

Des recherches précédentes, il ressort que la sémantique de la relation d'agrégation joue un rôle prépondérant dans l'analyse orientée objet. C'est cette sémantique qui nous permet de distinguer entre les objets composés de l'application et les ensembles d'objets ayant une sémantique unitaire tels que les collaborations et les frameworks. Mais avant de nous attarder sur la sémantique à donner à la relation d'agrégation, mentionnons que lorsque la relation d'agrégation n'est pas distinguable des autres types de relations, les objets composés ne sont pas séparables des ensembles d'objets ayant une sémantique unitaire.

Nous affirmons qu'une telle distinction est nécessaire, car les aspects à spécifier dans chacun des cas ne sont pas identiques. Pour les collaborations et les frameworks, les propriétés multi-objets sont à spécifier, ainsi que la distribution des fonctionnalités à travers les objets et les interactions, celles-ci incluant les contraintes structurelles et comportementales entre les objets. Tandis que pour les objets composés, la structure de ces objets, la hiérarchie (l'abstraction à travers le composé) et l'apport de chaque composant dans la composition doivent être spécifiés.

Sémantique de la relation d'agrégation

En général, comme le notait Rumbaugh [Rumb88], la relation d'agrégation est perçue comme une forme particulière d'association entre les objets. Les relations d'agrégation sont modélisées par:

1) des attributs, lorsque la relation d'agrégation entre les deux objets ne dispose pas de contraintes telles que le partage ou la dépendance des composants;

2) un objet, lorsque la relation d'agrégation des deux objets est soumise à des contraintes qui ne peuvent être réparties entre les objets reliés.

Les modes de représentation suggérés sont inadéquats si les relations d'agrégation ont un impact direct sur le comportement des objets considérés, par exemple, quand une propagation des opérations doit avoir lieu entre les objets participants à la relation d'agrégation [Rumb88]. Quelques fois, seule est figée la sémantique des relations d'agrégation dans les opérations des objets participants. Cela a pour principal désavantage de rendre flou et implicite la représentation des relations d'agrégation du domaine considéré, étant donné que la représentation par un attribut d'une relation entre deux objets ne permet pas d'englober dans cet attribut la sémantique de la relation. L'information relative à une relation est répartie dans plusieurs objets, répartition qui rend hardue la compréhension et l'entretien de la relation.

Par ailleurs, Sakkinen [Sakk89] note que l'agrégation est modélisable par l'héritage multiple lorsque l'identité des composants n'est pas prise en considération ou bien quand un composant est remplaçable par un composé dans le cadre de certaines interactions. Et cela d'autant plus que la composition, mécanisme de combinaison des objets et de leurs descriptions, s'apparente à l'héritage multiple.

À la fin du processus d'analyse, les analystes ont tendance à regrouper artificiellement des objets. Ces groupes sont appelés ensembles, grappes, sous-systèmes, objets collaborants, etc. Toutefois, il arrive que ces regroupements aient une sémantique d'objet. Ces regroupements sont faits en fonction du degré de connectivité entre les objets. Nous soulignons cet aspect, car d'une part, il aide à de découvrir des agrégations insoupçonnées lors de l'analyse du domaine et, d'autre part, il est utilisé pour manipuler un modèle d'objets à un niveau élevé de granularité (abstraction) .

Dans ce qui suit, nous examinons l'apport des outils d'analyse à la spécification des objets composés et des ensembles objets ayant une sémantique unitaire.

Analyse orienté objet de type descendante (top-down)

Elle procède par raffinements successifs et utilise des niveaux d'abstraction différents [deCh91]. Dans ce contexte, l'agrégation en tant que mécanisme d'abstraction constitue un élément clé. Aucune représentation graphique particulière n'est assignée à l'agrégation. L'agrégation est dénotée ici par le concept d'*ensemble*, qui est une grappe d'entités pouvant être des objets ou d'autres ensembles. Nous retrouvons deux types d'ensemble correspondant à deux formes de composition: (1) les ensembles dotés de propriétés au même titre que les objets, c'est-à-dire disposant d'attributs et d'opérations; (2) les ensembles au sens mathématique.

L'analyste est libre de déterminer les propriétés de l'agrégation et de les représenter comme propriétés d'un objet. L'ensemble capture uniquement le fait que des entités du monde sont décomposables en d'autres entités. L'interconnexion des composants n'est pas mentionnée, d'où la difficulté de mettre en évidence la contribution de chaque composant à la composition.

Le langage de spécification TROLL-2

Le langage de spécification TROLL-2 représente explicitement par des constructions définies du langage la composition des objets, les interactions entre les composants ainsi que les interactions des composants et de l'objet composé [Hart92, Hart94]. Ces interactions sont décrites à l'aide de formules de logique temporelle qui leur imposent un ordonnancement temporel.

Lors de la description d'un objet, une section réservée à cet effet permet de décrire les composants ainsi que les contraintes associées à leur qualité de composant de cet objet. TROLL-2 reconnaît l'inclusion du comportement du composant dans celui du composé. En fait, le composé peut substituer n'importe lequel de ses composants. En d'autres termes, le comportement de l'objet composé inclut celui de ses composants. Cette sémantique correspond à celle de l'héritage multiple. Dans TROLL-2, nous sommes restreints à une forme spécifique de composition.

Actions-conjointes

Reino Kukio-Sunio rejette la conception d'une opération attachée uniquement à un objet et nous offre un mécanisme de spécification des interactions entre objets qui associe chaque opération aux objets qui interagissent dans cette opération [Kurk93]. C'est une interaction multi-objets. Cela permet de faire abstraction du mécanisme de communication et du partage des "responsabilités" entre les objets qui coopèrent.

Il s'agit d'adopter une forme de "rendez-vous" généralisé comme moyen de communication. Chaque objet qui participe dans une interaction amène de l'information, impose des contraintes et en retire de l'information. Le rendez-vous a lieu lorsque toutes les exigences des objets sont satisfaites et que chaque objet est prêt pour l'interaction. Ce rendez-vous est appelé *joint-action* (action-conjointe de plusieurs objets). Ce mécanisme est implanté dans le langage de spécification DISCO [Kurk93]. Il a été utilisé avec succès dans la spécification des systèmes réactifs et des protocoles de communication.

Modèles de rôles

Ils sont utilisés dans la méthode dite *Object-Oriented role modeling and analysis* (OOram) [Reen92, Ande92]. Les modèles de rôles partitionnent le domaine du problème. Chaque partie est modélisée par une *structure d'objets interagissants*. Un rôle est une sorte de bloc-légo pouvant être assemblé avec d'autres blocs (rôles) pour former des architectures d'objets. Ces architectures peuvent être génériques et réutilisables. Un rôle, c'est aussi un point de vue sur un objet. Il est indépendant des objets qui joueront ce rôle. Les rôles sont assemblés en modèles de rôles. Cela permet de représenter à l'aide d'un modèle de rôles une abstraction architecturale décrivant le contexte dans lequel l'objet jouera un certain rôle. Ce modèle capture les "patrons" d'interactions et assigne des responsabilités (attributs et opérations) aux différents objets par le biais des rôles.

Les mêmes objets peuvent être décrits dans des modèles de rôles différents, il faut combiner divers modèles de rôles pour obtenir la spécification individuelle

d'un objet. Cette opération de composition des divers modèles de rôles est appelée *synthèse*. Elle suit des règles bien précises de combinaison des rôles. Dans l'opération de synthèse, plusieurs rôles pouvant être combinés, cela donne l'impression de créer des objets composés. C'est ainsi que l'on parle abusivement de composition des objets dans la méthode OOram alors qu'il s'agit de composition des comportements d'objets. Par ailleurs, certains rôles peuvent être décomposés en plusieurs autres rôles.

2.4.2. Conception

La phase de conception est similaire à celle d'analyse, sauf que l'emphase est mise sur le comportement des objets et que leurs interactions sont détaillées. La conception modélise le domaine de la solution qui inclut tous les objets identifiés lors de l'analyse. À cela, elle rajoute les objets purement internes. Les descriptions sont logiques et indépendantes d'une implantation particulière. Généralement, les comportements des objets et leurs interactions sont décrits de manière précise afin qu'ils rencontrent les exigences du système. Les objets issus de l'analyse sont réexaminés, raffinés, étendus ou réorganisés dans le but d'accroître la réutilisation et de tirer profit de l'héritage. Tout ceci est réalisé par le biais de certaines décisions, dites de conception, qui raffinent la description de l'application issue de l'analyse.

Décisions de conception

Les principales décisions de conception qui influencent la composition des objets sont décrites dans ce qui suit, de même que les transformations qu'elles apportent à la description de l'application obtenue lors de l'analyse:

La composition/ décomposition

Nous assistons souvent à une décomposition des objets en parties fines pour faciliter leur implémentation. Lorsque cette décision de conception est appliquée, nous avons généralement la création de relations d'agrégation.

Lorsque deux ou plusieurs objets interagissent de manière étriquée, il faut recourir à leur composition pour minimiser le couplage entre les objets et par la

même occasion le flot de messages entre-eux. Certains concepteurs n'hésitent pas à utiliser l'héritage multiple ou bien à placer des méthodes additionnelles pour faciliter l'interaction de ces objets. Comme pour la décomposition, de nouvelles relations d'agrégation sont créées. En outre, lors de la composition les composants disparaissent et sont remplacés par leur composé.

L'encapsulation/entrelacement

L'encapsulation est une forme de décomposition où un objet est caché à l'intérieur d'un autre, étant donné qu'une décision de conception est dissimulée dans un composant. L'entrelacement est le fait de partager un ou plusieurs composants.

La généralisation/spécialisation

La spécialisation peut consister en la décomposition des objets. La généralisation, du fait de la factorisation des comportements, modifie parfois certaines relations d'agrégation en les remplaçant par d'autres types de relations.

La représentation

Il s'agit ici de la sélection d'une représentation d'un objet ou d'un concept avec les opérations qui leurs sont associées. La décision de représenter une relation par un attribut, un objet ou par l'utilisation de l'héritage multiple est une décision de conception mais que nous retrouvons à l'analyse à cause de la carence en construits sémantiques pour modéliser les relations d'agrégation d'un domaine d'application.

La fonction/relation

Les relations sont souvent remplacées par des opérations d'objets pour faciliter leur manipulation. Cette transformation cause la perte du caractère symétrique de la relation. Ceci s'applique aussi aux relations d'agrégation.

Compte tenu des transformations apportées par les décisions de conception, il est nécessaire d'enregistrer l'impact de ces décisions tout le long du processus de développement. Le concept d'objet composé et de composition d'objet doit

se prêter à une telle manoeuvre. Voyons maintenant comment les méthodes de conception orientée objet procèdent pour la composition des objets.

HOOD: Hierarchical Object Oriented Design

Elle procède par raffinements successifs et insiste sur la structure hiérarchique des objets pour appréhender la complexité dans les domaines liés à l'avionique et à l'astronautique [Robi92]. À la conception, l'intérêt porte sur deux éléments: premièrement, la structure de contrôle de l'application, structure représentée soit par un graphe d'interconnexion soit par une hiérarchie d'utilisation des objets; deuxièmement, la composition des objets, illustrée par un graphe d'inclusion ou une hiérarchie d'inclusion.

La représentation d'une décomposition d'objets se fait sur plusieurs niveaux par un arbre de conception: le *HOOD Design Tree* (HDT). L'inclusion d'objets est représentée soit par l'inclusion graphique des objets dans des diagrammes soit par la subordination des objets dans un arbre de conception. Le lien entre les opérations d'un objet et celles de ses composants est établi à l'aide de la relation *implantée-par*. Cette relation établit une correspondance univoque ou multivoque: univoque, si chaque opération de l'objet est réalisée par une seule opération de ses composants; multivoque, si une opération au niveau de l'objet est réalisée par plusieurs opérations de composants, ce, par l'intermédiaire d'une composante appelée objet interne qui ne possède aucun composant. Les concepts de composition d'objets présents dans HOOD sont intuitifs.

Contrats

Un contrat au sens de Helm et al. est la spécification des dépendances comportementales et architecturales entre des objets qui coopèrent [Helm90, Holl93]. Dans un contrat, nous retrouvons les exigences relatives aux objets qui coopèrent en termes de propriétés et d'interactions entre ces objets. Le fait de localiser cette information dans un contrat permet un regroupement des interactions qui sont reliées de manière sémantique ou causale. Ce regroupement d'interactions est nécessaire lorsqu'une tâche particulière dans un système ou un invariant de celui-ci requiert la coopération de plusieurs objets.

Dans chaque interaction, un objet réalise une partie de la fonctionnalité requise tout en se basant sur les autres objets qui, eux, fourniront le restant de la fonctionnalité.

Un contrat insiste sur l'interdépendance des objets et offre les mécanismes nécessaires à la capture de ces interdépendances. Les interactions des objets ne sont plus encapsulées dans les opérations de ces mêmes objets : elles sont décrites explicitement et de manière localisée dans le contrat. Ainsi, il devient possible de réutiliser des "patrons" (modèles) d'interactions des objets. L'idée sous-jacente aux contrats est de disposer d'une description mathématique et localisée des interactions des objets de manière à rendre prouvable certaines caractéristiques de ces interactions. Il n'est point question ici d'avoir une représentation fidèle des interactions qui peuvent avoir lieu dans une situation réelle ou hypothétique du monde, mais plutôt d'offrir au concepteur ou au programmeur un outil de description des interactions au sein d'une application.

Activités

Kristensen reconnaît l'existence d'agrégats d'interactions entre des entités, agrégats disposant d'une identité conceptuelle. Il le nomme activité [Kris94] et affirme que ce concept mérite une plus grande considération, car il permet de structurer le domaine.

L'auteur s'étonne de voir que les méthodologies existantes ignorent les agrégats d'interactions (activités). Pour combler cette lacune, il propose une représentation de ce concept. Il isole deux aspects essentiels dans la description d'une activité, à savoir: (1) les participants à l'activité et (2) les interactions de ces participants. Ces interactions constituent l'activité proprement dite. Kristensen suggère de ne pas apporter d'extensions majeures aux langages orienté objet pour représenter les activités. Pour ce faire, il montre qu'une activité est représentable par un objet actif dont les attributs sont les participants à l'activité et dont la fonctionnalité décrit les interactions relatives à cette activité. Par cet artifice, il demeure possible de spécialiser et composer des activités à la manière des objets.

Les activités complètent la description des objets individuels. Le concept d'activité ne semble pas nouveau mais, considérée dans une perspective de modélisation conceptuelle, la proposition de Kristensen est originale. Selon Kristensen, le concept d'activité est qualitativement adéquat comparé à l'agrégation des événements utilisée dans OMT [Rumb91] et comparé à l'approche des contrats [Holl93]. Certains langages tels que CLOS [Bobr88] et Beta [Kris93] se prêtent facilement à la représentation des activités. Le concept d'activité est aujourd'hui complètement implanté dans le langage Beta.

2.5. Synthèse et pistes de recherche

2.5.1. Faiblesses des approches actuelles

Il existe quatre écoles de pensée en ce qui a trait à la composition des objets. La première école découle de la modélisation à l'aide de l'approche entité/relations du domaine d'application comme c'est le cas pour la méthode Fusion [Cole94]. Ici, un objet composé est une relation attribuée. Cette relation regroupe les objets qui sont manipulables par le biais de cette relation attribuée. Cette approche met l'emphase sur l'interconnexion des composants au niveau structurel. Elle demeure appropriée dans le contexte des bases de données.

La seconde école met l'emphase sur la relation existant entre l'objet et ses composants. Cette relation est dénommée agrégation, ou de son appellation anglaise, *is-part-of*. C'est le résultat des recherches dans le domaine des bases de données. Ici, la représentation des objets à différents niveaux d'abstraction est importante ainsi que le lien entre ces niveaux. Par ce moyen peut être appréhendée la complexité des objets. La relation is-part-of établit le lien entre les différents niveaux d'abstraction.

La troisième école adopte une vision d'implantation de la composition. Tout objet ayant des attributs de type objet est un objet composé. La structure de l'objet est reflétée par ses attributs. C'est l'école comptant le plus grand nombre d'adhérents. Mais c'est aussi celle qui ne reflète pas la composition des objets.

En effet, cette approche ne distingue pas la composition d'objet d'autres relations qui servent à relier des objets. Ces relations sont implantées par le même artifice : l'attribution.

La quatrième école utilise le concept d'héritage multiple pour représenter la composition des objets. De telles utilisations de l'héritage multiple sont rapportées dans [Carg91, Rumb93a, Sakk89]. Cette approche de composition a l'avantage de se fier au paradigme objet en utilisant l'héritage multiple comme mécanisme de composition. Mais seul ce mécanisme est disponible. Cette situation ne reflète pas la diversité de formes de composition. Par ailleurs, les composants d'une telle composition sont inaccessibles une fois l'objet composé instancié. Cargill a démontré que la plupart de ces situations étaient modélisables en usant de l'héritage simple et de l'agrégation au sens de la troisième école de pensée [Carg91].

Au niveau des langages à objet, peu de travaux significatifs ont été réalisés au sujet de la composition des objets. Néanmoins, le concept d'activité permet de représenter les interactions d'objets.

Un tour d'horizon de l'état de l'art révèle que les approches revues, examinées une à une, n'ont pas d'apport significatif dans la spécification du comportement des objets composés. Les principales caractéristiques des objets composés ne sont pas considérées dans leur totalité. Ces approches mettent l'accent sur un seul aspect tout en négligeant les autres. Plus précisément, sont laissés de côté les interactions des composants, les mécanismes de composition (formes de composition) ainsi que le rapport entre le comportement du composé et celui de ses composants. *Dès lors, il n'est pas étonnant que ces approches prises individuellement ne puissent satisfaire à la spécification du comportement des objets composés.*

2.5.2. Pistes de recherche

En tenant compte des recherches précédentes, nous tentons dans cette thèse d'élaborer une approche de spécification des objets composés. Nous y parvenons par :

(1) La recherche d'une définition adéquate de la composition et des objets composés. Par une définition adéquate nous entendons une définition facilitant la spécification du comportement des objets composés, y compris la relation avec le comportement de leurs composants. Cette définition doit être à la fois adaptée au contexte des systèmes distribués et suffisamment générale pour être utilisée ailleurs.

(2) Un examen approfondi du concept d'objet composé pour répondre aux interrogations suivantes :
- Comment sont distribuées les fonctionnalités dans une composition ?
- Quel rôle joue cette distribution dans la spécification d'un composé ?
- Quel est le lien entre les contraintes structuro-comportementales des composants et le comportement de l'objet composé ?
- Quelles est la sémantique de différentes formes de composition et leur impact sur le comportement de l'objet composé ?

(3) La proposition de moyens par lesquels le concept d'objet composé sera intégré au processus de développement orienté objet.

(4) La possibilité de formaliser correctement la composition des objets.

En somme, il nous faut un concept d'objet composé capable de représenter la diversité d'agrégats qui se retrouvent dans le domaine de l'application. Ce concept doit aussi pouvoir enregistrer les différentes transformations subies par l'objet composé et qui sont imposées par les décisions de conception. Ce même concept doit pouvoir être implanté sans que cela ne nécessite des transformations majeures. La manipulation de ce concept doit être définie par des règles bien précises qui doivent être cohérentes avec les concepts et les mécanismes sous-jacents au paradigme objet.

Nous avons noté que la structure et le comportement d'un objet composé sont reliés. Nous examinerons ce lien, car il nous aidera à mesurer l'adéquation de

la définition du concept d'objet composé et de la sémantique de la relation d'agrégation. Une fois formalisé, ce lien nous servira dans le raisonnement sur des spécifications formelles d'objets composés. Ce lien guidera aussi le raffinement des objets et la réutilisation des spécifications d'objets par la composition de celles-ci.

2.6. Approche proposée

L'approche proposée repose sur la définition d'un cadre conceptuel pour la composition des objets. Dans un premier temps, nous motivons les fondements philosophiques du cadre conceptuel. Nous nous basons sur l'inadéquation des méthodes actuelles de développement orienté objet pour (1) la spécification des objets composés lors du développement d'une application en utilisant une approche de développement descendante, (2) la description des cadres d'applications et (3) la spécification des propriétés multi-objets, parce qu'elles ne peuvent capturer le lien qui existe entre la structure et le comportement d'un objet composé.

Ensuite, nous présentons les différents concepts et principes formant ce cadre conceptuel. Ces derniers insistent sur les aspects prépondérants dans la description des objets composés. Parmi ces aspects, nous retrouvons (1) la structure de l'objet composé en termes de composants et d'interconnexions de ces derniers, (2) les propriétés des composants qui sont aussi des propriétés de l'objet composé et (3) les propriétés de l'objet composé qui sont indépendantes de celles de ses composants.

Le matériel contenu dans cette section est tiré de la publication départementale numéro 949 parue en novembre 1995 et dont un extrait figure dans les rapports de l'*International Conference on Engineering of Complex Systems* de novembre 1995 [Rama95b].

2.6.1. Recours au modèle de Bunge

Tel que nous le montrons dans ce chapitre, de nombreuses approches sont proposées pour la spécification des objets composés. La diversité de ces

approches provient des différentes interprétations du concept de composition des objets. Cela crée une confusion, en particulier, pour la sémantique de la relation d'agrégation.

Afin d'éliminer cette confusion, nous proposons une définition de l'objet composé qui par son fondement philosophique va permettre de reconcilier les différentes interprétations de ce concept et de spécifier adéquatement le comportement des objets composés.

Wand mentionne que de nombreux concepts du paradigme objet tels que l'objet, l'attribut et la relation ont un sens commun [Wand89]. Une interprétation particulière de ces concepts serait contraignante au niveau de la modélisation des applications. Par exemple, prenons le principe général énoncé par Bunge qui veut que les propriétés appartiennent aux objets et qu'elles ne peuvent exister sans l'objet auquel elles sont attachées. Ce principe, lorsque mis en oeuvre dans l'analyse orientée objet, nous empêche de modéliser des propriétés d'objets par des objets à part entière, car les propriétés sont distinctes des objets. Adopter un tel principe nous force à avoir un modèle objet disposant d'un construit différent de celui de l'objet pour modéliser des propriétés d'objet.

Dans sa formalisation du paradigme objet, Wand recourt au modèle de Bunge pour clarifier les concepts sous-jacents au paradigme objet et pour dériver les implications d'une interprétation particulière de ces concepts. Nous trouvons cette procédure adéquate, surtout pour formaliser le paradigme objet, car les axiomes de ce paradigme ne portent pas sur la signification des concepts fondamentaux comme la notion d'objet et de ce qu'est un objet. Ils assument que nous savons ce qu'est un objet. Ils nous indiquent seulement comment les représenter et les manipuler.

Le modèle de Bunge offre des critères permettant de juger de l'adéquation d'une définition particulière de la composition des objets. Par ailleurs, ce modèle systématise les concepts sous-jacents à la composition des objets.

Finalement, nous utilisons les principes de ce modèle dans le développement orienté objet.

2.6.2. Concepts et Principes

Le cadre conceptuel s'inspire largement du modèle de Bunge sommairement décrit dans la section 2.3.1. Il est complété par un ensemble de principes.

> *Un objet qui est composé avec l'un de ses composants donne toujours l'objet composé initial.*

Ce principe introduit la notion d'objet composé et de composant. Il fournit un critère permettant, muni d'un opérateur de composition, de savoir lorsqu'un objet est un composant d'un autre objet, et de savoir lorsqu'un objet est composé.

> *Un objet composé a une structure qui est l'organisation de ses composants et les relations entre les composants et l'objet composé.*

Ce principe définit la notion de structure d'un objet composé qui est en fait son organisation interne en termes de composants et de relations entre les composants et le composé. La structure d'un objet composé peut être vue comme étant l'extension aux objets composés de la notion d'état de l'objet, car ce que nous appelons état d'un objet dans le paradigme objet est souvent un ensemble d'attributs et de contraintes entre ces attributs. Une distinction doit être établie entre les objets sans états et les objets avec un état, par exemple, un entier peut être vu comme un objet sans état, alors qu'un fichier est un objet avec un état, car il peut être ouvert ou fermé. Dans ce contexte, l'état de l'objet étant compris comme l'ensemble des attributs et des contraintes; une structure est un ensemble d'attributs et de contraintes sauf que les attributs en question sont des objets. En plus de cela, la structure définit la collaboration des composants entre-eux et avec le composé.

> *La structure de l'objet permet de distinguer les objets qui sont des objets composés de ceux qui ne le sont pas.*

Ce principe fait ressortir la notion de structure comme étant l'élément clé de la composition. Il faut distinguer la structure hiérarchique de l'objet composé de sa structure horizontale. La structure hiérarchique représente la relation entre l'objet composé et ses composants. Cette dernière est appelée relation d'agrégation. Tandis que la structure horizontale représente les relations et les interactions des composants dans le contexte de la composition. Ces interactions ayant lieu dans le contexte de la composition représentée par l'objet composé, elles peuvent affecter le comportement de l'objet composé. Elles définissent l'activité interne à l'objet composé.

La relation d'agrégation définit un ordre partiel.

Un objet composé a des propriétés résultantes et des propriétés émergentes.

Plusieurs auteurs ne reconnaissent pas l'existence de propriétés émergentes. Il adoptent une approche réductionniste de la composition en général. Interprétée dans le cadre du paradigme objet, cette approche de composition donne lieu à des compositions qui utilisent l'héritage multiple comme mécanisme de composition. Le résultat d'un héritage multiple est entièrement déterminé à partir des classes ayant été héritées. Selon l'approche réductionniste, la composition devient une sorte de logique où l'emploi des opérateurs de composition permet de fabriquer des objets composés. De tels opérateurs découlent des formes génériques de composition. Ce point de vue se retrouve dans la philosophie sous-jacente à divers langages de spécification formelle dont :

- LOTOS [ISO89a] avec la composition parallèle, entrelacée, séquentielle et le choix,
- Disco [Kurk93] avec la superposition,
- TLA avec la conjonction des spécifications [Lamp94].

Le fait de reconnaître l'existence des propriétés émergentes permet d'introduire de nouvelles propriétés au niveau de l'objet composé [Ablo39, Angy39]. Ces

dernières caractérisent l'objet composé en tant qu'objet à part entière et indépendamment de ses composants. Ce point de vue a donné lieu à des approches de composition des objets dont le parfait représentant est la composition par attributs, dans laquelle les composants sont représentés par les attributs de l'objet composé. Dans ce type d'approche, l'objet composé n'est qu'un objet comme tout autre.

En plus de cette distinction, nous en introduisons une seconde à travers le principe ci-après:

> *Parmi les propriétés résultantes, nous distinguons les les propriétés inhérentes et les agrégats de propriétés.*

Les propriétés inhérentes sont des propriétés dont la sémantique dépend d'un composant. Les agrégats de propriétés correspondent aux propriétés multi-objets. Elles sont des propriétés dont la sémantique est établie par composition des propriétés des composants.

A part le modèle de Bunge, il existe d'autres modèles d'objets composés, notamment les modèles des sciences cognitives comme celui de Odell [Odel94] inspiré des travaux de Winston, Chaffin et Herrman [Wins87]. Ce modèle propose une toute autre caractérisation de la composition. La composition y est réduite à trois caractéristiques qui sont la configuration, l'homéoméricité et l'invariance. Nous estimons que de telles approches ne sont d'aucun apport significatif en génie logiciel, car elles sont axées sur la classification des objets et non sur leur manipulation. Dans la définition du cadre conceptuel, nous ne nous inspirons pas de ces modèles.

2.6.3. Le modèle proposé

A travers les principes énoncés, nous introduisons dans le paradigme objet les notions suivantes:

- Un objet composé est avant tout un objet.

- Un objet composé dispose d'une structure qui le distingue des objets non-composés (simples).
- La structure d'un objet composé est faite des composants (la cardinalité et la spécification individuelle) et des contraintes structurelles et comportementales entre ces composants.
- Un composant est un objet qui fait partie de la structure d'un autre objet.
- En plus de la structure, les propriétés d'un objet composé sont de deux ordres: les propriétés implicites, parmi lesquelles nous retrouvons les propriétés inhérentes et les agrégats de propriétés, et les propriétés explicites, qui correspondent aux propriétés émergentes telles que proposées par Bunge. Suivant cette classification, un objet simple dispose uniquement de propriétés explicites.

La notion d'objet (c'est-à-dire telle que connue dans le paradigme objet) est une forme particulière d'objet composé.

Dans cette extension du paradigme objet, le comportement d'un objet composé est relié à la structure de cet objet par l'identification de la contribution de chaque composant au comportement de l'objet composé. Cette identification se traduit à travers la classification des propriétés des objets composés en propriétés inhérentes et agrégat de propriétés. Le principe consistant à établir un lien direct entre la structure et le comportement des objets composés constitue une différence fondamentale entre notre cadre conceptuel et les approches décrites dans la littérature pour la composition des objets.

Placée dans le contexte de la spécification du comportement des objets composés, la notion de propriété telle qu'utilisée dans cette thèse est une abstraction de l'état et du comportement d'un objet. En utilisant le modèle proposé, le comportement d'un objet composé est de deux ordres: le comportement implicite et le comportement explicite. Le comportement implicite regroupe le comportement inhérent et l'agrégat de comportements. Le comportement global de l'objet composé résulte de la composition parallèle des comportements implicite et explicite.

Dans les articles qui forment le reste de cette thèse, nous référons au comportement implicite par ses particularisations qui sont le comportement inhérent et l'agrégat de comportements. Nous référons au comportement explicite par son appellation correspondante dans le modèle de Bunge, en d'autres mots, le comportement émergent.

La mise en oeuvre de ce modèle dans le développement orienté objet est relaté dans le reste de cette thèse. En particulier, le chapitre 3 illustre la mise en oeuvre de ce modèle au niveau de l'analyse orientée objet, cela, à travers l'extension de la notation orientée objet associée à la méthode OMT. Le chapitre 4 formalise la notation orientée objet développée dans le chapitre 3. Cette formalisation est faite par la définition d'un langage de spécification formelle supportant le modèle proposé. Étant donné que les questions reliées au nommage des objets jouent en rôle prépondérant dans la définition des algorithmes et dans la représentation des structures récursives d'objets, il faut examiner l'impact de ce nouveau modèle sur les mécanismes de nommage offerts par le paradigme objet. Cela est fait dans le chapitre 5. Dans le chapitre 6, vous retrouverez une application concrète de ce modèle à travers le processus de dévéloppement orienté objet.

Chapitre 3

Extension de OMT pour la spécification des objets composés

Sommaire

Ce chapitre examine l'intégration du cadre conceptuel dans une méthode de développement orienté objet. Il est la version intégrale de l'article [Rama96b] dans lequel nous montrons comment les concepts et principes du cadre conceptuel sont représentables et applicables. Le cadre conceptuel décrit dans le précédent chapitre est abstrait. Lorsqu'est assumé un modèle objet donné, les concepts et principes sous-jacents à ce cadre sont aisément interprétables.

Pour appliquer notre cadre conceptuel à une méthode orientée objet, il nous faut définir dans le modèle objet supporté par la méthode ce que nous entendons par la structure d'un objet. Ensuite doit être étendue la définition d'un objet afin de considérer la structure de celui-ci. Pour modéliser la structure d'un objet, il faut pouvoir parler des interconnexions et interactions d'objets. Si le modèle sous-jacent ne dispose pas de ces concepts, il faut de nouveau étendre celui-ci. Enfin devra être établie la distinction entre les attributs, les opérations, et les interconnexions selon la classification proposée par le cadre conceptuel. Cela requiert des ajouts à la notation et à la sémantique des différents concepts de la méthode.

La différence entre le cadre conceptuel et OMT [Rumb91] réside dans :
- la notion d'objet composé,
- la relation d'agrégation,
- la spécification du comportement d'un objet composé, spécification impliquant la concurrence entre les composants et l'objet composé,
- les associations impliquant les objet composés du fait qu'elles découlent des composants ou de l'objet composé.

Ce chapitre montre que, dans OMT, l'expression des interactions des objets est basée sur la communication explicite par message ou par des conditions d'exécution des transitions entre les statecharts. Cette approche est détrimentale

pour la réutilisation et la modification des spécifications d'objets composés. A sa place, nous suggérons des contrats qui définissent les contraintes entre les objets interagissant. La sémantique de ces contrats est la composition concurrente (AND-composition) des statecharts des participants au contrat tout en tenant compte des contraintes entre les états des participants.

Par ailleurs, au niveau du comportement de l'objet composé, pour classifier ce comportement selon le cadre conceptuel, il faut distinguer les états et transitions qui vont dans chaque catégorie (inhérent, agrégat et émergent) et adopter un mécanisme qui permet de décrire leurs combinaisons. Une façon de le faire a été d'imposer que le statechart global d'un objet composé soit la composition parallèle (AND-composition) de trois statecharts : un premier qui décrit le comportement inhérent, un second qui est le comportement agrégé et un troisième qui est le comportement émergent. La combinaison de ces trois statecharts repose sur le fait qu'il n'y a pas de relation entre le comportement inhérent, agrégé et émergent d'un objet composé.

Un autre aspect qui a été considéré est la représentation des propriétés inhérentes. Nous avons deux possibilités, à savoir: Si nous décidons que ces propriétés sont offertes par l'objet composé, alors nous devons interdire l'accès aux composants qui offrent ces propriétés. Mais si nous autorisons l'accès aux composants, alors il n'est plus question de permettre à l'objet composé d'offrir ces propriétés. C'est la question de la visibilité des composants. Dans notre extension de la méthode OMT, appelée Extended Object Modeling Technique (XOMT), un objet composé peut avoir des composants visibles et d'autres qui demeurent cachés. La notation supportée par OMT a été étendue pour distinguer dans une composition les composants qui sont visibles de ceux qui demeurent cachés. Notez que seuls les objets faisant partie de la composition, c'est-à-dire le composé et ses composants, peuvent interagir avec les composants cachés.

D'un point de vue méthodologique, cette extension de la méthode OMT facilite la pratique des concepts et principes du cadre conceptuel. Les modifications proposées à OMT se résument à (1) l'adoption d'un mécanisme abstrait de communication entre objets, (2) la distinction explicite dans les modèles objets

entre les propriétés inhérentes, les agrégats de propriétés et les propriétés émergentes. Ces extensions sont apportées à la notation et à la sémantique des concepts.

L'article [Rama96b] veut avant tout être une sorte de diatribe adressée à la composition des objets dans OMT. C'est ainsi que l'article met l'emphase sur l'étape d'analyse. Le passage à l'implantation, notamment les différentes décisions de conception et leur réalisation dans un langage de programmation orienté objet n'est pas mentionné. Les conclusions ne portent pas directement sur des aspects particuliers du développement orienté objet, mais elles insistent sur l'adéquation de procéder à l'extension d'une méthode existente pour favoriser la pratique des concepts et principes sous-jacents au cadre conceptuel. La méthode orientée objet ainsi produite peut être utilisée pour documenter des frameworks et des designs patterns ou encore dans la modélisation des applications où une approche de type descendante serait recommandée, par exemple dans l'ingénierie des systèmes distribués.

La conférence TOOLS-USA96 pour laquelle cet article a été accepté regroupe des experts dans le domaine de la technologie objet. C'est une conférence axée sur la pratique et l'expérience industrielle de la technologie objet. Cette conférence de renommée internationale a un comité technique de révision des soumissions qui comprend au moins quatre juges par article. Ses membres proviennent des milieux académiques et industriels. La conférence est sponsorisée par l'IEEE Computer Society.

Extending OMT for the Specification of Composite Objects

Dunia Ramazani
Gregor v. Bochmann

Abstract:
In engineering and telecommunication applications, it is common to have composite objects. Existing object-oriented methods propose many approaches for modeling these objects. However, these approaches fail to capture the linkage between the structure and the behavior of composite objects. In [Rama95a], a conceptual framework for the description of composite objects prescribes how this linkage can be established by means of a set of fundamental concepts. In order to make this framework more usable in practice, this paper shows how OMT can be adapted and extended to describe composite objects according to this framework. A great deal of these adaptations and extensions require only minor notational and semantic changes to the method. This work also shows how more requirements in connection with composite objects can be captured, made explicit, and precisely stated using an extended OMT.

Keywords: Composite objects, Object-oriented methods, Object-oriented specifications.

This work was funded by the Ministry of Industry, Commerce, Science and Technology, Quebec, and the Natural Sciences and Engineering Research Council of Canada under the IGLOO project organized by the Centre de Recherche Informatique de Montreal.

1. Introduction

The object-oriented approach describes applications by means of objects which collaborate to provide functionality of the application [Aski92]. Some of these objects are defined in terms of other objects. Former objects are called *composite objects*, and the latter are *component objects*. Experience in the specification of engineering and telecommunication applications has shown that *the linkage*

between the structure and the behavior of composite objects plays a key role for understanding the semantics of these applications [Rama95b].

As a matter of fact, in engineering applications, we find objects consisting of physical assemblies of other objects. Such assemblies form composite objects. The behavior of these assemblies can be expressed in terms of the behavior of their components. Since, the behavior of components depend upon their environment, connections among the components affect their behavior. These connections follow *principles of engineering and physics* which can be complex [Rama95b]. In order to understand the behavior of a given composite object, one has to understand the behavior of each component along with their connections. These two form the structure of composite objects. As a consequence, in engineering applications, the behavior of physical assemblies depends upon their structure. In telecommunication applications, the reasoning is similar, except that the connections among components follow *protocols*. In addition, applications are explicitly structured using *containment relationships*. A containment relationship localizes an object inside another object. It allows the description of the containing object behavior in terms of that of the contained object, since the former object has access to the latter object.

Existing object-oriented methodologies propose many approaches to the specification of composite objects. These approaches can be classified into four categories as described in [Rama95b]. For example, the Fusion method [Cole94] represents composition as the abstraction of a given relationship among component objects. This way of handling composition shows the interconnections between the component objects. It focuses on the structure and neglects the behavior of the composite objects. In OMT [Rumb91], composition of objects is described by means of is-part-of (aggregation) relationships relating the composite object to its components. This approach has the advantage of describing, in an hierarchical manner, the structure of an object. In addition, it captures the fact that the composite object has access to its components. In methods such as Booch's OODA [Booc94] and HOOD [Robi92], composition of objects is represented by attribution, i.e., component objects are represented by attributes of the composite object. Attribution captures the hierarchical

organization of composite objects. However, it precludes any distinction between composition and interconnection of objects. There are situations where composition of objects is modeled by multiple inheritance. Such situations are reported in [Carg92, Rumb93a, Sakk89]. Modeling composite objects by multiple inheritance is appropriate when the identity of component objects is not important and the focus is on the resulting properties of the composite object. This way of handling composition, while treating both structural and behavioral aspects of composition, can create problems when reusing such a specification. For more details on these problems, the reader is refered to [Carg91, Carg92, Rumb93a, Sakk89]. All these approaches distinguish themselves by focusing on some aspects and neglecting other aspects of composition. Among these aspects, the contribution of the structure of the composite object to its behavior is ignored. To alleviate this problem, we propose in [Rama95a] a new perspective on composition of objects. In the sequel, we briefly present this approach.

The approach focuses on the linkage between the structure and the behavior of these objects. Among the mechanisms which allow this linkage, we find (1) *visibility* of components, (2) *promotion* of component properties to the status of composite properties (inherent properties), and (3) *aggregation* of component properties in order to form properties of the composite object (aggregate properties). These mechanisms determine *the way requirements placed upon structure and behavior of composite objects relate.* Approaches to the description of composite objects which neglect these aspects will fail to capture the connection between the structure and the behavior of these objects. This failure may have an impact upon *reusability* and *modifiability* of composite object descriptions.

This approach combines features of object-oriented methods and it adds to these the concepts and mechanisms necessary for relating structure and behavior of composite objects. Many of these concepts and mechanisms forming this approach already exist within object-oriented methods or they are allowed by the object paradigm. *It is the way they are organized and utilized in the framework which is novel.* As a consequence, object-oriented methods can be adapted to describe composite objects in accordance with the approach. The amount of

changes and extensions required depends upon the selected method. In this paper, we show how the OMT [Rumb91] can be used and extended in order to describe composite objects according to the approach.

In OMT, structural aspects of an application are described apart from its dynamic aspects. Structural aspects are described using object models. An object model is a diagrammatic representation of classes portraying their attributes, operation signatures, and associations. Dynamic aspects are described by means of dynamic models. A dynamic model describes the temporal evolution of the objects in an application in terms of the changes they undergo in response to interactions with the other objects inside or outside the application. The dynamic model is defined by statecharts, each of which describes the behavior of a class. Features of object and dynamic models allow the description of a great deal of aspects of composite objects in accordance with the framework proposed by our approach as shown in the following table.

OMT models	Features	Aspects of composite objects
Object model	*Class*	- Component classes - Composite class
	Association	- Associations between components - Aggregation association
Dynamic model	*Statechart*	- Behavior of components - Communication by event sending between the components - Behavior of composite objects using the dominant object - Communication between the composite and its components through the dominant object

Table 3.1: Coverage of the framework by features of object and dynamic models

Step	Observation	Remarks
Components and interactions between the components	Could be extended	• *Communication between components is restricted to explicit sending of events between objects. This is inadequate for expressing more abstract interactions like constraints between behaviors.*
Inherent and aggregate properties	Inadequate	• *Not covered, except the is-part-of (aggregation) association* • *Extensions are required for:* - *Visibility/hiding of components* - *Promotion of components properties* - *Aggregation mechanisms for component properties* - *Explicit distinction between inherent and aggregate properties*
Emergent properties	Adequate	• *However, OMT should be extended in order to distinguish between emergent and other properties of the composite.*

Table 3.2: Evaluation of OMT with respect to the framework

The table 3.2 summarizes our evaluation of OMT with respect to steps of the conceptual framework. Extensions required can be achieved using appropriate notational and semantic changes to OMT, except for visibility/hiding of components. The notion of visibility/hiding of components requires fundamental changes to the object paradigm itself, since it is related to aliasing, identification, reference, and typing of objects. Such changes to the paradigm are out of the scope of this paper.

The structure of this paper is as it follows. We begin by describing an application example which is used throughout the paper. It consists of a portable cassette player. This device has features which help to illustrate significant aspects of the approach. Next, we present our three step process for the description of composite objects. This is followed by the description, using OMT, of the portable cassette player according to our approach. Throughout the description, we describe how to represent characteristics of composite objects and we also

reveal shortcomings of OMT. As a consequence, we propose extensions to OMT. Finally, a summary of our contributions closes the paper.

2. The Portable Cassette Player

We assume that the reader is familiar with cassette players. We omit certain details which do not contribute to the essential points of this paper. A portable cassette player consists of two parts, a case and a earphone, as shown in Figure 7.1. It offers functions for managing the playing of cassettes. In the context of our presentation, the earphone can be considered as a simple object. The case is a composition consisting of the following components:

- a cassette compartment into which a cassette may be placed;
- a motor which moves the tape of the cassette;
- a head which reads the content of the tape and produces the corresponding sound;
- an amplifier controlled by a volume controller. It regulates the volume level of the sound produced by the head which is then sent to the earphone;
- a control panel for managing the play function. It consists of five buttons, namely play, forward, rewind, pause and stop.

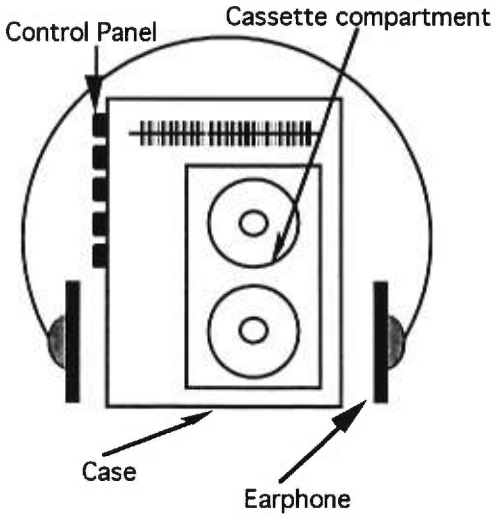


Figure 3.1: A portable cassette player

The normal operation of the cassette player is as follows. When a cassette is

present in the compartment, the user may activate the play function. This is done by pushing the button play. This action turns on the motor and causes the motor to move at normal speed. This action also activates the head. It sets the head to operate in read mode. Then, the sound produced by the head is transmitted to the amplifier which regulates this sound according to the level of volume determined by the volume controller, and transmits it to the earphone. The user may move forward a cassette by pushing the forward button. This action turns on the motor and causes the tape to move forward at fast speed. He may also move backward a cassette by pushing the rewind button. This function is similar to forward except that the tape moves backward. To stop these functions, the user has to press the stop button. The pause button is used to temporarily disable the playing function by turning off the motor. To resume the play function, the user has to deactivate the pause function by pushing one more time on the button.

Each part of the portable cassette player and the player itself are considered as physical objects. As such, they have a weight and a position in space. The portable cassette player has also other properties such as a price which the owner may want to change. The panel consists of five buttons namely, forward, play, rewind, pause, and stop. Each button can only be in two states "On" or "Off". The play, forward, and rewind buttons are mutually exclusive in the sense that two of these buttons can not be in the "On" state at the same time. In addition, when the stop button is "On", forward, play, and rewind buttons must be "Off". We also assume that the control panel has a part number which serves for its identification.

3. Aspects of Composite Objects

Among the aspects which characterize composite objects, our experience has shown that three distinct aspects are important [Rama95a]. These aspects are: (1) its structure; (2) its inherent and aggregate properties; (3) its emergent properties. We may define a composite object as follows.

Definition 1: ***Composite object***

A composite object is an object with an internal structure. It has three kinds of

properties, namely inherent, aggregate and emergent properties. A property may denote an attribute, an operation, a behavior, a structural relationship, a behavioral interaction or a sequence of interactions. The structure of a composite object consists of:

- the components (type and number of instances within the composite object);
- the interconnections and/or dynamic interactions between the components of the composite object.

Definition 2: ***Inherent property***
An inherent property is a property of the composite object such that the semantics of this property is given by a property of some component of the composite object. When the component on which depends this property is absent in the composite object, the inherent property is undefined.

Definition 3: ***Aggregate property***
An aggregate property is obtained by a combination of corresponding properties of all components. The aggregation mechanisms used for combining these properties are defined at the composite object level and they depend on the way the components are interconnected.

Definition 4: ***Emergent property***
An emergent property is a property of the composite object which does not directly depend upon the properties of the components.

What really makes a composite object different from a simple object is the possible presence of inherent and aggregate properties. As a consequence, a composite object showing only emergent properties can be treated as a simple object. The requirements in connection with each of these aspects may lead to complex specifications. Separation of concerns is used in the description of composite objects by subdividing the description into three distinct steps, each focusing on a specific aspect. These steps are the following.

Step 1: Structure of composite objects

In the process of specifying composite objects, we first begin by describing the structure of these objects in terms of components and interactions among these components. The step consists of three activities which are as follows.

(1) *Identification of the components* : Here, we specify the number of components and the names or identifiers which shall be used to address these components within the specification of the composite object.

(2) *Description of individual properties of components* : Individual properties of components consist of requirements in terms of attributes, operations, and behavior that these objects must support in order to be components of the composite.

(3) *Description of collective properties of components* : Collective properties of components consist of structural relationships (object relationships or associations) and behavioral interactions among components. We mean by a behavioral interaction a constraint between two or more behaviors. It affects the behavior of the involved objects. This constraint can be described in terms of dependencies between operations, states, and sequences of operations and states of the involved objects.

Step 2: Inherent and aggregate properties

Next, we have to describe how component properties relate to composite properties through inherent and aggregate properties. This step consists of the following activities:

(1) *Definition of the characteristics of the is-part-of relationship binding components to the composite* : This includes determining visibility and hiding of component objects. Here *visibility* is taken in the sense of controlling access to an object. An object x is visible to an object y if and only if y has a reference to x and y can access x using this reference. An object x is invisible to an object y if and only if (1) y has no reference to x, or (2) y has a reference to x, but y is not allowed to access x using this reference.

(2) *Description of inherent properties* : Note that properties of visible components are automatically considered as inherent properties since they are available to clients of the composite object.

(3) *Definition of aggregate properties* : Care should be taken when defining how component properties compose to form aggregate properties. The composition mechanism applied on these properties needs to be compatible with the composed properties. When composing component properties, the collective properties of the components may conflict with the composition mechanism. As an example, when we use concurrent aggregation of component behaviors, if the components have dependencies, care should be taken to avoid deadlock situations.

Step 3: Emergent properties

Finally, we describe the emergent properties by extending the inherent and aggregate properties through the definition of new properties. This is a case of specialization. These new properties must be compatible with inherent and aggregate properties. Property extension follows subtyping rules, and consistency checking in relation to existing properties is done according to these rules. Emergent properties are described as usual properties of simple objects.

4. Using OMT to describe composite objects

Before illustrating how OMT can be used to describe composite objects in accordance with the approach as described in Section 3, we shall present how concepts of OMT, in connection with composite objects, relate to those of our approach. Among these concepts, notion of composite object, semantics of aggregation relationship, behavior of composite objects, and associations involving composite objects are compared. The material of this section comes from [Rumb94, Rumb95a, Rumb95b, Rumb95c, Rumb95d and Rumb95e] which describe the second generation of OMT.

4.1. OMT versus our approach

What is a composite object?

In OMT, a composite object is an extended form of aggregation where the composite is viewed at a higher level of abstraction than the parts. The whole and its parts are at the same semantic level and can coexist at runtime. According to Rumbaugh [Rumb94], *Composites have no additional semantics but instead serve to organize your understanding of a model.* An aggregate (or composite object) is a set of objects taken together that is viewed as a single high-level object. The dominant object in a composite object is an object which holds information common to the entire composite. Also, the composite is distinguishable from its dominant object.

We view a composite object as an object formed by the superposition of three kinds of properties namely inherent, aggregate, and emergent. A composite is distinct from its components. But components are indiscernible from the composite, i.e. when the components are involved in interactions which are triggered by objects outside the composite, these interactions also involve the composite. Composite objects represent concrete or abstract things of the world which can be described in terms of other things. Special attention is given to relationships and interactions among components.

The concept of composite objects proposed by OMT is appropriate for describing objects which, when associated with one another may form a conceptual entity for analysis, design, and implementation purposes. These conceptual entities may not necessarily exist in the situation represented. This approach contrasts with our approach, since the concept of composite object in our approach can be used to model both existing composite objects and/or composite objects created by the specifier, like in OMT.

Comments on the is-part-of relationship

In OMT, the terms is-part-of and aggregation are used indistinctly to denote composition of objects. Is-part-of relationships are represented by aggregation which is a special form of association. Aggregation is anti symmetric and transitive. There are two kinds of aggregation namely physical aggregation, i.e.

aggregation with multiplicity of one, and catalog aggregation, i.e. aggregation with multiplicity of many. The main distinction between these forms is the possibility of sharing components.

In our opinion, in addition to the properties introduced by OMT, is-part-of relationships are non-reflexive at the instance level. There are many forms of is-part-of relationships. These forms are characterized by seven aspects namely dependency, sharing, physicality, visibility, functionality, homogeneity, and separability. More on these characteristics of is-part-of relationships can be found in [Rama95a]. Special attention is given to visibility of components due to its intricate interaction with aliasing, identification, reference, and typing of objects. In the presence of physical (or concrete) composite objects, some of its clients may have a partial view of the composite by interacting only through its visible components. In order to have an accurate description of such situations, we need to consider the visibility of components. That is why in our approach the emphasis is on this aspect.

Behavior of composite objects

In OMT, the behavior of a composite is similar to the behavior of simple objects. It is described by means of the statechart of its dominant object. There is implicit concurrency between the components. Behavioral interactions are expressed using:

(1) informal text companion to the object, dynamic, and functional models. This can give rise to human interpretation errors during later stages of the development;

(2) operational restrictions by controlling the execution of actions through preconditions and sending of events.

We recognize three kinds of behavior for composite objects namely inherent, aggregate and emergent behaviors. A composite object is logically distributed among its components, therefore there is concurrency in a composite.

Associations involving composite objects

In OMT, there are implicit and explicit associations for composite objects. Implicit associations involve the components, while explicit associations involve only the dominant object. Our approach introduces three kinds of associations. These associations are distinguished by the participating objects which can be the components and/or the composite. Inherent associations are associations which involve some of the components. Aggregate associations involve all the components and the composite. Finally, emergent associations involve only the composite.

To summarize, the main differences between OMT and our approach lie in:

a) *raison d'être of composite objects.* In OMT, composite objects are analysis, design, and implementation artifacts. This contrasts with our approach where the concept is in addition also used to represent composite objects existing in hypothetical and/or real-world situations.

b) *properties that characterize composite objects.* In OMT, composite objects are indistinguishable from non-composite objects. We distinguish between composites from non-composites through the presence of inherent and aggregate properties in composite objects.

c) *linkage between structure and behavior of composite objects.* This aspect is neglected by OMT.

4.2. Description of composite objects

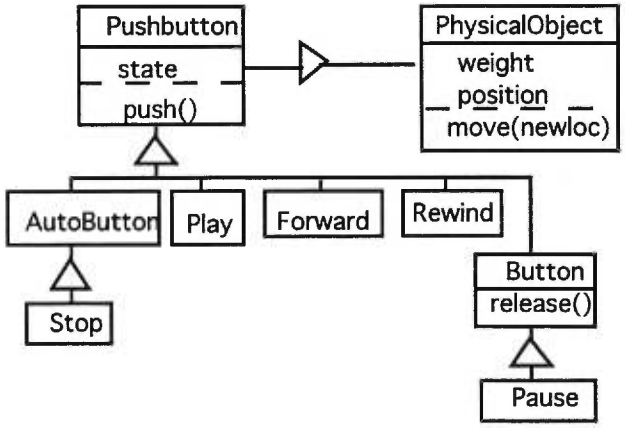
In order to show how OMT can be used to describe composite objects in accordance with our approach, we shall follow the process defined in Section 3. For each step, we describe how OMT can be used and what are the adaptations and extensions required. In addition, we use the portable cassette player to illustrate the ideas.

Step 1: Structure of composite objects

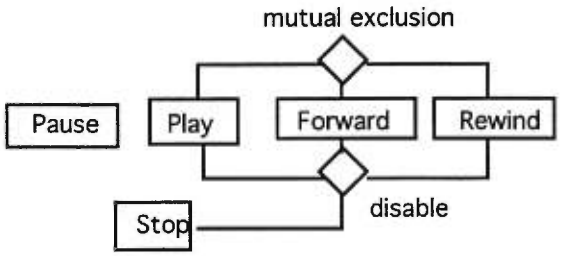
Individual properties of components are adequately represented using the concept of class of OMT in conjunction with statecharts associated to classes. The class captures attributes and operations of components, while the statechart captures the behavior of components. Structural properties are represented by means of

OMT associations. In OMT, behavioral interactions are represented by explicit communication between the statecharts or by using the state of one class in the guard of transitions of another class as proposed by Rumbaugh [Rumb95b]. This way of representing behavioral interactions is harmful to reusability and modifiability. The problem is that at analysis phase, behavioral interactions have to be expressed in a more abstract way so that the description does not introduce implementation bias. One way to achieve this is to describe these interactions by constraining the behavior of the involved objects. This is done by using predicates which constrain features of various statecharts. The way the conditions imposed by these predicates are realized are left for other phases of the development.

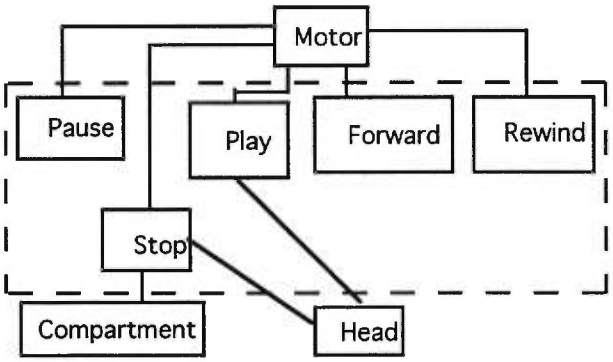
Structures of composite objects are represented by (1) object models describing attributes and operations of components, and structural relationships between components; and by (2) dynamic models describing behavior of components and behavioral interactions among the components. The objects outside the composite object are separated from the components by a dashed rectangle surrounding the latter objects. For illustration, consider a control panel. Its structure is presented in the next figures. Each type of button is represented by a distinct class in order to facilitate the presentation of structural relationships and behavioral interactions among the components. A distinction is made between configurational and external relationships. Configurational relationships are associations among components, while external relationships are associations between components and objects outside the composite.



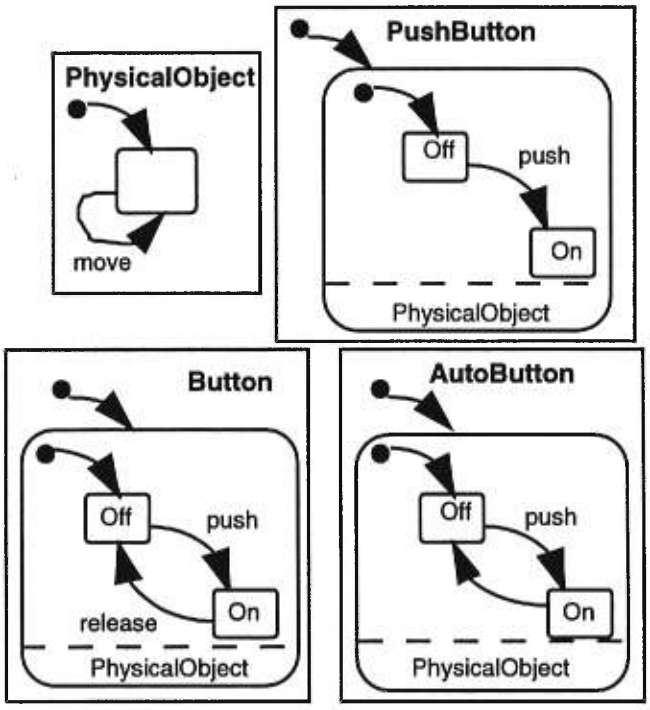
Attributes and operations of components



Configurational relationships



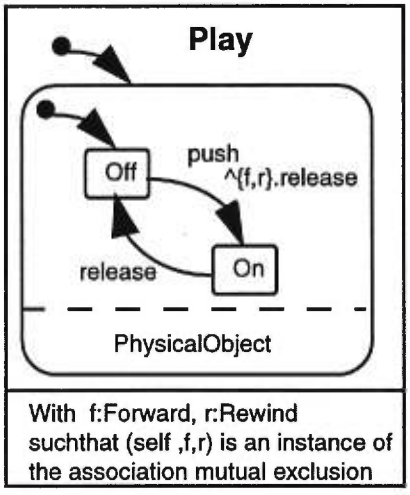
External relationships



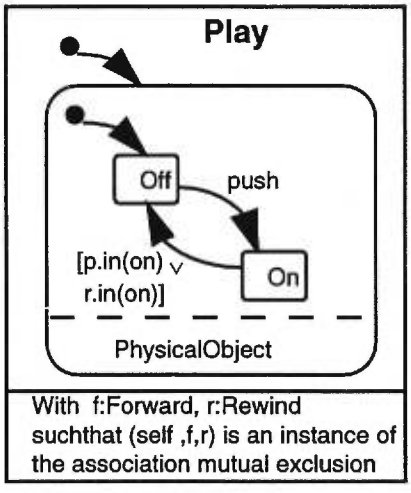
The difference between PushButton and AutoButton is that in an AutoButton, the button automatically comes back to the state "Off" at the completion of the push

operation.

In OMT, we have two alternatives for representing the constraints among the buttons. Consider for example the mutual exclusion between Play, Forward, and Rewind buttons. One way is to introduce a *release* operation which will be triggered when another button is *On*. This makes the statechart of the Play button to look like the statechart *Sending events*. The other alternative which is illustrated in the statechart *Guards on transition* consists of introducing a spontaneous transition with a guard using the states of the mutual exclusive buttons. The notation *object.in(state)* is used in OMT to denote the predicate which is true when the object is in the state "state" and false otherwise. According to Cook and Daniels [Cook94], descriptions of message sending confuse specification issues with implementation tactics. It appears that the approach *Sending events* over-specifies the interactions between the buttons since it introduces unnecessary sequencing between the transitions of the buttons.

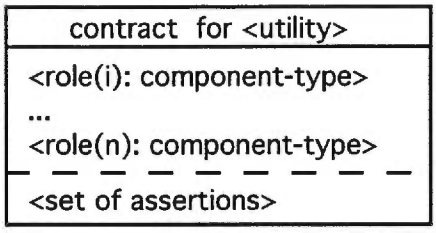


Sending events



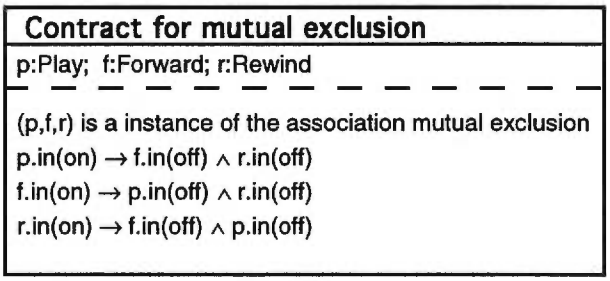
Guards on transition

Instead of using this approach, we propose an extension to the notation which consists to describe interactions between classes by means of contracts. The concept of a contract is similar to the Contract technique introduced by Holland [Holl93]. In our approach, we represent contracts using the ad hoc textual convention illustrated below. In this ad hoc notation, <utility> is the name of the association abstracting the interactions between the classes.

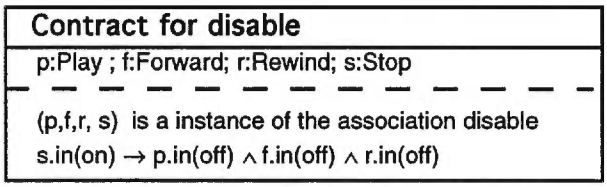


Contracts in OMT

In a contract, an assertion can be any predicate allowed in OMT specifications. For instance, the contracts describing the associations "mutual exclusion" and "disable" are represented by the following diagrams.



(I)



(II)

A contract represents the concurrent composition of its participant statecharts such that the behavior of the participants conforms to the constraints explicitly stated in the contract. AND-composition of statecharts corresponds to concurrent composition of statecharts as defined in [Rumb95b]. The composed objects can interact explicitly by sending events. They can interact implicitly if one object has a guard condition that depends on the state of another object. In our contract notation, the interactions between the statecharts are expressed in an abstract manner using predicate logic as well as OMT constraints. The global statechart represented by the contract may have less states than the statechart obtained using only AND-composition without communication between the member statecharts. This is due to its set of assertions (i.e. the constraints). For instance, the

statechart (I) has 4 states while the AND-composition without communication between Play, Forward, and Rewind buttons has 8 states. The statechart (II) has also 4 states compared to the AND-composition of its member statecharts which has 16 states.

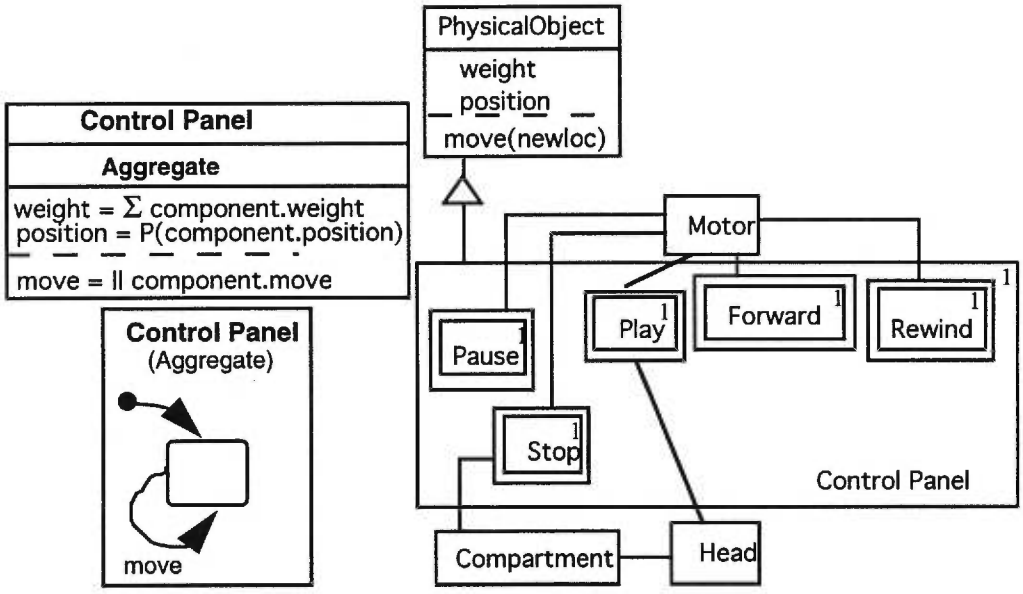
Step 2: Inherent and aggregate properties

We found that the notation used in OMT for representing aggregation relationships is adequate. This is done by graphically including the component classes within the composite class. Multiplicity of components is indicated by a number at the right corner of the class. However, the notation does not cover visibility of components. Therefore, we extend OMT as follows. Within a composite, visible components are represented by using double rectangles and hidden components by simple rectangles.

OMT does not distinguish between inherent and aggregate properties of a composite object. Therefore, we need also extensions to the notation. *Concerning attributes, operations, and behavior due to visibility of components, the notation proposed for visible components implicitly conveys the idea that properties of visible components are also properties of the composite.* For attributes, operations, and behavior which originate from the hidden components of the composite, we require a specific box and statechart to represent these. Inherent relationships which involve visible components are represented by lines which cross the composite object boundary and end up at the visible component. Inherent relationships which involve hidden components are represented by lines ending up at the composite boundary. The mapping with the corresponding component is achieved by drawing a dashed line within the composite from the component to the end point of the relationship at composite boundary. An end point is represented by a small rectangle at the composite object boundary. Keep in mind that aggregate properties may imply constraint propagation, as described in [Rumb88, Mili90], from the composite to its components. Aggregate attributes, operations, and behavior are represented like inherent properties originating from the hidden components, except that these attributes and operations are annotated with aggregation assertions explaining how the component properties are combined. Aggregate relationships are represented by

lines ending up at the composite boundary. At this end point, the annotation {A} indicates that it is an aggregate relationship. In OMT a dominant object can be used to represent aggregate attributes, operations, and behavior. However, the notation does not distinguish between aggregate and inherent properties.

For illustration, consider again the control panel. All its components are visible. Therefore, the properties of the buttons become inherent properties of the control panel. Its weight and position constitute aggregate attributes. It has only one aggregate operation which is "move". In the definition of these aggregate properties "component" stands for any component of the control panel, i.e. the "component" may be Pause, Play, Forward, Rewind or Stop. We may also use contracts to define the semantics of these properties.

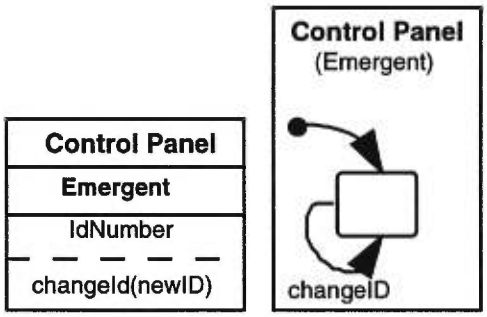


Step 3: Emergent properties

In OMT, emergent attributes, operations, and behavior can be represented by attributes, operations, and behavior of the dominant object. Using this artifact, the distinction between aggregate and emergent properties is not clear. Instead, we propose to extend the method.

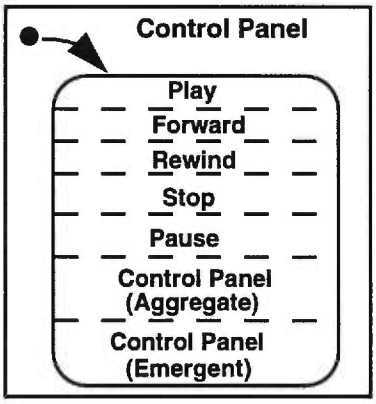
We use a diagrammatic notation similar to the one proposed for aggregate properties with an indication that these properties represent emergent properties.

For the control panel, when considering its identification number and the operation used for updating this number, this looks like illustrated below. Unfortunately, in this case, there is no emergent relationship.



Behavior of composite objects

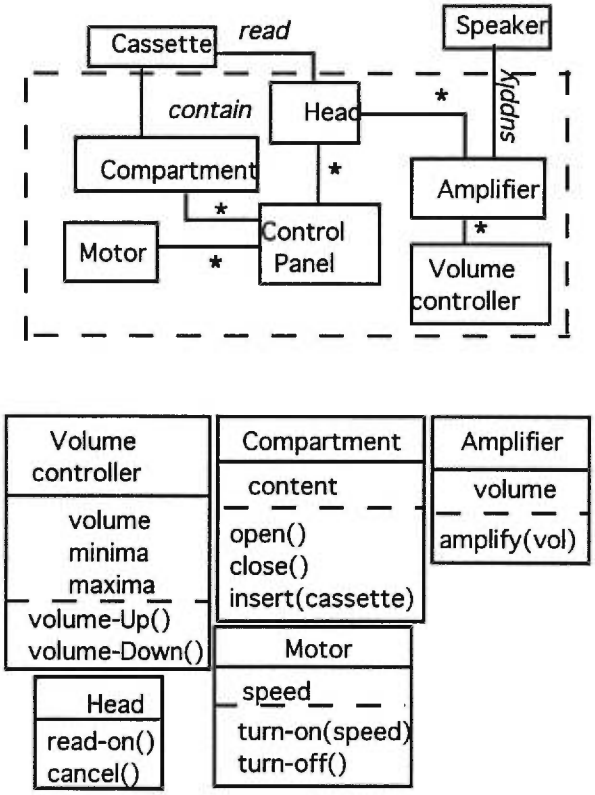
In our approach, the behavior of composite objects is split into three parts namely inherent, aggregate, and emergent behaviors. The behavior of the composite object is represented by the statechart resulting from concurrent-composition of the behavior of its visible components, inherent behavior, aggregate behavior, and emergent behavior. For the control panel, this looks like illustrated below.



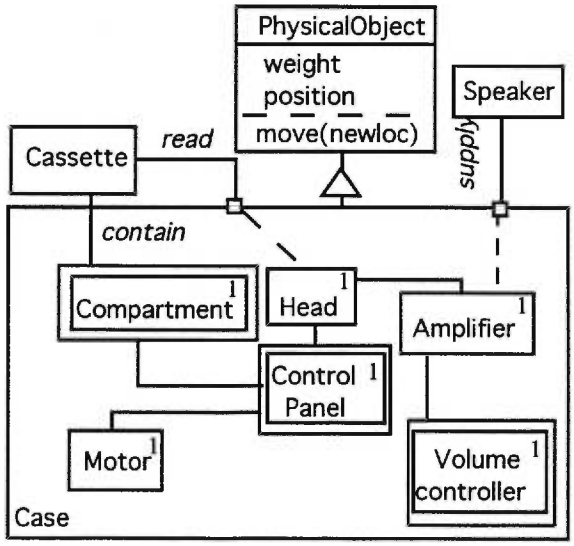
Other composite objects in the context of the portable cassette player

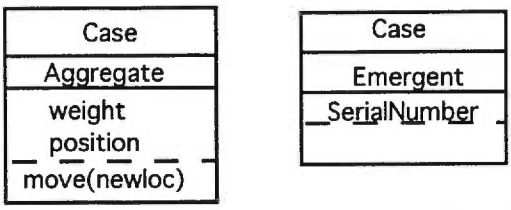
For sake of brevity, we make some simplifications in the presentation. The case has six components, namely the control panel, the motor, the compartment, the head, the amplifier, and the volume controller. There are a lot of behavioral interactions between the components of the composite objects case and cassette player which require an explicit contract. However, due to lack of space, we only

indicate in the figures the associations which require a contract. This is done by labeling with a "*" symbol the line representing the association.

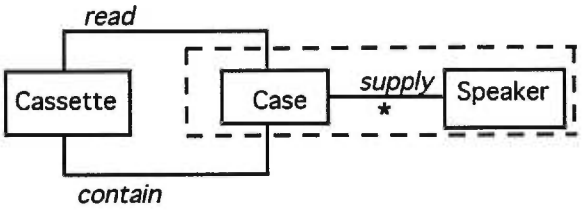


I: Structure of the case

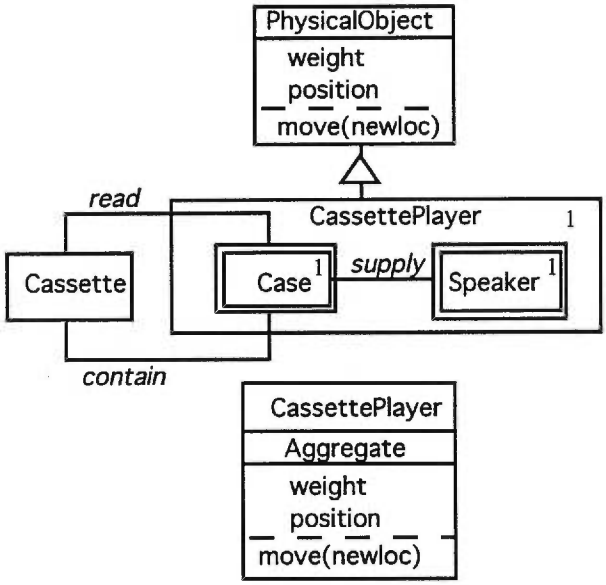




II: Inherent, aggregate and emergent properties of the case



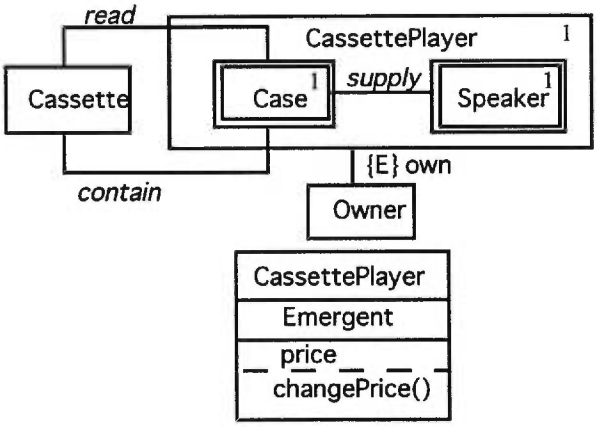
III: Structure of the cassette player



IV: Inherent and aggregate properties of the cassette player

Only the control panel, the compartment, and the volume controller are visible components of the case. As in the case of the control panel description, the properties of these visible components become inherent properties of the case. Therefore, the case has three inherent relationships, namely contain, read, and supply. The association "contain" links the visible component compartment to a cassette. In addition, the head is allowed to read the cassette and the amplifier to supply the earphone with sound. These properties of hidden components are

mediated by the case and become inherent properties of the case. Using the consideration that all parts of the cassette player are physical objects, we may define the weight, the position, and its change as aggregate properties of the case. We consider only one emergent attribute for the case, which is its serial number. The overall behavior of the case is defined by superposing the behavior of its visible components (control panel, compartment, and volume controller), its inherent behavior realized using hidden components (head and amplifier), as well as its behavior as a physical object (aggregate properties). We omit the statechart representing this behavior since it is similar to the statechart of the control panel.



V: Emergent properties of the cassette player

The cassette player has two components, the case and the earphone. They are related by the fact that the case provides the earphone with the sound. Both the case and the earphone are visible components of the cassette player, therefore the properties of these visible components become inherent properties of the cassette player. Aggregate properties are similar to those defined for the case. We have chosen to model the ownership of the cassette player as an emergent property. Besides this property, we may consider the price attributed to the cassette player and to allow updating this price when necessary. These constitute additional emergent properties of the cassette player. The overall behavior of the cassette player is defined by superposing the behavior of its visible components (case and earphone), its behavior as a physical object (aggregate properties), as well as the possibility of owning the cassette player and defining its price.

More on visibility of components

The concept of visibility of components, as introduced by our approach, has much to do with subtyping and it should not be confused with visibility of class features proposed for object-oriented programming languages. In our approach, when a component is visible, this means that its substitution by a more specialized one allows the composite to offer more properties. This contrasts with visibility of class features which introduces three kinds of visibility namely public, protected, and private. Public features are accessible to clients and can be used in derived classes. Protected features are inaccessible to clients but can be used in derived classes. Private features are only accessible within the class. Visible components are public features of composite classes while hidden components may be protected or private features of composite classes. Visibility of class features does not convey the idea that the composite shares its interface with its visible components and that the latter objects are substitutable by the former object.

We may draw an analogy between visibility of components and subtyping (inheritance). The composite class is a subtype of all its visible component classes since they have common properties and in most of the cases the composite offers all the properties of its visible components. When a composite class defines more than one visible component, this situation is similar to multiple inheritance. However, a composite object created by multiple inheritance can not have multiple instances of the same class as its components. In addition, the components can not be replaced. As a consequence, we do not recommend the modeling of visibility of components using visibility of class features or subtyping (inheritance). The concept of visibility of components has to be further examined and suitable concepts, mechanisms, and principles have to be defined.

5. Conclusion

Various aspects of composite objects are explicitly captured using an extension of OMT. This allows to envision their usage in the specification of requirements with respect to composite objects. Returning to the purpose of this paper, the

contributions are twofold:

(1) On the one hand, OMT is extended with:
- a concept of composite object which encompasses both concrete objects of an application, and abstractions in the form of aggregations created for analysis, design, and implementation purposes;
- an explicit linkage between structure and behavior of composite objects by means of a set of fundamental concepts;
- an understanding of the major differences between composite and non-composite objects according to structural and behavioral characteristics of these objects;
- more abstract forms of communication between objects;
- a new concept of visibility/hiding of component objects; etc.

(2) On the other hand, our approach to the description of composite objects described earlier [Rama95a] is made more usable in practice through its integration within OMT.

In the literature, the work done by Mili and colleagues [Mili90] is close to the conceptual framework. In [Mili90], there is an in depth study of how the structure of composite objects may affect or influence their behavior. In particular, they assume that some behavioral and functional relationships between objects are the consequences of the structure relationships. Structure relationships include the aggregation relationship and the connections between the components. We may relate this approach to our conceptual framework as follows. In [Mili90], the concept of change propagation between related objects is key to the specification of constraints between the composite and its components and it implicitly indicates the connection between the structure and the behavior of composite objects. Considering our conceptual framework, we may classify the properties of a composite object into two classes. The first class involves properties for which change propagation is necessary from the composite to its components. The second class involves properties for which there is no change propagation. This latter class of properties corresponds to emergent properties in our framework. The former class of properties can be

further subdivided into two categories. One for which the change propagation are limited to one component, corresponding to inherent properties, and one for which the change propagation affects all components. This latter category corresponds to aggregate properties. While the two approaches, ours and [Mili90] can be reconciled, there is an important difference between the two approaches. In [Mili90], the structure of an object is not visible to outside objects such that an outside object can not request operations that directly manipulate the component objects. In our approach, we allow that certain components may be visible; these components may be accessed and manipulated from the objects outside of the composition.

Our experience with engineering and telecommunication applications has shown that extended OMT allows to capture explicitly more requirements in connection with composite objects. It has also shown that the extended OMT improves reusability and modifiability of composite object descriptions due to the application of (1) separation of concerns and (2) superposition through concurrent-composition of component behaviors. In the future, we plan to look at other object-oriented methods, such as the Fusion method [Cole94], and Object Oriented Design with Applications [Booc94]; and also to update an existing tool supporting OMT in order to provide means for using the approach proposed in this paper. We shall also further examine the implications of visibility/hiding of components upon the object paradigm and how these concepts can be fully integrated within object-oriented methods and programming languages.

Chapitre 4
Le langage de spécification CSL

Sommaire

Après avoir intégré dans une méthode orientée objet les concepts et principes du cadre conceptuel, il fallait adopter une notation orientée objet qui nous aiderait à rédiger des spécifications respectant les concepts et les principes du cadre conceptuel. Cette notation devait avoir une sémantique formelle et permettre le raisonnement sur les spécifications produites. L'idée était d'autoriser un traitement formel de la composition.

Pour ce faire, nous pouvions formaliser directement les concepts et principes du cadre conceptuel en utilisant par exemple la logique dynamique typée. Une telle approche avait été expérimentée avec succès par Wieringa dans la formalisation d'un modèle conceptuel [Wier95]. L'autre approche de formalisation consistait à intégrer dans une technique de description ou dans un langage de spécification formelle les concepts et principes du cadre conceptuel.

Nous avons opté pour cette dernière approche, car elle est un prolongement de la méthode OMT. Elle offre des avantages qui sont, entre autres, la formalisation du lien entre la structure et le comportement des objets composés, ainsi que l'incorporation de ce lien dans les techniques de décharge de preuve et de vérification de propriétés des objets composés. De plus, cette formalisation sert de base à la mise en oeuvre de générateur de code, de prouveur de théorème et d'interprète de spécifications de compositions.

Après avoir choisi une approche de formalisation, il a fallu se pencher sur l'influence de la composition sur le processus de spécification formelle, ceci du point de vue de la structuration des spécifications formelles, de leur raffinement et de leur composition. Cette interrogation porte sur l'impact de la structure de l'application en termes de compositions sur celle de la spécification formelle. Une approche formelle de spécification est dite compositionnelle, premièrement, lorsque la preuve de consistance est reliée à la structure de la

spécification; deuxièmement, s'il est possible de déduire les propriétés de la composition à partir des spécifications des composants et de la manière selon laquelle ces composants sont assemblés. Ainsi, notre formalisation du cadre conceptuel à travers l'intégration dans un langage de spécification formelle devait être compositionnelle.

Pour décrire formellement une composition, il faut avant tout pouvoir parler d'objets ayant une structure, d'où le problème de la description formelle de telles structures. Nous avons indiqué dans le chapitre 2 que les structures d'objets composés sont faites de composants, d'interconnexions et d'interactions des composants. Nous nous sommes penchés sur la spécification formelle des associations. Le lecteur retrouvera la discussion à ce sujet dans l'article [Rama97] qui n'a pas été repris dans cette thèse. Cet article déplore la multiplicité des interprétations du concept d'association. Il propose un cadre conceptuel pour les associations et fournit quatre approches de formalisation des associations décrites à l'aide d'Object-Z [Duke94]. Les approches de spécification des associations sont utilisées dans la spécification de la structure des objets composés. Une extension d'Object-Z a été produite pour spécifier des objets composés. Le choix d'Object-Z a été dicté par le fait qu'il est utilisé pour la normalisation. Le problème avec ce langage est son inadéquation pour décrire les interactions entre objets. En effet, Object-Z est adéquat pour spécifier des propriétés structurelles. Mais, au niveau des interactions d'objets, ce langage ne précise pas comment elles se produisent, ni comment les opérations sont invoquées, ni ce que représente le comportement d'un objet.

Pour pallier à ce problème, nous avons utilisé un autre langage de spécification, nommément TLA (Logique temporelle des actions), qui est bâti sur la notion d'interaction. Par ailleurs, la logique temporelle permet d'exprimer l'ordonnancement temporel des interactions. Nous avons entrepris l'extension de ce langage. Le fait qu'il ne soit pas orienté objet nous a conduit à y ajouter une couche d'orientation objet qui tient compte du typage des objets, de la représentation des classes par des modules ayant un record pour représenter les attributs, des assertions pour représenter les invariants, des actions et une expression de comportement. Ce langage a été baptisé OOTLA. Toutes les

extensions apportées au langage TLA sont assez difficiles à percevoir du premier coup d'oeil, car une spécification OOTLA est en fait un ensemble de modules dont l'un ajoute une couche d'orientation objet. Par ailleurs, la communication entre les objets se fait par le biais de variables partagées. Tout cela rend difficile l'utilisation des spécifications OOTLA. Nous avons décidé de produire un nouveau langage de spécification en utilisant certains principes de OOTLA. Dans ce nouveau langage en plus des caractéristiques de OOTLA, nous avons mis en oeuvre les principes suivants:

1) Les interactions des objets se font dans un contexte particulier. Ce contexte est défini par une association liant les objets qui interagissent. Ainsi, les interactions des objets n'ont lieu que dans le contexte des associations qui lient ces objets. Les interactions des objets sont décrites en tenant compte de la synchronisation des actions des objets impliqués.

2) Le concept d'environnement d'un objet est maintenant pris en compte dans chaque action (méthode) supportée par l'objet.

3) Les objets sont composés par le biais de leurs interactions. Nous avons représenté au niveau de chaque action supportée par un objet les hypothèses faites par l'objet sur son environnement.

4) Les interactions sont explicitement définies en utilisant des constructions spécifiques pour les représenter.

Étant donné que le langage obtenu obéit aux principes du cadre conceptuel pour la composition des objets, nous l'avons baptisé Composition Specification Language (CSL). CSL ayant pour origines OOTLA, nous avons défini des règles de traduction du langage en TLA. Cela permet de profiter des environnements de développement disponibles pour le développement et la vérification des spécifications TLA. La théorie de composition du langage CSL repose sur des principes de composition similaires à ceux énoncés par Abadi et Lamport (Composition Principle) [Abad95], Lam et Shankar (Theory of Interfaces) [Lam94] et Fiadeiro et Maibaum (Theory of module

interconnection) [Fiad92].

Le reste de ce chapitre est la version intégrale de la publication départementale numéro 1134 [Rama98a] soumise à ACM Transactions on Software Engineering Methodologies (ACM TOSEM). Mais, étant donné que le chapitre 6 reprend les idées principales de cette publication et qu'il a déjà été publié dans FMOODS99, le comité éditorial de la revue nous a suggéré de faire des modifications qui permettraient de distinguer les contributions de chacune des publications. FMOODS est organisée par l'IFIP à travers le groupe TC6 WG6.1. Elle a pour objectifs de présenter les avancements dans les domaines des méthodes formelles, des systèmes distribués et de la technologie objet.

Nous travaillons actuellement sur l'application du langage CSL à la documentation et à la vérification des spécifications de design patterns et des frameworks.

Composition Specification Language

D. Ramazani and G. v. Bochmann

Abstract

This paper presents the Composition Specification Language (CSL). Before the exposition of this language, the concepts and principles underlying its design are motivated. CSL is based on the assumption that object interactions occur only in the context of object associations. When this principle is applied to the specification of composite objects, we can relate the structure of composite objects to their behavior. The structure of a composite object records its components, their associations, as well as their expected behavior in the context of the composite object.

Objects are composed through their interactions. This is based on viewing the structure of a composite object as the backbone of the composition. By hiding certain interactions in a structure and allowing the refinement of such structures, we are able to describe the behavior of composite objects. These observations have led to the design of CSL. It has been used for describing components, interactions among these, and object compositions. CSL is a specification language essentially geared towards applications with rich dynamic behavior involving complex interactions between the application's components. A case study using CSL and demonstrating how it can be integrated in the object-oriented development life-cycle can be found in [Rama99].

The language formal semantics is based on its translation into TLA. In addition, CSL has a composition rule which is comparable to the composition rules proposed in: (1) Abadi and Lamport's composition theorem [Abad95], (2) Lam and Shankar's theory of interfaces [Lam94], and (3) Fiadeiro and Maibaum's theory of interconnection [Fiad92].

Keywords: CSL, Object composition, Object interactions, TLA.

1. Introduction

The object-oriented approach describes applications by means of objects which collaborate to provide the functionality of the application. Some of these objects are defined in terms of other objects. Such objects are called composite objects, and their constituents are component objects.

There are various approaches to object composition [Rama95a]. Each approach is based on a specific composition metaphor. For instance, using the connection metaphor, i.e. composing objects by connecting them, rules of object composition reduce to the rules of interoperability of objects. More specifically, when the client/server approach is used to connect objects, compositions are based on define/use patterns between object specifications [Alle94]. As indicated in [Rama96a], this metaphor is based on not distinguishing object interconnection from object composition. It has the consequence that abstraction gained from composing objects is not used when reasoning about the specifications (see [Rama96a] for more detail).

On the other hand, some authors focus on the capability of the object paradigm to model the real world. This has led to an ontological approach to object composition [Wand89] [Rama96b]. The ontological approach characterizes an object by its properties. Here, the term property is an abstraction for states and behavior of a given object including structural constraints involving this object. Composite object properties are classified into inherent, aggregate, and emergent properties. Inherents represent the properties inherited by the composite object from its components. Aggregates are composition of component properties. Emergents represent properties which are specific to the composite object, i.e. they do not depend on either inherents or aggregates. The ontological framework has been criticized for being decoupled from software engineering. For instance, it is not clear how this classification of properties affects the quality of the specification or the design. In addition, its impacts on the architecture of the application being specified, designed, implemented or tested are unknown. However, the ontological approach

remains appealing because it conforms to the way we model the real world.

In [Rama95a], an attempt to bridge the gap between software engineering principles and the ontological framework is made. This is achieved by recognizing that a composite object has a structure in which each component plays a specific role. Using this technique, both connection and ontologic composition metaphors can be used together for describing composite objects. However, this requires linking the structure of composite objects to their behavior. Among the mechanisms which allow this linkage, we find:

(1) *Visibility* of components, i.e. some of the components can be manipulated by objects outside the composition. By objects inside the composition, we mean the component objects and the composite object;

(2) *Aggregation* of component properties in order to form properties of the composite object. Component properties can be combined to form properties of the composite object.

These mechanisms determine how requirements placed upon structure and behavior of composite objects relate. It is interesting to notice that these two mechanisms rely on how the objects communicate with one another.

Close examination of existing interaction mechanisms have shown that multiparty and synchronous interactions, i.e. rendezvous interactions [ISO89a], are suitable for describing interactions which occur in the context of an object association [Rama97]. In this paper, we show how a complete specification language can be designed based on synchronous interactions and by imposing that these interactions occur only in the context of object associations. The use of multiparty interactions has allowed the specification of the behavior of components in the context of the structure of the composite object. Objects are composed through their interactions. This is based on viewing the structure of a composite object as the backbone of the composition. From the structure, we may hide certain interactions and/or add new actions to it. This means we are considering the structure as an object by itself. Using this

technique, we are able to describe the behavior of composite objects. These observations have led to the design of a specification language for object compositions, named Composition Specification Language (CSL).

This specification language has been used for describing components, interactions among these, and object compositions. CSL is a specification language essentially geared towards applications with rich dynamic behavior involving complex interactions between the application's components. A case study using CSL and demonstrating how it can be integrated in the object-oriented development life-cycle can be found in [Rama99].

The language formal semantics is based on its translation into TLA [Lamp94]. The choice of TLA is motivated by both theoretical and practical reasons. TLA has some nice reasoning rules which can be used to prove properties of CSL specifications once they are translated into TLA. The CSL composition rule is comparable to the composition rules proposed in: Abadi and Lamport's composition theorem [Abad95], Lam and Shankar's theory of interfaces [Lam94], and Fiadeiro and Maibaum's theory of interconnection [Fiad92].

In addition to its notational suitability with respect to composition of objects, CSL distinguishes from other specification languages by the following characteristics. It is close to object-oriented notations such as UML [Booc96]. It takes into account the assumptions about the environment made by an object. It supports hierarchical specification development including the refinement of an individual object specification into a composite object specification. All these characteristics make the language suitable for applications in the domains of engineering including software engineering, network management, multimedia, and communication protocols.

The balance of this paper is as follows. In Section 2, we first provide the principles and concepts underlying the design of CSL. Next, the language syntax and informal semantics are presented. The language is illustrated by means of the specification of a buffer, and the composition of two buffers. We also discuss about the notational suitability of the language for object

composition. Its formal semantics based on its translation into TLA is given in Section 3. We also compare the compositional reasoning of TLA to that of CSL. In Section 4, we describe how to discharge various proofs based on TLA reasoning rules once the specifications are translated into TLA. We close this paper with a comparison of CSL with other approaches to composition.

2. Object Composition

2.1. Underlying principles

Object interactions

We assume that object interactions occur only over object associations. This assumption is grounded on the fact that object interactions portray collaborations between the involved objects. Intuitively, the terms and conditions of these collaborations must be stated and established somewhere. The object association is the preferred place where such terms and conditions may be layed down.

Object collaboration supposes some form of communication. Message passing is one of the well-known mechanism used for object interaction. There is another object communication mechanism, namely joint-actions [Jarv90], which is related to the rendezvous communication in LOTOS [ISO89a]. One version of joint-actions, called "abstract events" is described in [Boch93b]. Object interactions using abstract events are characterized by:

(a) There is no asymmetric caller/callee relationship: It is not said which object makes the decision for the execution of an interaction.
(b) There may be more than two objects participating in a given interaction, i.e. multiparty interactions.
(c) Each participating object may impose certain conditions which must be satisfied when the interaction occurs. Each participating object may also define some local state changes that occur during the execution of the interaction.

The above concept of interaction does not include the notion of caller - callee.

This means that it is not specified how an interaction is "triggered". Only the conditions for its occurrence are specified. This way of describing object interactions is abstract, and non deterministic, leaving such question as triggering of actions, their scheduling and the involved actors for a later, more detailed specification. Along with us, other authors [Cook94] have also adopted a similar mechanism for describing object interactions.

Linking the structure of composite objects to their behavior

Once we are able to describe the interactions that occur in the context of object associations, we may use the same approach for describing the interactions between components in a composition. Therefore structures of compositions, and interactions that occur inside, can be described.

In [Rama95a], we have shown that the structure of a given composition is linked to its behavior because the behavior of the composition is realized in terms of the behavior of its components and how these are interconnected. The approach proposed above can be extended to the description of the behavior of compositions in terms of the behavior of components while taking into account the structure of the compositions. This is achieved by objectifying the structure of the composition so that abstract events can be encapsulated (hidden) and new actions can be added. Hiding is used as an abstraction mechanism.

2.2. The Specification Language CSL

Syntax and informal semantics

In CSL, we assume that a simple object (non-composite) can be represented as a composite object which has no components, i.e. its structure is empty. Based on that assumption, we use a single template for specifying objects. The template is portrayed below. Only object compositions have the "inherits" section which indicates the actions of the components which are also actions of the composition. The specification parameters, local variables, actions, and abstract events are by default visible unless tagged with the keyword **hide**. An abstract event which is hidden is also called an internal abstract event. Generic

specifications are also allowed in CSL, i.e. a specification may have parameters. The "constants" clause is self-explanatory. It introduces constants in the specification. The "define" clause introduces new literals. For instance, we may introduce an enumeration of values. The "contains" clause indicates the structure of the composition. It lists the components, while the constraints on these components can be stated in the "invariants" clause.

```
spec spec_name(<parameter list>)
        constants name = <value>
        define typename : <type>
        extends  spec_name_i, .., spec_name_j
        contains
                role_name_k : spec_name_t
                ..
        inherits
                action_i from role_name_i.action_k
                ..
        local  variables
                <variable declaration>
        initial
            <assignment to local variables> <conditions on the components>
        invariants
                <predicates on the local variables>
                <predicates on the components>
        behavior
           abstract events
            association name_u
              participants: <component list>
                [hide] event ae_i (<parameter list>) = actions:<action list>
                                                       constraint:  <predicate>
                        ..
                        event ae_k (...) = ...
              ..
          -- other actions
          action name (<parameter list>) =  enabled:<predicate>
                                            defined:<predicate>
                                            changes:<predicate>
                ..
end spec spec_name
```

Figure 4.1: Object template

Each component has a role in the composition. This is syntactically captured by the role name associated with the component. The "abstract events" clause denote the abstract events occurring between the components. The "local variables" clause denote the states of the object. The clause "initial" defines the

initial state. It assigns the initial values to the local variables and it may define the initial states of the components in the context of the composition. The "invariants" clause is used to record safety properties of the object, and its components. The invariants hold whenever no action is in process of execution.

In a CSL specification, the "extends" clause denotes specialization. The features of the specifications listed in the "extends" clause are augmented with new features introduced by the given specification. The "enabled", "defined" and "changes" predicates of an action can be strengthened. Invariants can be more constrained, etc. All the properties of the extended specifications are visible. Multiple specialization is allowed, and name conflicts are resolved by prefixing the properties of specifications. Containment is also inherited along with the structures of the extended specifications juxtaposing one with another. Similar to actions, abstract events can be also more constrained. For local variables, only new variables can be added. The initial conditions can be augmented but not redefined. The invariants can be more constrained and new ones added.

The "behavior" clause defines the actions which are supported by the object including its abstract events. In fact, abstract events are generalized actions. They involve more than a single object contrasting with actions which are provided by a given object. In other words, abstract events are multi-object actions while actions are single-object actions. In this paper, for any CSL specification M, we refer to its actions through the set *actions(M)*. Actions may have parameters. Actions are defined in terms of "enabled", "defined", and "changes" predicates. These predicates involve the local variables of the specification. In action definitions, parameters and local variables occur primed only in the "changes" predicate where a primed name denotes the value after the execution of the action. In general, the action's "changes" predicate relates the new values of parameters and local variables to the values before the execution of the action. The "defined" predicate indicates the hypotheses about the environment for the action. In CSL, each object has an environment. This environment is dynamic. It consists of all the objects which may participate in

some abstract event with the given object. With respect to the environment, each object makes assumptions. Since objects communicate through actions, the assumptions about the environment made by an object are included in the definition of its actions. The "defined" clause in action definitions serves for this purpose. In other words, the assumptions about the environment of an object are the collection of the "defined" clauses of its actions.

We use a combination of mathematics and predicate logic to define the semantics of actions. Each action has the form :

action action-name (<parameters>)=	**enabled**: <predicate> **defined**: <predicate> **changes**: <predicate>

The meaning of an action is:

(a) when "enabled" is satisfied and "defined" is also satisfied, then the action can be executed and "changes" will be true afterwards;

(b) when "enabled" is satisfied and "defined" is not satisfied, the action can be executed, but the result is undefined;

(c) when "enabled" is not satisfied, the action is blocked (it can not be executed).

The overall behavior of the object is that first the initial conditions are established. In any state either an action is executed or the local variables remain unchanged. When an action occurs, the invariants must be satisfied in the state before the execution of the action and in the state after the execution of the action. At the level of an object, we assume interleaving concurrency between the actions supported by that object. For a composition, there is interleaving between the abstract events, and the actions.

An abstract event is the synchronization of two or more actions. It is specified by a list of actions which are synchronized and a constraint over the parameters of the synchronized actions. The abstract event, with a given set of parameter values, is enabled in states where all its composing actions are enabled and the constraint is true.

Abstract events occur in the context of object associations. Therefore, they are

grouped according to the associations in which they occur. Each association has participants. The objects involved in the abstract events should at least participate in the association in which the abstract event occurs. Each abstract event has the form:

event abstract event-name(<parameters>) =	**actions**: <action list> **constraint**: <predicate>

The action list determines which objects and which actions constitute the abstract event. Because abstract events are a generalization of the concept of action, its enabling condition is the conjunction of the enabling conditions of its actions plus the supplemental conditions imposed by the constraint. Notice that because we allow the actions involved in an abstract event to agree on the value of its parameters, it may be the case that an abstract event is blocked because two actions impose conflicting values for the parameters. Its definedness condition is also the conjunction of the definedness conditions of its actions, and finally its "changes" predicate corresponds to the conjunction of the "changes" predicate of its actions.

Internal consistency of an abstract event

A hidden abstract event is said to be internally consistent if the involved actions are such that there are no values of the parameters and local variables which make the "defined" predicate of one action false while making the "enabled" predicate of all the actions involved in the abstract event true. This is referred to as *internal consistency of the abstract event.*

In other words, for a CSL specification M, we assume that the notation e_i represents an internal abstract event. The notation $e_i.actions$ denotes the actions involved in the abstract event e_i. We refer to the predicates of an abstract event or an action by using the dot "." notation, e.g. $e_i.constraint$ represents the constraint of the abstract event e_i. $a_n.enabled$ and $a_n.defined$ respectively represent the "enabled" and "defined" predicates of the action a_n. A hidden abstract event e_i is internally consistent if:

$$\forall\, a_k \in e_i.\text{actions} : (e_i.\text{constraint} \wedge (\bigwedge_{n=1}^{m} a_n.\text{enabled})) \Rightarrow a_k.\text{defined}$$

where m is the number of actions involved in e_i and $a_n \in e_i$.actions.

Internal consistency of a composed specification

The notion of internally consistent abstract events leads to internal consistency of a CSL specification. A CSL specification is internally consistent if all its internal abstract events which are executed bear internal consistency. Since we have introduced abstract events as generalization of actions, let the notation enabled(e_i) and defined(e_i) respectively denote the fact that the abstract event e_i is enabled and defined. The notation $\text{events}_h(M)$ denotes the set of internal abstract events of the CSL specification M. A CSL specification M is internally consistent if it maintains the internal consistency invariant which is

$$\forall\, e_i \in \text{events}_h(M) : \text{enabled}(e_i) \Rightarrow \text{defined}(e_i).$$

In order to discharge the proof of the internal consistency invariant, one has to check that when an internal abstract event is enabled, it is internally consistent.

Global consistency for a specification

The concept of internal consistency has to be contrasted with global consistency. A specification is globally consistent if assuming that the environment satisfies the assumptions made by the specification, all the actions and abstract events executed by the specification result in a well-defined behavior of the specification. A global system specification may still be globally consistent, i.e. no action or abstract event will be executed with some "defined" predicate not being true, if some of abstract events are not internally consistent.

Component consistency

The notion of global consistency of a CSL specification is interesting, in particular in the context of a composed specification. For instance, assuming the component specifications M_i, .., M_n. When building a composed specification M which uses the former specifications as components, we have to discharge that the component specifications M_i, .., M_n are globally

consistent in the environment provided by the composed specification M. Let us call this property the *component consistency*. It is a proof obligation which has to be discharged for any composed specification.

Global consistency for a composed specification

When dealing with composed specifications, discharging global consistency can be cumbersome. However, by distinguishing between actions and abstract events which depend on the components (inherent and aggregate events) from those which are independent of the components (emergent events), we can show the global consistency of a composed specification in two steps:

a) showing component consistency in the context of the composed specification. This take care of the inherent and the aggregate events;
b) showing that assuming the environment of the composed specification satisfies the assumption made by the emergent events, when these events are executed, they result in a well-defined behavior of the specification.

2.3. Notational suitability of CSL

CSL departs from other approaches to the description of composite object behaviors because it is grounded on the linkage between the structure and the behavior of these objects. In a typical development using CSL, the associations and the interfaces that the objects provide have to be defined before describing the behavior of the objects. The behavior of objects is an outgrowth of the constraints indicated in the associations binding the objects to one another.

CSL has been defined to closely follows our interpretation of object composition. To that extent, it allows us to produce specifications which have plausible intuitive interpretation with respect to the conceptual framework for object composition introduced in [Rama95a]. It does not contains any intricate encoding to specify concepts, constructs, and mechanisms related to object composition.

Notational suitability is the degree to which a specification language makes it possible for the specifier to express the intent (semantics) in a precise, concise,

understandable modular and easily modifiable way. In this context, we must ask ourselves what are the concepts, principles, and mechanisms which must be adequately described for object composition. The conceptual framework proposed in [Rama95a] answers this question. It identifies the following aspects:

1) **The description of the individual behavior of components**: CSL has features for describing a component independently of the environment in which the component is supposed to work with minimal conditions describing what is required for it to work properly. The minimal conditions are specified for each action of the component in the form of the "defined" predicate. It indicates when the action behavior is defined. The parameters of the action define the values exchanged with the environment. The enabling conditions specify the states and parameters for which the action is possible; when it is false, the action is blocked.

2) **The description of interactions between components**: This is achieved in CSL through the synchronization of actions in the form of abstract events. These synchronizations are derived from the associations which bind the objects to one another.

3) **The contribution of each component to the behavior of the composition**: This is achieved in CSL by the "inherits" clause by which the composition exposes the actions of its components. In addition, several alternatives can be used for describing the aggregation of actions of the components. One alternative consists to define the aggregate action at the level of the composition. Then define another action which is synchronized with the component actions such that the former action is only enabled by the execution of the aggregate action. This intuitively captures the fact that the composition propagates the action to its components.

4) **Describing what makes the composition different from the sum of its components**: In a CSL specification, you may introduce actions, invariants, and local variables for a composite object which have no relation

with its components.

In addition to its notational suitability with respect to composition of objects, the language is distinguished from other specification languages by the following characteristics:

- It is close to object-oriented notations, such as UML. It supports basic concepts such as associations and collaborations which are not supported by other object-oriented formal specification languages.
- It takes into account the assumptions about the environment made by an object. Further, it localizes the assumptions about the environment. In other languages these assumptions, if not ignored, are scattered throughout the specification with no indication that they do represent environmental assumptions. Furthermore, CSL makes the assumptions about the environment explicit and they are taken into account when reasoning about the specification properties.
- CSL supports hierarchical specification development including the refinement of an individual object specification into a composite object specification.

All these characteristics makes the language suitable for applications where :

1) There are complex structural and behavioral constraints between objects.
2) The hierarchical organization and composition are key to structuring the behavior of the application.
3) There are dynamic interactions between objects at different levels of the hierarchy.

These characteristics have been witnessed in engineering including software engineering, network management, multimedia, and communication protocol applications.

2.4. Example

The example used in this paper is a FIFO buffer offering two operations, namely *put(x)* and *get(x)*. The buffer has a capacity which is defined as the number of items it can hold. The operation *put(x)* is enabled until the capacity of the buffer is reached. The operation *get(x)* is enabled only when the buffer is

not empty. The CSL specification of the buffer is illustrated below.

```
spec Buffer(n:Nat)
        local variables
                buffer[1..n]: Nat
                size: Nat
        initial
                (size = 0)
        invariants
                (size ≤ n) ∧ (∀ m ∈ Nat: (0 < m ≤ size) ⇒ (buffer[m] ∈ Nat))
        behavior
          action put(i:Nat) = enabled: size < n
                              defined: size < n
                             changes: (buffer[size + 1]'= i) ∧  (size' = size + 1)
          action get(o:Nat) = enabled: size > 0
                              defined: size > 0
                              changes: (o' = buffer[1]) ∧ (size' = size - 1) ∧
                                                (∀ m:  (1 ≤ m < size) ⇒
                                                (buffer[m]' = buffer[m + 1]) )
end spec Buffer
```

An arbitrary number of buffers can be composed yielding a larger buffer. Hereafter, we portray the composition of two buffers. This is exemplyfied by the following specification.

```
spec CompositeBuffer(n:Nat)
        contains
                buffer[1..2]: Buffer(n)
        local variables
                size: Nat
        initial
                (∀ m ∈ Nat : (1 ≤ m ≤ 2) ⇒ buffer[m].initial) ∧ (size = 0)
        invariants
                size = buffer[1].size + buffer[2].size
        inherits
                put(i:Nat) from buffer[1].put(i)
                get(o:Nat) from buffer[2].get(o)
        behavior
          abstract events
            association connection
              participants: buffer[1], buffer[2]
                hide event transfer(v:Nat) = actions:     buffer[1].get(v)
                                                          buffer[2].put(v)
                                             constraint:  true
end spec CompositeBuffer
```

Notice that the assumptions about the environment made by the CSL specification Buffer are strong enough to allow us to compose these specifications. In the case we would have the specification FoolBuffer illustrated below, we may not be able to arbitrarily compose FoolBuffer. Because in this case, the actions *put(x)* is enabled even if the FoolBuffer reaches its capacity, and the action *get(x)* is enabled when the buffer is empty.

```
spec FoolBuffer(n:Nat)
        local variables
                size: Nat
                buffer[1..n]: Nat
        initial
                (size = 0) ∧ (∀ m ∈ Nat: (0 < m ≤ n) ⇒ (buffer[m] ∈ Nat))
        invariants
                (size ≤ n)
        behavior
                action put(i:Nat) = enabled: true
                                defined: size < n
                                changes: (buffer[size + 1]'= i) ∧ (size' = size + 1)
                action get(o:Nat) = enabled: true
                                        defined: size > 0
                                        changes: (o' = buffer[n]) ∧ (size' = size - 1) ∧
                                                        (∀ m: (1 ≤ m < size) ⇒
                                                        (buffer[m]' = buffer[m + 1]) )
end spec FoolBuffer
```

Assuming that in order to compose two FoolBuffer, we define an abstract event *transfer* similar to the one introduced for CompositeBuffer. In this case the abstract event transfer is enabled even if the FoolBuffer b[1] is empty or the FoolBuffer b[2] is full. However, CSL permits us to constrain abstract events in order to be able to compose actions which in essence seem incompatible. This is illustrated by the specification EncapsulatedBuffer below.

```
spec EncapsulatedBuffer(n:Nat)
        contains
                buffer[1..2]: FoolBuffer(n)
        local variables
                size: Nat
        initial
                (∀ m ∈ Nat : (1 ≤ m ≤ 2) ⇒ buffer[m].initial) ∧ (size = 0)
```

```
        invariants
                (size = buffer[1].size + buffer[2].size)
        inherits
                put(i:Nat) from buffer[1].put(i)
                get(o:Nat) from buffer[2].get(o)
        behavior
          abstract events
            association connection
              participants: buffer[1], buffer[2]
                hide event transfer(v:Nat) =
                        actions:        buffer[1].get(v)
                                        buffer[2].put(v)
                        constraint:     (buffer[2].size < n) ∧ (buffer[1].size > 0)
end spec EncapsulatedBuffer
```

Notice that you may be unable to compose an EncapsulatedBuffer with a CompositeBuffer, because the actions *put(x)* and *get(x)* of EncapsulatedBuffer are the ones provided by the FoolBuffer. As another variation of the idea of encapsulating FoolBuffer, consider the specification ClosedSystem portrayed below. In this specification, a FoolBuffer has a user which is able to use the FoolBuffer while taking care of its assumptions about the environment. The specification ClosedSystem does not communicate with the outside world.

```
spec User(n:Nat)
        local variables
                inTransit: Nat
        initial
                (inTransit = 0)
        invariants
                (inTransit ≤ n)
        behavior
                action send(x:Nat) =  enabled: inTransit < n
                                      defined: true
                                      changes: inTransit' = inTransit  + 1
                action receive(x:Nat) = enabled: inTransit > 0
                                      defined: true
                                      changes: inTransit' = inTransit - 1
end spec User
```

```
spec ClosedSystem(n:Nat)
        contains
                user: User(n)
                buffer: FoolBuffer(n)
        initial
```

```
            user.initial ∧ buffer.initial
      behavior
        abstract events
          association communicate
            participants: user, buffer
              hide event give(v:Nat) = actions: user.send(v)
                                                buffer.put(v)
                                       constraint: true
              hide event take(v:Nat) = actions: buffer.get(v)
                                                user.receive(v)
                                       constraint: true
end spec ClosedSystem
```

It is interesting to contrast the internal abstract events of the specifications EncapsulatedBuffer and ClosedSystem. In the former specification, we can deduce from the definition of the abstract event *transfer* that it is internally consistent. However, in the case of the internal abstract events *give* and *take*, a proof obligation is required (see below). On the other hand, the specification ClosedSystem is globally consistent. If we consider the slight variant specification FoolClosedSystem which contains two FoolBuffers instead of a single FoolBuffer, this latter specification is not globally consistent because the user may overflow the buffer[1] or try to receive from an empty buffer[2].

```
spec FoolClosedSystem(n:Nat)
      constants m = n/2
      contains
            user: User(n)
            buffer[1..2]: FoolBuffer(m)
      initial
            user.initial ∧ buffer[1].initial ∧ buffer[2].initial
      invariants
            ((buffer[1].size + buffer[1].size) ≤ n) ∧ (n = 2m)
      behavior
        abstract events
          association connect
            participants: buffer[1], buffer[2]
              hide event transfer(v:Nat) = actions: buffer[1].get(v)
                                                    buffer[2].put(v)
                                       constraint: (buffer[1].size > 0)
                                                   (buffer[2].size < m)
          association communicate
            participants: user, buffer[1], buffer[2]
              hide event give(v:Nat) = actions: user.send(v)
                                                buffer[1].put(v)
```

```
                                    constraint:  true
        hide event take(v:Nat) =    actions:     buffer[2].get(v)
                                                 user.receive(v)
                                    constraint:  true
end spec FoolClosedSystem
```

3. Formal semantics

To reason about CSL specifications, we need an underlying execution model and an underlying logic for proving properties. We have favored the translation of CSL specifications into TLA (Temporal Logic of Actions) [Lamp97]. The choice of TLA is motivated by its computational model, its built-in notion of behavior including the temporal ordering of actions. In addition, there is no explicit action triggering, as in the case of CSL abstract events.

3.1. Overview of TLA

TLA defines the behavior of systems in terms of states. In particular, it distinguishes an initial state. A state is an assignment of values to variables. From one state either the system executes an action which brings the system into another state, or it does not execute an action. In this latter case, the system stutters, i.e. it remains in the same state.

TLA formulas are built up from state functions using boolean operators ($\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$) and the operators ' (prime), $\Box$, and $\exists$, where $\Box$ is the usual "always" temporal logic operator. A state function is like an expression in a programming language. It assigns a value to each state. A state predicate is a boolean-valued state function. An action is a boolean-valued expression containing primed and unprimed variables. It is a predicate over a pair of states <s,t> where the primed variables refer to t and the unprimed variables to s. A pair of states <s,t> satisfying an action A is called an A step. An action A defines how the values of variables in the state t, at the end of the action, are related to the values of variables in the state s, the state at the beginning of the action. A is enabled in state s if there exists a state t such that <s,t> is an A

step. The state predicate *Enabled* A is true for state s iff A is enabled in s.

In TLA, a variable represents some part of the universe, and a behavior represents a possible complete history of the universe. There is no formal distinction between systems, specifications, and properties. They are all represented by TLA formulas. Each system is characterized by a set of variables and a behavior (set of actions) which modifies these variables.

A typical TLA specification has the form Init $\wedge \Box[\text{Actions}]_{<V>} \wedge$ F where V is a set of variables, Init is a predicate defining the values of the variables at the initial state, Actions is a disjunction of all specified actions, the predicate $[\text{Actions}]_{<V>}$ is an abbreviation for the predicate Actions $\vee$ (v=v'), and F represents the fairness requirements on the actions listed in Actions. Lamport has observed that restricting concurrency to interleaving of actions simplifies the reasoning. We also adopt the interleaving model of concurrency. In addition, TLA introduces temporal existential quantification which is used to hide variables in a specification. Also, actions can be composed yielding a new action, e.g. let a_1 and a_2 be two actions, $a_1 \wedge a_2$ is a new action which represents the concurrent execution of the two actions a_1 and a_2.

In TLA, specialization between specifications is logic implication, i.e. S_1 specializes S_2 iff $S_1 \Rightarrow S_2$. If different sets of variables are used in the two specifications to represent the universe, then a refinement mapping must be introduced [Abad95].

In TLA, modules are introduced to structure specifications. The modules consist of various sections. The "parameter" section is used to define the local variables. The "assume" section can be used to further constrain the local variables, and to establish relationships between local variables of distinct specifications. The "assume" section is also used to introduce instances of other modules when we would like to use the properties of these modules.

The "predicate" section is used to introduce the initial conditions of the specification. Then follows the "actions" section which introduces the actions

of the specification. In addition, all the actions which leave some of the local variables unchanged contain the predicate Unchanged(x_i, ..,x_n) which is an abbreviation for ($x_i'= x_i$) $\wedge$ .. $\wedge$ ($x_n' = x_n$). The "temporal" section contains the definition of the behavior. It consists of a TLA specification Init $\wedge$ $\Box$[Actions]$_{<v>}$ $\wedge$ F .

3.2. Composition in TLA

Consider two TLA specifications S_1 and S_2,

$S_1 = Init(S_1) \wedge \Box[Actions(S_1)]_{<v(S_1)>}$

$S_2 = Init(S_2) \wedge \Box[Actions(S_2)]_{<v(S_2)>}$

where $Init(S_1)$ denote the initial conditions for S_1's variables, $Actions(S_1)$ is the disjunction of actions defined in S_1, and $v(S_1)$ is the tuple of S_1's variables.

The composition of S_1 and S_2 is formally defined as

$S_1 \wedge S_2 = Init(S_1) \wedge Init(S_2) \wedge \Box[Actions(S_1) \vee Actions(S_2)]_{<v(S_1) \cup v(S_2)>}$.

Composition is based on interleaving the actions of S_1 with those of S_2. At the initial state of the composition, the initial conditions imposed by S_1 and S_2 are satisfied. However, in such a composition, there is no assertion on the possible environments in which the system specified by S_1 and S_2 may evolve.

Because a system may not behave as required in arbitrary environments, assumption-guarantee specifications are introduced in TLA. They are based on the operator $\pm>$ between specifications. An assumption-guarantee specification has the form E $\pm>$ M where E represents the specification of the system's environmental requirements and M represents its guarantees. The communication between E and M is achieved by shared variables.

The specification E -> M asserts that M holds at least as long as E does. The specification E_{+v} asserts that when E is not satisfied, the variables in v stop changing. When E and M are both safety properties and v is a tuple of variables containing all free variables of M then E $\pm>$ M equals E_{+v} -> M. Assumption-guarantee specifications are not composed like ordinary TLA specifications.

The composition rule for two assumption-guarantee specifications $E_1 \stackrel{\pm}{>} M_1 \wedge E_2 \stackrel{\pm}{>} M_2$ is : if

$$(1)\ (E \wedge (M_1 \wedge M_2)) \Rightarrow E_1$$
$$(2)\ (E \wedge (M_1 \wedge M_2)) \Rightarrow E_2$$
$$(3)\ (E \wedge (M_1 \wedge M_2)) \Rightarrow M$$

then

$$((E_1 \stackrel{\pm}{>} M_1) \wedge (E_2 \stackrel{\pm}{>} M_2)) \Rightarrow (E \stackrel{\pm}{>} M)$$

This rule can be interpreted as follows: If the high-level environment (E) and the guarantees of all the components (M_1 and M_2) imply the environmental requirements of each of the components (E_1 and E_2) and the guarantees (M), then the composition of the components implements (specializes) the high-level specification (M) in the high-level environment. More details on this composition rule can be found in [Abad95].

3.3. Translating CSL specifications into TLA

The translation of CSL specifications is based on a mapping between CSL concepts and TLA concepts. This mapping is as follows:

- CSL specifications are translated into TLA modules;
- TLA modules may have constant parameters which can be used to represent CSL constants. In addition, since the specification parameters in CSL remain constant, they can also be represented by constant parameters of TLA modules;
- TLA has the notions of records, sets, tuples and functions. These notions can be used to represent the types introduced by the "define" clause of a CSL specification;
- TLA modules can be extended by other modules. These latter modules are specializations of the former modules. This can be used to translate the "extends" clause of CSL specifications;
- TLA modules can contain local instantiations of other modules. We use this feature for representing the "contains" clause of CSL specifications;
- TLA modules permit the definition of actions in terms of other actions. We use that for representing the "inherits" clause;

- TLA modules have local variables. This is used to represent CSL local variables;
- TLA modules may define an initial state. This is used to represent the "initial" clause of CSL specifications;
- TLA modules may impose invariants properties. This is used to represent the "invariants" clause of CSL specifications;
- Since we consider abstract events to be a generalization of actions, actions and abstract events are both translated into TLA actions. When translated into TLA, CSL action parameters are introduced as additional variables into the TLA specification.

The main difference between CSL and TLA is on the semantics of actions. CSL actions have explicit parameters. *In addition, each CSL action indicates what are its assumptions about the environment since the objects are composed through their actions.* This semantics introduces possible undefined behavior for actions. The actions also have an explicit enabling condition.

Notation

We write $LV(a_i)$ for the set of local variables appearing in the CSL action a_i, V for the set of all local variables of the entire CSL component. $CH(a_i)$ is a subset of $LV(a_i)$; it denotes the local variables which are modified by the action a_i. $PAR(a_i)$ is the set of the parameters of the action a_i. Parameters which do not appear primed in the "changes" predicate of the action can be considered as input parameters. The others are output parameters. We write $PAR_{in}(a_i)$ for input parameters and $PAR_{out}(a_i)$ for output parameters of the action a_i. Recall that a CSL action is defined by three predicates, **enabled**, **defined** and **changes**, which will be respectively referred to in the sequel by a_i.**enabled**, a_i.**defined**, a_i.**changes**.

Translation

A CSL action a_i is translated into a TLA action which we denote by **TLA-action**(a_i) and which is defined as follows:

$\forall\ LV(a_i) : \forall\ PAR(a_i)$: $\quad \wedge\ a_i$.**enabled**

$$\wedge\ (a_i.\textbf{defined} \Rightarrow\ a_i.\textbf{changes}\ )$$
$$\wedge\ \textbf{unchanged}((V - CH(a_i)) \cup PAR_{in}(a_i))$$

Notice that in this translation, explicit TLA **unchanged** predicates are added in order to indicate which variables remain unchanged by the action. Using this translation approach, the TLA state predicate *Enabled* **TLA-action**(a_i) is equal to a_i.**enabled**. In addition, when a_i.**defined** is not satisfied, any state can be considered as a next state of the TLA specification.

With respect to the behavior of an object, TLA and CSL have the same semantics, i.e. the initial conditions hold first and then either an action is executed or the specification variables remain unchanged. Therefore, the translation is one to one for the object behavior.

Once the CSL actions are translated into TLA, abstract events can also be translated into TLA. Abstract events are represented by the conjunction of the involved actions and the constraint of the abstract event. In the sequel, the translated abstract event e_i will be referred to as **TLA-action**(e_i).

Hiding of abstract events in a composition is achieved by hiding the parameters of the actions involved in these abstract events. Assume that the notation PAR(e) is extended to abstract events. Hiding an abstract event e correspond to writing $\exists$ PAR(e): TLA-action(e).

The initial clause of a CSL specification M has a direct mapping into TLA since it consists of a set of predicates. Its translation is denoted by **TLA-Init**(M). The behavior of the CSL specification M is denoted by **TLA-spec**(M). It is equivalent to **TLA-Init**(M) $\wedge\ \Box[\bigvee_{i=o}^{n}$ **TLA-action**$(a_i)]_{<V(M)>}$
where a_i is in actions(M) and n is the cardinality of the set actions(M).

As an example, we discuss in the following the translation of the buffer specification presented in Section 2.4. We have the followings:

V(Buffer) = {{buffer[1], buffer[2]}, size}
TLA-Init(Buffer) is Buffer.**initial** which corresponds to (size = 0).
The actions considered for Buffer are *put* and *get*.

Buffer.invariants = $(size \leq n) \wedge (\forall\ m \in Nat: (0 < m \leq size) \Rightarrow (buffer[m] \in Nat))$

The actions are translated as follows:

TLA-action(put) = $\wedge (size < n)$
$\wedge (size < n) \Rightarrow (buffer[size + 1]' = i) \wedge (size' = size + 1)$
$\wedge$ unchanged(i)

TLA-action(get) = $\wedge (size > 0)$
$\wedge ((size > 0) \Rightarrow (o' = buffer[1]) \wedge (size' = size - 1) \wedge (\forall\ m:\ (1 \leq m < size) \Rightarrow (buffer[m]' = buffer[m + 1]))$

At first sight, there is no *unchanged* predicate for the action get. But close examination of the semantic of the "changes" predicate leads to the conclusion that the predicate unchanged(buffer[2]) can be added to the action definition. This illustrates that the automatic translation may require the intervention of the specifier so that the specification can be fine-tuned. In addition, the type constraints can be added to the action specification. A type constraint is captured by an additional predicate in the action definition, e.g. requiring that the parameter i be a natural number means adding the predicate ($i \in Nat$) to the action definition. The behavior of a buffer is captured by the TLA formula
TLA-spec(Buffer) = **TLA-Init**(Buffer) $\wedge\ \Box[\exists$ i, o: **TLA-action**(put(i)) $\vee$ **TLA-action**(get(o))$]_{<V(Buffer)>}$

The TLA specification of the Buffer based on the module structure provided by TLA is portrayed below. In the module specification, the CSL specification parameter n is introduced as a constant since we already know that specification parameters can not be changed. All the variables of the CSL specification, i.e. variables in V(Buffer) are introduced as variables of the TLA module. Type constraints for the variables can be added in the "predicates" section of the module as illustrated. The "invariants" of the CSL specification are introduced as they are formulated in the "assume" section. This gives the following TLA

specification.

Module Buffer
parameters
n : Constant
buffer[1]: Variable
buffer[2]: Variable
predicates
(buffer[1] ∈ Nat) ∧ (buffer[2] ∈ Nat)
Init ≡ TLA-Init(Buffer)
assume
Buffer.invariants
actions
put(i) ≡ TLA-action(put)
get(o) ≡ TLA-action(get)
temporal
Behavior ≡ TLA-spec(Buffer)

The abstract event *transfer* is translated into

TLA-action(transfer) = ∃ v: ∧ buffer[1].get(v) ∧ buffer[2].put(v)
∧ (v ∈ Nat)

The contains clause is translated into local instantiation of the modules corresponding to the components. Assume the notation **TLA-module**(M) representing the TLA module of the CSL specification M. The clause

contains
buffer[1..2]: Buffer[n]

is translated into

assume
LOCAL buffer[1] **instantiate** TLA-module(Buffer(n))
LOCAL buffer[2] **instantiate** TLA-module(Buffer(n))

The TLA specification **TLA-module**(CompositeBuffer) is illustrated below.
TLA-Init(CompositeBuffer) = CompositeBuffer.**initial**.
TLA-spec(CompositeBuffer) = **TLA-Init**(CompositeBuffer) ∧ ❑[∃ i, o, v: **TLA-action**(put(i)) ∨ **TLA-action**(get(o)) ∨ **TLA-action**(transfer(v))]$_{<V(CompositeBuffer)>}$. Notice that the "inherits" clause gives rise to new actions.

Module CompositeBuffer
parameters

```
        n : Constant
        size : Variable
assume
        LOCAL buffer[1] instantiate TLA-module(Buffer(n))
        LOCAL buffer[2] instantiate TLA-module(Buffer(n))
        CompositeBuffer.invariants
predicates
        (x ∈ Nat) ∧ (size ∈ Nat) ∧ (n ∈ Nat)
        Init ≡ TLA-Init(CompositeBuffer)
actions
        get(o) ≡ buffer[1].get(o)
        put(i) ≡ buffer[2].put(i)
        transfer(v)≡ TLA-action(transfer(v))
temporal
        Behavior ≡ TLA-spec(CompositeBuffer)
```

4. Proof of properties based on TLA reasoning rules

4.1. Properties implied by a specification

In general, safety properties are proven as invariants of the specification. In TLA, the specification and its properties are specified using the same language. This means, the property P of a specification is proven using logic implication. In other words, saying that a property P is implied by a CSL specification M means that **TLA-spec**(M) $\Rightarrow$ P.

For instance, consistency (both global and internal) of a specification can be seen as a property implied by the specification. Hereafter, we denote global consistency of a CSL specification M by the safety property $\Box$Defined(M). As defined in Section 2, a specification is globally consistent if assuming that the environment satisfies the assumptions made by the specification, all the actions and the abstract events executed by the specification result in a well-defined behavior of the specification. Let **TLA-env**(M) represents an environment satisfying the assumptions made by the specification M. Proving that M is globally consistent in the environment **TLA-env**(M) amounts to proving:

(**TLA-env**(M) $\wedge$ **TLA-spec**(M)) $\Rightarrow \Box$Defined(M)

where Defined(M) = $\forall$ $a_i \in$ actions(M): **TLA-action**(a_i) $\Rightarrow$ a_i.**defined**.

Notice that when proving the global consistency of a specification, the reasoning uses the environment (assumptions about the environment) of the object, because an object may not be consistent in arbitrary environments.

In TLA, invariance properties, such as the ones above, are proven using the following reasoning rule INV1 described in [Lamp94] :

$$\frac{I \wedge [N]_f \Rightarrow I'}{I \wedge \Box[N]_f \Rightarrow \Box I}$$

This rule is used to prove that a TLA specification satisfies an invariance property $\Box I$. The hypothesis asserts that a $[N]_f$ step cannot falsify I. The conclusion asserts that if I is true initially and every step is a $[N]_f$ step, then I is always true.

For instance, in order to prove that the abstract events *give* and *take* of the CSL specification ClosedSystem defined in Section 2.4 are executed consistently, i.e. the ClosedSystem specification is globally consistent, we use the invariance rule of TLA as follows. The ClosedSystem specification developed in Section 2.4 is a special case where global and internal consistency of a specification are equivalent. It is so because the ClosedSystem specification does not interact with its environment. The proof rule for showing that an internal abstract event e_i is executed consistently is

$$\forall\, a_k \in e_i.\text{actions} : (e_i.\text{constraint} \wedge (\bigwedge_{n=1}^{m} a_n.\text{enabled})) \Rightarrow a_k.\text{defined}$$

This in fact corresponds to: *Enabled* **TLA-action**(e_i) $\Rightarrow (\bigwedge_{k=1}^{m} a_k.\text{defined})$

We check consistency only for the accessible states. This has to be contrasted with consistency for all states which may be independent of the environment.

Taking into account the translation into TLA of the abstract events *give* and *take*, the rule for the abstract event *give* is:

$\exists$ n $\in$ Nat: $\exists$ v $\in$ Nat:

((true) $\wedge$ (user.send(v).enabled $\wedge$ buffer.put(v).enabled)) $\Rightarrow$

(user.send(v).defined ∧ buffer.put(v).defined)

Based on the definition of the involved actions, the rule reduces to:

∃ n ∈ Nat: (inTransit < n) ⇒ (buffer.size < n) .

This latter proof rule is discharged based on the invariant ❑(inTransit = buffer.size). The invariant is shown using TLA INV1. First, we show that TLA-Init(ClosedSystem) ⇒ (inTransit = buffer.size). This follows from the definition of TLA-Init(ClosedSystem) which is (inTransit = 0) ∧ (buffer.size = 0). Now by applying INV1, i.e. we know that when the specification ClosedSystem stutters, both inTransit and buffer.size remain the same. When the abstract event *give* occurs, we have (inTransit' = inTransit + 1) ∧ (buffer.size' = buffer.size + 1) which mean (inTransit' = buffer.size'). Also, when the abstract event *take* occurs, we have (inTransit' = inTransit - 1) ∧ (buffer.size' = buffer.size - 1) which mean (inTransit' = buffer.size'). Now by applying the rule INV1 with I = (inTransit = buffer.size), i.e.

$I \wedge [\text{TLA-action(give)} \vee \text{TLA-action(take)}]_{<\text{inTransit, buffer.size}>} \Rightarrow I$

$I \wedge \Box[\text{TLA-action(give)} \vee \text{TLA-action(take)}]_{<\text{inTransit, buffer.size}>} \Rightarrow \Box I$

we deduce that the invariant ❑(inTransit = buffer.size) holds. The proof that the internal abstract event *take* is executed consistently is similar.

In TLA, the closure of the safety properties implied by a TLA formula M is denoted by C(M). C(M) is such that for any safety property P implied by the TLA formula M, we have C(M) ⇒ P. In the context of CSL, we denote the closure of the safety properties implied by a CSL specification S by Property(S). Once translated into TLA, the closure of the safety properties implied by the CSL specification S is C(**TLA-spec**(S)). This notion is useful when comparing CSL specifications.

4.2. Specialization

Specialization in TLA is general enough to cover CSL specialization. Given two CSL specifications M and N, we say that N specializes M if :

a) the specification N is globally consistent in any environment which satisfies the assumptions about the environment made by the specification M;

b) when executed in that environment, N behaves as M.

More formally, N specializes M if for all environments **TLA-env**(M) that satisfy (**TLA-env**(M) $\wedge$ **TLA-spec**(M)) $\Rightarrow$ $\Box$Defined(M)
the following holds :

(**TLA-env**(M) $\wedge$ **TLA-spec**(N)) $\Rightarrow$ $\Box$Defined(N) and
(**TLA-env**(M) $\wedge$ **TLA-spec**(N)) $\Rightarrow$ **TLA-spec**(M).

Note that when M and N do not have the same sets of variables and actions, we need a refinement mapping RM, and the above definition becomes

(**TLA-env**(M) $\wedge$ **TLA-spec**(N) $\wedge$ RM) $\Rightarrow$ $\Box$Defined(N) and
(**TLA-env**(M) $\wedge$ **TLA-spec**(N) $\wedge$ RM) $\Rightarrow$ **TLA-spec**(M).

4.3. Component consistency

A closer look at a composed specification M with components M_1, .., M_n reveals that the "contains", "inherits" and "abstract events" clauses record the contribution of the components to the composed specification. In particular, the "abstract events" clause describes in terms of actions the interactions between the components. In addition, some of these interactions can also be implied by the invariants of the composed specification. As mentioned in Section 3.1, a TLA specification consists of a set of variables and actions acting on these variables. The module structure is used to partition the set of variables so that the actions which refer to the same subset of variables are regrouped under a module. From this observation, it results that we can partition the TLA specification of the composed specification M, i.e. **TLA-spec**(M) into two specifications **TLA-spec**(M_{int}) and **TLA-spec**(M_{emg}) where **TLA-spec**(M_{int}) regroups the variables and the actions involved in the interactions between the components M_1, .., M_n of M. **TLA-spec**(M_{emg}) regroups the remaining variables and actions. In fact, the specification **TLA-spec**(M_{int}) represents the inherent and the aggregate behaviors of M.

In this context, assuming the environment **TLA-env**(M) which satisfies the assumptions about the environment made by M, discharging component consistency in the context of M amounts to showing:

$$(\textbf{TLA-env}(M) \wedge (\bigwedge_{i=1}^{n} \textbf{TLA-spec}(M_i)) \wedge \textbf{TLA-spec}(M_{int}))$$

$$\Rightarrow \bigwedge_{i=1}^{n} \Box\text{Defined}(M_i)$$

4.4. Global consistency for a composed specification

Based on the technique used to discharge component consistency for M_1, .., M_n in the context of the composed specification M, when we partition the TLA specification **TLA-spec**(M) into **TLA-spec**(M_{int}) and **TLA-spec**(M_{emg}), since **TLA-spec**(M_{int}) represents the inherent and the aggregate behaviors of M, **TLA-spec**(M_{emg}) represents the emergent behavior of M. In other words, **TLA-spec**(M_{emg}) represents the variables and the actions of **TLA-spec**(M) which are independent of its components. With this in mind, showing that M is globally consistent in the environment **TLA-env**(M) is achieved in two steps:

a) discharging component consistency in the context of M and **TLA-env**(M);

b) discharging that the emergent actions are executed consistently, i.e.

(**TLA-env**(M)$\wedge$**TLA-spec**(M_{emg})) $\Rightarrow \Box$Defined(M_{emg})

When the composed specification M does not introduce an emergent behavior, its global consistency corresponds to its component consistency.

4.5. Composition

In order to show that a higher-level specification M_h is implemented by a composed lower-level specification M_l, it suffices to substitute, in the definition provided in 4.2, the specification N by the composed specification M_l, and M by the higher-level specification M_h. This leads to the following rule: M_h is implemented by M_l if for all environments **TLA-env**(M_h) that satisfy (**TLA-env**(M_h) $\wedge$ **TLA-spec**(M_h)) $\Rightarrow \Box$Defined(M_h)

the following holds :

(**TLA-env**(M_h) $\wedge$ **TLA-spec**(M_l)) $\Rightarrow \Box$Defined(M_l) and

(**TLA-env**(M_h) $\wedge$ **TLA-spec**(M_l)) $\Rightarrow$ **TLA-spec**(M_h).

As indicated in 4.4, when M_l does not introduce an emergent behavior, we may reason using the component specifications of M_l.

4.6. Proving properties of the Buffer example

The various proof rules sketched in this section allow us to discharge:
(1) the global consistency of the individual specification Buffer(1),
(2) the internal consistency of the composed specification CompositeBuffer(1),
(3) component consistency for Buffer(1) in the context of CompositeBuffer(1),
(4) that Buffer(2) is implemented by CompositeBuffer(1).

We only illustrate the proof of point (4) which means that two buffers of length one implement a buffer of length two. This proof illustrates the principles used in the other proof obligations. We discharge the proof obligation through the following steps:

(1) To derive the TLA specifications of Buffer(1), Buffer(2), and CompositeBuffer(1): This was already done in Section 4.1. The TLA specifications for Buffer(1) and Buffer(2) can be deduced from the TLA specification **TLA-module**(Buffer). CompositeBuffer(1) is deduced from **TLA-module**(CompositeBuffer).

(2) To show the global consistency of the composition **TLA-module**(CompositeBuffer): In fact, this is trivial because the abstract event transfer is internally consistent. On the other hand, we assume **TLA-env**(CompositeBuffer) and **TLA-env**(Buffer). The global consistency of CompositeBuffer(1) can be derived from the global consistency of Buffer(1) because CompositeBuffer does not introduce an emergent behavior. Therefore, global consistency corresponds to component consistency which is proven using the rule (**TLA-env**(Buffer(2))∧(**TLA-spec**($\text{CompositeBuffer}_{\text{int}}$)∧ **TLA-spec**(Buffer) ∧**TLA-spec**(Buffer))) ⇒ ❑Defined(Buffer). This is left as an exercise to the reader.

(3) To show that CompositeBuffer(1)) behaves as Buffer(2) in the environment **TLA-env**(Buffer(2)): Abadi and Lamport have shown that it suffices to find a suitable refinement mapping between the two specifications to discharge the proof of specialization [Abad95]. The proof of this property is based on the refinement mapping which consists of a coupling invariant (*CI*) between

CompositeBuffer(1) and Buffer(2), i.e. a predicate which establishes the relationship between variables of both specifications. Such a coupling invariant *CI* can be defined as follows: (CompositeBuffer(1).size = Buffer(2).size) ∧ (CompositeBuffer(1).buffer[1].buffer • CompositeBuffer(1).buffer[2].buffer = Buffer(2).buffer).

The notation "•" denotes concatenation of buffers. In this case concatening two arrays gives a larger array. With this coupling invariant, we can show that,

(a) (**TLA-env**(Buffer(2)) ∧ **TLA-spec**(CompositeBuffer(1)) ∧ *CI*) ⇒
❑Defined(CompositeBuffer(1)) and

(b) (**TLA-env**(Buffer(2)) ∧ **TLA-spec**(CompositeBuffer(1)) ∧ *CI*) ⇒
TLA-spec(Buffer(2)).

We use the coupling invariant to match actions of the environment of Buffer(2) with actions of CompositeBuffer(1). The property (b) is discharged using the reasoning rule TLA2 [Lamp94] by having respectively P = **TLA-env**(Buffer(2)) ∧ *CI*, and Q = true. The rule TLA2 is

$$\frac{P \wedge [A]_f \Rightarrow Q \wedge [B]_g}{\Box P \wedge \Box[A]_f \Rightarrow \Box Q \wedge \Box[B]_g}$$

where f and g are tuples of variables, and A and B are disjunctions of actions.

5. Relation with other approaches to composition

We restrict the comparison to some well-known appproaches to composition, namely Abadi and Lamport Composition theorem [Abad95], Lam and Shankar Theory of Interfaces [Lam94], and Fiadeiro and Maibaum Theory of Module interconnection [Fiad92]. The comparison is done along three lines: hypotheses about the environment, object interactions, and composition rules.

It should be noticed that the work of Allen and Garlan [Alle94], while not covered in this comparison, is also close to our work. They recognize the importance of software architecture description for complex systems. The

specification of an architecture consists of two parts (1) the description of components, and (2) the description of connectors linking components. The central idea is to define architectural connectors as explicit semantic entities, i.e. they have a behavior. Behavior descriptions are formulated using a subset of CSP. This allows the specification of connectors as collections of protocols that characterize each of the participants involved in the interactions.

5.1. TLA assumption-guarantee specifications

To reason about specifications which include explicit assumptions about the environment, TLA introduces the operator $\pm>$ as explained in Section 3. Recall that a specification E $\pm>$ M is a specification of an open system where E denotes the assumptions about the behavior of the environment and M denotes the behavior of the component. In CSL, the assumptions about the environment are at the level of actions, while in TLA, assumption-guarantee specifications are introduced to take into account the assumptions about the environment at the level of the whole specification. This means undefined situations are dealt at the level of the whole specification, i.e. undefined situations have coarser granularity in TLA assumption-guarantee specifications.

One may suggest that TLA assumption-guarantee specifications should be comparable with the specifications obtained by translating CSL specifications into TLA. However, assumption-guarantee specifications require explicit synchronization between the environment and the module actions. This synchronization is achieved by sharing variables. This contrasts with the fact that CSL action synchronization is translated into action conjunction, resulting in a more abstract specification. E $\pm>$ M specifications, because of the explicit synchronization through shared variables, have an implementation bias.

However, the reasoning used in TLA assumption-guarantee specifications is similar to the one used in CSL compositional reasoning. For instance, the component consistency is equivalent to the hypotheses (1) and (2) of the TLA Composition Theorem described in Section 3. Since in CSL, when considering composed specifications which do not introduce emergent properties, the global consistency of the composed specification corresponds to component

consistency. The hypothesis (3) of the Composition Theorem of TLA corresponds to the specialization rule for a composed specification as explained in section 4.5.

Since we use TLA in this paper, we take the opportunity to illustrate the Buffer example using a TLA assumption-guarantee specification so that it can be compared with the translated CSL specification. The specification of the assumptions about the environment are in the module E. The guarantees provided by the system are in the module M. The modules E and M illustrated below communicate through shared variables *size*, *out* and *in*. When instantiating the modules, we take the care that the same variables are shared between the modules. In TLA assumption-guarantee specifications, the actions of the environment are interleaved with actions of the system. In this particular example, the actions of the environment are *get* and *put*, while the actions of the system are *insert* and *remove*. Notice that the variables of the assumption-guarantee specification must be controlled (modified) by either the environment or the system. There is no variable which can be modified by both the environment and the system. In our example, the environment controls the variable *in*. The system controls the variables *buffer*, *size* and *out*.

Module E

parameters

n : Constant
size : Variable
out: Variable
in : Variable

predicates

Init ≡ true

actions

put ≡ ∃ x:Nat: (size < n) ∧ (in' = x) ∧ unchanged(size, out)

get ≡ ∃ o:Nat: (size > 0) ∧ (o'=out) ∧ unchanged(size, out, in)

temporal

Actions ≡ put ∨ get

Behavior ≡ Init ∧ ❑[Actions]$_{<in>}$

In other words, the environment consists of the variable *in*, and the actions *get* and *put*. In addition, it can read the system's variables *out* and *size*. The system

consists of the variables *out*, *size* and *buffer*. Its actions are *insert* and *remove*. It can read the environment's variable *in*.

Module M

parameters

buffer : Variable
n : Constant
size : Variable
out: Variable
in : Variable

predicates

Init ≡ (size = 0)

actions

insert ≡ (buffer[size + 1]' = i)∧ (size' = size + 1)∧ unchanged(out, in)

remove ≡ (out' = buffer[1]) ∧ (size' = size - 1) ∧ unchanged(in) ∧
(∀ m : (1 ≤ m < size) ⇒ (buffer[m]' = buffer[m + 1]))

temporal

Actions ≡ insert ∨ remove

Behavior ≡ Init ∧ ❑[Actions]$_{<buffer,size,out>}$

The assumption-guarantee specification is introduced as a third module which has local instantiations of the assumptions (E) and the guarantees (M) modules. It also defines the variables which are shared between the environment (E) and the system (M). Notice that the behavior of the module E $\pm$> M is the TLA formula (env.Behavior$_{+<buffer,size,out>}$ –> mod.Behavior). In this formula, the operator –> is not logical implication but a TLA operator. This formula can be reduced to a canonical TLA formula (Init ∧ ❑[Actions]$_{<v>}$ ∧ F) using the Proposition 3 and the Lemma 11 in [Abad95].

Module E $\pm$> M

parameters

buffer : Variable
n : Constant
size : Variable
out: Variable
i : Variable

assume

LOCAL env **instantiate** E(out, n, size, i)
LOCAL mod **instantiate** M(buffer, out, n, size, i)
(size ≤ n)

predicates

(buffer ∈ Nat) ∧ (size ∈ Nat) ∧ (n ∈ Nat) ∧ (i ∈ Nat)
Init ≡ E.Init ∧ M.Init
temporal
Behavior ≡ env.Behavior$_{+<buffer,size>}$ –> mod.Behavior

5.2. Lam and Shankar theory of interfaces

Theory of Interfaces proposed by Lam and Shankar addresses the problem of the formal specification of interfaces [Lam94]. Interfaces are viewed as abstract behavioral relationships between modules. In addition, the interface prescribes the behavior that have to be implemented by the modules. An interface is a set of events and allowed sequences of these events. Events are partitioned into input and output events. At an interface, two modules interact. One is the provider. It controls the output events. The other is the consumer. It controls the input events. The distinction between providers and consumers allows to have separable interfaces.

A module also defines a set of events. However, it includes internal events. In addition, it has a state transition system which defines the allowed behaviors of the module. A module offers an interface if it controls the output events of that interface and its input events correspond to the input events of the interface. In addition, whenever an output event of the module is enabled to occur, the event's occurrence would be safe, i.e. if the event occurs next, the resulting sequence of interface event occurrences is a prefix of an allowed event sequence of the interface. Whenever an input event of the module can occur safely, the module does not block the event's occurrence. The notion of a module using an interface while offering another interface is based on the definition of offering an interface. Lam and Shankar introduce the notion of compatible modules. Two modules are compatible if they do not control the same events. Compatible modules can be composed based on the relation between used and offered interfaces, i.e. if one module offers an interface which is used by the other then these two modules can be composed.

In this approach, an interface I is a location where modules interacts. Considering two modules M_1 and M_2, assuming the rôles of provider and

consumer, their interface can be described by an association in CSL parlance with its abstract events. Since CSL does not distinguish between providers and consumers, we are not able to say that a module M offers an interface U, nor that it uses an interface L. Based in this mapping between CSL concepts and the concepts of the Theory of Interfaces, the composition of compatible modules corresponds to CSL composition of object specifications since interfaces corresponds to abstract events. In many respects the Theory of Interfaces is similar to the underlying theory of CSL since both are outgrowth of object interactions mechanisms.

5.3. Fiadeiro and Maibaum theory of interconnection

Fiadeiro and Maibaum Theory of Interconnection [Fiad92] is an adaptation of work by Goguen and Burstall on composition of algebraic specifications [Gogu86]. In Theory of Interconnection, algebraic specifications are replaced by temporal logic specifications of objects. An object specification is a theory. It consists of axioms which define the possible states and the actions supported by the object. Composition is grounded on Categorial theory .

The approach assumes that each object specification is a theory. By introducing morphisms between these specifications, the category of object specifications is built. This category is shown to be cocomplete. It has a colimit which is an object specification representing the composition of all its component object specifications. To apply the theory to object composition, the following artifact is used. Assume that we have two theories T_1 and T_2 representing object specifications. By introducing a third theory T as well as morphisms σ_1:T -> T_1 and σ_2:T ->T_2 , we may build the category of T, T_1 and T_2 . This category has a colimit which is the composition of T_1 and T_2 through T. Notice that the theory T defines the synchronization between T_1 and T_2 attributes and actions.

The relation between this theory of interconnection and CSL is on synchronization of actions. Introducing of a third theory to connect and compose two object specifications corresponds to the definition of the abstract events betwen the object specifications in CSL. However, the composition rules used takes the colimit of a category, which is not the same thing as

synchronizing actions and then hiding the synchronized actions.

6. Conclusions

The quest for mechanisms allowing the linkage between the structure and the behavior of composite objects has led us to adopt abstract events (rendezvous interactions) as a communication means for objects. The approach to the specification of object interactions, because of its intertwinning with the specification of object associations, has proven to be adequate for specifying the structure of composite objects. It appears that by adding actions to a structure and by hiding certain abstract events occuring in a structure, we are able to specify the behavior of composite objects. In other words, the structure specification can then be extended with aggregates and emergents, as proposed in [Rama96b], completing the specification of the composite object.

A specification language, CSL has been designed in order to capture specifications of composite objects according to the principles and concepts described in Section 2.1. These concepts and principles are the core of the composition rule of CSL. The definition of the semantics of CSL is based on its translation into TLA. This allows the usage of TLA's specification, design, and verification tools. However, our work is restricted to safety properties. In order to take into account liveness properties, the language has to be extended to support weak and strong fairness for actions and abstract events. The compositional reasoning used in CSL is similar to the compositional reasoning of some well-known approaches to composition of specifications, including Abadi and Lamport's Composition Principle [Abad95], Lam and Shankar's Theory of Interfaces [Lam94], and Fiadeiro and Maibaum's Theory of module interconnection [Fiad92].

Future developments related to this work include building tools for specification development and verification. Also, in this context, code generation from CSL based specifications of object-oriented applications should be examined.

Chapitre 5

Nommage et composition des objets

Sommaire

Nous traitons ici des exigences à respecter pour le nommage des objets dans le contexte des objets composés, exigences qui sont : (1) la visibilité et l'encapsulation des composants, (2) l'appartenance de deux objets à une même composition, (3) le partage des composants entre différentes compositions, et (4) la référence des objets d'une composition à une autre. Ces exigences ne peuvent être prises en charge par le mécanisme de nommage des objets tel que proposé par le paradigme objet, car le concept d'identité de l'objet y est utilisé pour des fins de nommage. Cette dernière ne dispose d'aucune information sur la structure des objets.

Selon nous, il faut reconnaître que ces exigences démontrent la dépendance de la référence aux objets vis-à-vis du contexte dans lequel ces objets se trouvent. Ce faisant, une désignation des objets à l'aide de noms est plus appropriée que l'identité, le nom pouvant dépendre d'un contexte alors que l'identité est une caractéristique intrinsèque de l'objet. Sur la base de ce constat, nous avons entrepris le développement de mécanismes de nommage des objets qui exploitent le concept de "sens de direction". L'utilisation de ce concept découle de l'interprétation des applications comme étant des structures évolutives de graphes étiquettés d'objets. Dans cette interprétation, chaque état de l'application est représenté par un graphe d'objet. Le concept de "sens de direction" est utile pour définir des étiquettages des graphes ayant les mêmes caractéristiques que les mécanismes de nommage des objets. Nous travaillons actuellement sur la construction d'un simulateur de mécanismes de nommage.

Ce chapitre est la version intégrale de la publication départementale numéro 1135 dont je suis l'auteur principal [Rama98b] et qui a été rédigée sous la supervision des professeurs Bochmann et Flocchini.

Object Naming and Object Composition

D. Ramazani, P. Flocchini, G. v. Bochmann
Département d'informatique et de recherche opérationnelle
Université de Montréal
C.P. 6128, Succursalle Centre-Ville
Montréal, Canada H3C 3J7
{bochmann, flocchini, ramazani}@iro.umontreal.ca

Abstract

In this paper, we show that a flat domain of object names is inappropriate in the context of object composition since new naming requirements must be fulfilled. These requirements can be summarized by: (1) visibility and hiding of components, (2) checking whether two objects belong to the same composition, (3) sharing of components between compositions, (4) reference from one composition to another composition. Object naming in the object paradigm is based on the use of the concept of object identity which can be considered as an intrinsic and universal name of the object. It does not record information about the structure of objects. This naming mechanism is inadequate for fulfilling the requirements stated above. For instance, once you have the identity of an object you can access that object. This means you can not achieve component hiding within the composition.

There are many alternatives for solving this problem including the design of complex naming mechanisms. Our solution consists of recognizing that referencing objects is contextual while the concept of object identity is context independent. Based on this observation, we view the execution of an application as an evolving structure of labelled graphs of objects. Each state of the application is represented by a graph of objects. This allows us to use the concept of sense of direction [Floc98a] for designing various kinds of naming mechanisms.

In this paper, we designed three naming mechanisms using sense of direction. The first is based on a chordal sense of direction. The second is hierarchical

naming without component sharing. The last is a slight modification of the second to allow component sharing. This is achieved by introducing special names for shared components.
Keywords: Composition, Naming, Object identity, Sense of Direction.

1. Introduction

Designing object-oriented distributed applications consists of defining precisely collective and individual behaviors as well as the states of the objects forming the application. The way we describe interactions between these objects plays a significant role in the specification of the behavior of objects; ultimately, it influences how we implement applications. Interactions between objects require that at least some of these objects be referred to. This is achieved by means of an object naming mechanism. Such a mechanism affects three aspects of the application, namely :

1) the definition of object interfaces: it determines how we hide certain details of objects;

2) the algorithms that implement the object operations, as well as the data structures used for representing the object state;

3) object communication with one another, as well as the representation of associations between objects.

In many object-oriented languages, the concept of object identity is used as an object name. Since object identities are intrinsic properties of objects, this corresponds to the use of global names for objects. The obvious implementation for the concept of object identity is a pointer (or reference) which represents a memory address. In the distributed context, this can be complemented with network addresses and task identifiers. As mentioned above, an object naming mechanism plays a significant role in the description of the collective and individual behaviors as well as the states of objects. In this context, one may ask if global names are still appropriate for applications with composite objects. Intuitively, the linkage between the structure and the behavior of composite objects established in [Rama95a] suggests that this structure will influence the

way objects communicate. This will in turn impact on how we designate and reference the communicating objects.

1.1. Summary of our experience

In this paper, we show that global names are inappropriate for applications having composite objects since new naming requirements must be fulfilled in these applications. These requirements include : (1) visibility and hiding of components, (2) checking whether two objects belong to the same composition, (3) sharing of components between compositions, (4) reference from one composition to another composition. These naming requirements are not totally fulfilled by the existing object naming mechanisms, especially those based on the concept of object identity as unique name provided by an object. For instance, if we consider the necessity to hide some components, object naming in the object paradigm is such that once you have the identity of an object you can access that object. This means that, if a component before its insertion into the composite object was known to an object outside the composition, the latter object can still access the component even after its insertion within the composition as a hidden component. There is no means for preventing such an access except by removing all the references from objects outside the composition to the component before its insertion within the composition. This operation virtually implies the access to all objects. On the other hand, if we want to check if two objects belong to the same composition, this can not be achieved by using the identity of these objects since the identity of an object has no record of the context in which the object is located, nor it records the structure of the objects.

There are many alternatives for solving this problem including the design of complex naming mechanisms. Our solution is simple and practical, it consists to recognize that referencing objects is contextual while the concept of object identity is context independent. Based on this observation, we view the execution of an application as an evolving structure of labelled graphs of objects. Each state of the application is represented by a graph of objects. Viewing the state of an application as a graph of objects allows us to use the concept of sense of direction [Floc98a] for designing various kinds of naming mechanisms. Sense of direction exhibits many properties that allow the design of naming mechanisms

which can: distinguish between objects, permit the transfer of object names between objects, handle sharing of components, and preserve the locality of names. We later show how many properties of sense of direction relate to the properties of naming mechanisms. In this paper, we describe three naming mechanisms using sense of direction. The first is based on a chordal sense of direction. The second is hierarchical naming without component sharing. The last is a slight modification of the first to allow component sharing, this is achieved by introducing special names for shared components.

We have considered naming based on a chordal sense of direction. It is based on a chordal labeling of the graph of objects which is defined by fixing some arbitrary cyclic ordering of the nodes and labeling each incident edge by the distance in the above cycle. This means for a given object system, the number of objects forming this system is used for fixing the cyclic ordering of the nodes, and local names attributed by one object to the objects it knows are based on this number. In addition, for this chordal labeling to be a sense of direction, the local names attributed by one object to the other objects must be relative distances in the cyclic ordering [Floc98a]. The main difference with hierarchical naming is that the naming is not linked to the structure of the application and its objects.

Our naming scheme based on hierarchies of objects considers the application as being a composition of objects which can later be refined into other compositions. At the initial level, there are the application and its objects. The application has a naming context which attributes names to the objects, i.e. a naming context is a mapping between names and objects. In a naming context, names are unambiguous. In addition, objects may be refined into compositions. For each composition, there is an associated naming context. Components of a given composition have local names in that composition. They may refer to each other using these local names. An object of one composition may reference an object of another composition through the compositions, i.e. reference between compositions requires a composite name which includes the names of the compositions which are involved in this reference (naming action). This simple mechanism allows to take care of many of the naming requirements except component sharing.

In these experiences, we found that allowing component sharing complicates a lot the naming mechanisms. This raises the question of the necessity of allowing component sharing between compositions. From a modeling point of view, compositions which share components might be considered as a single composition. Therefore, component sharing is a modeling artifact which can be ignored. This is the approach taken by many authors [Rumb94].

1.2. Organization of this paper

It will be pretentious to provide in this paper an in depth analysis and a complete account of all the aspects of object naming in the context of composite objects. The reader is referred to [Wier95, Kent91] for other aspects of object naming. Therefore, we shall purposely focus on the naming mechanisms we propose with regard to the new requirements in the context of object composition.

The paper is organized as follows. Section 2 introduces the basic concepts of object naming which are relevant for understanding the rest of this paper. Next, Section 3 describes the new requirements of object naming imposed by object composition. Section 4 introduces the concept of sense of direction and its application to naming mechanisms. It describes how chordal naming and hierarchical naming can be used in the context of object composition. In addition, it describes related work. Section 5 concludes the paper and describes future research.

2. Object Naming

2.1. Characterizing object naming

Many aspects characterize object naming [Kent91, Wier95]. First are the syntax and the semantics of object names. Generally, name syntaxes are implementation dependent. For example, object names may be user defined, e.g. containment relationships may be taken into account when attributing names to objects. On the other hand, object names may be system generated. Furthermore, in the context of object databases and distributed systems, we may distinguish physical object names which are implementation dependent from logical object names which tend to be independent of any implementation. For example, the concept of object

identity has been used as a special name provided by each object. In object-oriented notations and programming languages, the aim of object identity is to provide a means for distinguishing between objects independently of their states and/or behaviors.

Apart from its distinguishing capability, the semantics of an object name may embed other properties. For instance, in the context of object databases and distributed systems, the current location of an object (e.g. within the database, within the application cache, or within the application memory) may be encoded in its name. Access control and object locking information may also be encoded along or within the name of an object.

Both the syntax and the semantics of names are grouped into a single concept, namely the naming convention which stands for the syntactic representation of names as well as their semantic interpretation. A name space or domain is a set of names complying with a given naming convention. The operations allowed on object names are performed in the context of a naming mechanism. Object names are used to refer to objects; to provide information about those objects (e.g. type information); to locate objects given only their names; and to access those objects. All these functionalities of an object naming mechanism are based on the operations one may perform on object names. Among these operations, we find comparison, assignment of a name to an object (also known as binding), resolution (i.e. finding the object behind the name), etc.

In addition to these properties, Wieringa and de Jonge [Wier95] require that an object naming mechanism should be monotonic. An object naming mechanism is monotonic when from one state of the application to another :
1) the name refers to the same object;
2) the object remains named by the same name.
This precludes object renaming and reuse of object names.

2.2. Structure of the set of object names

The set of object names may be flat or structured. By structured, we mean it is partitioned according to some criteria. For example, some authors propose to

structure object names based on the hierarchical relationships that exist between the named objects. Others propose to partition the set of object names into contexts or spaces defining the validity of the name. By a valid name, we mean a name which can be successfully resolved.

Generally, when the set of object names is structured, this is reflected on the syntax and the semantics of object names. Flat sets of object names go along with flat (or primitive) object names. Structured sets of object names may imply partitioned object names which in turn are commonly structured as a series of primitive object names. In addition, the names can be global or local. When the set of names is partitioned into sets, these sets define the validity of names. A local name is only valid in a given set of names, while a global name is valid in any set of names with respect to the naming convention considered.

2.3. Object identity and smart pointers

As explained before, in the object paradigm, the concept of object identity is used for naming objects. The object's identity can be considered as a global name provided by the object. However, in many ways, the concept of object identity does not fulfill the needs of application designers. As a matter of fact, the advent of smart pointers in C++ illustrates the need for other kinds of object names [Meye96]. In other words, object identity is inadequate as an object name since it conveys only the information that allows to distinguish between objects and no more. In C++, smart pointers are objects that are designed to look, act, and feel like built-in pointers (i.e. object names), but offer greater functionality. In particular, they allow to gain control on:

- construction and destruction of pointers to objects;
- copying and assignment.
- dereferencing (pointer resolution), for instance in database applications, you may decide to bring the object in memory. The smart pointer transparently handles the object location.

As a matter of fact, the object database standard relies on the use of smart pointers to transparently access transient and persistent objects in database applications. With smart pointers, software designers express the insufficiencies

of the object naming mechanism provided by C++, i.e. the use of the concept of object identity as an object name. This naming mechanism conforms to what is expected with the object paradigm. Is this the plea that we need changes to the object paradigm with respect to object naming? We believe so, because with the use of smart pointers, we are altering the way we perceive and understand object names. Smart pointers may add access control and other information to the object names. In addition, smart pointers may reflect the organization of the set of objects.

3. Operations on names

In this Section, we describe the various operations which can be executed on object names. These operations are classified into general operations, and operations specific to object composition. Once the syntax and semantics of names are defined, it should be stated how the names are used in specification, design and programming of applications. The operations on names reported in this section cover these steps of application development.

3.1. General operations on names

Four generic operations can be defined on names, namely attribution, comparison, resolution, and communication. Attribution denotes the operation through which a name is associated to an object, i.e. the object is named. This operation is sometimes referred to as binding a name to an object. It occurs when an object is created or when another object wants to reference the latter object using a name it assigns to that object. When global naming is used, object names are usually system generated, and objects usually have unique names. When local naming is used, an object may have multiple names according to the objects which are referencing this latter object.

Once a name is attributed to an object, it can be used to define the semantics of object actions. For instance, one object may want to know if two objects are equal. This can be achieved by comparing the names of these two objects. Comparison of names is based on the equivalence relation defined between names. This relation assumes that two names belong to the same equivalence

class when they denote the same object. This relation is used for defining a comparison relation between names. When global naming is used, such an equivalence relation between names is easy to implement since it consists of partitioning the set of names. However, when local naming is used, comparing names can be cumbersome. We will see later how it can be achieved in the context of local naming.

In addition to comparing names, an object may want to access to another object through its name. For instance, it may want to query the value of an attribute of the a given object, or it may want to call an operation of a given object. To achieve that, the object should be able to find an object by using its name. This is called name resolution, it means having access to an object through its name. In this case, when global naming is used, it is implemented by means of a mapping between names and objects. When local naming is used, since an object may have multiple names, resolving a name can be a non-trivial task.

Objects may also exchange information about other objects. This can be achieved by exchanging the names of objects. This means sending to a given object the name of another object, even its own name. To that extent, the receiving object must be able to understand the name which is communicated to it. This corresponds to communication of names. When global naming is used, every object is able to understand the names it receives, they are communicated as they are. In the context of local naming, names understood by one object may not be by another object. This requires translation of names so that the latter object may now understand the name which is received.

3.2. Issues related to object composition

Visibility and hiding of components

When considering a composition, the objects that belong to the composition are the component objects and the objects which do not belong to the composition are the objects outside the composition. Visible components are the components whose names can be exported to objects outside the composition. Hidden

components are the components that must remain unknown to objects outside the composition. Component hiding implies the following restrictions:
R1: An object not belonging to a given composition may not interact with a hidden component .
R2: There is no restriction on a hidden component to interact with other objects outside the composition.

Visibility and hiding of components requires that we clarify how we reference a component from another component, from the composite object, and from an object outside the composition. Such a reference operation may use global names or local names. For instance, since in the object paradigm, the object identity is used as a name of the object, this corresponds to global naming. Another naming mechanism may use local names which in turn may be role-based or not. In a composition, the names of components are role-based when they are relative to the role played by the component in the composition. The names of components are not role-based when they are based on their identity. For instance, if we substitute a component in this context, we should use another name to reference the substituting component, while in the former case, the old name is still meaningful.

Once the mechanism used to reference the components is known, we must decide how we make components visible or hidden. In fact, there are many advantages to achieve visibility or hiding by including this capability into the naming mechanism. When the names are resolved, we can deny or grant the access to an object. This means each time an interaction has to take place between two objects, the object naming mechanism must make sure that such an interaction is desirable. This supposes that the object naming mechanism must be aware of the componenthood of the interacting objects. We require that the restrictions related to hiding of object must be enforced by the object naming mechanism since they can not be statically checked. In fact, visibility and hiding of objects, when the objects are manipulated through their names, can be viewed as an extension of the name resolution mechanism, i.e. when names are resolved, we can deny or grant the access to an object based on the object using the name and the target object.

Checking if two objects belong to the same composition

It may be useful to know if two objects belong to the same composition. This will ease the design of algorithms dealing with recursive data structures. With composition, we may take advantage of the preorder relation between objects based on the hierarchical relationships between the objects. For instance, when we are copying a hierarchy of objects, it is important to know if an object was already copied based on the composition to which it belongs. In addition, many distributed applications are based on hierarchies. For instance, network management assumes that the network is made of various components which in turn are made of other components. The failure of one component may cause the failure of the composition and also the malfunctioning of the other components. In such a context, we are interested to know whether a faulty network card is in the same equipment as a given communication port. Checking whether two objects belong to the same composition is an operation which can be seen as an extension of the operation of comparing object's names.

Checking if one object is a component of another object is a variant of checking whether two objects belong to the same composition. One of the best approaches to checking whether two objects belong to the same composition should be based on operating at the level of the object names. This suggests that the object name records the information about the object hierarchy. This can not be achieved when object identities are used as object names. In addition, global names are independent of the organization of objects in the application. This means, global names can not record information about the object hierarchy. This leaves as only alternative to use local names.

Sharing of components between compositions

From the modeling point of view, we may be interested to model compositions which share components. There is no particular problem with component sharing when we use global naming, especially object identities, to name objects. However, when local naming is used and objects are named on a per composition basis, component sharing implies that a component may have more than a name since it may belong to more than one composition. How can we check that two

names resolve to the same object? In addition, how does the shared component resolve the names of other objects. This is a matter of concerns since two components of different compositions may have the same name.

Referencing objects between compositions

Interactions between objects are not limited to objects belonging to the same composition. Objects belonging to different compositions may also communicate. This means the object in one composition should be able to reference an object in another composition. When global naming is used, there is no particular problem. However, with local naming, and in particular when names are attributed on a per composition basis, it is not clear how an object in a given composition may reference an object in another composition. In this context, composition of names may allow us to cope with reference between compositions, i.e. the objects in other compositions are refered to using a combination of the name of the composition and the local name of the object in the composition.

3.3. Alternatives to the object naming in the object paradigm

In this section, we have exposed various issues which need to be addressed with respect to naming in the context of object composition. These issues can be considered as additional naming requirements which must be fulfilled by any naming mechanism which has to deal adequately with object composition. In order to fulfill these new naming requirements, we may change the way we interpret object naming. A shift in our thinking could consist of viewing an object name as a channel which allows to communicate with an object. Only communication paths between objects are named and not the objects themselves. Or, we could recognize that the new naming requirements can not be fulfilled by the available object naming mechanism, since they represent orthogonal requirements to that actually fulfilled by the concept of object identity. As a consequence, a different concept may be more adequate for tackling these new naming requirements. For instance, we may use different concepts for different purposes, i.e. the identity to distinguish intrinsically one object from another, and (contextual) names to reference the objects. In addition, object identity is not contextual because the concept of object identity is aimed at distinguishing the

objects independently of their use. However, object reference is contextual. It implies a context, a lifetime, a relationship, and a meaning. As a consequence this context can be used to find out the object denoted by a name.

A variety of alternatives are at our disposal for fulfilling these new naming requirements imposed by composite objects. We may adopt the "do not care" approach, ignoring these requirements and leaving each software designer with the trouble of devising specific modeling and implementation artifacts for coping with these naming requirements. Instead, we prefer to address the problem by proposing another approach to designating objects, even at the expense of changing the way we view object naming in the object paradigm.

It is interesting to notice that based on the syntax, semantics and operations on names, a graph remains a good abstraction for discussing about these concepts. The name syntax, semantics and operations can be described by means of graphs. For instance, assume a graph G=(E,N). Its nodes N represent the objects. The edges E represent the references between the objects. Assume also that the edges are labelled using labels at the start and at the end of the edge. The labeling convention represents the naming convention in the sense that when global naming is used, all the edges ending at a given object have the same labels. When the edges ending at the same node may have different labels according to the starting nodes, this corresponds to local naming. The syntax of labels corresponds to the syntax of names. In addition, one object may reference an object which is not directly reachable from it based on the graph G. Such references may be handled by adding a new edge to the graph, or by considering the sequence of edges leading to the referenced object from the referencing object. Operations on names are interpreted as follows. Attributing a name to an object consists of drawing an edge to that object. Comparing the names becomes an operation on the labels of edges and the nodes of the graph. Resolving a name is traversing the graph to the required object. Communication of names is interpreted as drawing a new edge from the receiving object to the object denoted by the received name.

On the other hand, operations on names in the context of object composition are interpreted as follows. Notice that these operations apply only on the hierarchical relationships between objects. Therefore, we may limit the graph of objects to a tree representing the hierarchical relationships between the objects. This tree is derived from the graph based on the assumption that each composition is able to reference its components and each component has a reference to the composition to which it belongs. This assumption is used to prune the graph with non-hierarchical references between the objects reducing the graph to a tree. Now, armed with this tree, we interpret the operations on names in the context of object composition as follows: visibility and hiding of components is linked to the way we traverse the tree. For instance, hidden objects can be tagged so that when we resolve a name, we can check if the object is visible or not from the standpoint of the object using that name. Checking if two objects belong to the same composition is achieved by operating on the tree. We shall see later how this interpretation of name syntax, semantics, and operations in terms of graph and graph operations leads to a formalization of naming.

4. An Approach to naming

4.1. Basic idea

We would like to handle the new requirements explained in Section 3 using a formal framework. In the past, naming mechanisms have been formalized with mathematical relations and partial functions, see [Wier95, Kent91] for more detail. Such a framework adequately captures the relationship between names and objects, and the issues related to the resolution of names, but it tends to be oriented towards global naming.

We have stressed that the new requirements require contextual reasoning for manipulating and attributing names. This has led us to model the state of an application as far as naming is concerned, as a labeled graph of objects. Instead of devising a completely new formal framework, we advocate the use of existing ones. Sense of direction, described in [Floc98a], appears to be a formal framework in which deep semantic issues of naming mechanisms can be addressed. Hereafter, we show how the new requirements can be described and

fulfilled using the concept of sense of direction. One of the advantages of sense of direction is that we can study both, global and local naming. In addition, many properties of naming mechanisms can be described in terms of sense of direction. It is less low-level than graph theory, or mathematical relations, and partial functions.

In this paper, we use two kinds of sense of direction, one is based on hierarchical naming, and the other on a chordal naming. The reasons for adopting a hierarchical design in the context of object composition is largely that hierarchies are easy to understand and to implement. Hierarchy helps to ensure the uniqueness of names, to reduce ambiguity, and to impose common names for all objects.

4.2. Sense of Direction

Sense of direction is a property of labelled graphs which has been extensively studied in the context of distributed computing (for a survey see [FlSa98]). It provides a framework in which answers to deep semantic questions concerning naming can be addressed, and solutions can be proposed based on semantic coherence and theoretical feasibility. As an example, communication of names may involve the transfer of a name to an object which interprets names in a different context than the sender. How can we prove that the naming mechanism is consistent with respect to the communication of names? We shall see later in this paper how this can be achieved in the formal framework provided by sense of direction.

Definition of Sense of Direction

Let G=(V, E) be a graph where nodes correspond to objects and edges to references between the objects. Let E(x) denotes the set of edges incident to node x. Every node associates a label to each incident edge. Let $\lambda_x(x,z)$ denote the label associated by x to the edge (x,z) in E(x). λ_x is a local edge labeling function such that $\lambda_x : E(x) \rightarrow \mathfrak{L}$ where $\mathfrak{L}$ is a set of labels. Let λ denote the set of all λ_x. We say that λ is a local orientation if each node can distinguish among its incident edges, i.e. $\forall x \in V, \forall e_1, e_2 \in E(x)$: $\lambda_x(x, e_1) = \lambda_x(x, e_2)$ iff $e_1 = e_2$.

A path in G is a sequence of edges in which the end point of one edge is the starting point of the next edge. Let P[x] denote the set of all paths with x as a starting point, and let P[x,y] denote the set of paths starting from node x and ending in node y. Let Λ be the extension of the labeling function λ from edges to paths. Λ_x : P[x] -> $\mathfrak{B}^+$ be the path-labeling function defined as follows: for every path $\pi \in P[x]$, $\Lambda_x(\pi) = (\lambda_x(x,x_1), \lambda_{x_1}(x_1,x_2), .., \lambda_{x_{m-1}}(x_{m-1},x_m))$ where $\pi = [(x,x_1), (x_1,x_2), .., (x_{m-1},x_m)]$.

Coding function

We now introduce the notion of coding and decoding functions. We also show how to use them in order to construct local names. A coding function is a function that maps a sequence of labels corresponding to a path in G into a value, in such a way that two sequences corresponding to different paths starting from the same node are mapped in the same value iff the ending point of the paths are the same.

A coding function f of a system (G, λ) is a function such that :
$\forall$ x,y,z $\in$ V, $\pi_1 \in P[x,y]$, $\pi_2 \in P[x,z]$, $f(\Lambda_x(\pi_1)) = f(\Lambda_x(\pi_2))$ iff y = z.

The coding function allows the node to understand, from the labels associated to the edges whether different paths from any given node x end in the same node or in different nodes.

Decoding function

A decoding function h for f is a function that given a label and a coding of a string (sequence of labels), returns the coding of the concatenation of the label and the string. It allows nodes to translate the local names of their neighbors.

More precisely, given a coding function f, a decoding function h for f is such that for all x, y, z $\in$ V, such that (x,y) $\in$ E(x) and $\pi \in P[y,z]$, we have $h(\lambda_x(x,z), f(\Lambda_y(\pi))) = f(\lambda_x(x,z) \cdot \Lambda_x(\pi))$, where "." is the concatenation operator.

Sense of direction [Floc98a]

A system (G, λ), has a sense of direction (*SD*) iff the following conditions hold:

1) λ is a local orientation,

2) there exists a coding function f,

3) there exists a decoding function h for f.

We shall also say that (f,h) is a sense of direction in (G, λ). Several examples of sense of directions (e.g., cartographic, chordal, neighboring) have been described in [Floc98a].

Naming scheme

In a given labelled graph G, we can define a local naming scheme as follows. Each node associates a local name to the other nodes in the system; let us denote by $\beta_x(y)$ the name that node x locally associates to y and by β the set of all β_x. The set $\{ \beta_x(y): y \in E(x)\}$ is also called local view of x. The collection of all local object naming schemes $\beta = \{ \beta_x: x \in V\}$ constitutes the object naming scheme.

In the case of a graph with sense of direction, an object naming scheme can be constructed based on the existence of the coding function f; more precisely, the name that an object gives to another object is either the label of the link between them (if there is such a direct link), or the coding of a sequence of labels leading to it. In other words, given a coding function f, a local object naming scheme β_x for x is constructed as follows:

$$\forall x,y: \beta_x(y) = \begin{cases} \lambda_x(x,y) \text{ if } y \in E(x) \\ f(\alpha) \text{ otherwise} \end{cases}$$

where α is the sequence of labels corresponding to an arbitrary path between x and y.

Given a graph representing an object system, many different labelings could be constructed to have different senses of direction, each one providing a different naming scheme.

Sense of direction allows to define names which can be used in contexts where they are not defined. When the labeling is a sense of direction, it is possible to understand from the labels associated to the edges, whether different paths from any given node x end in the same node or in different nodes. Using this characteristic, we may distinguish between objects. In a labeled graph with sense of direction, there is a function which maps the sequences of labels associated to the paths from x to y to the local name $\beta_x(y)$ used by x to refer to y. This latter characteristic allows an object to refer to objects which are not directly reachable from it. If there is sense of direction, node x, based on the label $\lambda_x(x,y)$ and on the name $\beta_y(z)$, can deduce that the received information is about the node locally called $\beta_x(z)$. This allows objects to exchange valid object names, i.e. names which can be understood.

4.3. Relation between naming mechanisms and sense of direction

As observed in Section 3, a labelled graph is a good abstraction of a naming mechanism. This has led us to examine how the concept of sense of direction which is a property of labelled graphs is linked to naming. The concepts underlying sense of direction can be related the properties which characterize object naming mechanisms.

An object naming mechanism has two aspects, the naming convention and the operations on names. In a framework with sense of direction, the name syntax is captured by the edge-labeling function λ and its extension to paths Λ. The semantics of names are captured by the node-labeling function β. In particular, the name attributed by the node x to the object y is $\beta_x(y)$. On the other hand, the operations on names such as attribution, comparison, and resolution of names are captured by the node-labeling function and its properties. Communication of names is handled by the existence of a coding and a decoding function. This latter function is used to translate the names so that an object can understand the names it receives from the other objects. In the sequel, we illustrate how a chordal naming with sense of direction can be used as a naming framework.

Sense of direction has many properties which help to handle some of the issues related to object composition. For instance, checking whether two objects belong to the same composition can be achieved by exploiting the fact that the naming convention is a sense of direction. An algorithm based on message broadcasting in a graph, as explained in [Floc98b] can be devised. Since the naming convention is a sense of direction, the algorithm is guaranteed to complete in linear time over the entire graph. This is possible because with sense of direction, we can distinguish between two nodes based on their names, and the names known by one node can be understood by another node. Checking whether an object is a component of another composed object is a special case of checking whether two objects belong to the same composition. Depending on the naming convention, specific algorithms can be devised for answering to these questions as we shall see later in this paper.

With sense of direction, we can also handle visibility and hiding of objects in a composition. Recall that we have introduced this property as an extension of the operation of resolving a name based on the object using that name and the target object. It suffices to extend the node-labeling function such that it can now query a map defining the visibility of objects in compositions. In addition, communication of names between compositions is handled by imposing the translation of names before they can be used by another object. This is achieved through the decoding function h of the sense of direction. In fact, $\forall$ x, y, z: $\pi \in$ P[y,z] , we have $f(\Lambda_y(\pi)) = \beta_y(z)$, and $\beta_x(z) = h(\lambda_x(x,y), f(\Lambda_y(\pi)))$, i.e. from $f(\Lambda_y(\pi))$ and $\lambda_x(x,y)$ we can deduce $\beta_x(z)$.

4.4. Exemplifying the use of sense of direction as a naming framework

Assume that each object uses its own local names to reference other objects. We also assume that an object uses different local names to refer to different objects. A chordal labeling of a graph G=(V,E), with |V| = n , is defined by fixing a cyclic ordering of the nodes and labeling each incident link (x,y) of x by the distance (modulo n) of the two nodes in the above cycle. It can be easily shown that a graph with a chordal labeling has a sense of direction (see [Floch94] for more detail). For instance, assume the graph G=(V,E), with |V| = n . Let γ: V->V be a

successor function defining a cyclic ordering of G and let $\gamma^k(x) = \gamma^{k-1}(\gamma(x))$ for k > 0. Let $\delta: V^2 \rightarrow \{0, .., n-1\}$ be the corresponding distance function, i.e. $\delta(x,y)$ is the smallest k such that $\gamma^k(x) = y$. The labeling λ in G is defined such that $\lambda_x(x,y) = \delta(x,y)$. The coding function f is defined as follows

$\forall \pi \in P[x_0], \pi = [(x_0,x_1), (x_1,x_2), .., (x_{m-1},x_m)]$:

$$f(\Lambda_{x_0}(\pi)) = f(\lambda_{x_0}(x_0,x_1), \lambda_{x_1}(x_1,x_2), .., \lambda_{x_{m-1}}(x_{m-1},x_m))$$
$$= \Sigma_{i=0,\, m-1}\, \lambda_{x_i}(x_i,x_{i+1}) \bmod n$$

Its corresponding decoding function h is:

$\forall (x_0,y_0) \in E(x_0), \forall \pi \in P[y_0], \pi = [(y_0,y_1), (y_1,y_2), .., (y_{m-1},y_m)]$:

$h(\lambda_{x_0}(x_0,y), f(\Lambda_{y_0}(\pi))) = \lambda_{x_0}(x_0,y_0) + f(\Lambda_{y_0}(\pi)) \bmod n$

To build an object naming scheme based on a chordal sense of direction we use the following. Given a coding function f, the local object naming scheme β_x for x is constructed as follows: $\forall$ x,y: $\beta_x(y) = f(\alpha)$ where α is the sequence of labels corresponding to a path between x and y. The collection of all local object naming schemes $\beta = \{\beta_x : x \in V\}$ constitutes the (global) object naming scheme. In this naming, we can not know if an object is a component of another object or if they belong to the same composition since chordal naming does not record the hierarchical relationships between objects. However, translation of names is fairly easy. For instance, when an object, say "x" wants to communicate to the object "y" the name of the object "z", this name can be translated. By definition of the decoding function h, $\forall (x,y) \in E(x)$ and $\pi \in P[y,z]$, $h(\lambda_x(x,z), f(\Lambda_y(\pi))) = \lambda_x(x,z) + f(\Lambda_y(\pi))$. By definition of the coding function f, $f(\Lambda_y(\pi))) = \beta_y(z)$, moreover $\lambda_x(x,z) + f(\Lambda_y(\pi)) = \beta_x(z)$. It follows that $\beta_x(z) = h(\lambda_x(x,z), \beta_y(z))$.

The figure below illustrates the application of this naming to a network consisting of three computers, namely c_1, c_2, and c_3. The computer c_1 is managing a large disk array "d"; c_2 is managing a laser printer "p"; and c_3 is managing a satellite dish "s".

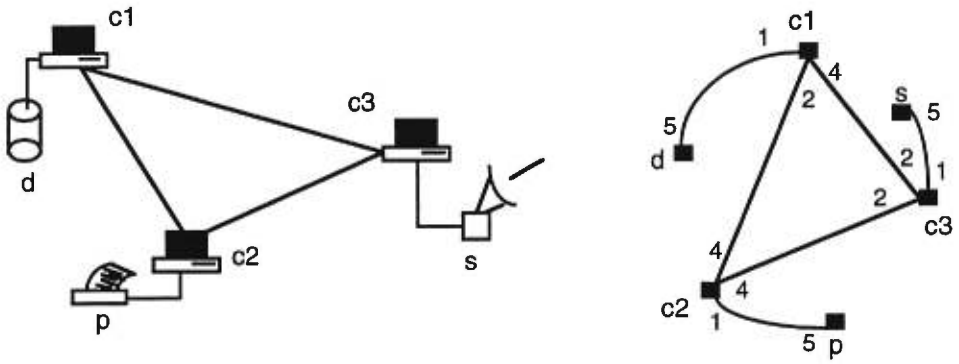


Figure 5.1: Chordal labeling of a computer network

In the graph with chordal labeling, the objects are represented by small squares. The local edge labels are used as follows. For instance, consider the object "c_1". When "c_1" references the object "d", it uses "1". When the object "d" references "c_1", it uses "5". When "c_1" references "c_2", it uses "2" and when "c_2" references "c_1", it uses "4", and so on. The translation of names occurs when the computer c_1 wants to send a document to the printer managed by the computer c_2. c_1 receives the name of the printer as "1" from the computer c_2. c_1 understands this name as $\beta_{c_1}(d)$ = "3".

4.5. Hierarchical Naming

In [Floc97], it is shown that in a tree any labeling with local orientation is a sense of direction. Remember that a local orientation in a labelled graph is characterized by the property that a node uses different labels for its adjacent edges. This observation can be used for devising a hierarchical naming from the graph of objects. This is achieved by deriving a tree that reflects the hierarchical organization of the application. In particular, we introduce a root object which represents the application itself. In addition, we assume that there is no sharing of components between compositions. The object is created in the context of a given composition. The application itself is considered as a composition.

Nodes of the tree are compositions. Each composition has a local labeling function λ_x characterized by the labels in λ_x are from the set L where $L = L_0 \cup L_0^- \cup \{self\}$. The special label *self* denotes the composition. This means a node

can reference itself using the label *self*. Elements of L_0 are written n, and $L_0^- = \{ n^- \mid n \in L_0 \}$. n^- is the symmetric form of n in L. n and n^- are labels of edges linking two nodes x and y such that if x is a composition and y is a component of x, the edge (x,y) is labeled n and the edge (y,x) is labeled n^-.

As shown in [Floc98a], this labeling is a variant of a contracted sense of direction. Its coding function f is defined as the sequence of labels corresponding to the shortest path in the tree between the naming and the named objects. Here also, the name $\beta_x(y) = f(\alpha)$ where α is the sequence of labels representing a path between x and y. The figure below illustrates the computer network with this hierarchical labeling.

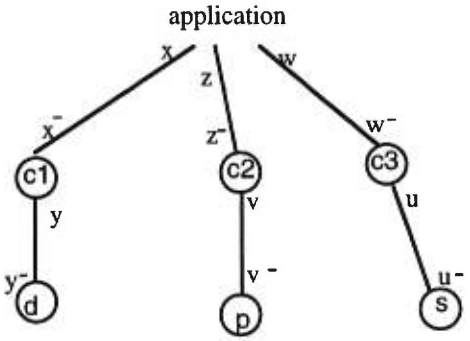


Figure 5.2: Example handled with hierarchical naming

As explained, the labeling of the graph may allow us to use simpler algorithms than one based on message broadcasting in a graph with sense of direction in order to check whether two objects belong to the same composition. Let $\beta_x(y)$, $\beta_x(z)$ be the names the object x attributes to the objects y and z with $x \neq y$, $y \neq z$, and $x \neq z$. The algorithm used by the object x to tell whether y and z belong to the same composition is based on the following properties. Keep in mind that we always assume that the application itself is a composition and all the objects are its components. This algorithm is applied at a given object x using the information provided by sense of direction. We assume tha each node is able to name the root of the hierarchy, i.e. $\beta_x(root)$ is known. From the object x, the objects y and z belong to the same composition iff:

- $\beta_x(root)$ is not a prefix of neither $\beta_x(y)$ nor $\beta_x(z)$;

or

- $\beta_x(root)$ is a prefix of both $\beta_x(y)$ and $\beta_x(z)$.

On the other hand, for the object x to tell whether y is a component of z, x can take advantage of the algorithm presented above because y and z have to belong to the same composition. Here, we have two cases:
(1) $\beta_X(y)$ has only labels in L_0^- and $\beta_X(z)$ has only labels in L_0. Here z is a component of y.
(2) $\beta_X(y)$ is a prefix of $\beta_X(z)$ and after cutting the prefix, the resulting $\beta_X(z)$ has labels only in L_0 or only in L_0^-. If $\beta_X(z)$ has labels only in L_0 then z is a component of y, otherwise y is a component of z.

4.6. Hierarchical naming with component sharing

To achieve sharing of components in the hierarchical naming proposed in this paper, we use the following artifact. We still consider the tree without component sharing. It is obtained by considering that objects have weight. Objects which are close to the root of the hierarchy are more weighted than those far from the root of the hierarchy. Based on that assumption, when a component is shared, it first belongs to the composition closest to the root of the hierarchy. This artifact allows us to derive a tree according to the previous hierarchical naming. Now, we complement this tree with chords (i.e. direct edges) linking a given composition to its components when such composition relationships are not portrayed by the previous tree. The label assigned to such a chord to the composition equals to the sequence of labels of a path between the two nodes linking the compositon to its component in the tree. Its symmetric is the inverse path.

This labeling of the hierarchy is also a sense of direction. Its coding function f is the same as before, i.e. the sequence of labels of a path between two nodes in the tree. The naming function β is also defined to be equivalent to the coding function f. However, with such a naming, we are not able to answer to the question whether two objects belong to the same composition using the algorithm proposed for hierarchical naming. To answer this question, we are forced to use another labeling. Another alternative consists of using the algorithm which is based on depth-first search and message broadcasting in a graph with sense of direction as explained in subsection 4.3.

Finally, we have also explored the combination the hierarchical and chordal namings, into a single naming. The approach to their combination consists of structuring the application in terms of its compositions. This structuring is captured in a hierarchy of compositions. We use hierarchical naming for referencing one composition from another. Within a composition, we use a chordal naming. This is made possible by extending the node labeling function β to pairs of names, e.g. $\beta_x(y)=(a,b)$ where a is the name of the composition where y is located and b is the cyclic ordering of y within the composition. This results in a naming convention which has the advantages of both hierarchical and chordal namings. However, this naming is not intuitive.

4.6. Related work

A problematic similar to visibility and hiding of components have been studied by Hogg in the context of removing aliasing in object-oriented languages [Hogg91]. In a programming language with destructive read[12], Hogg introduces the idea of islands. An island consists of a container, i.e. set of objects, and a manager, which represents a the bridge to the rest of the world. An island is the transitive closure of a set of objects accessible from a bridge. A bridge is an object which can be considered as the container object for a set of objects. As such it is the unique access point to the objects in the container that make up the island; no object that is not a member of the island holds any reference to an object within the island except the bridge. An object outside the island can have a reference, by means of a variable with destructive read, to an object inside the island only through the bridge which manages the island. It can use the reference only once because of the mechanism of destructive read. This means if we associate a bridge to the composition, it can decides when to make available the references to its components, therefore enforcing visibility and hiding of components. Bridges and islands were originally devised as a technique for making proof systems for object-oriented languages practical. The solutions proposed are implementation oriented, i.e. they are described in the context of imperative object-oriented programming languages. They are not abstract enough to be useful in analysis and design of applications.

---

[12]Variables with destructive read are such that once they are read their values are gone.

Chien and Dally have also studied the naming in the context of composite objects [Chien90]. They propose the concept of a concurrent aggregate to model a homogeneous collection of objects. A concurrent aggregate represents a group of objects (called representatives). Messages sent to the aggregate are directed to arbitrary representatives. The representatives of an aggregate can communicate by sending messages to one another. This is achieved through local naming within an aggregate. The representatives in an aggregate have indices from 0 to groupsize-1. groupsize is the number of representatives in an aggregate. The expression (sibling group 0) would return the name of the 0th representative in an aggregate. Representative names are ordinary object names or identities. Chien and Dally concentrate on intra-aggregate naming while we are concerned with more broader issues.

Local referential integrity introduced by Kappel and Schrefl is a concept which is in relation with composition of objects [Kapp92]. It attempts to apply the idea of referential integrity within the scope of a composite object. Referential integrity is satisfied if every object referenced exists. Accordingly, the concept of local referential integrity means that in any object reference, the referenced object and the referencing object belong to the same composite object. In addition, the authors mention that within their framework, objects are named using global and unique names which is proven in this paper to be inadequate for handling the new requirements in the context of object composition.

RM-ODP Naming [ISO98a] is close to hierarchical naming as proposed in this paper. In addition, RM-ODP Naming introduces the concept of overlapping of naming contexts. When interpreted in the context of object composition it means having components which have the same names from one composition to another. However, RM-ODP Naming does not address the problem of checking if two objects belong to the same composition. In addition, the treatment of name communication is rather informal.

5. Conclusions

In this paper, we gave a new formulation of naming requirements in the context of object composition. These requirements can be summarized by: (1) visibility and hiding of components, (2) checking whether two objects belong to the same composition, (3) sharing of components between compositions, (4) from one composition referencing an object in another composition. We show that global names are inappropriate for fulfilling these requirements.

In an attempt to solve the problems related to these naming requirements, we designed naming mechanisms which are based on viewing the execution of an application as an evolving structure of a labeled graph of objects. Each state of the application is represented by such a graph. This allowed us to take advantage of the concept of sense of direction in the design of the different naming schemes. Sense of direction exhibits many properties that allow the design of naming mechanisms which can: distinguish between objects, permit the transfer of object names between objects, handle sharing of components, and preserve the locality of names. In this paper, we show how many properties of sense of direction relate to the properties of naming mechanisms. We describe three naming mechanisms using sense of direction. The first is based on a chordal sense of direction. The second is hierarchical naming without component sharing. The last is a slight modification of the first to allow component sharing, this is achieved by introducing special names for shared components. The main difference between chordal and hierarchical namings is that the hierarchical naming is not linked to the structure of the application and its objects. In these experiences, we found that allowing component sharing complicates a lot the hierarchical naming mechanisms. This raises the question of the necessity of allowing component sharing between compositions. From a modeling point of view, compositions which share components might be considered as a single composition.

Future developments of this work are concentrated in the area of formalization, especially the integration of the naming mechanisms which are proposed into specification languages. In this context, π-Calculus [Miln93] may play an

important role. Since π-Calculus is built upon the notion of naming and because it provides a convient semantic substrate for object-oriented programming (see for [Miln93] more detail), the formal framework provided by sense of direction can be specified using π-Calculus, and later integrated in the formal semantics of an object-oriented programming language. Walker explored first the use of the π-Calculus to give semantics to concurrent object-oriented programming [Walk90]. In his formalization, he deals only with global names. We believe that sense of direction can be implemented as a naming discipline above the computational model provided by π-Calculus. Such a formalization will be more general than the one using sense of direction solely, because it will include the mechanisms of object interactions and the handling of multiple interfaces for objects. This formalization can later be used to describe the RM-ODP computational model. Future work of this research will explore the ideas sketched above.

Chapitre 6

Étude de cas dans la composition des objets

Sommaire

Nous réexaminons l'intégration du cadre conceptuel dans une méthode de développement orienté objet à la lumière de la théorie de composition proposée dans le chapitre 4. Il faut souligner que dans cette théorie de composition les interactions des objets ne sont significatives que dans le contexte des associations reliant ces objets. Ce principe est utilisé pour modifier la méthode XOMT. Le processus de développement de XOMT est allégé et deux étapes principales sont envisagées: la description de la structure et la spécification du comportement. La description de la structure se fait en fonction des diverses compositions se trouvant dans l'application. Les associations identifiées lors de cette étape servent ensuite pour indiquer les différentes interactions des objets. Cette dernière étape est complétée par des spécifications en CSL des différentes compositions.

La preuve des propriétés de la spécification est aussi fournie. Nous montrons comment tirer profit de la traduction des spécifications CSL en TLA afin d'utiliser les règles de raisonnement de TLA dans la preuve des propriétés des spécifications. Des outils de spécification, vérification et de génération d'implantation utilisant TLA peuvent alors être employés pour des spécifications CSL. Ce chapitre illustre la mise en oeuvre des travaux présentés dans cette thèse dans le but d'élaborer une étude de cas.

Ce chapitre est la version intégrale d'un article réalisé sous la supervision de mon directeur de recherche et il paraîtra dans les Actes du *Workshop on Formal Methods for Object-Based Distributed Systems (FMOODS'99)* de février 1999 [Rama99]. J'en suis l'auteur principal.

OBJECT COMPOSITION: A CASE STUDY

Dunia Ramazani, Discreet Logic
Gregor v. Bochmann, University of Ottawa

Abstract

In [Rama96b], we have presented an object-oriented method, called eXtended Object Modelling Technique, XOMT for shorthand. In this paper, we modify the developmental approach used in XOMT to include the specification of the behavior of composite objects based on synchronous interactions. The new developmental process consists of describing the application structure in terms of objects and associations between these objects. Object associations are then further refined by describing the object interactions that occur in the context of these associations. Object and association behaviors are specified in CSL, a specification language based on rendezvous interaction. CSL specifications can be translated in TLA. This adds a reasoning capability to the development process. The translation into TLA is motivated by the existence of a variety of specification and verification tools for TLA.

Keywords: Case study, CSL, Formal reasoning, Object composition, OMT, TLA, XOMT.

1. Introduction

In software engineering, experimentation is a necessary adjunct to process improvement. Objective and meaningful case studies can help us understand particular object-oriented notations, methods, or languages. In [Rama96b], the presentation of eXtended Object Modelling Technique, XOMT for shorthand was aimed at presenting the concepts and principles underlying XOMT. Now we have designed a new specification language for capturing the behavior of composite objects, namely CSL which stands for Composition Specification Language. This paper shows how these latter results can be combined with

XOMT. A specification case study is used for conveying our ideas, and demonstrating the usability of the results in the context of XOMT.

The integration of CSL is achieved at the expense of modifying the developmental approach used in XOMT. We departed from the OMT-based [Rumb91] development process to a lighter process based on describing the application and object structures in terms of objects and associations between these objects. Object associations are then further refined by describing the object interactions that occur in the context of these associations. Object behaviors and association behaviors are specified in CSL, which can be translated into TLA [Lamp97]. This adds a reasoning capability to the development process, i.e. we may then use TLA tools for verifying the specifications. The application selected in this case study is the specification of a lift system.

The paper is structured as follows. In Section 2, we begin by reviewing the essential concepts and principles of XOMT. It is followed by a summary of features provided by CSL including its formal semantics based on its translation into TLA. We then summarize the development process which uses CSL. In Section 3, the development process is put in practice through the specification of a lift system. This includes the description of the application requirements, the informal specification supported by XOMT and the specifications in CSL, including their translation into TLA and a demonstration of some formal reasoning. We close the paper with the lessons learned in this experiment and point to some future work.

2. Overview of XOMT and CSL

2.1. Concepts and Principles underlying XOMT

Roughly speaking, XOMT is OMT with a few add-ons allowing to specify composite objects such that there is a linkage between the structure and the behavior of these objects. In XOMT, properties of composite objects are classified into inherent, aggregate, and emergent properties. A composite object is an object with an internal structure which consists of the components, and the interconnections including the dynamic interactions between the

components of the composite object. The linkage between component properties and composite object properties is established by distinguishing inherent properties, i.e. properties of the composite object which semantics is provided by properties of its components, from aggregate properties, properties of the composite object obtained by combining the properties of its components using aggregation mechanisms, and from emergent properties which are properties of the composite object which do not depend on component properties.

This new classification requires notational changes to object models in order to capture visibility and hiding of components, promotion of component properties to the status of composite object properties, and aggregation mechanisms used to combine the component properties. Other areas of improvement of OMT include the specification of the communication between objects, especially the communication between the components of a given composition. A more abstract interaction mechanism based on behavior constrainment is proposed, namely Contract specifications where behavioral interactions are expressed in a more abstract way so that the description does not introduce implementation bias. It is achieved by constraining, through predicates, the behavior of the interacting objects. Contracts are related to object associations abstracting the interactions between the classes. They represent the concurrent composition of the participant statecharts (individual behaviors) such that the behavior of the participants conforms to the constraints explicitly stated in the contracts. Constraints are expressed using a combination of first order predicate logic and OMT constraints.

2.2. Steps and notation in XOMT

The description of composite objects proceeds in three steps [Rama95a]. The configuration describes the structure of the composite in terms of components, associations and behavioral interactions among these. The next step, juxtaposition shows how the composite is linked to its components through the aggregation association, and inherent and aggregate properties. The last step, emergence concerns the specification of emergent properties.

Aggregation association is represented like in OMT. Visibility/hiding of components is specified by having the component class represented respectively with doubled and simple framed rectangles. The attributes and operations of composites are classified using three separate rectangles. Inherent, aggregate, and emergent attributes and operations are represented using respectively inherent, aggregate, and emergent rectangles, that is three rectangles in place of one, like for classes in OMT. Inherent associations are of two kinds, those resulting from the visibility of components and those involving hidden components. The former are represented by associations crossing the boundary of the composite and ending at some visible component inside the composite. The latter are represented by associations ending at the boundary of the composite and continued within the composite by a dashed line to some hidden component. Aggregate and emergent associations are represented using associations ending at the boundary of the composite and respectively decorated with the annotations {A} and {E}.

To describe the behavior, we use parallel composition of statecharts. This is represented by having the parallel statecharts separated by a dashed line while being enclosed in the composite statechart [Hare88]. Inherent behavior is represented by the statechart of visible components and a specific statechart (annotated with "promoted") for the behavior originating from hidden components. Aggregate and emergent behaviors are respectively specified using specific statecharts. The overall behavior of the composite object consists of parallel composition of inherent, aggregate, and emergent behavior.

2.3. Composition Specification Language (CSL)

2.3.1. Concepts and principles

CSL has its roots in the assumption that object interactions occur only over object associations. In CSL, object interactions are called abstract events Boch93b] and they correspond to joint-actions [Jarv92, ISO89a]. They are characterized by:

(a) There is no asymmetric caller/callee relationship: It is not said which object makes the decision for the execution of an interaction.

(b) There may be more than two objects participating in a given interaction, i.e. multiparty interactions.
(c) Each participating object may impose certain conditions which must be satisfied when the interaction occurs. Each participating object may also define some local state changes that occur during the execution of the interaction.

Once we are able to describe the interactions that occur in the context of object associations, we may use the same approach for describing the interactions between components in a composition. Using this approach, we are able to describe structures of compositions. Composition in CSL is achieved by connecting the objects and then hiding certain abstract events which involve these objects. Hiding is used as an abstraction mechanism.

2.3.2. Notation

The notation proposed by CSL is portrayed below. In CSL, we assume that a simple object (non-composite) can be represented as a composite object which has no components. Based on that assumption, we use a single template for specifying objects. Only object compositions have the "inherits" section which indicates the actions of the components which are also actions of the composition.

spec spec_name
    constants name = <*value*>
    define typename : <*type*>
    extends $name_i$, .., $name_j$
    contains
        $name_k$: $spec\_name_t$
        ..
        $name_p$: $spec\_name_s$
    inherits
        $action_i$ **from** $name_i.action_k$
        ..
        $action_q$ **from** $name_q.action_l$
    local variables
        <*variable declaration*>
    initial conditions
        <*assignment to local variables*>
        <*conditions on the components*>
    invariants
        <*predicates on the local variables*>
        <*predicates on the components*>

```
behavior
        abstract events
                association name
                        event aei = <abstract event definition>
                        ..
                        event aek = <abstract event definition>
                ..
                association name
                        event aei = <abstract event definition>
                        ..
                        event aek = <abstract event definition>
        -- other actions
        action name (<parameter list>) = <action definition>
        ..
        action name (<parameter list>) = <action definition>
end spec name
```

Figure 6.1. Object template.

The "constants" and "define" clauses are self-explanatory. They introduce constants and new types in the specification. The "extends" clause denotes specialization. Features of the specifications listed in the "extends" clause are augmented with new features introduced by the specification. Actions can be redefined, invariants more constrained, etc.

The "contains" clause indicates the structure of the composition. It lists the components, while the constraints on these components can be stated in the "invariants" clause. "abstract events" denote the abstract events occurring between the components. These abstract events are structured according to the associations between the objects inside the composition. "local variables" denote the states of the object or the composition. The clause "initial conditions" defines the initial state. It assigns the initial values to the local variables and may define the initial states of the components in the context of the composition. "invariants" are used to record the safety properties of the object, and its components.

The "behavior" clause defines the actions which are supported by the object. Actions may have parameters. The semantics of the behavior clause are: First the initial conditions are established, in any state either an action occurs, or the local variables remain unchanged. The invariants must be always satisfied in

the state before the execution of the action and in the state after the execution of the action. At the level of an object, we assume interleaving concurrency between the actions supported by that object. For a composition, there is interleaving between its abstract events, and its actions.

Until now we have not said too much about actions. Actions are defined in terms of "enabled", "defined", and "changes" predicates. These predicates involve the local variables of the specification. In action definitions, the "defined" predicate indicates the hypotheses about the environment for the action. In particular, the "changes" predicate will normally relate the new values of parameters and local variables to the values before the execution of the action. We use a combination of mathematics and predicate logic to define the semantics of actions. The meaning of an action is:
(a) when "enabled" is satisfied and "defined" is also satisfied, then the action can be executed and "changes" will be true afterwards;
(b) when "enabled" is satisfied and "defined" is not satisfied, the action can be executed, but the result is undefined;
(c) when "enabled" is not satisfied, the action can not be executed.

An abstract event is the synchronization of two or more actions. It is specified by a list of actions which are synchronized and a constraint over the parameters of the synchronized actions. The abstract event is enabled in states where all its composing actions are enabled and the constraint is true. The defined clause of one action involved in an abstract event should be consistent with the definition of the other actions involved in the abstract event, i.e. there are not values of the variables appearing in the defined predicate which are such that the defined predicate is false and the enabled predicates of the actions involved in the abstract event are true. This is referred to as internal consistency of the definition of an interaction.

The specification parameters, local variables, actions, and abstract events are by default visible unless tagged with the keyword **hidden**.

2.3.3. Formal semantics

To reason about CSL specifications, we need an underlying execution model as well as an underlying logic for proving properties. We have favored the translation of CSL specifications into TLA [Lamp97], an existing formal specification language. The choice of TLA (Temporal Logic of Actions) is motivated by its computational model which is based on interaction, its built-in notion of behavior including the temporal ordering of actions, and nonetheless the availability of a wide range of specification and verification tools [Merst94].

The main difference between CSL and TLA is on the semantics of actions. CSL actions have explicit parameters. *In addition, each CSL action indicates what are its assumptions about the environment since the objects are composed through their actions.* This semantics introduce possible undefined behavior for actions. The action also has an explicit enabling condition.

When translated into TLA, CSL action parameters are introduced as additional variables of the TLA specification. In order to represent, in TLA, the case where the "defined" predicate of an action is not satisfied, we take the following approach for translating the action semantics. We write $LV(a_i)$ for the set of local variables appearing in the CSL action a_i, V for the set of all local variables of the entire CSL specification. $CH(a_i)$ is a subset of $LV(a_i)$; it denotes the local variables which are modified by the action a_i. $PAR(a_i)$ is the set of the parameters of the action a_i. Parameters which do not appear primed in the "changes" predicate of the action can be considered as input parameters. The others are output parameters. Based on this observation, $PAR(a_i)$ can be decomposed into two sets $PAR_{in}(a_i)$ for input parameters and $PAR_{out}(a_i)$ for output parameters. Recall that a CSL action is defined by three predicates, namely **enabled**, **defined**, **changes** which will be respectively referred to in the sequel by a_i.**enabled**, a_i.**defined**, a_i.**changes**. The translation of an CSL action a_i into TLA is defined as follows:

$$\forall\ LV(a_i) : \forall\ PAR(a_i): \quad \wedge\ a_i.\textbf{enabled}$$
$$\wedge\ (a_i.\textbf{defined} \Rightarrow\ a_i.\textbf{changes}\ )$$
$$\wedge\ \textbf{unchanged}((V - CH(a_i)) \cup PAR_{in}(a_i))$$

In this translation, notice that explicit TLA **unchanged** predicates are added in order to indicate which variables remain unchanged by the action. Let the notation **TLA-action**(a_i) denote the resulting TLA action when a CSL action a_i is translated into TLA. In TLA, the operator *Enabled*, e.g. *Enabled* a, is used to denote the fact that the action a is enabled, i.e. it may be executed. Using the approach described above, *Enabled* **TLA-action**(a_i) corresponds to a_i.**enabled**. In addition, when a_i.**defined** is not satisfied, any state can be considered as a next state.

With respect to the behavior of an object, TLA and CSL have the same semantics, i.e. the initial conditions hold first and then either an action is executed or the specification variables remain unchanged. Therefore, the translation is one to one for the object behavior.

Once the CSL actions are translated into TLA, abstract events can also be translated into TLA. Abstract events are represented by the conjunction of the involved actions and the constraint of the abstract event. Local variables of an abstract event are introduced using TLA temporal existential quantification $\exists$ which denotes hiding of variables in TLA. Hiding of abstract events in a composition is translated by hiding the parameters of the actions involved in these abstract events.

Specialization in TLA is based on the existence of a refinement mapping between the specifications. It is general enough to cover CSL specialization. In other words, let **TLA-spec**(M) denote the CSL specification M translated into TLA, if the CSL specifications M and N are such that N is a specialization of M, then **TLA-spec**(N) $\Rightarrow$ **TLA-spec**(M).

In CSL, we have a composition rule which can be used to show that the composition of a given number of specifications is a specialization of another higher-level specification. The intuition behind this composition rule is: given a CSL composition of M_1 and M_2, if

1) we have internal consistency in that composition which means that each component behaves well in an environment including the other components with respect to the abstract events relating the latter components to the former;
2) and the composition, considering visible the actions of the components and the abstract events which remain , is a specialization of another high-level specification M
then the composition of M_1 and M_2 is a specialization of M.

Once internal consistency in a composition is shown, specialization suffices to have a sound composition rule. Internal consistency can be demonstrated by showing that none of the abstract events has a participating action for which the "defined" predicate is not satisfied. Once the CSL specifications are translated into TLA, it can be demonstrated as an invariant of the resulting TLA specification. This is achieved using the TLA rule INV1 [Lamp97]. As explained, specialization is proven at the level of the TLA specifications. This has for consequence that CSL compositional reasoning can be achieved at the level of the resulting TLA specification using TLA rules.

On the other hand, to reason about specifications which include explicit assumptions about the environment, TLA introduces the operator $\pm>$ [Abad95]. A specification $E \pm> M$ is a specification of an open system where E denotes the assumptions about the behavior of the environment and M denotes the behavior of the component. In CSL, the assumptions about the environment are at the level of actions, while in TLA, $E \pm> M$ specifications are introduced to take into account the assumptions about the environment at the level of the whole specification. This means undefined situations are dealt at the level of the whole specification, i.e. undefined situations have coarser granularity in TLA $E \pm> M$ specifications. One may suggest that TLA $E \pm> M$ specifications should be comparable with the specifications obtained by translating CSL specifications into TLA. However, $E \pm> M$ specifications require explicit synchronization between the environment and the module actions. This synchronization is achieved by sharing variables. This contrasts with the fact that CSL action synchronization is translated into action conjunction, resulting in a more abstract specification. $E \pm> M$ specifications,

because of the explicit synchronization through shared variables, have an implementation bias.

2.4. Developmental approach and Tools

The developmental approach which is used in this case study proceeds as follows. First the structure model is built. It describes the application objects and the associations between them including aggregations. The notation which is used is the one proposed in [Rama96b]. In the structure model, the different associations may have application-specific behavior and behavioral constraints as described in [Rama96b]. These allow to identify the actions of the objects participating in the associations and the interactions between these objects in terms of abstract events. The description of these interactions is achieved in the interaction model.

Once the structure and interaction models are built, we then use CSL to describe in more detail the objects, composite objects, actions of these objects, and the interactions in terms of abstract events. As noted earlier, XOMT follows closely the OMT rules. In OMT, the behavior of objects is described by means of statecharts [Hare88]. By adopting CSL as a vehicle for specifying object behavior, interactions, and compositions, we should say how the computational model of CSL compares to statecharts. CSL describes the behavior of an object by its initial conditions, the invariants, and its actions. The initial conditions represent the starting state in a statechart. An action can be interpreted as a combination of OMT events and the operations performed when the event occurs.

In OMT, events are atomic and they can be specialized. In CSL, actions are atomic and they can be specialized. However, in OMT events may denote an elapsed time. This is not actually handled in CSL, although it can be achieved by timing the action executions. The OMT event trace diagram corresponds to the traces allowed by an object. They can be derived from its CSL specification.

In CSL, states are implicitly captured by the values allowed for the local variables. The organization of states into disjoint substates or composite states

can be achieved through predicates on the values of the local variables. It can be demonstrated that a substate inherits the properties of its superstate. However, composite states can only be specified in the context of compositions. For instance, the initial conditions of a composition may imply the initial conditions of its components. The initial state of the composition is a composite state bundling the initial states of its components.

In CSL, concurrency is based on interleaving, i.e. at the level of a single object there is no concurrency. When considering a composition, the abstract events occurring in the composite are interleaved with its actions. In an abstract event, more than two components may simultaneously interact. This means, there is concurrency between the interacting components. OMT assumes that an atomic object is a finite state machine with a queue for incoming events. Composite objects may have more queues of events corresponding to each of its components. This contrasts with CSL where even if an object is a sequential system (or process), it has no queue. It accepts or blocks the execution of an action. In addition, it may display an undefined behavior or undergo non-deterministic changes by executing alone one of its actions.

OMT assumes that objects communicate by sending events. In addition, they can interact implicitly if one object has a guard condition that depends on the state of another object, such as being in a given state. Our experience with XOMT has shown that describing synchronous interactions with such a model is cumbersome. In CSL, interaction is based on the concept of abstract event [Boch93b]. Abstract events are only meaningful in the context of an object association. *This has the consequence that the structure of an application shapes its dynamic behavior.*

Apart from events, in a statechart, there are transitions. They correspond to the changes predicates defined in the semantics of actions in CSL. The so-called λ transitions in OMT correspond to the non-deterministic changes allowed by an object in CSL.

As a rule of thumb, the interaction specification is done when we may easily determine the enabling and defined states of the objects involved in each abstract event. If we can not determine the enabling and definedness of states of objects involved in the abstract events, we may not determine which abstract events do occur. After providing the CSL specifications, they are translated into TLA formulas. With the TLA specifications which are obtained, the specifier may now use different TLA tools to verify the specifications. Results of this process can be reflected directly into CSL specifications.

3. Case Study

As an application of this case study, we have selected the lift problem which is one of the problems proposed for demonstrating the adequacy of a specification method for complex systems. We omit certain details which do not contribute to the essential points of this paper.

3.1. Problem statement

A lift system is to be installed in a building with m floors. It is aimed at moving people from one floor to another. It is used under the following constraints:

1. Each lift has a set of buttons, one for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is canceled when the corresponding floor is visited by the lift.
2. Each floor has two buttons (except for ground and top floors), one to request an up-lift and one to request a down-lift. These buttons illuminate when pressed. The illumination is canceled when a lift visits the floor and is either moving in the desired direction, or has no outstanding requests.
3. When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests.
4. All requests for lifts from floors must be serviced eventually, with floors given equal priority.
5. All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel.

6. Each lift has an emergency button which, when pressed causes a warning signal to be sent to the site manager. The lift is then deemed "out of service". Each lift has a mechanism to cancel its "out of service" status.
7. The doors shall be closed when the lift moves.
8. The lift is stopped when it reaches the "out of service" state. Furthermore requests made from the lift carriage are then cleared, and no new requests from the lift carriage are accepted.

Many requirements of a lift are not mentioned, since the designer is expected to know what a lift is.

3.2. Developing the XOMT specification

3.2.1. Structure model

The structure model is about modeling the associations between objects as well as the structure of these objects in terms of other objects. We use the following rule of thumb for establishing associations between objects. We are justified in establishing an association between two objects, A and B, if and only if we want to express structural or behavioral constraints between the two objects. On the other hand, the structure of objects depend on the specifier and the level of detail that is required for the model.

We first provide an high-level structure model of a building since the user, the lift and the floors are in the context of a building. In structure models, objects are represented by rectangles and aggregation is shown by embedding one object into another. The structure model for the building can be interpreted as follows. We have a building which is a composite object consisting of Floor and Lift objects. When necessary the cardinality of the components are shown in a structure model. The object associations are shown in a structure model. Floor objects are associated between them. Each Floor is associated to the Lift through several associations.

The main object associations abstract the constrainment between the floor and the lift. When a user pushes the button at a floor, this propagates through the association "control" linking that floor to the lift. In addition, when the lift

arrives at a given floor, if it stops and opens its doors, the floor doors must also be opened. This is captured by the association "synchronized". A user may be located at a given floor or traveling through the building using the lift. These information are respectively captured by the associations "located" and "use".

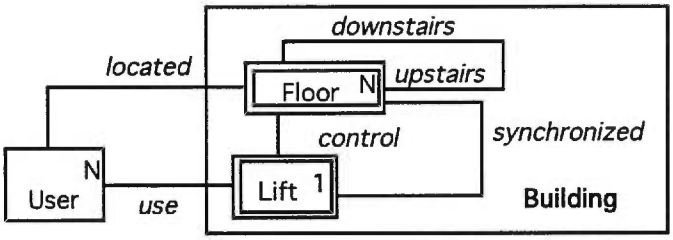


Figure 6.2. High level structure model of the lift application.

After providing this high-level view, we provide more details by describing the Lift object. As a rule of thumb, we use one structure model per composition when the composition is complex enough , or we regroup several compositions in a single structure model. In the structure model below, the Lift and its components are detailed. Please do not confuse Floor(i) with Floor objects, the former represent buttons inside the Lift while the latter is the real Floor.

In the structure model of the application, illustrated in Figure 2, the lift being a visible component of the building, it follows that the visible components of the lift are also visible at the level of the building. A lift has a ControlPanel and doors. The ControlPanel has various buttons including two buttons which control the opening and closing of the lift doors.

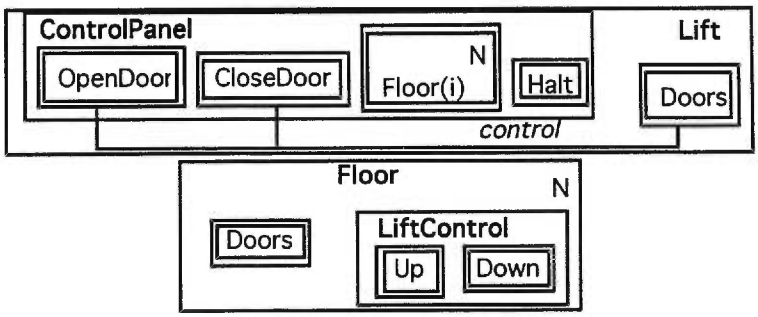


Figure 6.3. Detailed structure model of the lift and the floor.

This is followed by a detailed structure model of the Floor. Each Floor object consists of Doors and a LiftControl. The LiftControl is an abstraction for the two buttons which control the Lift at each Floor. Notice that in order to avoid too much details on this structure model, we have left out the case of the first and the last Floors which have only one button per LiftControl. However, this can be captured by making the components of the LiftControl optional.

The structure models describe one lift per building. In a multi-lifts building we may have considered an additional object responsible for the coordination of the lifts. Such an object may be called a LiftManager.

3.1.2. Interaction specification

An interaction specification portrays the interactions between the objects forming the application. These interactions are described in terms of abstract events involving the objects of the applications. These abstract events occur only in the context of object associations. The associations which embed abstract events include but are not limited to the following associations: (1) between lift doors and floor doors, (2) between up button and the lift, (3) between down button and the lift, (4) between floor(i) button and the lift, (5) between openDoor button and the lift, (6) between closeDoor button and the lift, (7) between halt button and the lift, (8) user and the lift.

The association 1 corresponds to "synchronized"; 2 and 3 correspond to "control"; 4, 5,6, and 7 correspond to the associations within the Lift composition; finally, the association 8 corresponds to "use". The abstract events are described based on the requirements and assumptions below:

Lift and the doors

- When the lift arrives at a given floor where the button is pressed and the lift is going toward the direction indicated by the button, this makes the lift stop at this floor and open its doors.
- When the lift arrives at a given floor which was a target direction, this makes the lift stop and open its doors.
- When the lift is stopped and the button *open door* is pressed, this makes the lift to open its doors.

- When the lift doors are opened (respectively closed), the corresponding floor doors are also opened (respectively closed).

Up and down buttons

- When the lift arrives at a given floor which was a target direction, and the button up or down is pressed, these buttons should be deactivated. If both are pressed, non-deterministic choice is applied to deactivate only one button.
- When the lift arrives at a given floor which is in the direction of the target floor, and the button leading to the target floor is pressed, this makes the button to be deactivated.

Control panel

- When *floor(i)* button is pressed, this causes the indicated floor to be a target destination.
- When the button *open* (*close*) *door* is pressed if the lift is stopped, this makes the lift open (close) its doors.
- When the button halt is pressed, this leads the lift to the nearest floor and stops the lift at that floor canceling all the target destinations.

To make the description of the behavior of the lift tractable, we have decomposed its movement into two specific discrete steps, namely move(i,j) and stop(k). move(i,j) models the displacement from the floor i to the floor j. It is achieved in one complete execution step and it is atomic. stop(k) models the action forcing the lift to stay at the floor k.

In the following, we describe a simplified Lift system which can be later refined to include all the requirements. It includes a button, the lift, the user and the doors specifications. A button is described by the following CSL specification :

```
spec Button(direction:Direction, floor:FloorNumber)
 define ButtonState = {idle,on}
 define Direction = {none, up, down}
  local variables
      state : ButtonState; serviced : Boolean
  initial conditions
      state = idle ∧ serviced = false
  behavior
```

```
action push =      enabled: state = idle
                   defined: true
                   changes: state' = on
action call =      enabled:  state = on
                   defined: true
                   changes: true
action serviced =  enabled:  state = on
                   defined: (serviced=false)
                   changes: (serviced'= true)
action release = enabled:  (state = on) ∧ (serviced = true)
                   defined: true
                   changes:  (state' = idle) ∧(serviced'= false)
end spec Button
```

The specification is parameterized to allow the generic specification of buttons according to the direction and the floor which are serviced by the button. The actions of the button can be described informally as follows. The *push* action consists to press the button. *call* is an action by which the button communicates with the lift. *serviced* is an action by which the lift notifies the button that it has stopped at the corresponding floor.

Based on the semantics of CSL actions introduced in Section 2, there are many specification variants for an action. For instance consider the *push* action. It can be specified as:

```
action push = enabled: true
              defined: true
              changes:  state' = on
```

The action is always enabled and it is never results in an undefined behavior.

```
action push = enabled: state = idle
              defined: true
              changes: state' = on
```

The action is enabled in states where the button is idle and it is always well-defined. In other words, you may call the action only when the button is idle.

```
action push = enabled: true
              defined: state = idle
              changes: state' = on
```

The action is always enabled. It results in undefined behavior in states where the button is on.

```
action push = enabled: state = idle
              defined: state = idle
              changes: state' = on
```

The action blocks in states where the button is on and it never results in undefined behavior.

It is up to the specifier to select the intended behavior. Actually, the first specification of the action push is the one which corresponds to the reality since we are allowed to push the button independently of its state.

In the requirements, we have stated that buttons are illuminated. This is specified by having a light in each button. The light object is described below.

```
spec Light
 local variables
      lit : Boolean
 initial conditions
      lit = false
 behavior
      action illuminate =   enabled: lit = false
                            defined: true
                            changes: lit' = true
      action deilluminate =       enabled: lit = true
                            defined: true
                            changes: lit' = false
end spec Light
```

The specification of a button which has a light is as follows :

```
spec ButtonWithLight
 extends Button
 contains
      light : Light
 invariants
      light.lit = (state = "on")
end spec ButtonWithLight
```

The invariant has to be satified by any implementation of this specification. In fact, there should be a relation between the implementation of the invariant and the two actions provided by the specification Light. In the remaining of this paper, we no longer describe the inner-workings of actions in terms of **enabled, defined** and **changes** predicates.

We assume the operators : *first_floor()* returning the first floor of the building, *last_floor()* returning the last floor of the building, and *next_floor(direction, f,*

n) returning the floor following a given floor based on the direction. Direction can be none, up, and down.

```
spec Floor(id:FloorNumber)
 contains
      up_button : ButtonWithLight(up, id)
      down_button : ButtonWithLight(down, id)
      doors : Doors
 end spec Floor
```

```
spec Doors
define DoorState = {closed, open}
 local variables
      state : DoorState
 initial conditions
      state = closed
 behavior
      action open
      action close
 end spec Doors
```

```
spec Lift
 contains
      doors : Doors
 local variables
      state : LiftStates, location : FloorNumber;
      direction : Direction, up_req : Requests; down_req : Requests
 initial conditions
      state = stopped ∧l ocation = first_floor()
      direction = up ∧ up_req = ∅ ∧ down_req = ∅
 behavior
      action open_door
      action close_door
      action select_destination(f)
      action out_of_service
      action called(f,d)
      action service(f,d)
      action move
      action atfloor
 end spec Lift
```

```
spec LiftSystem
   contains
      lift : Lift
      user : User
      floors : n : i..j Floor(n)
   abstract events
```

```
association control
        event callup =      actions: floors[n].up_button.call
                            lift.called(f, d)
                            constraint: (f = floors[n].up_button.floor) ∧
                            (d = floors[n].up_button.direction)
        event calldown = actions: floors[n].down_button.call
                                  lift.called(f, d)
                            constraint: (f = floors[n].down_button.floor) ∧
                            (d = floors[n].down_button.direction)
    ... the other abstract events are defined similarly
end spec LiftSystem
```

3.3. Developing the formal specification

In the following, we only portrays the translation of Button and Light specifications.

3.3.1. TLA Specifications

In the TLA specifications below, each CSL specification is mapped into a module. An *extends* clause corresponds to the TLA extends clause. A *contains* clause is mapped into instantiation of the modules corresponding to the components. Local variables are introduced as parameters of the module. At this point, the translation becomes one to one since the initial conditions clause is translated into an Init predicate, and the actions are translated based on the approach sketched in Section 2. They are included in the temporal section of the module. For the CSL specification Light, V = {lit}, LV(illuminate) = {lit}, LV(deilluminate) = {lit}, PAR(illuminate) = ∅, and PAR(deilluminate) = ∅. This leads to the following translation :

```
                    Module Light
parameters
        lit : Variable
predicates
        Init ≡ lit = false
actions
        illuminate ≡ ∧ (lit = false) ∧ (true ⇒ lit' = true)
        deilluminate ≡ ∧(lit = false)∧ (true ⇒ lit' = true)
temporal
        Actions ≡ illuminate ∨ deilluminate
        Behavior ≡ Init ∧ ❑[Actions]<bulb>
```

In the translation of the CSL specification Button, we have taken care of adding extra predicates state ∈ ButtonState and floor ∈ FloorNumber in order to reflect the typing of the local variables in the TLA specification.

Module Button

parameters

direction : Variable
floor : Variable
state : Variable
serviced : Variable
calling: Variable

predicates

state ∈ ButtonState ∧ floor ∈ FloorNumber
Init ≡ state = "idle" ∧ serviced = false ∧ calling = false

actions

push ≡ ∧ (state = "idle")
∧ (true ⇒ ((state' = "on") ∧ (calling' = true)))
∧ Unchanged(serviced, floor, direction)

call ≡ ∧ (state = "on") ∧ (calling = true)
∧ (true ⇒ ((d = direction) ∧ (calling' = false)))
∧ Unchanged(serviced, floor, direction)

serviced ≡ ∧ (state = "on")
∧ ((serviced = false) ⇒ (serviced'= true))
∧Unchanged(state,floor, direction))

release ≡ ∧ (state = "on" ∧ serviced = true))
∧ ((true ⇒ ((state' = "idle") ∧(serviced'= false)))
∧Unchanged(floor, direction))

temporal

Actions ≡ push ∨ call ∨ serviced ∨ release
Behavior ≡ Init ∧ ❑[Actions]<state,serviced, calling>

An abstract event is translated using conjunction of actions. For instance, the abstract event *callup* in the specification LiftSystem is translated into the following TLA action :

callup ≡ ∃ f: FloorNumber, d: Direction , n ∈ {i, i+1, .., j}:
∧ floors[n].up_button.call
∧ lift.called(f, d)
∧ (∧ (f = floors[n].up_button.floor)
∧ (d = floors[n].up_button.direction))

Based on the TLA specifications, various properties of the lift system may be verified.

3.3.2. Proving various properties

For the lift system, deadlock freedom can be interpreted as the conjunction of two safety properties:

a) In the initial state at least one abstract event (represented by a TLA action) is enabled.

b) Each abstract event leads the objects into a state where at least one abstract event is enabled.

In order to prove this property, we use the TLA INV1 rule shown below [Lamp94].

$$\frac{I \wedge [N]_f \Rightarrow I'}{I \wedge \Box[N]_f \Rightarrow \Box I}$$

This rule is used to prove that a program satisfies an invariance property $\Box I$. The hypothesis asserts that a $[N]_f$ step cannot falsify I. The conclusion asserts the if I is true initially and every step is a $[N]_f$ step, then I is always true.

In our case, I represents the disjunction of the enabling conditions of the actions provided by the program. It suffices to choose the invariant property I as at least one abstract event is enabled to prove the deadlock freedom property using the above rule. For the LiftSystem, this can be expressed as the following TLA formula

Enabled callup $\vee$ *Enabled* calldown

Notice that (*Enabled* callup) can be reduced to the conjunction of *Enabled* floors[n].up_button.call, *Enabled* lift.called(f,d), and (f = floors[n].up_button.floor) $\wedge$ (d = floors[n].up_button.direction)) by predicate logic since an abstract event is enabled when all its constituant actions are enabled and its constraint is true.

Liveness properties such as "when the button of a floor is pressed the lift will eventually stop at this floor" are proved using TLA's WF1 or SF1 rules described in [Lamp94]. To ease the proof of this property, we decompose the property into two properties denoted by a) and b) below. The WF1 rule is used to prove properties of the form P *leads-to* Q from a weak fairness condition $WF_f(A)$. The reader is referred to [Lamp94] for more details on weak fairness conditions for TLA actions. Here A denotes a specific action. An A step is understood as the execution of the action A. It can be applied when an A step that starts with P true makes Q true. The WF1 rule is as follows:

$P \wedge [N]_f \Rightarrow (P' \vee Q')$

$P \wedge \langle N \wedge A \rangle_f \Rightarrow Q'$

$P \Rightarrow Enabled \langle A \rangle_f$

$\Box[N]_f \wedge WF_f(A) \Rightarrow P$ *leads-to* Q

The two liveness properties a) and b) are proven using the following assumptions:

a) When we press the button of a floor then the floor number will eventually be listed in the request list of the lift: This property, i.e. $\exists$ f : Floor : P *leads-to* Q, is proven with TLA WF1 by taking P as

(floors[n].up_button.state = "on" $\wedge$ f = floors[n].up_button.floor)

$\vee$ (floors[n].down_button.state = "on" $\wedge$ f = floors[n].down_button.floor)

Q as

f $\in$ lift.up_req $\vee$ f $\in$ lift.down_req

and A as the abstract event callup or calldown, respectively, where:

callup $\equiv$ $\exists$ f: FloorNumber, d: Direction , n $\in$ {i, i+1, .., j}:

$\wedge$ floors[n].up_button.call

$\wedge$ lift.called(f, d)

$\wedge$ ($\wedge$ (f = floors[n].up_button.floor)

$\wedge$ (d = floors[n].up_button.direction))

calldown ≡ ∃ f: FloorNumber, d: Direction , n ∈ {i, i+1, .., j}:
∧ floors[n].down_button.call
∧ lift.called(f, d)
∧ (∧ (f = floors[n].down_button.floor)
∧ (d = floors[n].down_button.direction))

$WF_f(A)$ is deduced from the fairness of floors[n].up_button.call, floors[n].down_button.call, and lift.call actions. In the LiftSystem specification, by default the TLA actions are supposed to have weak fairness.
b) If the floor number is listed in one of the request lists of the lift then the lift will stop at the corresponding floor: This property is internal to the lift and it can be deduced from the sequence of states of the lift module, i.e. its behavior. It can be also broken down into smaller properties.

Proofs are tedious and complicated. A great deal of these proofs can be mechanical and take advantage of the structure of the formulas to decompose the proofs. There are many tools which can assist in proving the properties of TLA specifications. At the University of Dortmund, a certain number of tools for developing, preparing, building, testing and verifying TLA specifications have been prototyped [Merst94]. Among these tools, we selected eTLA+ which is an interpreter allowing the interpretation of specifications combined with graphical visualization of their execution. Once the CSL specifications are translated into TLA, TLA specifications can then be translated into eTLA+ which is used as input for the eTLA+ interpreter. The eTLA+ interpreter allows symbolic debugging including stepwise or continuous execution and tracing. Non-determinism is handled by allowing the user to select the next action to execute or if he may prefer a scheduling strategy. CSL specifications can also be translated in TLALight, another variant of TLA [Merst94]. TLALight specifications are implemented using a C++ translator which derives a distributed implementation prototype as a set of communicating processes in a workstation network.

4. Conclusion

In this paper, we have modified the developmental approach used in XOMT in order to integrate a new approach to the description of composite object behaviors. XOMT has now two distinct views which portray the structure and the dynamic behavior of applications. The developmental approach is compositional since we can connect different pieces of the design by establishing associations between objects of these pieces. In addition, the approach can be used for component-based systems.

Throughout this case study, the structure of composite objects serves for describing the dynamic behavior of these objects. There is an implicit relation between the CSL formal specifications and the XOMT structure specifications (structure models in XOMT). The characteristics of component associations are such that their presence simplifies proofs of behavioral properties, as compared to general associations. In addition, we can derive useful properties of the composition based on that of its components. For instance, specification invariants of the LiftSystem imply properties of its components. This is due to the encapsulation of the local variables in the components. Once the Deadlock freedom of components proved individually, there is no need to prove this again when the components are incorporated in the LiftSystem specification, i.e. the proof of deadlock freedom for the LiftSystem is limited to the actions and the abstract events defined in that specification. Furthermore, based on the interconnections between components, we may prove progress properties for the LiftSystem. This is illustrated with the proof of the property "when we press the button of a floor then the floor number will eventually be listed in the request list of the lift".

In addition, the approach proposed in this paper is based on the linkage between the structure and the behavior of composite objects. Such an approach was shown to be adequate when dealing with complex systems since the complexity of systems is concentrated on the interactions between components of such a system. CSL specifications can be translated into TLA, for which tools can be used for verification purposes as well as for checking the composition of different pieces of the design.

In this paper, we sketched how different tools can be combined for supporting the XOMT developmental approach. In order to support this development process, the adaptation of a case tool, namely MetaEdit tool [Meta95a] is underway. The MetaEdit tool supports various object notations and methods, in addition it is parameterizable in the sense that you may define your own notation and method by defining a meta model of the notation. The MetaEdit tool views an object-oriented method as a set of notations and specifications constructed using these notations. We use this latter feature for defining a meta model for XOMT.

Future development of this work includes the design and the implementation of a tool supporting CSL with an integrated environment for verifying the specifications. This may require the design and implementation of an automatic translator of CSL specifications to TLA specifications.

Chapitre 7

Contributions de cette thèse

A travers ce chapitre, nous examinons notre contribution à la composition des objets et en particulier notre apport au développement orienté objet. Nous faisons ressortir le mérite de chacune des idées principales des articles présentés dans cette thèse.

Nous commençons par revoir le cadre conceptuel et son impact sur le développement orienté objet. Ensuite, le langage CSL et son intégration dans le développement orienté objet sont revus. Sa théorie de composition est comparée avec d'autres approches formelles de composition. Le chapitre se termine par les différents problèmes reliés à la composition des objets qui n'ont pu être abordés dans cette thèse.

7.1. Le cadre conceptuel

D'emblée, il faut souligner que notre principale contribution est un cadre conceptuel pour la composition des objets. Ce cadre propose une extension du paradigme objet où la notion d'objet composé est basée sur l'existence d'une structure au sein de l'objet. Cette structure est faite des composants et des interconnexions de ces composants, y compris leurs interactions.

Cette extension du paradigme objet souligne le lien entre la structure et le comportement d'un objet composé. Le principe consistant à établir un lien direct entre la structure et le comportement des objets composés constitue une différence fondamentale entre le cadre conceptuel et les approches décrites dans la littérature pour la composition des objets. Lorsque ce principe est mis en oeuvre dans une méthode de développement orienté objet, il permet de relier la description d'un niveau d'abstraction donné à celle d'un autre.

Le cadre conceptuel est abstrait. Lorsqu'est assumé un modèle objet, les concepts et principes sous-jacents à ce cadre peuvent être interprétés sans aucune difficulté. Nous en avons fait la preuve avec la méthode XOMT (voir chapitres 3 et 6).

Le cadre conceptuel n'identifie pas une étape particulière du développement orienté objet où il serait applicable: il est applicable à chacune des étapes. Cette applicabilité découle du fait que le cadre conceptuel repose sur une extension du paradigme objet et non celle d'une approche de développement ou encore d'une notation orienté objet.

Le cadre conceptuel avait pour objectif initial de réconcilier les différentes approches de composition. De ce fait une grande similarité existe entre ce cadre et différentes approches de composition. Il organise l'existant dans un tout cohérent. Pour ce faire, il se doit d'être le plus abstrait possible mais sans nuire à son applicabilité.

7.2. Impact sur le développement orienté objet

Pour les fins de la discussion, nous nous restreindrons à un processus de développement orienté objet comprenant l'analyse et la conception seulement. Nous n'abordons pas la phase d'implantation ainsi que le problème de la réutilisation des spécifications d'objet composés, parce que c'est une problématique qui englobe les techniques de raffinement des spécifications des objets composés, leur documentation et leur test.

7.2.1. Analyse

Au niveau de l'analyse, le cadre conceptuel permet de clarifier ce qui distingue une analyse incluant la composition objets d'une analyse telle que nous la connaissons. En d'autres mots, la prise en compte de la composition exige-t-elle de nouveaux concepts, principes, et heuristiques qui supportent la démarche?

A priori, une fois le modèle objet étendu pour inclure la notion de structure d'un objet et la possibilité d'établir le lien entre la structure et le comportement des objets, l'analyse peut procéder comme d'habitude. La différence se situe au niveau de la qualité de la spécification obtenue. Celle-ci est plus claire : elle reflète les différentes compositions qui se trouvent dans le domaine du problème. En outre, elle peut maintenant être structurée en termes de différentes compositions qui se trouvent dans le domaine du problème. Chaque niveau décrivant une composition, les liens entre les différentes compositions se retrouvent au niveau où ces compositions sont représentées comme de simples objets.

D'un autre côté, la structuration de la spécification facilite sa validation: à chaque niveau, seuls les détails essentiels sont considérés. Par la même occasion, cela permet de gérer la complexité des applications. Il faut néanmoins reconnaître que ce ne sont pas tous les domaines d'applications qui permettent de tirer profit de ces avantages. Le domaine de télécommunications, plus particulièrement celui de la gestion des réseaux, se prête bien à une telle analyse, vu l'organisation hiérarchique des réseaux de télécommunications. C'est aussi le cas des entreprises, du contrôle des procédés industriels, et de celui des logiciels dits middleware, lesquels sont souvent organisés en couches, par exemple, les systèmes de gestion des bases de données, les systèmes d'exploitation, les systèmes de gestion de configuration, etc.

7.2.2. Conception

Au niveau de la conception, les décisions qui sont affectées par la composition des objets sont guidées par le cadre conceptuel. Une bonne partie de ces décisions de conception sont répétitives: les concepteurs ont tendance à vouloir reproduire les mêmes décisions en fonction de leur expérience, même lorsque la situation ne s'y prête pas. Du fait que le cadre conceptuel dans sa mise en oeuvre permette de documenter ces décisions, cela facilite leur enregistrement à l'aide de design patterns. Elles peuvent alors être conservées dans un répertoire pour utilisation future.

La qualité d'une conception se reconnaît par la facilité avec laquelle elle peut être modifiée et implémentée. Notre expérience a établi que les conceptions les plus propices au changement sont celles où les compositions sont identifiées de manière explicite et où le lien entre la structure et le comportement des objets composés est indiqué. De plus, leur implémentation est facilitée. Elle se fait en différentes étapes allant de la composition initiale aux compositions détaillées.

7.2.3. Comparaison entre XOMT et UML

Quoique les premiers travaux sur la méthode XOMT soient antérieurs à UML, nous comparons XOMT avec UML au niveau du traitement de la composition. Il convient de souligner que l'un des objectifs principaux de UML est de regrouper les notations orientées objet en utilisant leur complémentarité. UML est considéré comme étant le sucesseur des méthodes orientées objet OODA, OMT et OOSE. UML permet la particularisation de ses concepts en fonction de l'application à modéliser.

Au niveau de la composition, UML correspond à OMT. Dans UML, la relation d'agrégation n'est qu'une association comme toutes les autres. Dans UML, il y a deux formes de relations d'agrégation: l'agrégation et la composition. L'agrégation correspond à une inclusion dans un ensemble. Les composants peuvent être partagés entre différents agrégats. La composition correspond au fait que les composants sont contenus dans le composé et qu'ils ne peuvent être partagés entre les composés. Dans une composition, il y a propagation des contraintes structurelles et comportementales. Un objet composé est une composition. En plus de cela, nous pouvons définir des associations et des interactions qui ont lieu uniquement dans le contexte de la composition. Les associations peuvent être décrites à l'aide de collaborations et les interactions à l'aide de graphes d'interactions. Une collaboration exprime comment différents objets interagissent l'un et l'autre du point de vue structurel. Une interaction ne peut être définie que dans le contexte d'une collaboration.

De ce qui précède, nous pouvons en déduire que la différence entre UML et XOMT est à trois niveaux:

a) la raison d'être des objets composés dans les deux méthodes sont différentes. Pour UML, les objets composés ne sont qu'un artifice de modélisation qui sert à abstraire un groupe d'objets. XOMT reconnaît l'existence d'objets composés dans le domaine du problème et dans celui de sa solution.
b) les propriétés qui caractérisent les objets composés. Dans UML, le comportement d'un objet composé est comparable à celui d'un objet non-composé. Dans XOMT, le comportement d'un objet composé est la composition de deux comportements. Les associations impliquant les objets composés n'ont pas de sémantique particulière dans UML alors que, dans XOMT, il faut distinguer trois types d'associations lorsqu'il s'agit d'un objet composé. A ce niveau, la richesse sémantique du modèle objet supporté par XOMT permet de tenir compte de contraintes structurelles et comportementales plus complexes que celles qui sont permises par UML.
c) le lien entre la structure et le comportement des objets composés. Ici, le fait que XOMT tienne compte de l'impact de la structure de l'objet sur la définition de son comportement permet d'établir des règles de validation du passage d'un niveau d'abstraction à un autre, en particulier lorsqu'un objet est raffiné en un groupe d'objets.

7.3. Le langage CSL

Le langage CSL est une notation formelle orienté objet qui a été conçue pour refléter les concepts et principes du cadre conceptuel. Pourquoi une nouvelle notation formelle? Nous verrons qu'il y a une différence entre l'approche de composition et de raisonnement proposée dans CSL et celles que nous retrouvons dans la littérature. Notre comparaison n'est pas exhaustive. Nous avons sélectionné les plus importantes approches de composition.

Fiadeiro-Maibaum (Théorie des catégories)

Leurs travaux [Fiad92] s'inspirent de Goguen [Gogu86] sur l'utilisation de la théorie des catégories dans la spécification des modules. Dans le chapitre 4, nous avons montré comment les concepts et les principes du cadre conceptuel peuvent être mis en oeuvre dans leur approche. A titre de rappel, les

composants sont identifiés et décrits par le biais des descriptions d'objets (théories). Les interconnexions et interactions des composants sont décrites par l'introduction de théories d'interconnexion et de morphismes reliant ces théories à celles des composants. La notion de colimite aide à construire la composition. La notion de réduit (reduct) indique quelles sont les propriétés inhérentes. Pour chaque agrégat de propriétés, il faut introduire une théorie qui effectue la composition des propriétés et autant de théories d'interconnexion et de morphismes. Pour les propriétés émergentes, il faut introduire une nouvelle théorie construite à partir de la composition et un morphisme établissant ce lien. Une telle spécification permet de raisonner sur:

- chacun des composants individuellement;
- les propriétés induites par l'interaction des composants;
- la contribution de chaque composant dans la composition par la notion de réduit;
- les agrégats de propriétés.

L'émergence étant séparée des composants, le raisonnement sur ces propriétés ne tient pas compte des composants.

Cette approche couvre tous les aspects du cadre conceptuel. Mais, elle a certains désavantages :

- une nouvelle théorie et des morphismes sont nécessaires pour interconnecter des objets[5].
- la composition est similaire à l'héritage multiple. Cela revient à réduire toutes les formes de composition à l'héritage multiple.
- il n'existe pas encore d'outils servant à supporter cette démarche. Le langage TROLL [Hart92] est dérivé des premières recherches dans ce domaine. Mais ce langage ne dispose pas d'opérateurs de description de morphismes, colimite et réduits.
- l'utilisation de la théorie des catégories donne à l'approche une saveur mathématique loin des techniques de spécification formelle usuelles.

---

[5]Cela revient à traiter les associations comme des objets.

Abadi-Lamport (Principe de Composition)

Cette approche a déjà été présentée dans la revue de l'état de l'art sur la composition des objets. Dans le chapitre 4 se trouve une comparaison illustrée à travers une spécification formelle. Dans ce chapitre, nous mettons l'emphase sur la théorie de composition et faisons ressortir les éléments de comparaison avec CSL.

Il convient de souligner que l'ensemble du raisonnement induit par les théorèmes proposés par Abadi et Lamport reposent sur le fait que chaque raffinement doit au minimum se comporter comme le système original lorsqu'il est placé dans l'environnement de celui-ci. Le reste de la preuve est ardue parce qu'il faut tenir compte de la concurrence, du fait que les systèmes réactifs ont des propriétés de sûreté et de vivacité (progrès) et du fait qu'il faut spécifier l'environnement dans lequel le composant est sensé fonctionner correctement.

Du point de vue de l'utilisation de cette approche dans le développement orienté objet, comme elle n'est pas orienté objet, nous avons défini une technique de représentation des spécifications orienté objet. Dans cette approche, la notion d'environnement joue un rôle prépondérant dans la spécification. Une application modélisée à l'aide de l'approche objet est faite de différents objets qui interagissent. L'application dispose d'un environnement et chaque objet pris individuellement dispose aussi d'un environnement. Pour l'application, l'environnement est fait d'un ensemble d'objet qui ne font pas partie de l'application mais qui interagissent avec les objets de l'application. Au niveau d'un objet, son environnement est fait de l'ensemble des objets avec lesquels il interagit.

A partir de la spécification d'une application, nous pouvons produire une spécification TLA en tirant profit des théorèmes de composition et de décomposition proposés par Abadi et Lamport. Nous utilisons la correspondance suivante: l'application est considérée comme étant un objet composé. Un objet est représenté par une spécification de type hypothèses-garanties. Chaque objet est représenté par son comportement et les hypothèses qu'il assume au sujet de son environnement.

L'interaction des objets est réalisée par le mécanisme de variables partagées dans le sens que l'objet est en mesure de lire les attributs des objets qui forment son environnement. Ces attributs déterminent l'exécution des actions dudit objet. Ce mode d'interaction suppose que la communication entre objets est faite par la lecture des valeurs des attributs des autres objets. Les actions de l'environnement sont entrelacées avec celles de l'objet. Il n'y a pas de synchronisation entre les deux.

Une association entre objets est représentée par la lecture (dépendance) des valeurs des attributs des objets. Les contraintes entre les objets sont de la forme: si l'objet x est dans l'état s_i alors l'objet y exécute l'action a_y. Ainsi, quand nous avons deux spécifications de type hypothèses-garanties correspondant aux objets i et j, composer ces objets revient à composer des spécifications de type hypothèses-garanties. Cela se fait conformément au théorème de la composition proposé par Abadi et Lamport. Ce théorème repose sur le fait que la conjonction des hypothèses de l'objet i avec les garanties de l'objet j est satisfaite. En général, si nous considérons deux objets i et j représentés par des spécifications de type hypothèses-garanties:

(a) i et j ne sont pas composables lorsque les hypothèses de i (ou de j) sont conflictuelles avec les garanties offertes par j (ou par i).
(b) lorsqu'il y a absence d'interactions entre i et j, c'est-à-dire indépendance des spécifications, les objets i et j sont composables.
(c) lorsque les hypothèses de i sont satisfaites par les garanties offertes par j et que les hypothèses de j sont aussi satisfaites par les garanties offertes par i, les objets i et j sont composables.

A bien d'égards, l'approche proposée par Abadi et Lamport est semblable à CSL. Avec leur approche, nous spécifions des applications selon le paradigme objet au détriment de l'encapsulation des objets. Si par contre, nous voulons préserver l'encapsulation des objets, l'approche de type CSL est préconisée. Avec CSL, les objets n'ont pas accès aux attributs des autres objets. De plus, ils communiquent par synchronisation de leurs actions via les paramètres des

actions synchronisées. CSL est une autre façon d'interpréter, conformément au paradigme objet, les mêmes principes que ceux énoncés par Abadi et Lamport. Quand il est interdit à un objet de lire les valeurs des attributs des autres objets et qu'en plus les interactions des objets sont restreintes aux associations entre les objets, la communication entre eux se fait par synchronisation des actions et par échange ou entente sur la valeur des paramètres des actions synchronisées.

Lam-Shankar (Théorie des interfaces)

La théorie des interfaces proposée par Lam et Shankar adopte une autre approche pour la composition [Lam94]. Elle a aussi été présentée dans la revue de l'état de l'art. Dans le chapitre 4, vous retrouvez une mise en relation avec CSL.

Lam et Shankar laissent entendre que l'approche est extensible aux objets. Nous allons nous prêter à un tel exercice dans ce qui suit. Un module représente la spécification d'un type. Il est instanciable en différents objets. Nous considérons uniquement les actions supportées par un type en assumant que les attributs sont représentables par des actions retournant la valeur des attributs. Les notions de sts(M)[6] et AllowedBehaviors(M)[7] correspondent au comportement de l'objet. Une interface représente une association entre les objets. Elle supporte deux rôles : fournisseur et utilisateur. Chaque rôle est défini par les événements placés sous sa responsabilité: Inputs(I)[8] et Outputs(I)[9]. Les séquences permises d'événements correspondent au comportement de l'association [Rama97]. Dire qu'un objet offre une interface I signifie que l'objet est un participant de l'association I et qu'il y joue le rôle de fournisseur. Dire qu'un objet offre une interface I en utilisant une interface L revient à dire que l'objet participe à deux associations I et L où il joue respectivement les rôles de fournisseur et d'utilisateur. Composer des objets équivaut à les encapsuler dans une composition tel qu'illustré ci-après:

---

[6]sts(M) définit les transitions possibles pour le module M.
[7]AllowedBehaviors(M) est un ensemble de comportements.
[8]Inputs(I) est l'ensemble des événements d'entrée pour l'interface I.
[9]Outputs(I) représente l'ensemble des événements de sortie pour l'interface I.

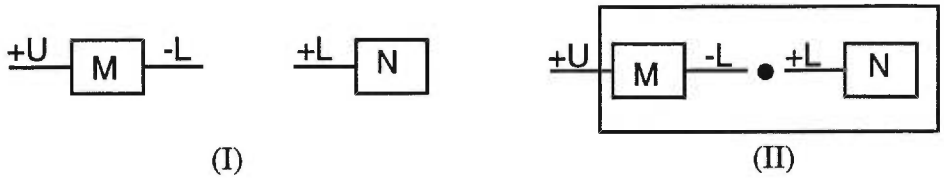


Figure 7.1: Composition des objets par association

Dans cette figure, (I) représente la situation où l'objet M offre l'interface U en utilisant l'interface L et l'objet de type N offre l'interface L. En (II), les deux objets sont composés pour donner un objet de type MN offrant l'interface U.

La relation d'implémentation des modules est similaire à l'héritage ou sous-typage. Elle permet de raffiner des types séparément pour les recomposer par après. Avec cette interprétation orienté objet de l'approche de Lam et Shankar, si nous voulons utiliser cette démarche pour représenter les compositions selon le cadre conceptuel, la structure de la composition est décrite par des interfaces. La notion d'offre d'une interface, étant conservée dans la composition, permet d'utiliser celle-ci pour parler des propriétés inhérentes. Pour les agrégats de propriétés, il est utile d'introduire un objet qui est en charge de la composition des propriétés. Quant aux propriétés émergentes, elles sont aussi introduites par un nouvel objet, car la relation d'implémentation n'introduit pas de nouveaux événements. Le modèle ne permet que le raisonnement sur une interface à la fois. Lorsque cette interface résulte de la composition de plusieurs interfaces disjointes, alors nous pouvons raisonner sur une composition. Il ne fait aucun doute que nous pouvons appliquer le cadre conceptuel dans cette approche de composition. Mais comme son domaine d'application est limité aux systèmes en couches, cette approche est limitée par rapport à CSL.

7.4. Articles non-repris dans cette thèse

7.4.1. Relation avec les systèmes distribués

Nous nous sommes aussi penchés sur la relation entre les objets composés et les systèmes distribués. Cela est décrit dans [Rama96a] qui illustre l'utilisation du modèle de référence ODP pour décrire les objets composés conformément

au cadre conceptuel. L'utilisation du modèle de référence à cette fin est permise par l'analogie entre les systèmes distribués et les objets composés. Les descriptions d'objets composés sont faites de descriptions d'objets ODP auxquelles sont adjoints des contrats. Ceux-ci relient la structure et le comportement de ces objets. Ces contrats ont pour participants les composants et l'objet composé.

Techniquement, cette interprétation des objets composés en termes de concepts du modèle de référence ODP (RM-ODP) est valide. Elle aide à décrire la structure, les propriétés inhérentes et les agrégats de propriétés d'un objet composé. Cependant, nous nous interrogeons sur son adéquation conceptuelle. En effet, cette interprétation repose sur l'utilisation de la notion d'interconnexion des objets pour représenter l'agrégation (composition). Ce faisant, la distinction entre les deux devient floue. Cette remarque s'applique aussi au langage de spécification de systèmes distribués *Darwin*. Pour *Darwin*, au lieu d'utiliser des contrats, la modélisation de l'agrégation est faite à l'aide de la notion de binding. Un binding représente une interconnexion d'objets.

Ces travaux nous ont aussi permis d'éprouver le cadre conceptuel dans la spécification des systèmes distribués. Ils montrent comment nous pouvons utiliser la notion d'interconnexion des objets pour représenter la composition des objets. Plusieurs auteurs conviennent de la difficulté d'établir une distinction nette entre la composition et l'interconnexion des objets. Dans RM-ODP et Darwin, aucune distinction nette n'est dressée entre la composition et l'interconnexion des objets.

Une telle distinction est nécessaire, car l'interconnexion des objets est reliée à la manière par laquelle nous désirons faire interagir les objets. Tandis que la composition est une question d'abstraction et de structure hiérarchique des objets. Réduire la composition à l'interconnexion des objets équivaut à ignorer l'abstraction et à restreindre la structure hiérarchique en une association entre l'objet composé et un de ses composants à la fois. Dans une hiérarchie, nous avons la notion d'encapsulation et de localité, ce qui n'est pas considéré dans l'interconnexion.

7.4.2. Spécification formelle des associations

L'un des objectifs de cette recherche était d'examiner les méthodes de vérification des propriétés multi-objets. Cet objectif n'a pu être atteint qu'à travers un traitement formel de la composition, traitement rendu possible par la formalisation du cadre conceptuel.

Pour décrire formellement une composition, il faut avant tout pouvoir parler d'objets ayant une structure, d'où le problème de la description formelle de telles structures. Nous nous sommes intéressés à l'adéquation de la notion d'association entre objets telle que présentée dans les méthodes de développement orienté objet pour décrire les interconnexions et les interactions entre les objets. L'article [Rama97] tente d'apporter une réponse à cette question. Il déplore la multiplicité des interprétations du concept d'association. Il propose un cadre conceptuel pour les associations et fournit quatre approches de formalisation des associations décrites à l'aide d'Object-Z. Il y est aussi démontré que ces approches peuvent être combinées. Néanmoins, il incombe au modélisateur de choisir l'approche adéquate pour représenter les associations de son application. Cela dépend aussi de l'application modélisée. Dans la plupart des cas, la combinaison de l'approche voulant que chaque instance de l'association soit un objet, à celle représentant toute l'association par un objet permet de capturer le maximum des propriétés sémantiques des associations.

La structure d'un objet composé ne peut être proprement décrite sans que cela ne soit fait pour les interconnexions et les interactions des composants. Parmi les qualités d'une bonne description, nous avons la fidélité, la précision, la rigueur et le caractère intuitif. Les interconnexions et les interactions des composants sont descriptibles par les associations de composants. Ainsi, en proposant des approches de spécification des associations, nous permettons par la même occasion une meilleure description de la structure des objets composés, structure qui est adéquatement représentée par un ensemble d'associations. L'agrégation est représentable comme une forme particulière d'association.

Étant donné que les spécifications faites sont formelles, il convient de se pencher sur la possibilité de raisonnement qu'elles offrent. Elles sont reliées à la technique de spécification formelle. Avec Object-Z, nous sommes en mesure de combiner des schémas en utilisant le calcul des schémas ou encore de décharger des obligations de preuves de raffinements, voire même de raisonner sur les propriétés invariantes des spécifications. Un des aspects qui aurait dû être considéré dans l'article [Rama97] est le lien entre les interconnexions et les interactions des objets. Object-Z est adéquat pour spécifier des propriétés structurelles. Mais au niveau des interactions d'objets, le langage ne précise pas comment elles se produisent, ni comment les opérations sont invoquées, ni ce que représente le comportement d'un objet. Cela nous a amené à définir le langage CSL.

7.5. Autres applications du cadre conceptuel

La nouvelle façon de voir la composition des objets telle que proposée par le cadre conceptuel a un impact sur le développement orienté objet actuel. En effet, la méthode XOMT et la notation formelle CSL permettent de résoudre les problèmes occasionnés par l'inadéquation du paradigme objet pour:

1. l'analyse orientée objet de type descendante: La distribution des fonctions (services) de l'objet composé à travers ses composants est représentée par la structure de l'objet composé. La validation du passage d'un niveau de description à un autre est faite par la théorie de la composition supportée par CSL.

2. les cadres d'application: Un cadre d'application est caractérisé par la fonctionnalité globale qu'il offre et par la distribution des fonctions qu'il impose aux objets. Ces deux éléments doivent être documentés explicitement et de manière appropriée pour faciliter l'utilisation des cadres d'application dans la réalisation de logiciels orientés objets. Du fait qu'un cadre d'application soit formé d'un ensemble d'objets qui interagissent, il peut être vu comme étant un objet composé. La spécification d'un tel objet documente la fonctionnalité

globale et la distribution des fonctions du cadre d'application, car la fonctionnalité globale correspond au comportement de l'objet composé et la distribution des fonctions au rôle joué par chaque composant est capturée par la structure de l'objet composé. Comme CSL permet la spécification du comportement des objets composés ainsi que du rôle joué par chaque composant, il peut être utilisé pour documenter les cadres d'application ainsi que les design patterns.

3. La vérification de propriétés multi-objets: Elle requiert avant tout la représentation de ces propriétés. La représentation de ce type de propriété est analogue à la représentation de celles des objets composés. CSL offre des mécanismes de vérification des propriétés de sûreté au niveau de la composition à partir de celles fournies par les composants et en fonction de la structure de la composition.

7.6. Problèmes ouverts

Parmi les problèmes laissés en suspens ou mis en exergue dans cette recherche figurent les points numéro 1 de la visibilité des composants, et 2 de l'impact sur le développement orienté objet en termes d'influence des idées proposées sur l'implantation des applications. Ces questions sont en partie considérées lors de l'examen des problèmes reliés au nommage des objets dans le contexte de la composition tel que présenté dans le chapitre 5.

Il n'en demeure pas moins que la question de la visibilité des composants est une question philosophique, puisque ne pas permettre la visibilité des composants suppose partiellement que la structure d'un objet composé est inconnue des autres objets, donc, que ces derniers ne peuvent en tirer profit. Les avantages reliés à la composition des objets deviennent restreints à la spécification individuelle des objets, confinant l'attrait de la composition à l'étape du raffinement (implémentation) des objets. Par ailleurs, permettre la visibilité des composants suppose résolue la question du nommage des composants visibles et de l'encapsulation des composants cachés. C'est toute la question du traitement des interactions des objets qui doit être revue.

Au niveau de l'implantation, cette recherche n'a pas examiné les modifications à apporter à un langage comme Java ou C++ pour qu'ils supportent les concepts et principes du cadre conceptuel. Nous pensons encore qu'un tel questionnement est prématuré. L'expérience qui sera acquise dans l'utilisation du cadre conceptuel, plus précisément à travers la méthode de développement XOMT (eXtended Object Modeling Technique, voir chapitres 3 et 6) et du langage de spécification CSL va permettre un ajustement du cadre conceptuel.

Au niveau de CSL, une étude approfondie de la relation entre la spécialisation et la composition des objets telle que définie dans cette thèse est requise. En effet, le traitement de la spécialisation dans le chapitre 4 ne tient compte que d'une seule forme de spécialisation: l'extension des comportements. La réduction des comportements n'est pas examinée. En outre, CSL nous offre un cadre formel permettant d'examiner le lien entre la notion de propriété multi-objets et celle de propagation des contraintes à travers les objets. Cet examen n'est pas réalisé dans cette thèse. Vu que le cadre conceptuel cible les applications complexes, le traitement du temps réel est important dans ce contexte. Ce n'est que lorsque ces questions seront clarifiées que l'intégration dans un langage connu pourra être expérimentée. Néanmoins, le cadre de développement offert par XOMT et CSL permet d'examiner la question de la vérification de spécifications, de leur réutilisation, et de leur composition.

Conclusion

Partis de l'inadéquation des approches de modélisation des objets composés proposées dans la littérature (voir chapitre 2), nous avons conclu que c'est le concept même d'objet qui était inapproprié pour la composition des objets. Afin de résoudre cette problématique, notre méthodologie a consisté à définir pour la composition des objets un cadre conceptuel tenant compte des approches de modélisation existantes, des critères permettant de réconcilier ces approches et de notre expérience dans la spécification des applications ayant des objets composés.

Nous avons proposé une extension de la notion d'objet pour y inclure la structure de l'objet. La structure d'un objet non composé est limitée à ses attributs. Par contre, celle d'un objet composé inclut les composants, leurs interconnexions et leurs interactions. Un autre aspect du cadre conceptuel est le lien entre la structure et le comportement des objets composés. Il est établi par l'identification de la contribution de chaque composant au comportement de la composition. Nous avons réalisé ce lien par la classification des propriétés des objets en propriétés inhérentes, et agrégats de propriétés.

Le cadre conceptuel étant une extension du paradigme objet, il nous a fallu examiner son impact sur le processus de développement et les notations utilisés dans le développement orienté objet. En d'autres mots, nous nous sommes penchés sur la mise en oeuvre du cadre conceptuel dans une méthode de développement orienté objet. Cet objectif a nécessité une formulation rigoureuse de la sémantique des concepts, des principes et heuristiques sous-jacents au cadre conceptuel. Cela a engendré d'autres activités de recherche.

Nos travaux montrent qu'une description adéquate de la structure d'objets composés doit indiquer comment ces objets interagissent. L'approche utilisée consiste à relier les interactions aux interconnexions des objets. Pour ce faire,

nous imposons que les interactions aient lieu uniquement dans le contexte des associations d'objets. Ce principe permet, entre autre, de relier la structure des objets à leur comportement. Les interactions sont établies par synchronisation des actions des objets interagissant. En tenant compte des hypothèses sur l'environnement faites par chaque objet et en ajoutant des mécanismes d'abstraction et de raffinement à cette approche, nous démontrons la possibilité de spécifier des compositions conformément au cadre conceptuel.

La théorie de composition sous-jacente à une telle approche permet de prouver des propriétés de la composition à partir de celles des composants. Il est intéressant de noter que cette théorie a beaucoup de ressemblances avec les théories de composition reposant sur la logique temporelle, la théorie des catégories, et la théorie des interfaces. En particulier, elle est très proche de celle proposée par Abadi et Lamport [Abad95]. Nous démontrons que les deux reposent sur les mêmes principes de base et qu'elles ne diffèrent que sur la mise en oeuvre conformément au paradigme objet des principes énoncés. Dans Abadi et Lamport [Abad95], la composition est faite au détriment de l'encapsulation des objets. Dans notre théorie, l'encapsulation est préservée en imposant que la communication entre les objets se fassent uniquement par les actions et leurs paramètres impliqués lors des interactions.

Nous avons repoussé les limites de la modélisation orienté objet par une meilleure prise en compte des aspects reliés à la composition des objets, surtout à la description du comportement des objets composés. Cette thèse offre un cadre conceptuel qui pourrait servir de fondement sémantique à des méthodes de développement orienté objet ainsi qu'à des langages de programmation orienté objet.

La composition des objets étant une problématique vaste, certaines questions qui portent sur l'impact de l'application des concepts et principes du cadre conceptuel sur le développement orienté objet n'ont pu être abordées. Il s'agit notamment des outils de génie logiciel, de l'intégration dans un langage de programmation orienté objet, de la réutilisation et du test des spécifications d'objets composés. Ces interrogations ouvrent la voie à d'autres recherches.

Parmi les problèmes laissés en suspens ou mis en exergue dans cette recherche figurent la visibilité des composants, et l'impact sur le développement orienté objet en termes d'influence des idées proposées sur l'implantation des applications. Au niveau de l'implantation, cette recherche n'a pas examiné les modifications à apporter à des langages comme Java ou C++ pour qu'ils supportent les concepts et principes du cadre conceptuel. Comme nous l'avons mentionné dans le chapitre 7, nous pensons encore qu'un tel questionnement est prématuré.

Il demeure que le cadre conceptuel permet de résoudre les problèmes occasionnés par l'inadéquation du paradigme objet pour:

- l'analyse orientée objet de type descendante: ici, la notion de structure d'un objet composé permet de représenter la distribution des fonctions (services) de l'objet composé à travers ses composants. La validation du passage d'un niveau de description à un autre est faite par la théorie de la composition supportée par CSL.
- la documentation des cadres d'application: cela est favorisé par certaines caractéristiques du langage CSL, notamment: la lisibilité des spécifications CSL, la spécification des hypothèses sur l'environnement des objets, la base formelle du langage, la composition et le raffinement des spécifications, la traduction en TLA (incluant la possibilité d'utiliser des outils de génie logiciel de TLA), la spécification des relations temporelles entre les actions, l'expression des contraintes structurelles entre les objets, la notion d'invariant, la spécification individuelle des objets et la prise en compte de manière explicite au niveau de chaque action des hypothèses sur l'environnement.
- la vérification de propriétés multi-objets: CSL offre des mécanismes de vérification des propriétés de sûreté au niveau de la composition à partir de celles fournies par les composants et en fonction de la structure de la composition.

Bibliographie

[Abad95] Abadi, M. et L. Lamport, Conjoining Specifications, *A CM TOPLAS* , Vol. 17, No 3, 507-534, May 1995.

[Ablo39] Ablowitz, R., The Theory of Emergence, *Philosophy of Science*, Vol. 6, No 1, 1-16, 1939.

[Alle94] Allen, R. et D. Garlan, *Formal Connectors*, CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, March 1994.

[Ande92] Andersen, E. et T. Reenskaug, System Design by Composing Structures of Interacting Objects, *Proceedings of ECOOP'92*, Springer Verlag, LNCS, Vol. 615, 133-152, 1992.

[Angy39] Angyal, A., The Structure of Wholes, *Philosophy of Science*, Vol. 6, No 4, 26-47, 1939.

[Aski92] Askit, M. et L. Bergmans, Obstacles in Object-Oriented Software Development, *Proceedings of OOPSLA'92*, ACM SIGPLAN Notices, Vol. 27, No 10, 341-358, October 1992.

[Bell92] Guttapalle, N., H. Kilov et J. Morabito, *The Materials: A Generic Object Class Library for Analysis*, Information Modeling Concepts and Guidelines, Science and Technology Series, ST-OPT-002010, Issue 1, October 1992, Bellcore.

[Bobr88] Bobrow et al., *Common Lisp Object System Specification X3Jl3*, Document 88-002R, ACM SIGPLAN Notices, Vol. 23, Special issue, September 1988.

[Boch92] Bochmann et al., *The IGLOO Project: Research Proposal*, Technical Description, CRIM, May 1992.

[Boch93b] Bochmann, G.v., *Abstract dynamic modelling of complex systems*, Publication départementale No 863, Dépt. IRO, Université de Montréal, Janvier 1993.

[Booc94] Booch, G., *Object-Oriented Analysis and Design with Applications*, Second Edition, The Benjamin/Cummings Publishing Co. Inc., 1994.

[Booc96] Booch, G., J. Rumbaugh et I. Jacobson, *The Unified Modeling Language for Object-Oriented Development*, Rational Corp., 1996.

[Broo87] Brooks, F., No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE COMPUTER*, Vol. 20, No 4, 10-20, 1987.

[Bung77] Bunge, M., *Treatise on Basic Philosophy, Volume 3, Ontology I: The Furniture of the World*, Dordrecht: D. Reidel Publ. Co, 1977.

[Bung79] Bunge, M., *Treatise on Basic Philosophy, Volume 4, Ontology II: A World of Systems*, Dordrecht: D. Reidel Publ. Co, 1979.

[Carg91] Cargill, T., A Case Against Multiple Inheritance in C++, *Computing Systems*, Vol. 4, No 1, 69-82, 1991.

[Carg92] Cargill, T., *C++ Programming Style*, Addison-Wesley, 1992.

[Chen76] Chen, P.P., The Entity-Relationship model - Toward a unified view of data, *ACM TODS* , Vol. 1, No 1, 9-36, March 1976.

[Cheo94] Cheon, Y. et G. Leavens, The Larch/Smalltalk Interface Specification Language, *TOSEM*, Vol. 3, No 3, 221-253, 1994.

[Chien90] Chien, A. et W. Dally, Concurrent aggregate, *Proc. of Second Symposium on Principles and Practice of Parallel Programming*, ACM SIGPLAN Notices, Vol. 25, No 3, 187-196, 1990.

[Cive93] Civello, F., Roles for composite objects in object-oriented analysis and design, *Proc. of OOPSLA'93*, Andreas Paepcke (Eds), ACM SIGPLAN Notices, Vol. 28, No 10, 376-393 , 1993.

[Coad92] Coad, P., Object-Oriented Patterns, *CACM*, Vol. 35, No 9, 152-159, 1992.

[Cole92] Coleman, D., F. Hayes et S. Bear, Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design, *IEEE TSE*, Vol. 18, No 1, 9-18, January 1992.

[Cole94] Coleman et al., *Object-Oriented Development: THE FUSION METHOD*, Prentice Hall, 1994.

[Cook94] Cook, S. et J. Daniels, *Designing Object Systems*, Prentice Hall, 1994.

[Dami94] Dami, L., *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*, Thèse, Centre universitaire d'information, Université de Genève, 1994.

[deCh91] de Champeaux, D., Object-Oriented Analysis and Top-down Software Development, *Proc. of ECOOP'91*, Pierre America (Ed.), Springer Verlag, LNCS, Vol. 512, 360-376, 1991.

[Duke94] Duke, R., G. Rose et G. Smith, *Object-Z: a Specification Language Advocated for the Description of Standards*, Technical Report 94-45, Software Verification Research Centre, the University of Queensland, Australia, December 1994.

[Feat87] Feather, M., Language Support for the Specification and Development of Composite Systems, *ACM TOPLAS*, Vol. 9, No 2, 198-234, April 1987.

[Fiad92] Fiadeiro, J. et T. Maibaum, Temporal theories as modularisation units for concurrent system specification, *Formal Aspects of Computing*, Vol. 4, No 3, 239-272, 1992.

[Floc97] Flocchini, P., B. Mans et N. Santoro, On the impact of Sense of Direction on Message Complexity, *Information Processing Letters*, Vol. 63, No 1, 23-31, 1997.

[Floc98a] Flocchini, P., B. Mans et N. Santoro, Sense of direction: formal definition and properties, *Networks*, Vol. 32, No 3, 165-180, 1998.

[Floc98b] Flocchini, P. et N. Santoro, Topological constraints for sense of direction, *International Journal on Foundations of Computer Science*, Vol. 9, No 2, 179-198, 1998.

[FlSa98] Flocchini, P. et N. Santoro, Sense of direction in Distributed Computing, *Proceedings of 12th international Symposium DISC '98*, Andros, Springer-Verlag, LNCS, Vol. 1499, 1-15, 1998.

[Gogu86] Goguen, J., Reusing and Interconnecting Software Components, *IEEE Computer*, Vol. 19, No 2, 16-18, 1986.

[Hare88] Harel, D., Statecharts: a Visual Formalism for Complex Systems, *Science of Computer Programming*, Vol. 8, No 3, 231-274, 1988.

[Hart92] Hartmann, T., R. Jungclaus et G. Saake, Aggregation in a behavior Oriented Object Model, *Proceedings of ECOOP'92*, Springer Verlag, LNCS, Vol. 615, 57-77, Utrecht, 1992.

[Hart94] Hartmann et al., *Revised Version of the Modeling Language TROLL (TROLL Version 2.0)*, Informatik-Berichte 94-03, Technische Universitaet Braunschweig, Germany, April 1994.

[Helm90] Helm, R., I. Holland et D. Gangopadhyay, Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, *Proceedings of ECOOP/OOPSLA'90*, ACM SIGPLAN Notices, Vol. 25, No 10, 169-180, 1990.

[Hend90] Henderson-Sellers, B. et J. Edwards, The Object-Oriented Systems Life Cycle, *CACM*, Vol. 33, No 9, 143-159, September 1990.

[Hoff88] Hoffman, D. et R. Snodgrass, Trace specifications: Methodology and models, *IEEE TSE*, Vol. 14, No 9, 1243-1252, September 1988.

[Hogg91] Hogg, J., Islands: Aliasing Protection in Object-Oriented Languages, *Proceedings of OOPSLA'91*, ACM SIGPLAN Notices, Vol. 26, No 11, 271-285, 1991.

[Holl93] Holland, I., *The Design and Representation of Object-Oriented Components*, Thèse, Northeastern University, Boston, 1993.

[ISO89a] ISO, *LOTOS - A Formal Description Technique Based on The Temporal Ordering of Observational Behaviour*, ISO JTC1, IS 8807, 1989.

[ISO89b] ISO, *Estelle - A Formal Description Technique based on Extended State Transition Model*, ISO IS9074, 1989.

[ISO96a] ISO, *Basic reference model of Open Distributed Processing, Part 2: Foundations*, ISO/IEC IS 10746-2, 1996.

[ISO96b] ISO, *Basic reference model of Open Distributed Processing, Part 3: Architecture*, ISO/IEC IS 10746-3, 1996.

[ISO98a] ISO, *Basic reference model of Open Distributed Processing, Part 1: Overview*, ISO/IEC IS 10746-1, 1998.

[ISO98b] ISO, *Basic reference model of Open Distributed Processing, Part 4: Architectural Semantics*, ISO/IEC IS 10746-4, 1998.

[ITU92] ITU, *CCITT Specification and Description Language (SDL)*, ITU-T Recommendation Z.100, 1992.

[IWSS87] Proceedings of the Fourth International Workshop on Software Specification and Design, Monterey, 1987.

[Jack95] Jackson, M. et P. Zave, Deriving Specifications from Requirements: an Example, *Proceedings of ICSE-17*, ACM Press, 15-24, 1995.

[Jaco92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, Reading, MA, 1992.

[Jarv92] Järvinen, H-M., *The Design of a Specification Language for Reactive Systems*, Thèse, Publication 95, Tampere University of Technology, 1992.

[John92] Johnson, R., Documenting frameworks using patterns, *Proceedings of OOPSLA'92*, ACM SIGPLAN Notices, Vol. 27, No 10, 63-76, Vancouver, B.C., October 1992.

[John93] Johnson, R. et W. Opdyke, Refactoring and Aggregation, *Proceedings of Object Technologies for Advanced Software*, Nishio, S. and Yonezawa, A. (Eds.), Springer Verlag, LNCS, Vol. 742, 264-278, 1993.

[Kapp91] Kappel, G. et M. Schrefl, Object/Behavior Diagrams, *IEEE Data Engineering'91 (ICDE91)*, IEEE Computer Society, 530-539, 1991.

[Kapp92] Kappel, G. et M. Schrefl, Local referential integrity, *Proceedings of 11th International Conference On the Entity-Relationship Approach*, Springer Verlag, LNCS, Vol. 645, 41-61, 1993.

[Kent91] Kent, W., A rigorous model of object reference, identity, and existence, *JOOP*, Vol. 4, No. 3, 28-36, June 1991.

[Kris93] Kristensen et al., *Object-Oriented Programming in the BETA programming language*, Addison-Wesley, Reading, MA, 1993.

[Kris94] Kristensen, B., Complex Associations: Abstractions in Object-Oriented Modeling, *Proceedings of OOPSLA'94*, ACM SIGPLAN Notices, Vol. 29, No. 10, 272-286, October 1994.

[Kurk93] Kurki-Suonio, R., Stepwise Design of Real-Time Systems, *IEEE TSE*, Vol. 19, No 1, 56-69, 1993.

[Lam94] Lam, S. et A. Shankar, A Theory of Interfaces and Modules. I -- Composition Theorem, *IEEE TSE*, Vol. 20, No 1, 55-71, January 1994.

[Lamp94] Lamport, L., The Temporal logic of actions, *ACM TOPLAS*, Vol. 16, No 3, 872-923, May 1994.

[Lamp97] Lamport, L. et L. Paulson, *Should Your Specification Language Be Typed?* SRC research Report 147, Digital System Research Center, May 1997.

[Leje67] Lejewski, C., Single axiom for mereological notion of proper part, *Notre Dame Journal of Formal Logic* 8, 279-285, 1967.

[Leon40] Leonard, H. et N. Goodman, The calculus of individuals and its uses, *Journal of Symbolic Logic* 5, 45-55, 1940.

[Liu92] Liu, L., *A formal approach to Structure, Algebra and Communication of Complex Objects*, ISBN 90-9005694-7, Infolab, Université de Tilburg, 1992.

[Mest94] Mester, A. et P. Herrmann, *Tools for TLA-based specifications*, RvS-TLA-94-35, University of Dortmund, February 1994.

[Meta95a] MetaEdit Personal 1.2, Customisable Case Tool to meet your requirements, MetaCase Consulting, Finland, 1995.

[Meta95b] Customer Bulletin 2/95, MetaCase Consulting, Finland,1995.

[Meye96] Meyers, S., *More Effective C++: 32 New ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.

[Mili90] Mili, H., J. Sibert et Y. Intrator, An Object-Oriented Model Based on relations, *Journal of Systems Software*, No 12, 139-155, 1990.

[Miln89] Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989.

[Miln93] Milner, R., Elements of Interaction, Turing Award lecture, *CACM*, Vol. 36, No 1, 78-89, January 1993.

[Misr81] Misra, J. et M. Chandy, Proofs of Networks of Processes, *IEEE TSE*, Vol. SE-7, No 4, 417-426, July 1981.

[Odel94] Odell, J., Six different kinds of Composition, *JOOP*, Vol. 5, No 8, 10-15, January 1994.

[Pete81] Peterson, J. L., *Petri net theory and the modelling of systems*, Prentice Hall, 1981.

[Press93] Pressman, L., *Software Engineering. A Practitionner Approach*, second edition, McGraw-Hill, Inc. 1993.

[Rama95a] Ramazani, D. et G.v. Bochmann, *A Conceptual Framework For Object Composition and Dynamic Behavior Description*, Publication départementale No 949, DIRO, Université de Montréal, Montréal, Canada, 1995.

[Rama95b] Ramazani, D., Contribution of Object-Oriented Methodologies to the Specification of Complex Systems, *Proceedings of Fifth Complex Systems Engineering Synthesis and Assessment Technology Workshop (CSESAW'95)*, 183-186, 1995.

[Rama96a] Ramazani, D. et G.v. Bochmann, Specification of Composite Objects Based on the ODP Reference Model, *Proceedings of FMOODS96*, 205-220, March 1996.

[Rama96b] Ramazani, D. et G.v. Bochmann, Extending Object Modelling Technique for the Specification of Composite Objects, *Proceedings of TOOLS-USA96*, July 1996.

[Rama97] Ramazani, D. et G.v. Bochmann, Approaches to the Specification of Object Associations, *Proceedings of FMOODS97*, July 1997.

[Rama98a] Ramazani, D. et G.v. Bochmann, *Composition Specification Language*, Publication départementale No 1134, DIRO, Université de Montréal, Montréal, Canada, Novembre 1998.

[Rama98b] Ramazani, D., P. Flocchini et G.v. Bochmann, *Object Identity and Object Composition*, Publication départementale No 1135, DIRO, Université de Montréal, Montréal, Canada, Novembre 1998.

[Rama99] Ramazani, D. et G.v. Bochmann, Object Composition: A Case Study, *Proceedings of FMOODS99*, à paraître.

[Reen92] Reenskaug et al., OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems, *JOOP*, Vol. 5, No 6, 27-41, October, 1992.

[Robi92] Robinson, P., *Hierarchical Object-oriented Design*, Chapman & Hall, 1992.

[Rumb87] Rumbaugh, J., Relations as Semantic Constructs in an Object-Oriented Language, *Proceedings of OOPSLA Conference*, ACM SIGPLAN Notices, Vol. 22, No 12, 466-481, 1987.

[Rumb88] Rumbaugh, J., Controlling Propagation of Operations using Attributes on Relations, *Proceedings of OOPSLA'88 Conference*, ACM SIGPLAN Notices, Vol. 23, No 11, 285-296, 1988.

[Rumb91] Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Rumb93a] Rumbaugh, J., Disinherited! Examples of misuse of inheritance, *JOOP*, Vol. 5, No 9, 22-24, February 1993.

[Rumb93b] Rumbaugh, J., Controlling Code, How to implement Dynamic Models, *JOOP*, 25-30, May 1993.

[Rumb94] Rumbaugh, J., Building Boxes: Composite Objects, *JOOP*, Vol. 7, No 7, 12-22, November/December, 1994.

[Rumb95a] Rumbaugh, J., OMT: The object model, *JOOP*, Vol. 7, No 8, 21-27, January, 1995.

[Rumb95b] Rumbaugh, J., OMT: The dynamic model, *JOOP*, Vol. 7, No 9, 6-12, February, 1995.

[Rumb95d] Rumbaugh, J., OMT: The functional model, *JOOP*, Vol. 8, No 1, 10-14, March/April, 1995.

[Rumb95e] Rumbaugh, J., OMT: The development process, *JOOP*, Vol. 8, No 1, 8-16, May, 1995.

[Rumb95f] Rumbaugh, J., Taking things in context: Using composites to build models, *JOOP*, Vol. 8, No 7, 6-11, November-December, 1995.

[Rumb96] Rumbaugh, J., Models for design: Generating code for associations, *JOOP*, Vol. 8, No 9, 13-7, 1996.

[Sakk89] Sakkinen, M., Disciplined Inheritance, *Proceedings of ECOOP Conference*, Cambridge University Press, 39-56, 1989.

[Tver84] Tversky, B. et K. Hemenway, Objects, Parts, and Categories, *Journal of experimental Psychology: General*, Vol. 113, No 2, 169-191, 1984.

[Walk90] Walker, D., *π-calculus Semantics of Object-Oriented Programming Languages*, Report ECS-LFCS-90-122, Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, October 1990.

[Wand89] Wand, Y., A proposal for formal object model, W. Kim, F.H. Lochovsky, Eds, *Object-Oriented Concepts, Databases, and Applications*, ACM Press and Addison-Wesley, 537-559, 1988.

[Wang93] Wang, Y. et D. L. Parnas, Simulating the Behaviour of Software Modules by Trace Rewriting, *Proc. of ICSE-15*, 14-23, 1993.

[Wier95] Wieringa, R. et W. de Jonge, Object identifiers, keys, and surrogates: object identifiers revisited, *Theory and Practice of Object Systems*, Vol. 1, 101-114, 1995.

[Wins87] Winston, M., R. Chaffin et D. Hermann, A taxonomy of part-whole relation, *Cognitive Science*, No 11, 417-444, 1987.

[Wirf90] Wirfs-Brocks, R., B. Wilkerson et L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[X.500] Information Processing - Open Systems Interconnections - *The Directory*, ITU Recommendation X.500/ISO/IEC/IS 9594, Geneva, Switzerland, 1993.

[Zave93] Zave, P. et M. Jackson, Conjunction as composition, *ACM TOSEM*, Vol. 2, No 4, 379-411, October 1993.

[Zave96] Zave, P. et M. Jackson, Where do Operations come from? A Multiparadigm Specification Technique, *IEEE TSE*, Vol. 22, No 7, 508-528, 1996.